Task-based hybrid linear solver for distributed memory heterogeneous architectures

Emmanuel Agullo, Luc Giraud, Stojce Nakov

Task-based hybrid linear solver for distributed memory heterogeneous architectures

E. Agullo, L. Giraud, S. Nakov

RESEARCH REPORT
N° 8913
May 2016
Project-Teams HiePACS

ISSN 0249-6399   ISRN INRIA/RR--8913--FR+ENG

Task-based hybrid linear solver for distributed memory heterogeneous architectures

E. Agullo[*], L. Giraud[*], S. Nakov[*]

Project-Teams HiePACS

Research Report n° 8913 — May 2016 — 13 pages

Abstract: Heterogeneity is emerging as one of the most challenging characteristics of today's parallel environments. However, not many fully-featured advanced numerical, scientific libraries have been ported on such architectures. In this paper, we propose to extend a sparse hybrid solver for handling distributed memory heterogeneous platforms. As in the original solver, we perform a domain decomposition and associate one subdomain with one MPI process. However, while each subdomain was processed sequentially (binded onto a single CPU core) in the original solver, the new solver instead relies on task-based local solvers, delegating tasks to available computing units. We show that this "MPI+task" design conveniently allows for exploiting distributed memory heterogeneous machines. Indeed, a subdomain can now be processed on multiple CPU cores (such as a whole multicore processor or a subset of the available cores) possibly enhanced with GPUs. We illustrate our discussion with the MaPHyS sparse hybrid solver relying on the PaStiX and Chameleon dense and sparse direct libraries, respectively. Interestingly, this two-level MPI+task design furthermore provides extra flexibility for controlling the number of subdomains, enhancing the numerical stability of the considered hybrid method. While the rise of heterogeneous computing has been strongly carried out by the theoretical community, this study aims at showing that it is now also possible to build complex software layers on top of runtime systems to exploit heterogeneous architectures.

Key-words: High Performance Computing (HPC); heterogeneous architectures; MPI; task-based programming; runtime system; sparse hybrid solver; multicore; GPU.

[*] Inria

Un solveur linéaire en tâches sur machines hétérogènes à mémoire distribuée

Résumé : L'hétérogénéité apparaît comme un challenge sur les calculateurs parallèles modernes. Aujourd'hui, peu de librairies scientifiques avec des fonctionnalités avancées ont été portées sur de telles architectures. Dans ce travail, nous proposons d'étendre un solveur linéaire hybride pour des machines hétérogènes à mémoire distribuée. Comme pour le solveur initial qui implante une méthode de décomposition de domaine, nous associons chaque sous-domaine à un processus MPI. Cependant, alors que le code initial "bindait" un sous-domaine à un cœur de calcul, le nouveau solveur s'appuie sur un solveur local à base de tâches, qui délègue les tâches aux unités de calcul via un moteur d'exécution. Nous montrons que le nouveau solveur "MPI+tâches" permt l'utilisation de machines hétérogènes à mémoire distribuée. Dasn ce contexte, un sous-domaine peut être traité par un multi-cœurs possiblement hétérogène. Nous illutrons notre propos avec le solveur hybride MaPHyS qui utilise PaStiX et Chameleon pour les calculs creux et dense d'algèbre linéaire.

Mots-clés : Calcul Haute performance (HPC); multi-GPUs; architectures hétérogènes; MPI; modéle en tâches; support d'exécution; systèmes linéaires creux; solveur creux hybride; machine multi-cœurs; GPU.

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 to high-performance, scalable solution of large sparse linear systems in parallel scientific computing is to combine 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 [40, 37, 22, 20]. 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 as we do in this paper. In this study we consider an actual fully-featured parallel sparse hybrid (direct/iterative) linear solver, MaPHyS[1] [3]

Starting from a baseline MPI version of the considered hybrid solver, the objective of this study is to propose a prototype extension for which each MPI process can handle heteregenous processing units with a task-based approach, delegating the task management to a runtime system. A preliminary experimental study asseses the potential of the approach.

This paper is organized as follows. Section 2 presents the solver considered in this study and its baseline parellel design. Section 3 presents background on task-based linear algebra and sparse hybrid solvers. Section 4 presents the design of the task-based extension proposed for our sparse hybrid solver. Preliminary results are discussed in Section 5 while concluding remarks on this work and perspectives are discussed in Section 6.

2 Baseline MPI hybrid (direct/iterative) solver

We now present the sparse hybrid (direct/iterative) method (Section 2.1) considered in this study and its baseline parellel design (Section 2.2).

2.1 Method

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

[1]https://project.inria.fr/maphys/

following block reordered linear system

$$\begin{pmatrix} \mathcal{A}_{\mathcal{I}\mathcal{I}} & \mathcal{A}_{\mathcal{I}\Gamma} \\ \mathcal{A}_{\Gamma\mathcal{I}} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_{\mathcal{I}} \\ x_{\Gamma} \end{pmatrix} = \begin{pmatrix} b_{\mathcal{I}} \\ b_{\Gamma} \end{pmatrix} \tag{2.1}$$

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 (2.1) leads to the reduced system

$$\mathcal{S} x_\Gamma = f \tag{2.2}$$

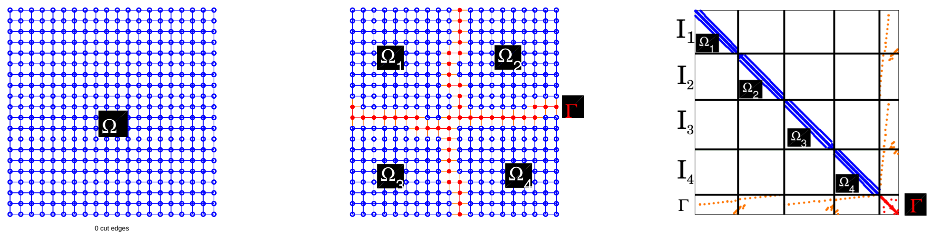
where

$$\mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma\mathcal{I}} \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} \mathcal{A}_{\mathcal{I}\Gamma} \quad \text{and } f = b_\Gamma - \mathcal{A}_{\Gamma\mathcal{I}} \mathcal{A}_{\mathcal{I}\mathcal{I}}^{-1} b_{\mathcal{I}}. \tag{2.3}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix.* This reformulation leads to a general strategy for solving (2.1). Specifically, an iterative method can be applied to solve (2.2). 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



(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 $\mathcal{A}_{\mathcal{I}\mathcal{I}}$.

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. Because interior vertices are only connected to vertices of their subset (either on the interior or on the boundary), matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ associated to the interior has a block diagonal structure, as shown in Figure 1a. Each diagonal block $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to a local interior.

While the Schur complement system is significantly smaller and better conditioned than the original matrix $\mathcal{A}$ [34, Lemma 3.11], it is important to consider further preconditioning when employing a Krylov method. We introduce the general form of the preconditioner considered in MaPHyS. The preconditioner presented below was originally proposed in [17] and successfully applied to large problems in real life applications in [21, 24, 35]. To describe the main preconditioner in MaPHyS, considering the restriction operator $\mathcal{R}_{\Gamma_i}$ from Γ to Γ_i, 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

$$\mathcal{M}_{AS} = \sum_{i=1}^{\mathcal{N}} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{2.4}$$

2.2 Baseline MPI parallelization

With all these components, the classical parallel implementation of MaPHyS can be decomposed into four main phases:

- *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 [36] and METIS [28];
- *the factorization of the interiors* and the computation of the local Schur complement factorizes $\mathcal{A}_i$ with the PaStiX [25] or the Mumps [9] sparse direct solver and furthermore provides the associated local Schur Complement $\mathcal{S}_i$ thanks to recent progress from the development teams of those sparse direct solvers;
- *the setup of the preconditioner* by assembling diagonal blocks of $\mathcal{S}_i$ via a few neighbour to neighbour communications and factorization of the dense local $\mathcal{S}_i$ using Mkl;
- *the solve step* consists of two steps: a parallel preconditioned Krylov method performed on the reduced system (Equation 2.2) to compute x_{Γ_i} where all BLAS operations are provided by Mkl, followed by the back solve on the interior to compute $x_{\mathcal{I}_i}$, done by the sparse direct solver.

3 Related work

To cope with the complexity of modern architectures, programming paradigms are being revisited. Among others, one major trend consists in writing the algorithms in terms of task graphs and delegating to a runtime system both the management of the data consistency and the orchestration of the actual execution. This paradigm has been intensively studied in the context of dense linear algebraand is now a common utility for related state-of-the-art libraries such as Plasma [2], Magma [1], DPLASMA [14], Chameleon [4] and FLAME [39]. Dense linear algebra algorithms were indeed excellent candidates for pioneering in this direction. First, their computational pattern allows one to design very wide task graphs so that many computational units can execute tasks concurrently. Second, the building block operations they rely on, essentially level-three Basic Linear Algebra Subroutines (BLAS), are compute intensive, which makes it possible to split the work in relatively fine grain tasks while fully benefiting from GPU acceleration. As a result, these algorithms are particularly easy to schedule in the sense that state-of-the-art greedy scheduling algorithms may lead to high performance, including on platforms accelerated with multiple GPUs [4].

This trend has then been followed for designing sparse direct methods. The extra challenge in designing task-based sparse direct method is due to indirection and variable granularities of the tasks. The PaStiX team has proposed such an extension of the solver capable of running on the StarPU [11] and PaRSEC [15] runtime systems on cluster of heterogeneous nodes in the context of X. Lacoste PhD thesis [29, 30]. In the meanwhile, the qr_mumps library developed by A. Buttari [16] aims at solving sparse linear least square problems and has been ported on top of those runtime systems in the context of F. Lopez PhD thesis [33, 6].

With the need of solving ever larger sparse linear systems while maintaining numerical robustness, multiple sparse hybrid variants have been proposed for computing the preconditioner for the Schur complement. PDSLin [31], ShyLU [37] and Hips [19] first perform an exact[2] 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 [37] also implements a structure-based approach for discarding values named *probing* and that was first proposed to approximate interfaces in DDM [18]. Instead of following such a two-fold approach, Hips [19] forms the preconditioner by computing a global ILU factorization based on the multi-level scheme formulation from [26].

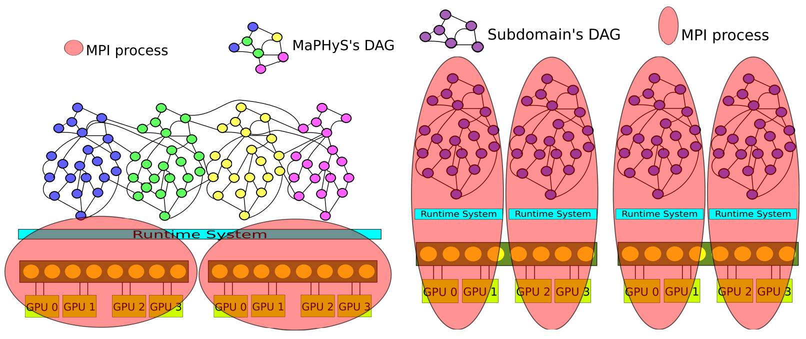
These sparse hybrid solvers have also been extended to cope with hierarchical supercomputers. Indeed, to ensure numerical robustness while exploiting all the processors of a platform, an important effort has been devoted to propose two levels of parallelism for these solvers. Designed on top of the SuperLU_DIST [32] distributed memory sparse direct solver, PDSLin implements a 2-level MPI (MPI+MPI) approach with finely tuned intra- and inter-subdomain load balancing [40]. A similar MPI+MPI approach has been assessed for additive Schwarz preconditioning in a prototype version of MaPHyS [23], relying on the Mumps [9, 10] and ScaLAPACK [13] 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 [19]. Finally, especially tuned for modern multicore platforms, ShyLU implements a 2-level MPI+thread approach [37]. A similar 2-level MPI+thread design has been investigated for MaPHyS in [35]. However, none of these extensions were tailored to exploit heterogeneous architectures.

4 Design of task-based sparse hybrid linear solver for distributed memory heterogeneous architectures

Although very efficient for exploiting multiple modern multicore nodes, the relatively low-level design of the 2-level parallelism sparse hybrid solvers discussed above cannot exploit heterogeneous architectures.

One solution for relieving this bottleneck would consist in fully abstracting the MPI scheme of the solver in terms of a DAG of tasks where vertices represent fine grain tasks and edges represent dependencies between them. Once the solver has been designed at such a high-level of abstraction, advanced fine-grain mapping strategies can be implemented as the burden of moving data in the system and ensuring their consistency is delegated to a runtime system. However, such a design prevents from relying on SPMD paradigms. As a consequence, it requires to fully rewrite the solver in terms of a DAG of tasks as illustrated in Figure 2a: the DAG representing

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



(a) Full task-based paradigm. (b) MPI + task paradigm (this study).

Figure 2: Illustration of the execution of MaPHyS on four subdomains with two different task-based paradigms on a platform composed of two nodes of eight cores and four GPUs per node

the whole numerical algorithm implemented in MaPHyS is written independently from the hardware architecture (we represent it on top of the runtime system and of the MPI processes). While there has been a lot of progress in that direction as discussed in Section 3 for dense (such as the DPLASMA [14] and Chameleon [4] task-based libraries, derived from Plasma [2] and Magma [1]) and sparse direct methods (such as the task-based version of PaStiX [29] and qrm_StarPU [5]), only limited work we are aware of has been devoted to study task-based Krylov methods, the third numerical pillar on top of which hybrid solvers are built on. Indeed, a task-based version of the CG algorithm for platforms equipped with several GPUs was proposed in [7] and a fault-tolerant task-based version of this algorithm was proposed in [27], but none of them could exploit distributed memory platforms.

A second solution would consist in relying on the modular design of the hybrid solver to use appropriate libraries depending on the target architecture leading to a collection of MPI+X, MPI+Y, ... solutions to support X, Y, ... architectures respectively.

In this paper, we propose to combine both solutions with a MPI+task approach in order to benefit from the high-level modular design of the hybrid solver and abstract the architecture with task-based local sparse and direct solvers which delegate the orchestration of the execution of the tasks within computational nodes to a runtime system. With such MPI+task approach, it is not only elegant to support X, Y, ... architectures in a consistent fashion, but also possible to exploit heterogeneous { X + Y} distributed memory architectures. Figure 2b illustrates the proposed design.

4.1 Overall design of the MPI+task extension of MaPHyS

In this section we explain how to create an MPI+task design of the MaPHyS solver. As illustrated in Figure 2b, this latter approach aims at abstracting the hardware architecture relying on task-based programming and delegating the orchestration of the task within computational nodes to a runtime system. However, contrary to the full task-based abstraction depicted in Figure 2a, each MPI process explicitly handles a DAG representing the numerical algorithm implemented in one MaPHyS subdomain. The MPI communications between subdomains are furthermore handled explicitly by MaPHyS (and not delegated to the task-based runtime system).

The goal of this preliminary study is to show the feasibility of the approach. To do so, we considered the baseline MPI version of MaPHyS and exploited the modular software architecture to substitute the multithreaded kernels with task-based versions of these kernels. We restricted our scope to the Symmetric Positive Definite (SPD) case.

In this paper we focus on the compute intensive numerical steps occurring after the *partitioning step*. Indeed, this stage is a symbolic pre-processing step; furthermore, to the best of our knowledge, none of the partitioners have yet been implemented on top of a runtime system:

- For the *factorization of the interiors*, we are relying on the task-based version of PaStiX proposed in X. Lacoste thesis [29] for which we further designed a task-based Schur complement functionality thanks to the support of the PaStiX development team.

- For the *setup of the preconditioner*, we use the task-based dense Cholesky solver from the Chameleon [4] library [3].

- For the *solve step*, the application of the preconditioner is performed by the Chameleon library and other operations involved in the iterative solution step such as level-one BLAS and matrix-vector product are executed with the multithreaded Mkl library.

All in all, we use task-based sparse (PaStiX) and dense (Chameleon) direct solvers, both of them expressed through the StarPU task-based runtime system that we now present.

4.2 The StarPU task-based runtime system

In the last decade, a large variety of task-based runtime systems have been developed. The most common strategy for the parallelization of task-based algorithms consists in traversing the DAG sequentially and submitting the tasks, as discovered, to the runtime system using a non blocking function call. The dependencies between tasks are automatically inferred by the runtime system through a data dependency analysis [8] and the actual execution of the task is then postponed to the moment when all its dependencies are satisfied. This programming model is known as a *Sequential Task Flow* (STF) model as it fully relies on `sequential consistency` for the dependency detection. This paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. The popularity of this model encouraged the OpenMP board to include it in the standard 4.0: the `task` construct was extended with the `depend` clause which enables the OpenMP runtime to automatically detect dependencies among tasks and consequently schedule them. Note that the abstract model that support this mechanism has been widely used before the inclusion in the standard. Among other, the StarSs [12] and StarPU [11] runtime systems have certainly strongly contributed to that progress. StarPU (read *PU) has been specifically designed for abstracting the underlying architecture so that in can execute task on any type of hardware (CPU core, GPU, ...). As a consequence, it is convenient for exploiting heterogeneous architecture. For this reason, we decided to use it for implementing the proposed task-based extension of MaPHyS.

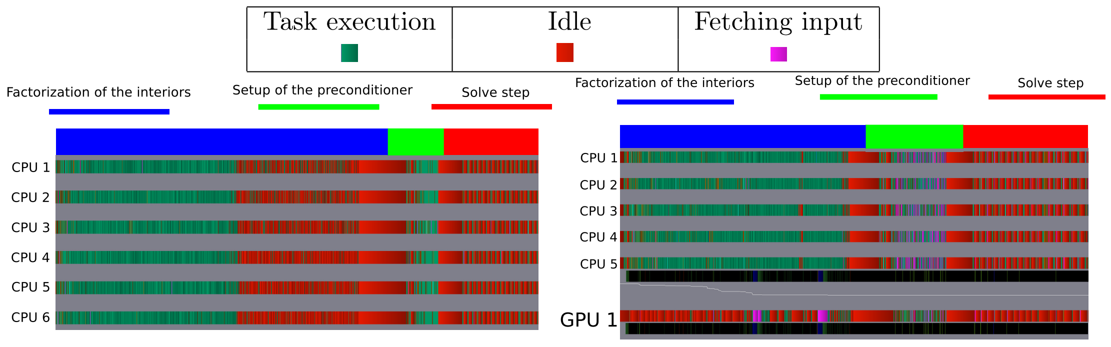
StarPU provides a convenient interface for implementing and parallelizing applications or algorithms that can be described as a graph of tasks. Tasks have to be explicitly submitted to the runtime system along with the data they work on and the corresponding data access mode. Through data analysis, StarPU can automatically detect the dependencies among tasks and build the corresponding DAG. Once a task is submitted, the runtime tracks its dependencies and schedules its execution as soon as these are satisfied, taking care of gathering the data on the unit where the task is actually executed.

[3]https://project.inria.fr/chameleon/

5 Experimental results

In this section we present preliminary results of our task-based prototype of MaPHyS. The tests presented in this section were performed on the `PlaFRIM 2` platform situated installed at Inria Bordeaux-Sud-Ouest, more precisely on the *sirocco* nodes. These nodes are composed of two Dodeca-core Haswell Intel Xeon E5-2680, for a total of 24 cores per node, and 128 GB of RAM memory. Each node is equipped with 4 Nvidia K40-M GPUs, each one having 12 GB of RAM. We consider the SPD `Audi_kw` matrix (size $n = 900K$ and non-zeros $nnz = 392M$) to illustrate the behavior of the proposed prototype solver. Both Chameleon and PaStiX use the version 1.1 of the StarPU runtime system described in Section 4.2. All tests were performed in double precision.

Figures 3a and 3b depict the traces obtained on one node, using only CPU cores or both GPUs and CPU cores, respectively. In both cases, the matrix has been decomposed in four subdomains. Each subdomain is associated with one MPI process in charge of a subset of six CPU cores (left trace 3a), or six CPU cores and one GPU (right trace 3b), respectively. The runtime system orchestrates the execution of the tasks on the different processing units. The traces represent the execution on one particular subdomain. In the heterogeneous case, each GPU has a CPU core dedicated to handle it (see Section 4.2).



(a) 6 CPUs per subdomain (b) 5 CPUs and a GPU per subdomain

Figure 3: Multicore execution trace associated with one subdomain of the MPI+task MaPHyS prototype processing the `Audi_kw` matrix. Four subdomains (hence four processes) are used in total.

The resulting traces show the versatility of the approach that composed multi-threaded and task-based numerical kernels. The processing units are abstracted and the same code may be executed indistinguishably on the homogeneous or on the heterogeneous cases. Although the implementation is still preliminary and not optimized, Table 1 shows that the resulting timings allow for accelerating all three numerical steps with the use of one GPU per subdomain in spite of the preliminary design. The *setup of the preconditioner* benefits from the highest acceleration as it mostly consists of a dense factorization accelerated with Chameleon. The *factorization of the interiors* has a limited (but not negligible) acceleration because PaStiX internal kernel has not been tuned for the Nvidia K40-M GPU. The *solve step* phase is accelerated thanks to the application of the preconditioner with Chameleon. The time differences between the fastest and slowest subdomain computation for the factorization of the interiors and preconditioner setup are related to the matrix partitioning that balances the splitting of the adjacency graph but not the calculation associated with each subgraph. They are some ongoing work in the graph community to address this issue that is out of the scope of this work.

		Multicore case	Heterogeneous case
Factorization of the interiors	min	19.6	23.3
	avg	37.2	31.7
	max	50.8	38.2
Setup of the preconditioner	min	4.80	1.10
	avg	7.02	3.63
	max	9.81	7.37
Solve step	min	13.1	11.8
	avg	13.2	11.8
	max	13.2	11.8

Table 1: Minimum, average and maximum time per subdomain for the MPI+task MaPHyS prototype for the multicore case (Figure 3a) and the heterogeneous case (Figure 3b) processing the `Audi_kw` matrix. Four subdomains (hence four processes) are used in total and a dense preconditioner is applied.

6 Concluding remarks

We have proposed an MPI+task extension of MaPHyS for exploiting distributed memory heterogeneous platforms. The modular design of MaPHyS allowed to use the task-based PaStiX and Chameleon sparse and dense direct libraries, respectively, in order to benefit from their ability to efficiently exploit the underlying heterogeneous architecture.

Although this prototype extension of MaPHyS is working properly and showed the feasibility of the proposed approach, designing a solid MPI+task version of MaPHyS would require further work. First of all, the proposed approach still follows a bulk-synchronous parallelism [38] (also sometimes designated as fork-join approach) pattern. Indeed, the calls to PaStiX and Chameleon, yet local to each subdomain, induce costly pre-processing. On the one hand, PaStiX need to perform a reordering of the variables to limit fill-in and a symbolic factorization. These steps are sequential in the present prototype. Although there exist parallel implementations of these steps, they are known to have a very limited parallel efficiency. To overcome the subsequent synchronizations, it would therefore be necessary to overlap these symbolic preprocessing steps with other numerical operations. On the other hand, following Plasma design, Chameleon first decomposes the dense matrix in tiles, which is also a synchronizing operation. As for Plasma, there exists an advanced interface allowing for tackling matrices already decomposed into tiles. Using this interface would certainly alleviate the bottleneck occurring within the *setup of the preconditioner* when calling the dense solver.

Other operations involved in the iterative solution step such as level-one BLAS and matrix-vector product could be implemented with a task-based approach. In the case of a dense preconditioner, these operations could also be implemented by calling BLAS operations implemented in Chameleon. However, in the present state, without using the advanced interface discussed above, the synchronizations would occur multiple times per iteration. A full task-based CG solver is presented in [7] and discuss in details how synchronization points can be alleviated.

To completely alleviate the synchronizations between the different sequences into which MaPHyS is decomposed, it would be necessary to further overlap communications with computations. This could be performed with a clever usage of asynchronous MPI calls. This approach is relatively difficult to implement and has been applied to overlap main stages in MaPHyS, both in the MPI+thread version and in the MPI+task prototype discussed in this section. However, relying on this paradigm for performing fine-grain overlapping would be challenging and certainly result in a code very complex to maintain. Alternatively, the MPI calls can be appended to the task flow. Doing so, the task-based runtime system can dynamically decide when to perform the actual MPI call to the MPI layer and interleave them with fine-grain computational tasks. Modern runtime systems such as StarPU and `PaRSEC` provide such an opportunity. However, even in that case, the task-flow would still have to be designed accordingly to the mapping between tasks and processes. On the contrary, the full task approach (see Figure 2a) would allows for fully abstracting the hardware architecture and makes the mapping issues practically orthogonal to the design of the task flow.

References

[1] MAGMA Users' Guide, version 0.2. http://icl.cs.utk.edu/magma, November 2009.

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

[3] E. Agullo, L. Giraud, A. Guermouche, and J. Roman. Parallel hierarchical hybrid linear solvers for emerging computing platforms. *Compte Rendu de l'Académie des Sciences - Mécanique*, 339(2-3):96–105, 2011.

[4] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[5] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 521–532, 2013.

[6] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. *ACM Transactions On Mathematical Software*, 2016. To appear.

[7] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, and Jean Roman. Task-based Conjugate-Gradient for multi-GPUs platforms. Research Report RR-8192, INRIA, 2012.

[8] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann, 2002.

[9] 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.

[10] 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.

[11] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[12] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An extension of the starss programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[13] 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.

[14] G. Bosilca, A. Bouteiller, A Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Distributed-Memory Task Execution and Dependence Tracking within DAGuE and the DPLASMA Project. *Innovative Computing Laboratory Technical Report*, 2010.

[15] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013.

[16] Alfredo Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM J. Scientific Computing*, 35(4), 2013.

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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

[25] 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.

[26] 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.

[27] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Exploiting asynchrony from exact forward recovery for due in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 53:1–53:12, New York, NY, USA, 2015. ACM.

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

[29] X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems.* PhD thesis, LaBRI, Université Bordeaux, Talence, France, February 2015.

[30] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes, 05/2014 2014.

[31] 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.

[32] 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.

[33] Florent Lopez. *Task-based multifrontal QR solver for heterogeneous architectures.* PhD thesis, University Paul Sabatier, Toulouse, France, 2015. submitted.

[34] T. Mathew. *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations.* Springer Lecture Notes in Computational Science and Engineering. Springer, 2008.

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

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

[37] 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.

[38] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[39] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009.

[40] 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.

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

ISSN 0249-6399