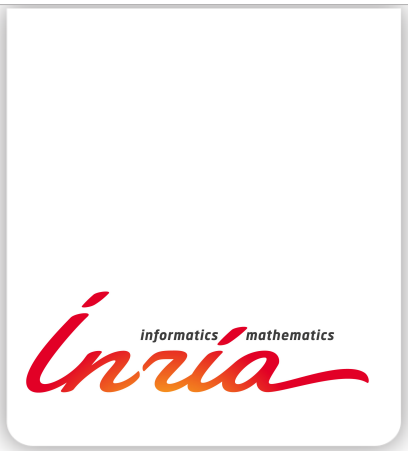
Hierarchical hybrid sparse linear solver for multicore platforms

E. Agullo, L. Giraud, S. Nakov, J. Roman

RESEARCH REPORT
N° 8960
October 2016
Project-Teams HiePACS

ISSN 0249-6399 ISRN INRIA/RR--8960--FR+ENG



Hierarchical hybrid sparse linear solver for multicore platforms

E. Agullo*, L. Giraud*, S. Nakov*, J. Roman*†

Project-Teams HiePACS

Research Report n° 8960 — October 2016 — 25 pages

Abstract: The solution of large sparse linear systems is a critical operation for many numerical simulations. To cope with the hierarchical design of modern supercomputers, hybrid solvers based on Domain Decomposition Methods (DDM) have been been proposed. Among them, approaches consisting of solving the problem on the interior of the domains with a sparse direct method and the problem on their interface with a preconditioned iterative method applied to the related Schur Complement have shown an attractive potential as they can combine the robustness of direct methods and the low memory footprint of iterative methods. In this report, we consider an additive Schwarz preconditioner for the Schur Complement, which represents a scalable candidate but whose numerical robustness may decrease when the number of domains becomes too large. We thus propose a two-level MPI/thread parallel approach to control the number of domains and hence the numerical behaviour. We illustrate our discussion with large-scale matrices arising from real-life applications and processed on both a modern cluster and a supercomputer. We show that the resulting method can process matrices such as tdr455k for which we previously either ran out of memory on few nodes or failed to converge on a larger number of nodes. Matrices such as Nachos_4M that could not be correctly processed in the past can now be efficiently processed up to a very large number of CPU cores (24,576 cores). The corresponding code has been incorporated into the MaPHyS package.

Key-words: High Performance Computing (HPC); multicore architecture; sparse linear solver; hybrid method; direct method; Krylov method; MPI; multi-threading; hardware locality.

* Inria Bordeaux Sud-Ouest, 200 avenue de la Vieille Tour, F-33400 Talence, France
† Bordeaux, INP

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vielle Tour
33405 Talence Cedex

Solveur linéaire hybride hiérarchique pour calculateurs multi-cœurs

Résumé : La résolution de grands systèmes linéaires creux est une opération centrale dans de nombreuses simulations numériques. Pour s'adapter à la structure hiérarchique des super-calculateurs modernes, des solveurs hybrides basés sur des méthodes de décomposition de domaine (DDM) ont été proposées. Parmi ceux-ci, ceux qui consitent à résoudre les problèmes intérieurs avec une méthode directe et le système réduit aux interfaces avec une méthode de Krylov préconditionnée ont montré un potentiel attractif puisqu'ils combinent la robustesse des méthodes directes et le faible encombrement mémoire des méthodes itératives. Dans ce rapport, nous considérons un préconditionneur de type Schwarz additif pour le complément de Schur qui est un candidat pour le passage à l'échelle, bien que sa robustesse puisse être altérée lorsqu'on utilise un grand nombre de domaines. Dans ce contexte, nous proposons une approche à deux niveaux de parallélisme qui combine de l'échange de messages et des threads afin de contrôler le nombre de domaines et donc le comportement numérique de la partie itérative. Nous illustrons cette approche sur des exemples de grande taille issus de problèmes réels via des expérimentations aussi bien sur des clusters de calcul que sur des super-calculateurs. Nous montrons en particulier que notre implantation est capable de traiter la matrice tdr455k pour laquelle l'implantation précédente soit dépassait les capacités mémoire des nœuds de calcul lorsque peu de sous-domaines étaient utilisés, soit ne convergeait pas lorsqu'on augmentait le nombre de domaines pour contourner le problème mémoire. Des matrices telles que Nachos_4M, qui étaient auparavant hors de portée, peuvent maintenant être traitées efficacement en utilisant jusqu'à un nombre important de cœurs (24,576 cœurs). L'implantation de cette méthode a été intégrée dans le logiciel MaPHyS.

Mots-clés : Calcul haute performance (HPC); mulicore architecture; algèbre linéaire creuse; solveur hybride; méthode directe; méthode de Krylov; multi-threading; localité matérielle.

1 Introduction

Parallel sparse linear algebra solvers are often the innermost numerical kernels in scientific and engineering applications; consequently, they are one of the most time consuming parts. In order to cope with the hierarchical hardware design of modern large-scale supercomputers, the HPC solver community has proposed new sparse methods. One promising approach towards the high-performance, scalable solution of large sparse linear systems in parallel scientific computing consists of combining direct and iterative methods. To achieve a high scalability, algebraic domain decomposition methods are commonly employed to split a large size linear system into smaller size linear systems that can be efficiently and concurrently handled by a sparse direct solver while the solution along the interfaces is computed iteratively [27, 25, 13, 11]. Such an hybrid approach exploits the advantages of both direct and iterative methods. The iterative component allows us to use a small amount of memory and provides a natural way for parallelization. The direct part provides its favorable numerical properties; furthermore, this combination provides opportunities to exploit several levels of parallelism.

With the need of solving ever larger sparse linear systems while maintaining numerical robustness, multiple variants for computing the preconditioner for the Schur complement of such hybrid solvers have been proposed. PDSLin [21], ShyLU [25] and Hips [10] first perform an exact[1] factorization of the interior of each subdomain concurrently. PDSLin and ShyLU then compute the preconditioner with a two-fold approach. First, an approximation $\widetilde{\mathcal{S}}$ of the (global) Schur complement $\mathcal{S}$ is computed. Second, this approximate Schur complement $\widetilde{\mathcal{S}}$ is factorized to form the preconditioner for the Schur Complement system, which does not need to be formed explicitly. While PDSLin has multiple options for discarding values lower than some user-defined thresholds at different steps of the computation of $\widetilde{\mathcal{S}}$, ShyLU [25] also implements a structure-based approach for discarding values named *probing* and that was first proposed to approximate interfaces in DDM [8]. Instead of following such a two-fold approach, Hips [10] forms the preconditioner by computing a global ILU factorization based on the multi-level scheme formulation from [17]. Finally, object of this study, MaPHyS [13] computes an additive Schwarz preconditioner for the Schur complement.

To ensure numerical robustness while exploiting all the processors of a platform, an important effort has been devoted to propose two levels of parallelism for these solvers. Relying on the SuperLU_DIST [22] distributed memory sparse direct solver, PDSLin implements a 2-level MPI (MPI+MPI) approach with finely tuned intra- and inter-subdomain load balancing [27]. A similar MPI+MPI approach has been assessed for additive Schwarz preconditioning in a prototype version of MaPHyS [14], relying on the Mumps [3, 4] and ScaLAPACK [5] sparse direct and dense distributed memory solvers, respectively. On the contrary, expecting a higher numerical robustness thanks to multi-level preconditioning, Hips associates multiple subdomains to a single process and distributes the subdomains to the processes in order to maintain load balancing [10]. Finally, especially tuned for modern multicore platforms, ShyLU implements a 2-level MPI+thread approach [25].

Whereas PDSLin and ShyLU can be virtually turned into to a pure direct method if no dropping is performed, and whereas Hips may expect robustness by relying on a multilevel scheme, additive Schwarz preconditioners are extremely local. As a result, their computation is potentially much more parallel (and scalable), but their application may lead to a dramatic increase of the number of iterations (or even to non convergence) if the number of subdomains becomes too large. The objective of the present study is to assess whether a 2-level parallel approach allows additive Schwarz preconditioning for Schur Complement methods to achieve an efficient trade-off between numerical robustness and performance on modern hierarchical multicore platforms. Following the parallelization scheme adopted in [25], we have designed an MPI+thread approach (Section 3) to cope with these hardware trends. Contrary to the MPI+MPI approach investigated in [14],

[1] There are also options for computing Incomplete LU (ILU) factorizations of the interiors but the related descriptions are out the scope of this paper.

such a parallelization allows for a better usage of the memory (*e.g.,* symbolic data structures such as the elimination tree used within a direct solver are shared between threads of a same process) and a better exploitation of the CPU cores local to a node at each step of the parallel computation [1, 9, 20], as we show in Section 4.

The rest of the paper is organized as follows. In Section 2, we present the numerical schemes used in sparse hybrid solvers; in particular, we introduce the additive Schwarz preconditioning for the Schur Complement and its MPI parallelization in MaPHyS. The 2-level MPI+thread extension (first contribution) of this baseline version is then introduced in Section 3. A detailed performance analysis (second contribution) on large test cases is then presented in Section 4 before concluding in Section 5.

2 Baseline flat MPI design of an hybrid solver

Let $\mathcal{A}x = b$ be the linear problem and $\mathcal{G} = \{V, E\}$ the adjacency graph associated with $\mathcal{A}$. In this graph, each vertex is associated with a row or column of the matrix $\mathcal{A}$ and it exists an edge between the vertices i and j if the entry $a_{i,j}$ is non zero. In the sequel, to facilitate the exposure and limit the notation we voluntarily mix a vertex of $\mathcal{G}$ with its index depending on the context of the description. The governing idea behind substructuring or Schur complement methods is to split the unknowns in two categories: interior and interface vertices. We assume that the vertices of the graph $\mathcal{G}$ are partitioned into $\mathcal{N}$ disconnected subgraphs $\mathcal{I}_1$, ..., $\mathcal{I}_\mathcal{N}$ separated by the global vertex separator Γ. We also decompose the vertex separator Γ into non-disjoint subsets Γ_i, where Γ_i is the set of vertices in Γ that are connected to at least one vertex of $\mathcal{I}_i$. Notice that this decomposition is not a partition as $\Gamma_i \cap \Gamma_j \neq \emptyset$ when the set of vertices in this intersection defines the separator of $\mathcal{I}_i$ and $\mathcal{I}_j$. By analogy with classical domain decomposition in a finite element framework, $\Omega_i = \mathcal{I}_i \cup \Gamma_i$ will be referred to as a subdomain with internal unknowns $\mathcal{I}_i$ and interface unknowns Γ_i. If we denote $\mathcal{I} = \cup \mathcal{I}_i$ and order vertices in $\mathcal{I}$ first, we obtain the following block reordered linear system

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

where x_Γ contains all unknowns associated with the separator and $x_\mathcal{I}$ contains the unknowns associated with the interiors. Because the interior vertices are only connected to either interior vertices in the same subgraph or with vertices in the interface, the matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ has a block diagonal structure, where each diagonal block corresponds to one subgraph $\mathcal{I}_i$. Eliminating $x_\mathcal{I}$ from the second block row of Equation (1) leads to the reduced system

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

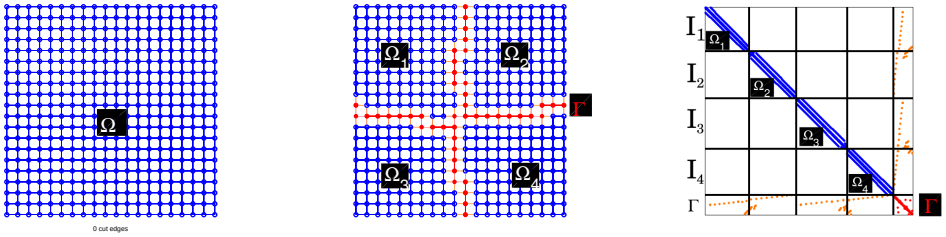
where

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

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 solve (2). Once x_Γ is known, $x_\mathcal{I}$ can be computed with one additional solve for the interior unknowns via

$$x_\mathcal{I} = \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} \left(b_\mathcal{I} - \mathcal{A}_{\mathcal{I}\Gamma} x_\Gamma \right).$$

We illustrate in Figure 1a all these notations for a decomposition into 4 subdomains. The local interiors are disjoint and form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$ (blue vertices in Figure 1b). It is not necessarily the case for the boundaries. Indeed, two subdomains Ω_i and Ω_j may share part of their interface ($\Gamma_i \bigcap \Gamma_j \neq \emptyset$), such as Ω_1 and Ω_2 in Figure 1b which share eleven vertices. Altogether, the local boundaries form the overall interface $\Gamma = \cup \Gamma_i$ (red vertices in Figure 1b), which is not a disjoint union.



(a) Initial global graph. (b) Graph subdomains. (c) Block reordered matrix.

Figure 1: Domain decomposition into four subdomains $\Omega_1, \ldots, \Omega_4$. The initial domain Ω may be algebraically represented with the graph $\mathcal{G}$ associated to the sparsity pattern of matrix $\mathcal{A}$ (a). The local interiors $\mathcal{I}_1, \ldots, \mathcal{I}_\mathcal{N}$ form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$ (blue vertices in (b)). They interact with each others through the interface Γ (red vertices in (b)). The block reordered matrix (c) has a block diagonal structure for the variables associated with the interior $\mathcal{A}_{\mathcal{I}\mathcal{I}}$.

While the Schur complement system is significantly smaller and better conditioned than the original matrix $\mathcal{A}$, it is important to consider further preconditioning when employing a Krylov method. We refer to [21], [25] and [10] for details on preconditioning strategies used in PDSLin, ShyLU and Hips, respectively, and we now introduce the additive Schwarz preconditioner considered in MaPHyS. This preconditioner was originally proposed in [7] and successfully applied to large problems in real life applications in [12, 15, 23]. To describe the main preconditioner in MaPHyS, we define $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} \mathcal{S} \mathcal{R}_{\Gamma_i}^T$, that corresponds to the restriction of the Schur complement to the interface Γ_i. If $\mathcal{I}_i$ is a fully connected subgraph of $\mathcal{G}$, the matrix $\bar{\mathcal{S}}_i$ is dense.

With these notations the additive Schwarz preconditioner reads

$$\mathcal{M}_{AS} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{4}$$

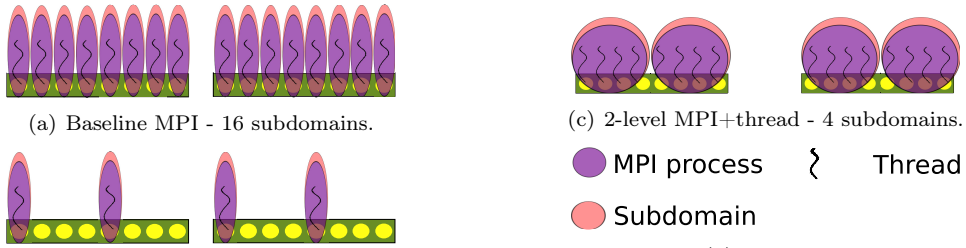
With all these components, the classical parallel implementation of MaPHyS can be decomposed into four main phases (see [15] for more details):

- *the partitioning step* consists of partitioning the adjacency graph $\mathcal{G}$ of $\mathcal{A}$ into several subdomains and distribute the $\mathcal{A}_i$ to different processes. For this, we are able to use two state-of-the-art partitioners, Scotch [24] and METIS [19];
- *the factorization of the interiors* and the computation of the local Schur complement factorizes $\mathcal{A}_i$ with the PaStiX [16] or the Mumps [3] sparse direct solver and furthermore provides the associated local Schur Complement $\mathcal{S}_i$ thanks to recent developments from the development teams of those sparse direct solvers;
- *the setup of the preconditioner* assembles diagonal blocks of $\mathcal{S}_i$ via a few neighbour to neighbour communications to form $\bar{\mathcal{S}}_i$ and factorize this local dense matrix with Mkl;
- *the solve step* is twofold: a parallel preconditioned Krylov method is performed on the reduced system (Equation 2) to compute x_{Γ_i} where all BLAS operations are provided by Mkl, followed by a back solve on the interiors to compute $x_{\mathcal{I}_i}$ with the sparse direct solver.

3 Design of a 2-level MPI+thread extension of MaPHyS

The baseline MPI model presented above presents some strong limitations on modern multicore supercomputers. Indeed, modern platforms tend to have more and more CPU cores. Yet, the

baseline model assigns exactly one subdomain per core being used. As a consequennce, if one wants to exploit all available computational cores, he has to decompose the matrix in as many subdomains. For instance, if two nodes of eight cores each are being used, 16 subdomains have to be used in order to exploit all the 16 cores (see Figure 2(a)). As shown later on this study (Section 4), this approach can (1) deteriorate or even prevent the convergence of the method at scale and (2) lead to excessive memory consumption, possibly leading the method to run out-of-memory. Solving those issues with the baseline MPI MaPHyS implementation would impose to assign fewer subdomains per node, hence fewer cores. For instance, on the same example platform, imposing four subdomains would limit us to use four cores only (see Figure 2(b)). To overcome



(a) Baseline MPI - 16 subdomains.

(b) Baseline MPI - 4 subdomains.

(c) 2-level MPI+thread - 4 subdomains.

(d) Caption.

Figure 2: Baseline MPI (a, b) versus MPI+thread (c) models on a 16 cores platform composed of two nodes of eight cores each. Imposing four subdomains limits the baseline MPI model to exploit four cores (b) but not the MPI+thread model (c).

these limitations while using all available cores, we have designed a 2-level MPI+thread method. With such a method, each process associated to a subdomain can use multiple cores thanks to multithreading. In our example, each subdomain would thus be processed by four cores using four threads per process (see Figure 2(c)).

In this section, we present the design of our MPI+thread extension of MaPHyS. Because MaPHyS is modular, we first need to select and possibly tune multithreaded versions of the external software packages MaPHyS relies on (Section 3.1). We then need to make sure that those libraries cooperate correctly in order to maximize the overall performance of our hybrid solver (Section 3.2). Among other important issues, the physical mapping of the threads on the processing units is critical. We present the three different binding strategies that have been studied in Section 3.3.

3.1 Multithreaded building block operations

We focus in this section on the description of multithreaded version of the external software packages that are used by MaPHyS. The 2-level parallel implementation will be effective for our hybrid solver if its three main computational phases can be efficiently performed in parallel: *factorization of the interiors*, *setup of the preconditioner* and *solve step*. We do not consider the parallel implementation of the *partitioning step* for two reasons. First, with the current state of MaPHyS, the domain decomposition is performed sequentially in a pre-processing phase, which computes the decomposition and sends the needed data on the corresponding processes. Furthermore, the current software version of the partitioners Scotch and METIS are not multithreaded. Let us quickly recall the main numerical kernels of the three computational steps of our solver and for each of them describe the multithreaded strategy.

For the *factorization of the interiors* with the current state of our package, we are able to use two sparse direct solvers, PaStiX and Mumps. These packages are able to perform Cholesky, LU or LDL^T factorizations of symmetric definite, symmetric and general sparse matrices respectively. Both solvers are state of the art software, implemented with a large number of numerical and

technical functionalities. The major difference between the two solvers is that MUMPS is based on the multifrontal approach while PASTIX is a supernodal solver. With the recent development of the MUMPS group, activities towards a multithreaded version have been accomplished [20], but when this study was developed, only the PASTIX package offered the possibility of using multithreading. In this study only PASTIX is considered and a more detailed description is given below.

Two different versions of the *setup of the preconditioner* are possible in our solver, sparse and dense. For each one, a different library is used. For the sparse version, the PASTIX solver is used, while for the dense version, the LAPACK routines of the MKL library are used. For the iterative solver, the BLAS operations are provided by the MKL library and the application of the preconditioner is performed by the MKL library or by the back-solve operation of the sparse direct solver that is used. Finally the black-solve on the interiors is performed by the sparse direct solver. All in all, we use the PASTIX sparse direct solver and the MKL library.

3.1.1 Multithreaded MKL

The MKL library provides highly optimized BLAS and LAPACK routines for Intel processors. The multithreaded version of the MKL library is implemented on top of the Intel OpenMP runtime system.

3.1.2 Multithreaded sparse direct solver

Sparse direct solvers usually consist of three distinct steps. First, the analysis step performs a reordering of the variables (by calling a third-party library such as SCOTCH or METIS) and a symbolic factorization in order to compute data structures that cope with expected fill-in. Second, the matrix is decomposed in factors in the so-called numerical factorization or factorization step for short. In MAPHYS, these first two steps occur in sequence during the *factorization of the interiors* (and separately during the computation of a sparse preconditioner). Third, the solve step computes the solution from the right-hand side and the factors. This third step occurs during the *back-solve on the interiors* (and separately during the application of the preconditioner if a spare preconditioner is used).

PASTIX is based on the supernodal technique. Once the block symbolic factorization has been performed, the block elimination tree and the supernode partition are available. Basically the supernodal method consists of regrouping several nodes (columns) of the elimination tree in a larger set called "supernode". From the matrix point of view, the supernodal approach can be viewed as grouping several contiguous columns in a single column-block, each of them corresponding to a supernode (see [16] for details). Instead of using inefficient kernels with low fetch/compute ratio, well suited BLAS-3 operations can then be used and exploit much more efficiently modern processing units. The supernodal factorization occuring in PASTIX consists of performing three operations for each column-block of the matrix, as depicted in Figure 3 for the first column-block. First the factorization of the diagonal block is computed (red block in Figure 3). Then a triangular solve is performed on the off-diagonal part of the column-block (yellow blocks in Figure 3). For each off-diagonal sub-block of the factorized column-block, an update to the facing column-block that owns the same rows is applied (green blocks in Figure 3). This is repeated for each column-block until the whole matrix is factorized. In PASTIX each operation is performed by one thread and the parallelism is expressed by performing several independent matrix operations simultaneously.

In the baseline version of PASTIX, multithreading was implemented for the factorization of the interiors $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ but the computation of the Schur ($\mathcal{S}_i$) was still performed sequentially. For the purpose of this study, we relieved this bottleneck by enabling concurrent updates of $\mathcal{S}_i$ as long as the updates are performed on disjoint matrix blocks (the three green blocks in the red bottom-right block in Figure 3). In that respect, we considered a 2D block decomposition of the square dense Schur complement matrix. The concurrency is protected by a simple deadlock-free algorithm consisting of performing active waiting until all the blocks accessed by an update are free for being updated. This additional software development designed to remove the final bottleneck

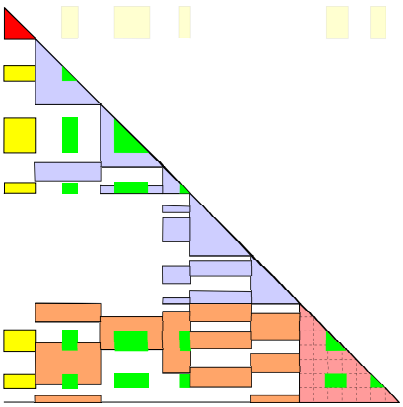


Figure 3: Operations related to the factorization of the first supernode (red block): triangular solve on the off-diagonal blocks (yellow blocks) and update to the facing column-blocks that owns the same rows (green blocks).

in the Schur complement capability of PaStiX has been eventually integrated in the public domain distribution of PaStiX.

3.2 Coexistence of different multithreaded libraries

The main issue of interoperability between PaStiX and Mkl is related to the different ways these libraries manage the parallelism. As explained above, PaStiX expresses the parallelism through individual tasks associated with different supernodes; those tasks are handled by single threads. The supernode calculation is performed by sequential (mono-threaded) BLAS-3 routines from the Mkl library. However, the multithreading exploited in the other steps of MaPHyS such as the *setup of the preconditioner* and the Krylov iterations is based on calls to multithreaded Mkl routines. Consequently, both mono and multithreaded Mkl subroutines need to be used in different contexts by MaPHyS.

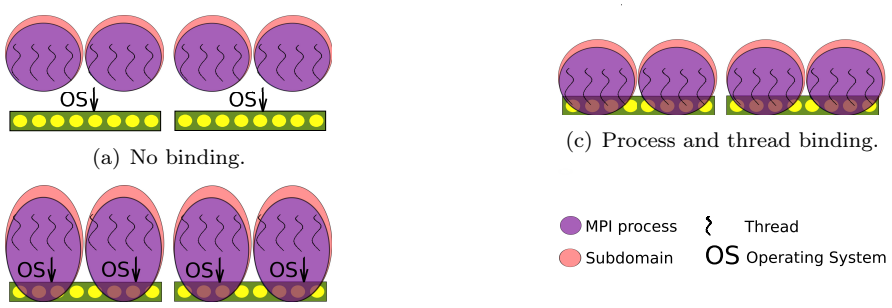
Because the Mkl library relies on OpenMP, there are two different ways to control multi-threading: through the OpenMP functionalities or directly using functions defined in the Mkl library such as $mkl\_set\_num\_threads(nb\_threads)$ or $omp\_set\_num\_threads(nb\_threads)$. The Mkl functions override the OpenMP functions. We point out below the adaptation that needs to be performed for the main three steps of MaPHyS.

First during the *factorization of the interiors*, PaStiX is used so the mono-threaded version of Mkl needs to be used. For the *setup of the preconditioner*, depending on the dense or sparse version of the preconditioner that is used, either Mkl or PaStiX is used respectively. In a similar manner, during the iterative method, Mkl or PaStiX is used depending on the preconditioner that is used. For the rest of the routines within the iterative part, the multithreaded version of Mkl is used. For the back-solve on the interiors, PaStiX is used with the mono-threaded Mkl.

3.3 Binding strategies

Three binding variants have been initially explored. The first strategy simply consists of not enabling any binding and let the operating system handle the mapping of processes and threads (see Figure 4(a)). The second strategy consists of binding processes to a subset of cores and let the operating system handle the mapping of threads (see Figure 4(b)). The third strategy consists of binding both processes and threads within processes (see Figure 4(c)).

The mechanism for binding processes and threads may differ depending on the considered target operating system and architecture. To perform a consistent and portable binding, we rely on the



(a) No binding.

(b) Process binding.

(c) Process and thread binding.

(d) Caption.

Figure 4: Binding strategies.

Portable Hardware Locality (`hwloc`) software package[6]. The `hwloc` library indeed provides a portable interface that is an abstraction of all the low level technical details (operating system and architecture). It also offers a consistent numbering of the physical cores, ensuring that cores that share cache memory will be numbered next to each other. This is not always the case with the raw information provided by the operating system. The numbering of `hwloc` is thus better suited for designing portable binding algorithms.

Modern multithreaded BLAS libraries commonly perform mapping of processes and threads. It is for instance the case of the MKL library on which we rely in this study. In order to control the binding by our own, we therefore prevent MKL from doing it. This is achieved by setting the environment variable $KMP\_AFFINITY$ to "disable".

We further discuss the impact of binding on performance in Section 4.2 where we show that the second strategy consisting of binding processes to a subset of cores and let the operating system handle the mapping of threads (see Figure 4(b)) is consistently optimum. For this reason, this strategy will be consistently used in the rest of this paper.

4 Performance analysis

In this section, a large set of experiments is presented to analyze the parallel and numerical performance of the proposed MPI+thread extension of MAPHYS. After selecting the most appropriate binding strategy (Section 4.2), we study the impact of multithreading on performance (Section 4.3). In that context, the number of subdomains is set to the number of nodes and we only vary the number of threads per subdomain. We then study the flexibility of our 2-level parallel implementation to achieve the best trade-off between the numerical and parallel behaviors (Section 4.4). For those experiments, we vary the number of subdomains and the number of threads per subdomain, so that the total number of cores remains constant.

4.1 Experimental setup

Matrix	`Matrix211`	`Audi_kw`	`Nachos`	`Haltere`	`Tdr455k`	`Nachos4M`	`Amande`
n	0.8M	0.9M	1.1M	1.3M	2.7M	4.1M	7.0M
nnz	129M	392M	40M	10M	113M	256M	58M
Symmetry	non-symmetric	SPD	non-symmetric	symmetric	non-symmetric	non-symmetric	symmetric
Arithmetic	real	real	complex	complex	complex	complex	complex

Table 1: Matrices used in this study.

The experiments presented in this paper were conducted on two platforms: `PlaFRIM`, located at Inria Bordeaux-Sud-Ouest and the `Hopper` machine from NERSC, located at the Lawrence Berkeley National Laboratory. The `PlaFRIM` platform is composed of 68 nodes. Each node is equipped with two Quad-core Nehalem Intel Xeon X5550 with a total of eight cores per node. Each node has a total of 24Gb of RAM memory. The nodes are interconnected in a fat tree fashion using an Infiniband QDR network delivering a 40Gb/s bandwidth. The `Hopper` platform is composed by 6384 nodes, each being two twelve-core AMD 'MagnyCours' 2.1-GHz processors. Each node has 24Gb of RAM. `Hopper` compute nodes are connected via a custom high-bandwidth, low-latency network provided by Cray.

We display in Table 1 the main characteristics of the test matrices considered in this section. They arise from different application domains ranging from structural mechanic analysis to electromagnetics calculation using both classical and discontinuous finite element modeling.

The *normwise backward error* is used for our study. As the iterative method is applied on the reduced system introduced in Equation 2, one option consists in using scaled residual norm associated to that system:

$$\varepsilon_f = \frac{\|\mathcal{S}x_\Gamma - f\|}{\|f\|}. \tag{5}$$

However, this criterion is related to the reduced system but not necessarily to the global system. Indeed, assuming the backsolve on the interior is performed in exact arithmetic, we have: $\frac{\|\mathcal{S}x_\Gamma - f\|}{\|f\|} = \frac{\|\mathcal{A}x - b\|}{\|f\|}$. Because $\|f\|$ depends on the number of subdomains, this criterion may thus not be appropriate for comparisons between executions whose number of subdomains differs. We rely on the alternative stopping criterion:

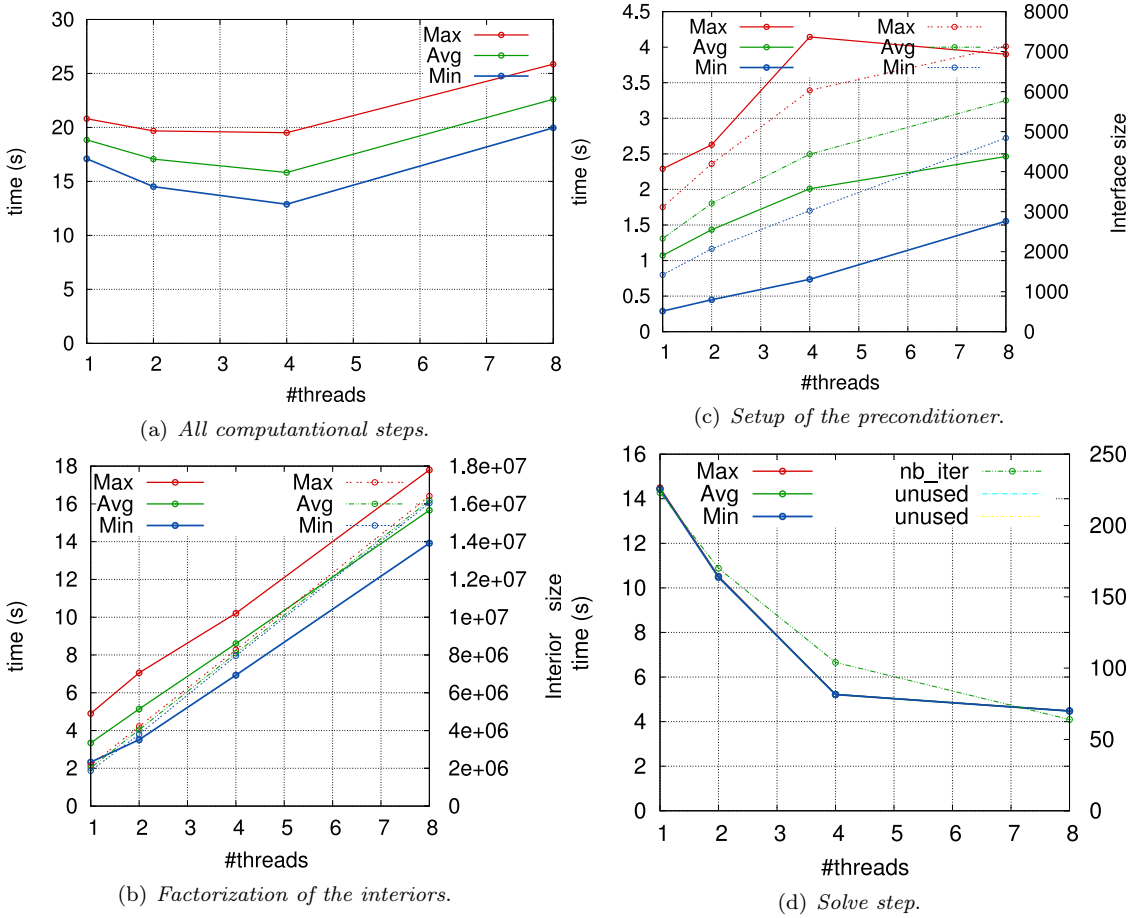
$$\varepsilon_b = \frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|}. \tag{6}$$

This latter stopping criterion indeed reflects the error on the global system because, assuming the backsolve on the interior is performed in exact arithmetic, we have: $\frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|} = \frac{\|\mathcal{A}x - b\|}{\|b\|}$. We use this latter convergence criterion which we set to $\varepsilon_b = 10^{-10}$ when not stated otherwise. The right preconditioned GMRES restart parameter is set to 500 and the maximum number of iterations is set to 7000. We use the version 4.0.1 of METIS to perform the domain decomposition and rely on the multithreaded version of MKL and the modified version of PASTIX discussed in Section 3.

4.2 Impact of the thread binding strategies

As discussed in Section 3, three binding strategies have been designed consisting either of fully delegating the mapping management to the operating system (see Figure 4(a)), binding processes to a subset of cores and let the operating system handle the mapping of threads (see Figure 4(b)) or binding both processes and threads within processes (see Figure 4(c)), respectively. Intensive experiments (not reported here) have been conducted on all matrices from Table 1 showing that the second strategy consisting of binding processes to a subset of cores and letting the operating system handle the mapping of threads (see Figure 4(b)) is consistently optimum. Indeed, this configuration outperforms the case where threads are also binded by the application for two reasons. First, binding threads at the application level prevents MKL from doing internal optimizations related to multithreading. Second, every time the number of threads used by the MKL library is changed, the binding needs to be performed again. This effects turns out to be especially significant in the iterative method when the sparse preconditioner is used, because it has to be done twice at each iteration for switching between calls to the dense operations performed by MKL and the call to the sparse direct solver for applying the preconditioner.

It also outperforms the case where the mapping is fully delegated to the operating system. We report here a detailed analysis on a particular example, considering matrix `Matrix211` processed on 8 nodes of `PlaFRIM` with a dense preconditioner. We compare the behavior of MAPHYS when



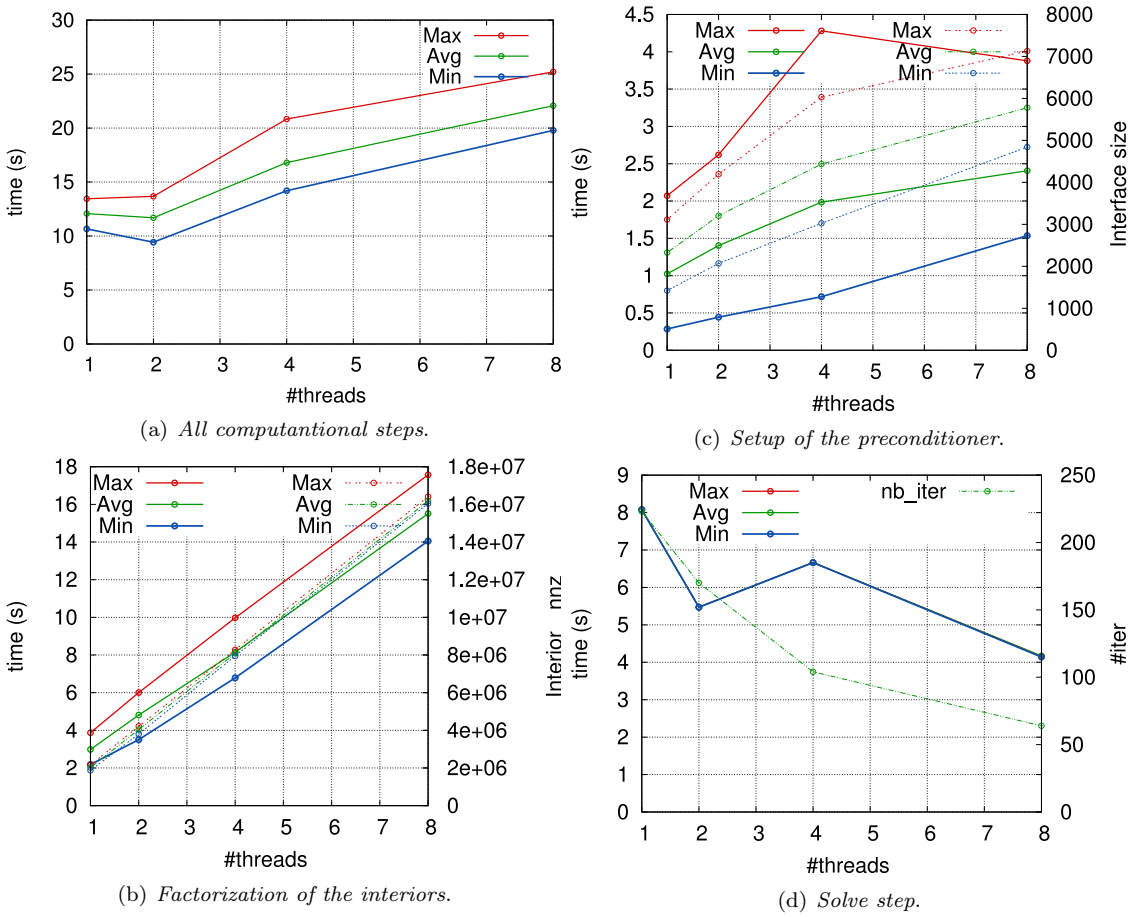
(a) *All computantional steps.*

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 5: Performance of MaPHyS when binding management is fully delegated to the operating system (configuration of Figure 4(a)) for matrix `Matrix211` on eight nodes of `PlaFRIM` with dense preconditioner. On each node, all eight cores are consistently used but the number of threads per subdomain (1, 2, 4 or 8, x-axis) is inversely proportional to the number of subdomains (8, 4, 2, 1).



(a) *All computantional steps.*

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 6: Performance of MaPHyS when processes are binded to a subset of cores (configuration of Figure 4(b)) for matrix `Matrix211` on 8 nodes of `PlaFRIM` with dense preconditioner. On each node, all eight cores are consistently used but the number of threads per subdomain (1, 2, 4 or 8, x-axis) is inversely proportional to the number of subdomains (8, 4, 2, 1).

no binding is performed (see Figure 5) to the case where processes are binded (see Figure 6). When only one process per node is used (eight threads per domain), both versions are equivalent and therefore achieve the same performance (right-most part of the plots). However, when multiple processes compete on the same node (*i.e.*, when one, two or four threads per domain are used), the performance gap may be significant, especially because of the *solve step*.

4.3 Multithreading performance

In this section, we investigate the performance of the multithreaded implementation on multicore nodes. First we are interested in the performance behavior of MAPHYS for the test cases described in Table 2. For those experiments, the number of subdomains is only four and we use the dense variant of the preconditioner. Processes (hence subdomains) have dedicated nodes. When the number of threads increases, more and more cores can thus be exploited (see Figure 7). The parallel performance, measured as a speed-up with respect to the single threaded reference, is plotted in Figure 8 using bar charts with the corresponding thread number on the top of the bars. For each matrix, we vary the number of threads from one to eight and compute the speed-ups for the three numerical steps of MAPHYS as well as the overall speed-up.

Matrix	Matrix211	Audi_kw	Nachos	Haltere	Tdr455k	Amande
Interior size (avg)	197K	233K	275K	320K	684K	1742K
Local Schur size (avg)	7K	6K	10K	4K	2K	12K
Schur size	14K	39K	11K	7K	4K	24K
#iter	36	24	28	9	5	16

Table 2: Size of the interior ($\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$), size of the local Schur complement ($\bar{\mathcal{S}}_i$) and total size of the Schur complement ($\mathcal{S}$) are given for each matrix when four subdomains are used. The number of iterations (#iter) needed with four subdomains and dense preconditioner is also given.

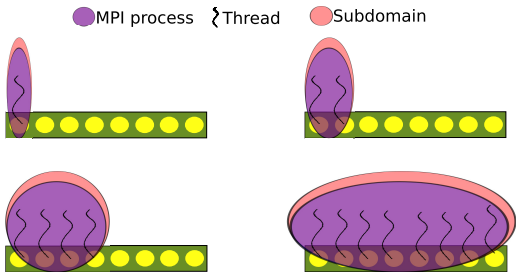
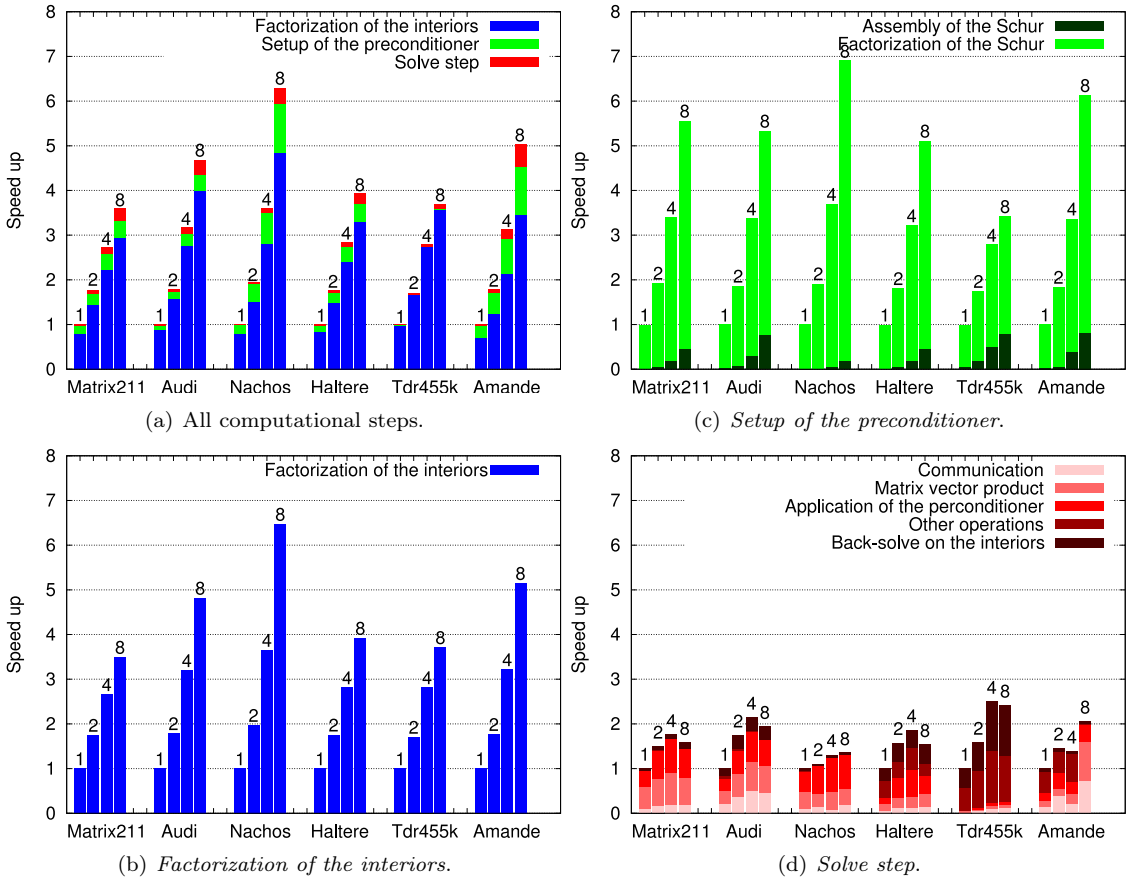


Figure 7: Using 1, 2, 4 or 8 threads and cores per process (set up in Section 4.3).

It can be seen that significant speed-ups are achieved for all test cases. An average speed-up of 4.53 is achieved for the overall execution time (see Figure 8(a)) of our matrix collection. In this configuration, the overall execution time is consistently dominated by the *factorization of the interiors*. An explanation for this is the fact that only four subdomains are used. Most of the unknowns are associated with the interior resulting to relatively small Schur complements. As a consequence, the *factorization of the interiors* takes more time to be performed compare to the *setup of the preconditioner*. Using only four subdomains and a dense preconditioner results with a preconditioner with good numerical quality, thus only a small number of iterations is needed in order to achieve the target accuracy.

Since the overall time is dominated by the *factorization of the interiors*, it has a behavior similar to the overall speed-up. An average speed-up of 4.6 is obtained for the *factorization of the interiors* (see Figure 8(b)) when 8 threads are used. The size of the interiors is large enough



(a) All computational steps.

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 8: Overall speed-up with using 1, 2, 4 or 8 threads (and cores) with respect to the time when 1 thread (and core) per process, for the matrices from Table 1. The dense preconditioner is used. Four nodes are used and one process per node is assigned. The number of threads (1, 2, 4 or 8) for each histogram is given on the top of the bars.

(see Table 2) for the multithreaded direct sparse solver. During the *setup of the preconditioner* with this configuration, the dense solver is used for the factorization of $\bar{\mathcal{S}}_i$. The size of the local Schur complements is large enough (at least a few thousands unknowns). As a result, the direct dense solver is able to exploit efficiently the multicore machines (see Figure 8(c)). For the *solve step*, an average speed up of only 1.94 (out of 8) is achieved (see Figure 8(d)). Two main reasons explain this behavior. First, the iterative method is expensive in terms of communication, which do not scale with the multithreaded version. Additionally, the operations that are done during the iterative method have a low level of computational intensity. These operations do not exploit efficiently the multicore machines. In addition to what has been presented, for the `Haltere` and `Tdr455k` matrices the back-solve on the interior represents a large portion of the *solve step*. This is a consequence of the fact that only 9 and 5 iterations are needed for the `Haltere` and `Tdr455k` matrix respectively (see Table 2). When the number of iterations increases, the portion that corresponds to the back-solve decreases. When more than a few dozen of iterations are used, the back-solve becomes inconsiderable compared to the time needed for the *solve step*.

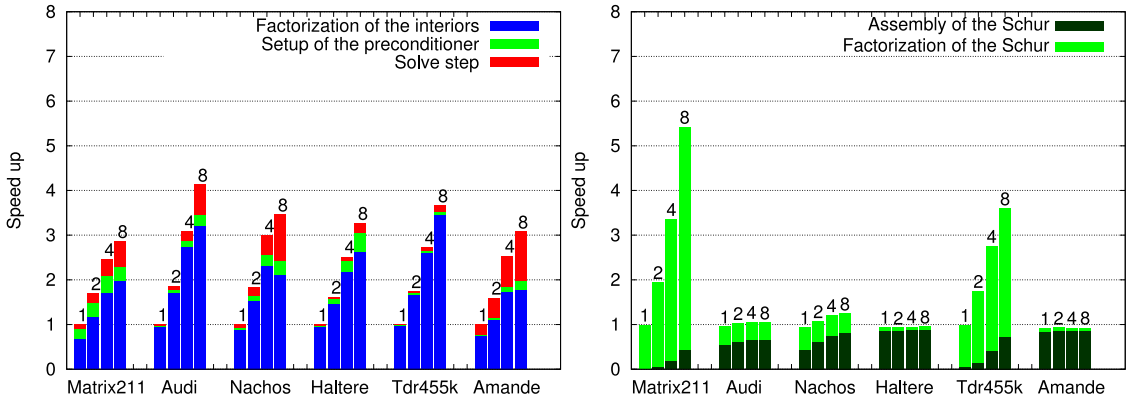
In the above study we have shown that MaPHyS is able to efficiently exploit the multicore platforms with a dense preconditioner and four nodes. To further explore the capabilities of the 2-level parallel implementation, we consider experiments at a scale related to the problem size and relying on a preconditioning strategy tuned for each matrix.The characteristics of this second set of experiments are reported in Table 3 while the observed speed-ups are displayed in Figure 9.

Matrix		`Matrix211`	`Audi_kw`	`Nachos`	`Haltere`	`Tdr455k`	`Amande`
#nodes		8	4	16	4	8	32
Preconditioner		dense	10^{-3}	10^{-2}	10^{-3}	dense	10^{-2}
Interior size (avg)		97K	233K	67K	320K	341K	216K
Local Schur	size (avg)	6K	6K	6K	3K	2K	5K
	kept entries (avg)	—	2.75%	2.5%	2%	—	1%
Schur size		23K	11K	49K	8K	10K	85K
#iter		64	49	53	11	9	94

Table 3: Configuration (#nodes and preconditioner) used in Section 4.3 and 4.4. The size of the interiors ($\mathcal{I}_i$), the size of the local Schur ($\bar{\mathcal{S}}_i$), the total size of the Schur complement ($\mathcal{S}$) and the number of iterations are given for each matrix.

Similarly to the situation described in Figure 8, we observe significant (but lower) speed-ups (see Figure 9(a)). Indeed, although the overall time is dominated by the *factorization of the interiors*, the number of subdomains increases with the number of nodes yielding smaller interiors (see Table 3) and resulting in a *factorization of the interiors* that is harder to parallelize efficiently with the use of multiple threads (see Figure 9(b)).

Concerning the preconditioner step, in the case where the sparsified preconditioner is used, the involved matrices almost do not benefit from the multithreading (see Figure 9(c)). The main reason is that sparsifing considerably decreases the amount of computation (factorization of the local assembled Schur $\bar{\mathcal{S}}_i$), which thus becomes dominated by the assembly step. Because the assembly step consists only of communications, it is not affected by multithreading. The second reason is that the local Schur complement become too small for a multithreaded sparse direct solver. For example, for the `Audi_kw` matrix, the average size of $\bar{\mathcal{S}}_i$ is 5,884. After applying the dropping with a threshold of 10^{-3}, approximately 2.75 percent of the entries are kept, resulting in a sparsified $\bar{\mathcal{S}}_i$ that contains around 692K of non zero values. This is not a large enough computational load for the sparse direct solver to exploit efficiently the multicore processors. As a consequence, the overall *setup of the preconditioner* is hard to accelerate with multithreading when a sparse preconditioner is used. On the other hand, thanks to the sparsification, the *setup of the preconditioner* turns out to be only a very limited proportion of the total execution time (see Figure 9(a)). For instance, when eight threads are used on four nodes with the `Audi_kw` matrix,



(a) All computational steps.

(c) *Setup of the preconditioner.*

(b) *Factorization of the interiors.*

(d) *Solve step.*

Figure 9: Overall speed-up when using 1, 2, 4 or 8 threads (and cores) with respect to the time when 1 thread (and core) per process for the matrices from Table 3. The configuration (#nodes and preconditioner) is given in Table 3 for each matrix. The number of threads (1, 2, 4 or 8) for each histogram is given on the top of each bar.

the dense factorization is performed in 2.55 seconds, while the sparse factorization takes only 0.95 seconds. Regarding the *solve step*, it follows the same behvavior for all the matrices for the same reasons as explained above.

4.4 Numerical and parallel flexibility of the 2-level implementation

In the previous section, we mainly focused on the parallel efficiency of the 2-level implementation when the number of cores is increased while keeping constant the number of nodes. We are now investigating the capability of the code to fully exploit the computing resources of a hierarchical multicore platform. To highlight the flexibility we fix the number of cores used for the solution of a given problem and we vary the number of subdomains and the number of threads per subdomain so that $nb\_nodes \times nb\_cores = nb\_threads\_per\_process \times nb\_domains$ (see Figure 10). Those iso-computing resource experiments show that our 2-level approach enables us to find the best trade-off between the numerical behavior of the solver (which depends on the number of subdomains) and its parallel performance (which depends on the number of threads per subdomain). For those experiments we consider the same test examples as in Table 3.

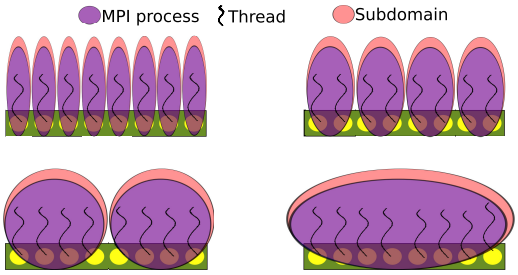
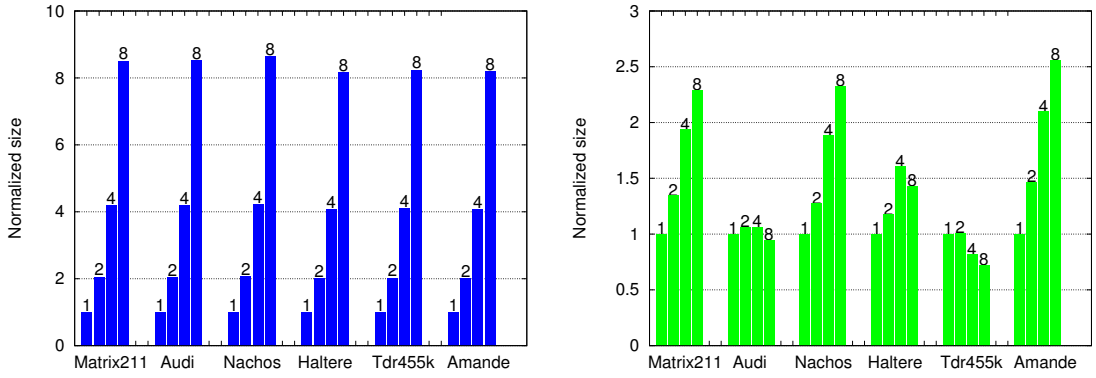


Figure 10: Setup in Section 4.4 illustrating the numerical and parallel flexibility of the 2-level implementation for exploiting all cores of a node with 1, 2, 4 or 8 subdomains per node.

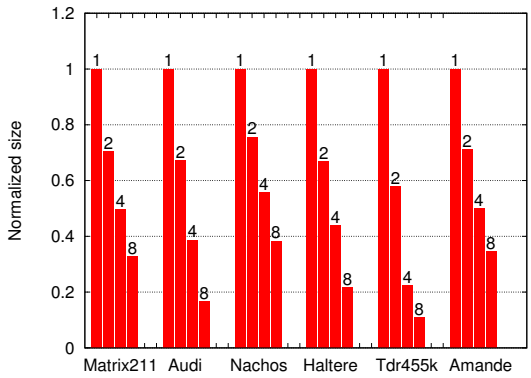
With this configuration, when the number of threads per process increases, the number of subdomains decreases; this tends to reduce the number of iterations but enlarges the elapsed time of the *factorization of the interiors*. For instance, for the `Audi_kw` matrix, the first histogram presents a run with 32 subdomains with one thread per process, while the last one, when 8 threads are used, corresponds to a run with 4 subdomains. Compared to the results presented in the previous section, for each run not only the number of cores assigned per process changes, but also the numerical problem solved. With an increased number of subdomains, the size of the domain interior is decreasing (see Figure 11(a)), while the size of the total Schur complement ($\mathcal{S}$) is increasing (see Figure 11(b)). With a larger Schur complement and the larger number of subdomains, the numerical difficulty of the iterative part tends be higher, increasing the number of iterations and possibly the overall solution time (see Figure 11(d)). With our 2-level parallel approach, we are able to explore the trade-off between larger subdomains and numerical efficiency of the iterative method in order to exploit the multicore machines in a most optimal way. The results are presented in Figure 12.

For the `Haltere` matrix, the best performance is obtained when one thread per process is used. More specifically, each step yields the best performance when one thread per process is used. Considering the *solve step*, increasing the number of subdomains does not strongly deteriorate the quality of the preconditioner for this matrix. For instance, in this case, when the number of subdomains varies from 4 to 32, the number of iterations increases from 11 to only 14 respectively (see Figure 11(d)). On that example, the best performance is obtained when the number of subdomains is the largest. Additionally, for this matrix with all configurations, the overall time is dominated by the *factorization of the interiors*. The *factorization of the interiors* yields best performance when the number of subdomains is the largest (see Figure 12(b)). The main reason for

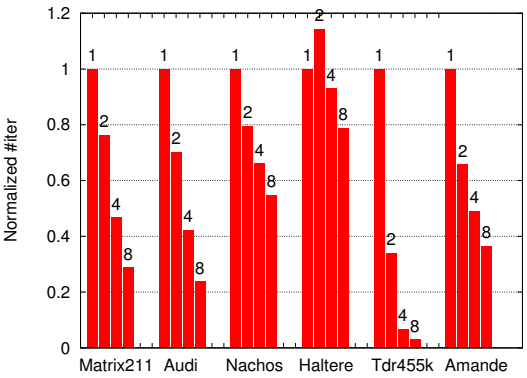


(a) Interior size.

(c) Size of the local Schur.



(b) Size of the total Schur.



(d) Number of iterations.

Figure 11: Average size of the interiors (11(a)), of the local Schur complement (11(c)), of the total Schur complement (11(b)) and number of iterations (11(d)) for each matrix shown when all available cores are used and the following statement is satisfied : $nb\_nodes \times nb\_cores = nb\_threads\_per\_process \times nb\_subdomains$. The number of nodes and the preconditioner are given in Table 3 and the number of threads used per process is given on top of each histogram bar. All histograms are normalized with respect with the histogram when one thread per process is used.

this behavior is the fact that when the number of subdomains is doubled, the size of each interior is approximately divided by two (see Figure 11(a)) and the number of threads are doubled. Since the complexity in terms of calculation for 3D problems for the sparse direct solver is squared (see Figure 12(d)), larger number of domain yields better performance.

For the `Nachos` matrix, the best configuration is also when one thread per process is used. Contrary to the the `Haltere` matrix, the *solve step* is expensive for this matrix. Nevertheless, with the increasing of the number of subdomains, the number of iterations does not decrease drastically (see Figure 11(d)). Since the solve step does not exploit efficiently multithreading (see Figure 9(d)), similar numbers of iterations will be executed in approximately the same amount of time (see Figure 12(d)). Due to this, the overall execution time is driven by the *factorization of the interiors*, and, as explained above, the factorization of the interior yields the best performance when one thread per process is used.

		#subdomains				
		8	16	32	64	128
Precond.	dense	9	21	107	315	6323
	10^{-4}	31	108	502	–	–
	10^{-3}	168	1002	–	–	–
	10^{-2}	6496	–	–	–	–

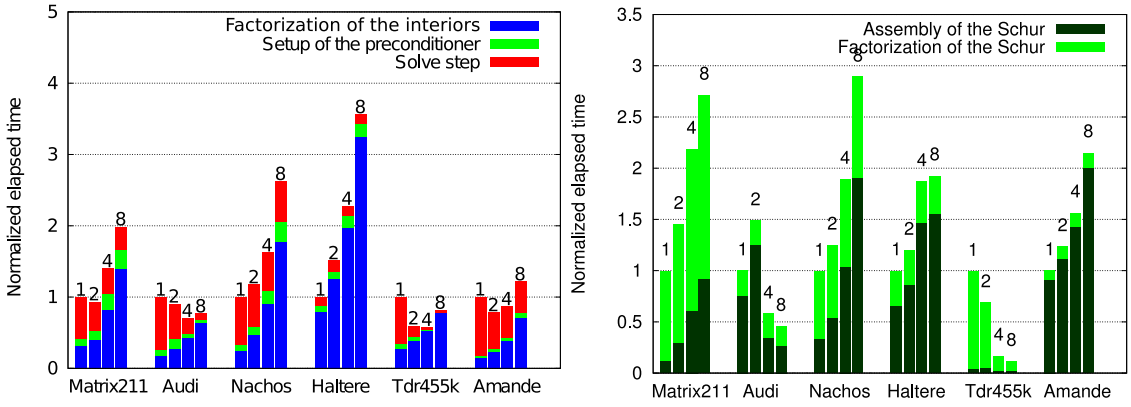
Table 4: Number of iterations for different configurations (#subdomains and preconditioner) for the `Tdr455k` matrix with the GMRES algorithm. "–" means that the algorithm fails to achieve convergence in 7000 iterations with restart of 500.

The `Amande` and `Matrix211` matrices both exhibit similar behaviors. The optimal performance is obtained when two threads per process are used. Compared to the `Nachos` matrix, the number of iterations decreases more quickly. As a consequence the *solve step* becomes less expensive in terms of time consumption when the number of subdomains decreases (see Figure 12(d)). Moreover, both matrices are issued from difficult numerical problems, resulting in a large number of iterations during the solve step. For instance, when one thread per process is used, 223 and 468 iterations are needed for the `Matrix211` and `Amande` matrices, respectively. Therefore, the benefits of reducing the cost of the solve step have larger consequences on the overall computational time (see Figure 12(d)).

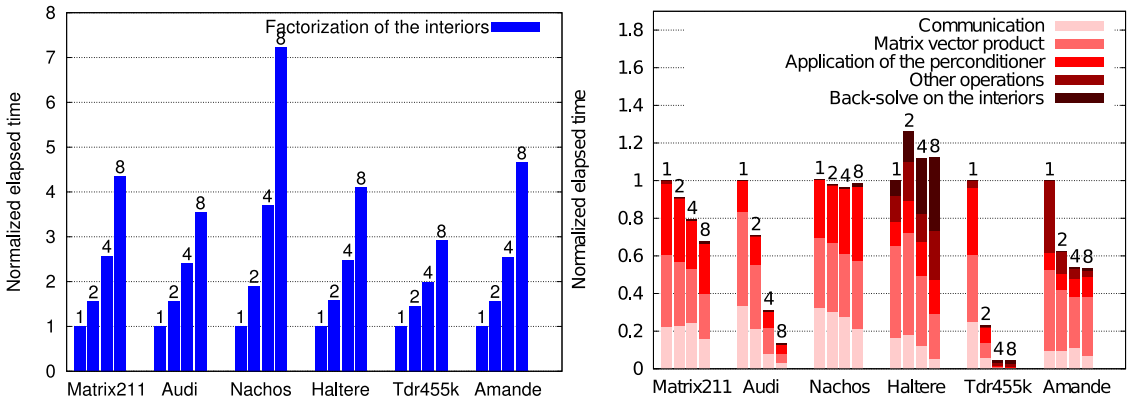
For the two remaining matrices, `Audi_kw` and `Tdr455k`, this phenomena is even stronger. Additionally, when the number of subdomains is increased, the size of the local Schur complement is not increased, even slightly decreased (see Figure 11(c)). Since the dense preconditioner is used for the `Tdr455k` matrix, the *setup of the preconditioner* yields better performance with the multithreaded version for this matrix. For these matrices increasing the number of subdomains has the most significant influence on the number of iterations. Therefore the solve step yields the best performance when eight threads per process are used.

Furthermore, the `Tdr455k` matrix is the most difficult matrix to solve in our collection for our algorithm. The results presented in Figure 12 correspond to a dense preconditioner. Performing dropping on the local Schur complement strongly decreases the quality of the preconditioner (see Table 4). Additionally, decomposing the matrix in larger number of subdomains increases strongly the number of iterations for each preconditioner, failing to converge in most cases. This matrix has strong numerical barriers for our algorithm. With the multithreaded version, we are able to break the numerical barrier by assigning larger number of cores per subdomain. On the `Hopper` platform we are able to both efficiently exploit as much as 384 cores (see Figure 13(a)) and in the same time lower the memory peak per node (see Figure 13(b)).

In order to push the limit of our algorithm, we have furthermore performed tests on the `Nachos4M` matrix (see Table 1) on the `Hopper` platform. In these series of experiments we have increased the number of nodes up to one thousand (see Figure 14). With the increased number of nodes, the number of subdomains is also increased up to 2^{13} when three threads per process are



(a) All computational steps.

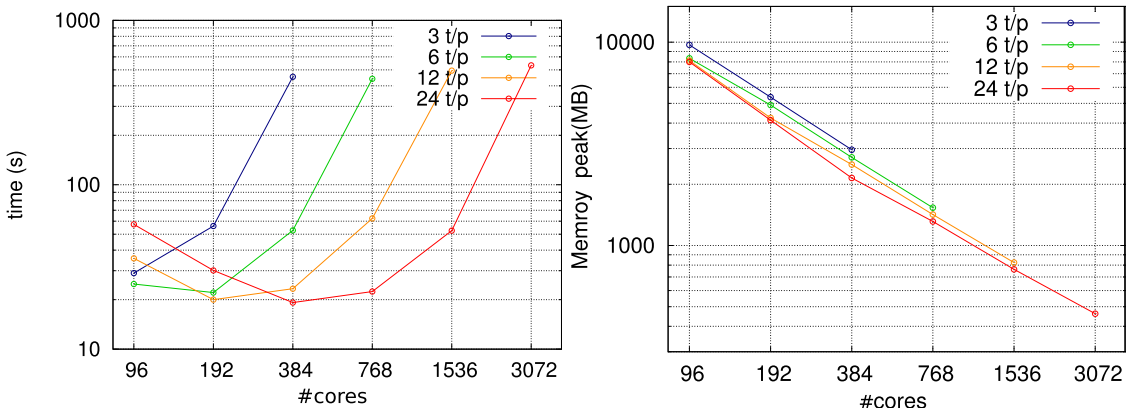


(b) *Factorization of the interiors.*

(c) *Setup of the preconditioner.*

(d) *Solve step.*

Figure 12: In these experiments all available cores are used and the following statement is satisfied : $nb\_nodes \times nb\_cores = nb\_threads\_per\_process \times nb\_subdomains$. The number of nodes and the preconditioner are given in Table 3 and the number of threads used per process is given on top of each histogram bar. Each histogram is normalized in comparison to the time obtained when one thread per subdomain is used.



(a) Elapsed time for all computational steps.

(b) Memory peak per node.

Figure 13: The maximum time 13(a) and memory peak per node 13(b) when the `Tdr455k` matrix is used with dense preconditioner. All the available cores per node are used and the statement is satisfied: $nb\_nodes \times nb\_cores = nb\_threads\_per\_process \times nb\_subdomains$. In the legend $\frac{t}{p}$ refers to number of threads (t) per process (p).

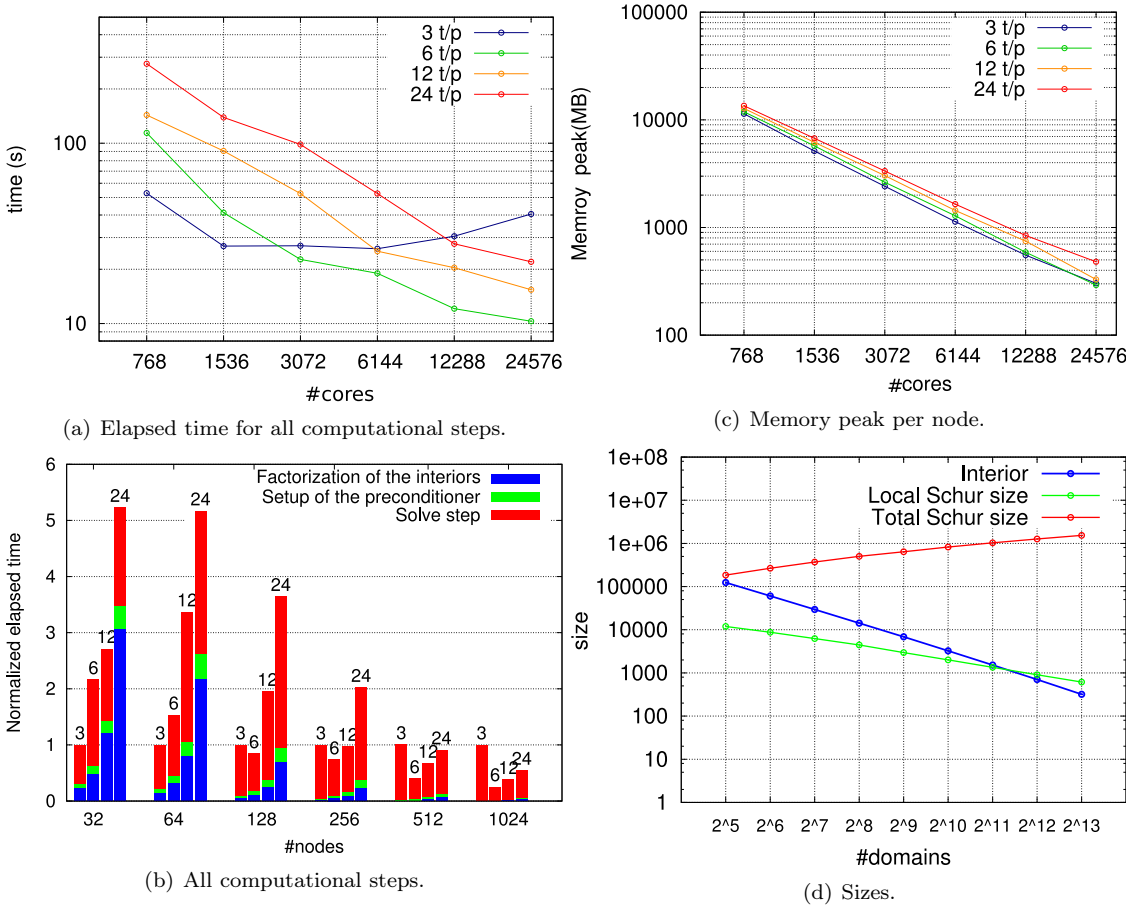
used. When the number of nodes is increased, the solve step dominates more and more the overall execution time (the red portion in each histogram in Figure 14(b)).

Although not illustrated up-to now, depending on the targeted accuracy, the best choice of the number of subdomains can vary for a given number of cores to be used. In Figure 15, we display the convergence history as a function of the elapsed time and iteration number for the `Audi_kw` matrix (see Figures 15(a) and 15(b) respectively). Lower the number of subdomains (*i.e.*, larger number of threads per subdomain), larger the time needed for *factorization of the interiors* and *setup of the preconditioner* is, and faster is the convergence. However, for a moderated target accuracy, this setup cost is not worth to invest. It might be more effective to enlarge the number of subdomains to shrink the setup time and having slower convergence rate but overall faster solution. For instance, for a 10^{-4} accuracy, the best choice is 64 subdomains (one thread each) while for a 10^{-9} accuracy the best choice is 16 subdomains (*i.e.*, 4 threads).

5 Conclusion

In this paper we have described a 2-level parallel implementation of the hybrid solver MaPHyS and shown how multithreaded libraries, namely Mkl and PaStiX, have been composed to design an efficient parallel implementation. The resulting MPI+thread parallel implementation does match the hierarchical structure of modern multicore parallel platforms. Not only it enables us to better exploit the computer architectures features but it also introduces an additional numerical flexibility to balance the numerical and parallel performances of the MaPHyS solver. Thanks to the new implementation we demonstrated that large computing platforms, up to a few tens of thousand cores, can be exploited to solve 3D linear systems that were not tractable before.

We have considered classical programming paradigms based on MPI and threads to design our 2-level parallel hybrid solver. An alternative approach would be to consider a more disruptive approach based on a task graph description of the algorithm that can be efficiently mapped and scheduled on multicore and manycore platforms by a modern runtime systems as discussed in [2] Because the preconditioner used in MaPHyS is local, we are also considering for symmetric positive definite problems to combine the 2-level parallel design proposed in the present article with coarse-grid correction techniques [26, 18].



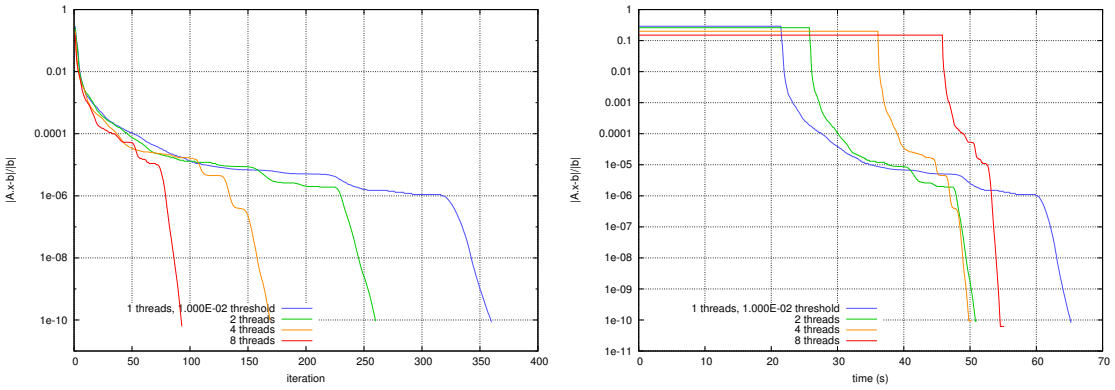
(a) Elapsed time for all computational steps.

(c) Memory peak per node.

(b) All computational steps.

(d) Sizes.

Figure 14: The maximum time 14(a) and memory peak per node 14(c) for the `Nachos4M` matrix with sparse preconditioner when a dropping is applied to the preconditioner with threshold of 10^{-2}. The detailed histogram for all computational steps is given in 14(b) and the sizes for the interior, the local Schur complement and the total size of the Schur complement are given in 14(d). The tests were performed on the `Hopper` platform. All the available cores per node are used and the statement is satisfied:
$nb\_nodes \times nb\_cores = nb\_threads\_per\_process \times nb\_subdomains$. In the legend $\frac{t}{p}$ refers to number of threads (t) per process (p).



(a) (b)

Figure 15: Convergence history with respect with the elapsed time for the `Audi_kw` matrix when four nodes are used in respect with the iteration (see Figure 15(a)) and in respect with the time (see Figure 15(b)). Dropping with a threshold of 10^{-2} is applied to the preconditioner.

Acknowledgement

The authors would like to thank Sherry Li (LBNL), Esmond Ng (LBNL) and Ichitaro Yamazaki (UTK) for constructive exchanges on the design of high-performance hybrid schemes. They are also grateful to the PaStiX team for their great support in the integration of the direct solver within MaPHyS. They also thank the NERSC and Plafrim support teams. All this work has been conducted in the context of the Depth Imaging Partnership (DIP) between Total and Inria and the authors would specially like to thank Henri Calandra (Total) for this support.

References

[1] PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.0. http://icl.cs.utk.edu/plasma, November 2009.

[2] Emmanuel Agullo, Luc Giraud, and Stojce Nakov. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In *HeteroPar'2016 worshop of Euro-Par*, Grenoble, France, August 2016.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[5] 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, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[6] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.

[7] L. M. Carvalho, L. Giraud, and G. Meurant. Local preconditioners for two-level non-overlapping domain decomposition methods. *Numerical Linear Algebra with Applications*, 8(4):207–227, 2001.

[8] Tony F. C. Chan and Tarek P. Mathew. The interface probing technique in domain decomposition. *SIAM J. Matrix Anal. Appl.*, 13(1):212–238, January 1992.

[9] Mathieu Faverge and Pierre Ramet. Dynamic scheduling for sparse direct solver on NUMA architectures. In *PARA'08*, LNCS, Trondheim, Norvège, 2008.

[10] J. Gaidamour and P. Hénon. A parallel direct/iterative solver based on a Schur complement approach. *2013 IEEE 16th International Conference on Computational Science and Engineering*, 0:98–105, 2008.

[11] Jérémie Gaidamour and Pascal Hénon. HIPS: a parallel hybrid direct/iterative solver based on a Schur complement approach. *Proceedings of PMAA*, 2008.

[12] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.

[13] Luc Giraud and A. Haidar. Parallel algebraic hybrid solvers for large 3D convection-diffusion problems. *Numerical Algorithms*, 51(2):151–177, 2009.

[14] Luc Giraud, A. Haidar, and S. Pralet. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing*, 36(5-6):285–296, 2010.

[15] Azzam Haidar. *On the parallel scalability of hybrid linear solvers for large 3D problems*. PhD thesis, Institut National Polytechnique de Toulouse, December 17 2008.

[16] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[17] Pascal Hénon and Yousef Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM J. Sci. Comput.*, 28(6):2266–2293, December 2006.

[18] Pierre Jolivet, Frédéric Hecht, Frédéric Nataf, and Christophe Prud'homme. Scalable domain decomposition preconditioners for heterogeneous elliptic problems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 80:1–80:11, New York, NY, USA, 2013. ACM.

[19] G. Karypis and V. Kumar. MeTiS – *Unstructured Graph Partitioning and Sparse Matrix Ordering System – Version 2.0*. University of Minnesota, June 1995.

[20] Jean-Yves L'Excellent and Wissam M. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, 2014.

[21] X. S. Li, M. Shao, I. Yamazaki, and E. G. Ng. Factorization-based sparse solvers and preconditioners. *Journal of Physics: Conference Series*, 180(1):012015, 2009.

[22] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

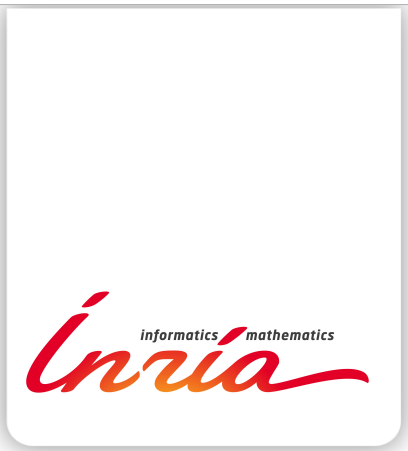
[23] Stojce Nakov. *On the design of sparse hybrid linear solvers for modern parallel architectures*. Theses, Université de Bordeaux, December 2015.

[24] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.

[25] S. Rajamanickam, E. G. Boman, and M. A. Heroux. ShyLU: A hybrid-hybrid solver for multicore platforms. *Parallel and Distributed Processing Symposium, International*, 0:631–643, 2012.

[26] Nicole Spillane, Victorita Dolean, Patrice Hauret, Frédéric Nataf, Clemens Pechstein, and Robert Scheichl. Achieving robustness through coarse space enrichment in the two level Schwarz framework. In *Domain Decomposition Methods in Science and Engineering XXI*, pages 447–455. Springer, 2014.

[27] Ichitaro Yamazaki and Xiaoye S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *VECPAR*, pages 421–434, 2010.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vielle Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399