


European Commission

Horizon 2020
European Union funding
for Research & Innovation

**E-Infrastructures**
**H2020-EINFRA-2015-1**

**EINFRA-5-2015: Centres of Excellence for computing applications**

# EoCoE

## Energy oriented Center of Excellence for computing applications

**Grant Agreement Number: EINFRA-676629**

## D1.8 - M36
## Software Technology Improvement

## Project and Deliverable Information Sheet

| | | |
|---|---|---|
| EoCoE | Project Ref: | EINFRA-676629 |
| | Project Title: | Energy oriented Centre of Excellence |
| | Project Web Site: | http://www.eocoe.eu |
| | Deliverable ID: | D1.8 - M36 |
| | Lead Beneficiary: | CEA |
| | Contact: | Matthieu Haefele |
| | Contact e-mail: | matthieu.haefele@maisondelasimulation.fr |
| | Deliverable Nature: | Report |
| | Dissemination Level: | PU* |
| | Contractual Date of Delivery: | M36 30/09/2018 |
| | Actual Date of Delivery: | 30/09/2018 |
| | EC Project Officer: | Carlos Morais-Pires |

* - The dissemination level are indicated as follows: PU – Public, CO – Confidential, only for members of the consortium (including the Commission Services) CL – Classified, as referred to in Commission Decision 2991/844/EC.

## Document Control Sheet

| | | |
|---|---|---|
| Document | Title: | Software Technology Improvement |
| | ID: | D1.8 - M36 |
| | Available at: | http://www.eocoe.eu |
| | Software tool: | LaTeX |
| Authorship | Written by: | Luc Giraud (INRIA), Leonardo Bautista Gomez (BSC), O. Abramkina (CEA/MDLS), Daniel Ruiz (IRIT), Yvan Notay (ULB), Salvatore Filippone (Cranfield University), G. Maait (Inria) |
| | Contributors: | Kai Keller (BSC), Maciej Brzeźniak (PSNC), Karol Sierociński (PSNC), Tomasz Paluszkiewicz (PSNC), Julien Bigot (CEA), R. Lacroix (CNRS/IDRIS), Y. Meurdesoif (CEA/LSCE), M.H. Nguyen (CNRS/LSCE), Iain Duff (RAL-CERFACS), Philippe Leleux (CERFACS), Fahreddin Sukru Torun (IRIT-CNRS), Daniela di Serafino (UNICampania), Pasqua D'Ambra (CNR), Ambra Abdullahi Hassan (UNITOV), E. Agullo (Inria), L. Giraud (Inria), M. Kuhn (Inria), L. Poirel (Inria) |
| | Reviewed by: | Matthieu Haefele (MdlS), Paul Gibbon (JSC), PEC members |

## Contents

## 1. Document release note

This document is the first report on software technology improvement. Some activities are already implemented and some others are still on going work. The final document D1.8 due for M36 will replace this document and contain the final status of all activities that have taken place in EoCoE.

## 2. Motivation

From the outset, the EoCoE project was equipped with a diverse set of HPC expertise in WP1 designed to tackle a variety of possible performance bottlenecks in the applications from the four domain pillars. These range from state-of-the-art computer science tools for performance analysis, parallel IO etc. . . , to advanced linear algebra and other applied mathematics methods. This permits a layered approach to application tuning, starting from initial blind analysis to identify problematic code portions, then subsequently delving deeper to undertake complete refactoring of critical, compute-intensive routines. The key feature of EoCoE has been the close interaction between WP1 and the application domains WP2-WP5, enabling real-world energy applications to effectively exploit the existing European computing infrastructure and better equip them for future hardware advances. Ultimately we expect this work to expedite advances in simulations of low-carbon energy systems and technology.

This deliverable gathers the status of software technology advances conducted within the project. By software technology we mean specific computer science libraries or packages used in the scientific applications developed by EoCoE partners. The packages supported in EoCoE - such as the linear algebra libraries AGMG and PSBLAS - existed before the project and will continue to exist after it formally ends. Typically this software has been developed as part of a research project and as such, is not always mature enough in term of software engineering and robustness. The aim of the activities conducted here is to improve this situation and bring these packages closer to production-readiness.

# 3. Fault Tolerance Interface

## 3.1 Package ID card

| | |
|---|---|
| Package name | Fault Tolerance Interface (FTI) |
| Functionalities offered | Multilevel Checkpointing in multiple formats and |
| Description | FTI is a multilevel checkpointing library with multiple features to reduce the stress on the parallel file system and reduce checkpointing overhead. |
| Number of users | 1-10 |
| Library dependencies | CMake, MPI |
| Package references | https://github.com/leobago/fti |
| Contact | • Leonardo Bautista Gomez (leonardo.bautista@bsc.es)<br>• Kai Keller (kai.keller@bsc.es) |

FTI stands for Fault Tolerance Interface[1] and is a library that aims to give computational scientists the means to perform fast and efficient multilevel checkpointing in large scale supercomputers. FTI leverages local storage plus data replication and erasure codes to provide several levels of reliability and performance. FTI is application-level checkpointing and allows users to select which datasets needs to be protected, in order to improve efficiency and avoid wasting space, time and energy. In addition, it offers a direct data interface so that users do not need to deal with files and/or directory names. All metadata is managed by FTI in a transparent fashion for the user. If desired, users can dedicate one process per node to overlap fault tolerance workload and scientific computation, so that post-checkpoint tasks are executed asynchronously.

## 3.2 Improvement achieved

| | |
|---|---|
| Contributors | Leonardo Bautista Gomez (BSC), Kai Keller (BSC), Maciej Brzeźniak (PSNC), Karol Sierociński (PSNC), Tomasz Paluszkiewicz (PSNC), Julien Bigot (CEA) |

During the reporting period several improvements were proposed to the FTI library implementation based on the automated code analysis, manual code review and testing the library with the built-in tests, dedicated testing applications as well as by integrating FTI with the Gysela application. The following paragraphs provide the details of this work.

First of all, the FTI library has been integrated with the continuous integration and static code analysis tools including Travis CI and Coverity scan. This enabled a more systematic approach to the further library improvement work.

In the second stage an extensive code analysis was conducted. It started with a static analysis of the library using Coverity scan and cppchek. At this stage 80 problems were found, mainly related to memory management, such as failure to free allocated memory segments of other resources. These problems were fixed and merged and to the code base. Another angle of the code analysis was to investigate MPI calls using a MUST checker. Within this analysis 2 problems were found and solved. Next the library I/O behaviour was checked using Darshan. As a result it was suggested to change the way of writing the level 4 checkpoints, by avoiding creating checkpoint file for each of the running processes,

as this led to excess number of the checkpoint files.

In the third phase, the library built-in examples were used for testing the library. While running these tests, several run-time problems were found and fixed, including crashes during the recovery process or invalid handling of the input options as well as failure to take checkpoints in several situations. Most of these issues were fixed and fixes were merged to the main branch of the code.

In the fourth phase, a code refactoring was. It included splitting the source code into the functionally independent subprojects as well as unifying the code building approach.

Above mentioned activities were performed by PSNC with the aid and in consultancy with the FTI library developers. FTI-Gysela integration. Another work related to FTI library was to integrate it with Gysela, a scalable computing application. This work was performed by CEA and FZJ. At PSNC several tests were performed based on the benchmarks integrated with the application, leading to several improvements of the library.

First of all, comparison of the execution time of several Gysela workflows with and without FTI-based checkpointing was conducted. Within these tests both synchronous and asynchronous mode of the library operation were tested (note that in the async mode the dedicated processes, i.e. one per node, are created, and they are used for taking asynchronous checkpoints). Weak scaling was also tested. The tests were conducted in two phases: smaller test cases (up to 128 nodes of the Eagle cluster at PSNC) as well as bigger test cases (256 and more nodes). In the small-scale tests no major differences of the execution time (with and without FTI) were observed. These tests however had to be repeated because of using improper input values for Gysela during the first approach to testing. There was also an attempt to run bigger scale tests, however (as of Nov 2016) most of them failed due to the several repeating problems with the Eagle cluster (related to the infrastructure issues, external to the project activities). These test might need to be repeated in future.

In the most recent phase of the FTI library testing six testing applications were developed and run along with the FTI library with different configuration files for the library (the file determines e.g. the mode of taking checkpoints: synchronous vs asynchronous etc.). In the following paragraphs details of the testing applications are provided.

The first test (addInArray) uses basic FTI functions in order to make checkpoints and restarts the program from the last saved checkpoint. The aim of this test is to check if recovery is successful and all protected variables are correctly recovered (note: protected variables are those that are 'marked' to be included in the checkpoints). The recovered values are compared with the values expected at a given iteration (acquired by a full, non-interrupted execution of the testing application). Within the second test (diffSizes) every of the running processes (X-Y) expands its array (Realloc), and notifies the FTI about resizing the variable (by using relevant FTI function) and changes values written in it. Even ranks have 3 times larger array than odd ranks. After several iterations the program is stopped and restarted from the last checkpoint written. After the restart, it is checked if recovery is successful. At the end all the processes send their arrays to root process that checks if the results are correct. Problems with recovery after variable size change were notified while performing the tests. The new FTI function (FTI_Realloc) was proposed in order to solve this issue, by enabling to explicitly notify the FTI library about the fact the size of the variable was changed. Within the third test (heatdis) the examples

from the FTI library are used in order to check correctness of the operation of the FTI functions such as FTI_Snapshot. Similarly to other tests the application is restarted after the simulated failure using the last checkpoint as an input. In this particular test, we tested if the snapshot functionality works properly despite scaling up the job size. It is important to note the snapshot function is designed to make a decision whether the actual checkpoint operation should be performed/triggered or not. In the current design and implementation the decision is based on the time criteria. During our tests we proved that triggering the checkpoints should not be based on the time criteria only as it is general it is hard to predict the application total execution time (or the time needed for particular iterations) if the job size is scaled up. Within the fourth test (lvlsRecovery) we examine the application recovery using all the checkpoint levels defined in the library. The computing job is stopped after some iterations instead of using FTI_Finalize function. In that way we keep all the levels saved on the persistent storage, while using FTI_Finalize function would cause removing the checkpoints made on different levels (L1, L2, L3). This lets us testing recovery from these various levels. The fifth test (nodeFlag) makes all the levels of the checkpoint and searches across the log files in order to make sure that there is only one process per node that goes through FTIs nodeFlag condition section. Example situation where such approach is needed is changing the folder for the storing the checkpoints. In that case only one process can make the change (this constitutes a 'critical section'). The sixth test (tokenRing) is very similar to addInArray test, but it uses FTI option to protect structures instead single variables. Within the tests some synchronization issues in the FTI library were discovered (e.g. some processes tried to use files or folder that did not exist yet) and fixes are provided.

In the last period the efforts on improving the FTI testing scripts and procedures were continued. In particular scripts for testing the FTI were created and add new tests were added to the automated testing mechanisms. The Travis CI configuration files were expanded in order to make automatic tests after every commit. We also managed to use 3 different compilers: gcc, clang and icc for compiling the FTI library and the testing applications, which improved scope and directions of the FTI testing.

## References

[1] Leonardo Arturo Bautista-Gomez and al. Fti: High performance fault tolerance interface for hybrid systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

# 4. XML IO Server (XIOS)

## 4.1 Package ID card

| | |
|---|---|
| Package name | XIOS |
| Functionalities offered | IO server and online post-processing |
| Description | XIOS is a hierarchical data managment library created by CEA/LSCE to handle its large needs in terms of data flow. Its asynchronous and flexible implementations enhance especially the uncoupling between computing needs ands data managment. |
| Number of users | The french climate community and a growing part of the european one. |
| Library dependencies | MPI, HDF5, NetCDF4 |
| Package references | `http://forge.ipsl.jussieu.fr/ioserver` |
| Contact | • O. Abramkina (olga.abramkina@cea.fr)<br>• R. Lacroix (remi.lacroix@idris.fr)<br>• Y. Meuredesoif (yann.meurdesoif@cea.fr) |

## 4.2 Improvement achieved

| | |
|---|---|
| Contributors | O. Abramkina (CEA/MDLS), R. Lacroix (CNRS/IDRIS), Y. Meurdesoif (CEA/LSCE), M.H. Nguyen (CNRS/LSCE) |

To be able to provide XIOS to a larger spectrum of applications than climate simulations, it was necessary to release some contraints on the XIOS implementation. Some on the heart of the library, like for the managment of grids or calendars, some on the output file backend. UGRID will illustrate this last point.

## 4.3 Grids composition

While a grid in previous versions of XIOS could only be composed of maximum one domain (a 2D plan, structured or unstructured) and one axis, XIOS is not any more limited to 3-dimension grids.

By allowing a grid to contain many domains and axis, XIOS provides a simple way to create high dimension grids. Moreover, with a new syntax, defining a multidimensional grid is easier than ever. For example, definition of a 6-dimension grid, as GYSELA's, can be done as following :

```
<grid>
<axis id="axis1" />
<axis id="axis2" />
<axis id="axis3" />
<axis id="axis4" />
<axis id="axis5" />
<axis id="axis6" />
</grid>
```

Users can easily define their own distribution of a grid by specifying the distribution of composing domain and/or axis. This deep modification has been the opportunity to also allow "zero-dimension" grids or scalar, which makes XIOS a tool to process and write various range of data.

Concerning grids, another obligatory "climate-specific" specification is lightened. In this way some meta-data related to longitudes and latitudes are optional, users choose the way to write out their data and associated meta-data.

### 4.4 Timeline managment

Since XIOS was originally developed to help dealing with the huge mass of data produced by climate simulations, the way it handled the simulation date and time was quite application-specific. Climate simulations are often used to study the evolution of the climate on Earth for large time scale, ranging from a few years to hundreds of years, with daily, monthly and/or yearly output frequencies. Due to this context, XIOS provided only Earth-based calendars and managed dates (for example the start date of the simulation) only as a fully-specified Earth date and time with the following format: "yyyy-mm-dd hh:mm:ss".

Although this calendar system was well-suited for climate simulations, it did not make much sense for some other simulations, for example those with a small simulation time or non Earth-based. In order to open XIOS to other scientific communities, we modified the calendar system so that is more flexible.

Some elements that used to be mandatory like the start date of the simulation are now optional to ease the configuration of simulations that are not tied to a specific date. In addition, the date/time format was reworked to allow partial date/time definition, for example with just a year or a date. It also allows defining an optional offset expressed as a duration (for example "2015-01-11 12:00:00 + 1d" or "2017 + 42h11m"). Being that the date/time definition can be completely omitted, it is possible to only specify the duration offset, making XIOS virtually calendar-free.

Additionally, we added a fully customizable calendar (possibly month-free and with leap-year support) that can be configured to be suitable for planets other than the Earth.

### 4.5 Unstructured extension : UGRID

A new file output format has been implemented into XIOS to meet the needs of communities working with unstructured grids. It follows the UGRID conventions for netCDF file format [1] and it allows users to store the topology of the underlying unstructured mesh. Currently XIOS supports 2D unstructured meshes of any shape (triangular, quadrilateral, etc) and their mixture.

A 2D mesh can be described in the simpest case by a set of points, or nodes in the UGRID terminology, and/or by a set of edges and faces. XIOS allows one to define data on any of these three types of elements (nodes, edges, and faces). XIOS generates a full list of connectivity attributes proposed by the UGRID conventions. For example, in case of a mesh composed of faces the stored connectivity attributes will be the following:

This work has been integrated into the LFRic model developed by the UK Met Office. Preliminary tests of the LFRic with XIOS on the I/O end on the Met Office Cray super-

```
edge_node_connectivity
face_node_connectivity
edge_nodes_connectivity
face_nodes_connectivity
face_edges_connectivity
edge_face_connectivity
face_face_connectivity
```

computer reveal good I/O performances. These results will be presented at ParCo2017, an international conference on HPC.

## References

[1] Ugrid conventions (v1.0). http://ugrid-conventions.github.io/ugrid-conventions/.

# 5. ABCD

## 5.1 Package ID card

| | |
|---|---|
| Package name | ABCD |
| Functionalities offered | Parallel sparse hybrid iterative and direct solver |
| Description | ABCD (Augmented Block Cimmino Distributed Solver) is a distributed hybrid (iterative/direct) solver for sparse linear systems. |
| Number of users | 1-10 |
| Library dependencies | MPI, MUMPS, BLAS, LAPACK, PaToH, Boost |
| Package references | https://bitbucket.org/apo_irit/abcd |
| Contact | • Iain Duff (iain.duff@stfc.ac.uk)<br>• Daniel Ruiz (daniel.ruiz@enseeiht.fr)<br>• Fahreddin Sukru Torun (ftorun@enseeiht.fr)<br>• Philippe Leleux (leleux@cerfacs.fr) |

ABCD Solver consists of two parallel methods which are parallel hybrid block Cimmino iterative method and parallel augmented block Cimmino (a pseudo-direct method). Both methods solve sparse systems of linear equations of the form $Ax = b$, where $A$ is a square sparse matrix, on distributed memory computers.

**Parallel Block Cimmino Hybrid Iterative Method**

This method follows the well-known block Cimmino method: a row projection method for solving linear systems, see [3] for more details. In this method $Ax = b$ is partitioned as blocks of rows:

$$\begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}. \tag{1}$$

and then the algorithm computes a solution iteratively from an initial estimate $x^{(0)}$ according to:

$$x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} A_i^{+} \left( b_i - A_i x^{(k)} \right). \tag{2}$$

Figure 1 shows a geometrical point of view of a sample iteration of Cimmino algorithm when there is two partitions.

The iterations can be reformulated as:

$$\begin{aligned} x^{(k+1)} &= \left( I - \omega \sum_{i=1}^{p} A_i^{+} A_i \right) x^{(k)} + \omega \sum_{i=1}^{p} A_i^{+} b_i \\ &= Q x^{(k)} + \xi, \end{aligned} \tag{3}$$

where $\xi = \omega \sum_{i=1}^{p} A_i^{+} b_i$ and $Q = I - \omega \sum_{i=1}^{p} A_i^{+} A_i$. Looking at the stationary point, this is

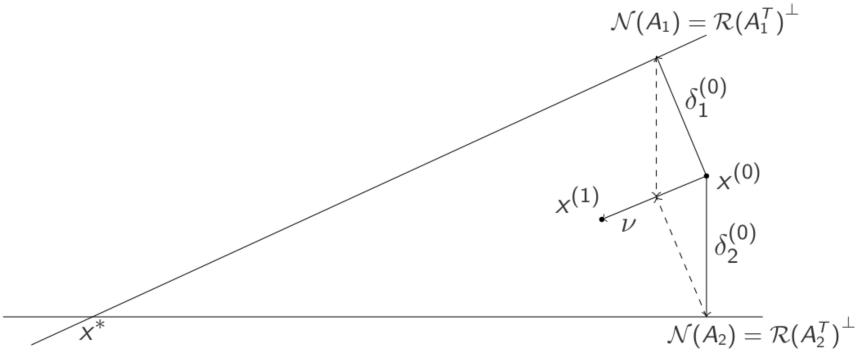



Figure 1: Geometric point of view of the block Cimmino Algorithm with $p = 2$

equivalent to the linear system

$$Hx = \xi, \tag{4}$$

where $H = I - Q$. Since $H$ is symmetric positive definite we can solve this system by using Conjugate Gradient (CG) iterative method. The CG accelerated block Cimmino algorithm is studied in details [3, 4, 5, 8]. One of the issues in the CG iteration is to compute the projections onto $A_i^T$. The chosen method is through the solution of an augmented system [2] of the form

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ r_i \end{pmatrix}, \tag{5}$$

where $r_i = b_i - A_i x^{(k)}$. The solution subvector $u_i$ of the augmented system gives the projection. These systems are symmetric indefinite and we can solve them using the direct parallel solver MUMPS, which makes efficient use of the parallelism and gives to our solver the hybrid property. Our goal in this solver is then to have partitions capturing the ill-conditioning of the matrix that will be tackled by the direct solver so that the CG can converge quickly.

Figure 2 illustrates the execution steps of the parallel block Cimmino algorithm. In the algorithm, if there are more MPI processes than row-blocks, ABCD adopts a master-slave approach for the distributed solution of the system. Each master processor owns one row-block and creates an augmented system which is assigned to one MUMPS instance, referred as master. Then each slave processor is assigned to a master processor with respect to load criteria. More slaves are assigned to highly loaded master processors. The slave processes are exploited to cooperate with the masters' factorization and solution within MUMPS.

The convergence of the block Cimmino iterative method depends heavily on the angles between the subspaces determined by the row-block partitioning. Intelligent row-block partitioning methods are proposed [6, 9] in order to improve the convergence of block Cimmino method. In the extreme case where subspaces would be orthogonal, only one iteration would be necessary to get to the solution [7] (pseudo-direct solver). In the next subsection, we will elaborate this method.

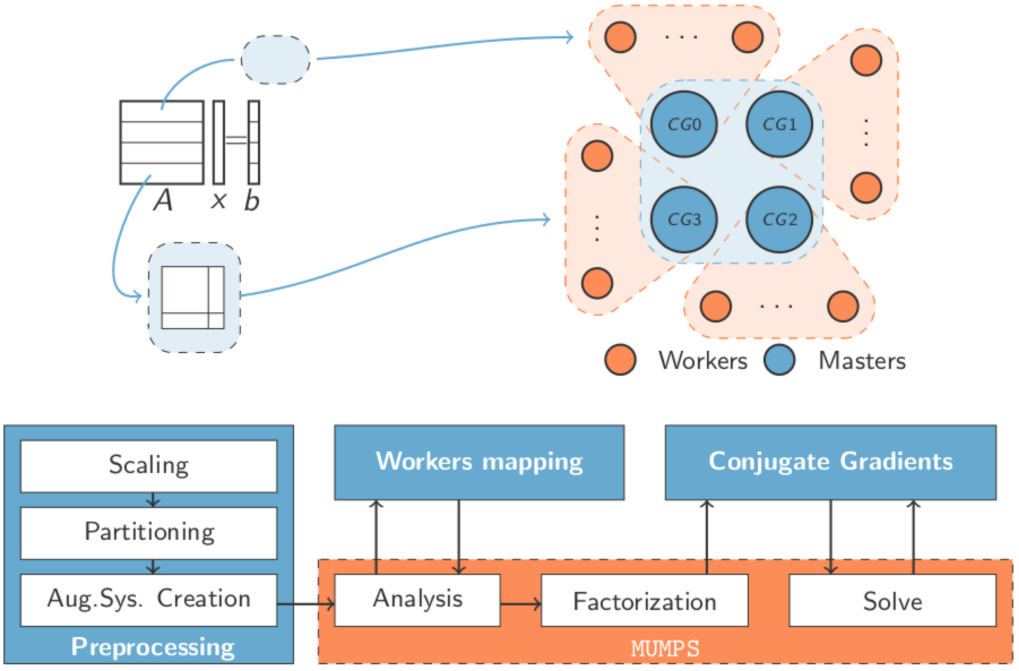



Figure 2: Execution steps of the parallel block Cimmino distributed solver

**Parallel Augmented Block Cimmino Pseudo-direct Method**

To understand the augmented block Cimmino algorithm, suppose that we have a matrix $A$ with three partitions, described as follows:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & & & & A_{1,3} \\ & A_{2,1} & A_{2,2} & A_{2,3} & & \\ & & & A_{3,2} & A_{3,3} & A_{3,1} \end{bmatrix}, \tag{6}$$

where $A_{i,j}$ the sub-matrices of $A_i$, $i$-th row-block partition, that is interconnected algebraically to the partition $A_j$, and vice versa.

The goal of the augmented block Cimmino algorithm is to make these three partitions mutually orthogonal to each other, meaning that the inner product of each pair of partitions is zero. We consider two different ways to augment the matrix to obtain these zero matrix inner products.

The first way to augment the matrix to make all the partitions mutually orthogonal to each other is obtained by putting the product $C_{ij} = A_{ij}A_{ji}^T$ on the right of the partition $A_i$ and adding $-I$ on the right of $A_j$ viz.

$$\bar{A} = \left[\begin{array}{cccccc|ccc} A_{1,1} & A_{1,2} & & & & A_{1,3} & C_{1,2} & C_{1,3} & \\ & A_{2,1} & A_{2,2} & A_{2,3} & & & -I & & C_{2,3} \\ & & & A_{3,2} & A_{3,3} & A_{3,1} & & -I & -I \end{array}\right]$$

The second way is to repeat the submatrices $A_{ij}$ and $A_{jj}$ reversing the signs of one of

them to obtain the augmented matrix $\bar{A}$ as in the following

$$\bar{A} = \left[\begin{array}{ccccc|ccc} A_{1,1} & A_{1,2} & & & A_{1,3} & A_{1,2} & A_{1,3} & \\ & A_{2,1} & A_{2,2} & A_{2,3} & & -A_{2,1} & & A_{2,3} \\ & & & A_{3,2} & A_{3,3} & A_{3,1} & -A_{3,1} & -A_{3,2} \end{array}\right]$$

Both ways make $\bar{A}_i \bar{A}_j{}^T$ zero for any pair $i$ and $j$, and so the new matrix has mutually orthogonal partitions.

Running our solver in the augmented block Cimmino mode will go through the following steps:

- Partition the system into strips of rows ($A_i$ and $b_i$ for $i = 1 \ldots, p$)
- Augment the different partitions according to the selected algorithm
- Create the augmented systems
- Analyse and factorize the augmented systems using the direct solver MUMPS
- Build an auxiliary matrix $S$ in parallel and use it to solve a reduced linear system. The result is then used to obtain the solution for the original linear system $Ax = b$.

For more details, we refer to [7, 10].

We consider the following row-blocks

$$\left[\begin{array}{cc} A & C \\ B & S \end{array}\right] \left[\begin{array}{c} x \\ y \end{array}\right] = \left[\begin{array}{c} b \\ f \end{array}\right],$$

where $x$ is ensured to be the same solution vector of $Ax = b$. We can denote by $\bar{A}$ the submatrix $[A\ C]$ where $C$ as been chosen to enforce the $p$ subspaces to be orthogonal as illustrated above, so that we have $\bar{A}^+ b = \sum_{i=1}^{p} A_i^+ b_j$. $f$ and $S$ are given by $f = -Y\bar{A}^+ b$ and $S = Y(I - P)Y^T$, with $Y = [0\ I]$. Finally the solution is given by

$$\left[\begin{array}{c} x \\ y \end{array}\right] = \bar{A}^+ b + (I - P)Y^T S^{-1} f$$

because of mutual orthogonality between row-blocks $\bar{A}$ and $[B\ S]$.

To obtain the solution practically, we currently build S and factorize it using a direct solver. The added value of this approach is the fact that the columns of $S$ can be built in an embarrassingly parallel fashion. The memory cost can be prohibitive in the case where $S$ is not small or sparse enough, but we observe in many cases that $S$ remains reasonable enough to make this approach computationally effective. The fact that $S$ is symmetric positive definite also offers the possibility of computing $S^{-1}f$ iteratively using conjugate gradients, without building $S$ explicitly.

### 5.2 Improvement achieved

| | |
|---|---|
| Contributors | Daniel Ruiz (IRIT, WP1), Iain Duff (RAL-CERFACS, WP1), Philippe Leleux (CERFACS, WP1), Fahreddin Sukru Torun (IRIT-CNRS, WP1) |

The package has been improved from software engineering, performance and maturity point of view. The following list summarizes our improvements:

- Improvements on scattering row-blocks among Master processes:
  - New efficient row-block distribution algorithm which ensures balanced workload on each Master processes when there are more number of blocks than the number of Master processes.
  - New communication minimizing row-block distribution scheme is implemented.
- Improvements on master-slave scheme:
  - Added an ability to convert some master processes to slave processes.
  - Improved node/MPI distribution for multi-node distributed memory architectures.
- Improved uniform partitioning method which works consistently for all kind of problems.
- Added an ability to apply manual partitioning from a file.
- Added an ability to use a starting guess vector for the CG accelerated block Cimmino.
- Improved matrix scaling using parallel MC77 algorithm.

We introduce Table 1 and Figure 3 in order to see the impacts of improvements over the parallel performance of ABCD by solving some real problems. In this experiment, we used three sparse nonsymmetric matrix, which are cage13, Hamrle3 and memchip, from SuiteSparse Matrix Collection [1]. Table 1 shows the parallel solution times of iterative block Cimmino algorithm for these problems. Figure 3 illustrates the performance gains in terms of solving time after improvements as percentages. As seen in these results, our advances on ABCD yields quite good performance improvement upto 75% on the performance of parallel solving.

Table 1: Parallel execution times in seconds for CG solution of block Cimmino algorithm before and after the improvements

| Problems | Old version | Improved version |
|---|---:|---:|
| cage13 | 9.21 | 8.65 |
| Hamrle3 | 4180.00 | 1040.00 |
| memchip | 4040.00 | 3860.00 |

## References

[1] Davis, Timothy A and Hu, Yifan. The University of Florida sparse matrix collection *ACM Transactions on Mathematical Software (TOMS)*, volume 38,1,1, ACM 2011.

[2] Mario Arioli, Iain Duff, and Peter PM de Rijk. On the augmented system approach to sparse least-squares problems. *Numerische Mathematik*, 55(6):667–684, 1989.

[3] Mario Arioli, Iain Duff, Joseph Noailles, and Daniel Ruiz. A block projection method for sparse matrices. *SIAM Journal on Scientific and Statistical Computing*, 13(1):47–70, 1992.

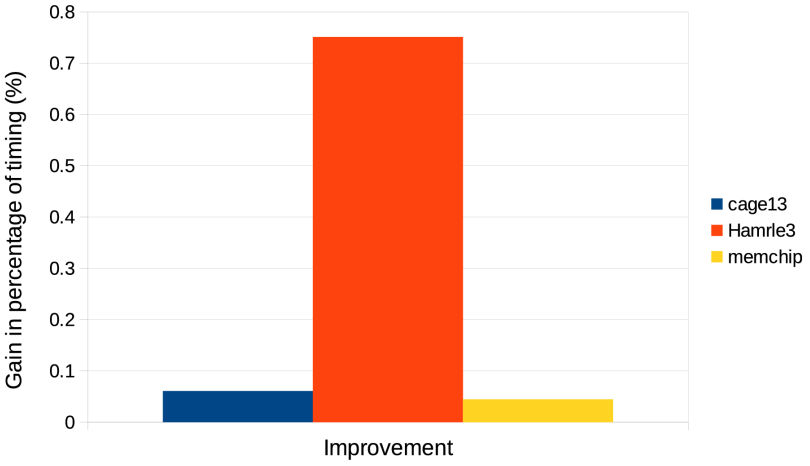



Figure 3: Performance gains of parallel block Cimmino after and before the modifications.

[4] Mario Arioli, Iain S Duff, Daniel Ruiz, and Miloud Sadkane. Block lanczos techniques for accelerating the block cimmino method. *SIAM Journal on Scientific Computing*, 16(6):1478–1511, 1995.

[5] Randall Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):168–193, 1992.

[6] LA Drummond, Iain S Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. Partitioning strategies for the block cimmino algorithm. *Journal of Engineering Mathematics*, 93(1):21–39, 2015.

[7] Iain Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block cimmino distributed method. *SIAM Journal on Scientific Computing*, 37(3):A1248–A1269, 2015.

[8] Daniel Ruiz and Miloud Sadkane. Techniques for accelerating the block cimmino method. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, volume 62, page 98. SIAM, 1992.

[9] F. Sukru Torun, Murat Manguoglu, and Cevdet Aykanat. A novel partitioning method for accelerating the block cimmino algorithm. *CoRR*, abs/1710.07769, 2017.

[10] Mohamed Zenadi. *Méthodes hybrides pour la résolution de grands systèmes linéaires creux sur calculateurs parallèles*. PhD thesis, École Doctorale Mathématiques, Informatique et Télécommunications (Toulouse); 142547247, 2013.

# 6. AGMG

## 6.1 Package ID card

| | |
|---|---|
| Package name | AGMG |
| Functionalities offered | Linear system solver |
| Description | AGMG implements an aggregation-based algebraic multigrid method. This method solves algebraic systems of linear equations, and is expected to be efficient for large systems arising from the discretization of scalar second order elliptic PDEs (see for [3, 1, 4, 2] for details and performance assessment).<br>The method is however purely algebraic and may be tested on any problem. No information has to be supplied besides the system matrix and the right-hand-side. |
| Number of users | above 1000 |
| Library dependencies | None |
| Package references | `http://homepages.ulb.ac.be/~ynotay/AGMG` |
| Contact | Yvan Notay (ynotay@ulb.ac.be) |

## 6.2 Improvement achieved

| | |
|---|---|
| Contributor | Yvan Notay (ULB) |

A multithreaded version of the software package has been developed. Formerly (till release 3.2.4), AGMG was either sequential or MPI-based parallel. The latter version scales pretty well (see [5]), but requires that the matrix of the system to solve is distributed on as many MPI ranks as there are available cores. This is not suited when the program calling the AGMG solver is parallelized only via multithreading, or uses an hybrid MPI+OpenMP programming model.

The new multithreaded version (releases 3.3.0 and above) is either pure OpenMP or hybrid MPI+OpenMP. The calling sequence for the pure OpenMP variant is the same as that for the sequential version, whereas the calling sequence for the hybrid variant is the same as that for the pure MPI version. Thus, in particular, the pure OpenMP variant allows one to obtain parallel speedup from a purely sequential program.

The used parallelization strategy is the same as for the pure MPI version: unknowns and corresponding matrix rows are distributed among the threads, and most computations are kept inherently parallel by constraining the aggregation algorithm to aggregate only unknowns assigned to a same thread. The Gauss–Seidel smoothing procedure is also truncated to become inherently parallel.

The new multithreaded version has been assessed on the large test suite used as basis of development for AGMG. This latter is a collection of large sparse linear systems stemming from the discretization of second order elliptic PDEs, and comprising:

- Problems on 2D/3D regular grids and on 2D/3D unstructured grids, some of them with strong local refinement;
- Problems with (big) jumps and/or (large) anisotropy in the PDE coefficients;

- Symmetric (SPD) and nonsymmetric problems (2D/3D convection-diffusion with dominating convection);
- finite difference and finite element (up to $p4$) discretizations.

Matrix sizes range from $5 \times 10^5$ to $3. \times 10^7$ , whereas the average number of nonzero entry per row ranges from 5. to 74. .

Timing results are displayed on Figure 4. One sees that for both sequential and multithreaded versions, the time per nonzero entry does not vary much despite the large variation in problems characteristics —the few pics correspond to challenging quasi singular convection-diffusion problems for which AGMG tends to outperform competitors, anyway. [1]

With the multithreaded version, the time needed per nonzero entry falls down to 0.1 microseconds on average. Using 8 cores, the speedup is roughly around 3.5 . This suboptimality is explained by the nature of the problem being solved: a sparse matrix problem with matrix stored in general sparse format and having only relatively few nonzero entries per row. It follows that the AGMG software code is strongly memory bound: beyond some point, having more computing power does not help if the memory bandwidth is not increased accordingly. (Observe that the test where ran on a simple workstation, without specific hardware enabling concurrent access of all cores to main memory.)

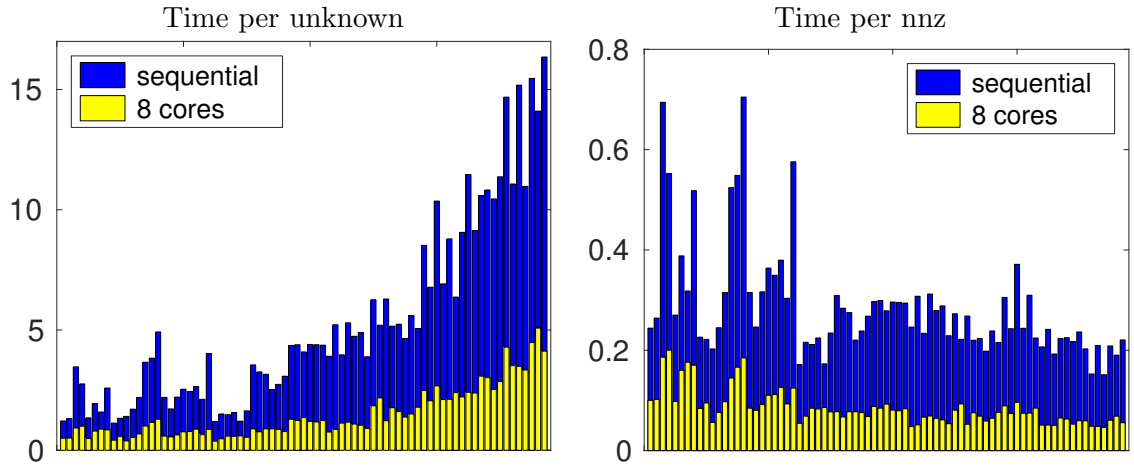



Figure 4: Total wall clock time to reduce the relative residual error by $10^{-6}$ – vs – problem index (problems ordered by increasing number of nonzero entry per row); times are reported in microseconds per unknown (left) or microseconds per nonzero entry (right); tests made on a desktop workstation – Intel XEON E5-2620 at 2.10GHz.

## References

[1] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109.

[2] ——, *Algebraic multigrid for moderate order finite elements*, SIAM J. Sci. Comput., 36 (2014), p. A1678–A1707.

[1] see `http://homepages.ulb.ac.be/~ynotay/AGMG/perf.html`

[3] Y. Notay, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.

[4] ——, *Aggregation-based algebraic multigrid for convection-diffusion equations*, SIAM J. Sci. Comput., 34 (2012), pp. A2288–A2316.

[5] Y. Notay and A. Napov, *A massively parallel solver for discrete poisson-like problems*, J. Comput. Physics, 281 (2015), pp. 237–250.

# 7. Maphys

## 7.1 Package ID card

| | |
|---|---|
| Package name | Maphys |
| Functionalities offered | Parallel sparse linear solveur |
| Description | Maphys is a hybrid direct/iterative solver that implements domain decomposition ideas at a pure algebraic form working only with the information associated with the user supplied sparse matrix. |
| Number of users | 1-10 ... |
| Library dependencies | MPI, MUMPS, PaStiX, BLAS, LAPACK, SCOTCH |
| Package references | `https://gitlab.inria.fr/solverstack/maphys/maphys` |
| Contact | • E. Agullo (emmanuel.agullo@inria.fr)<br>• L. Giraud (luc.giraud@inria.fr)<br>• M. Kuhn (matthieu.kuhn@inria.fr)<br>• G. Marait (gilles.marait@inria.fr)<br>• L. Poirel (louis.poirel@inria.fr) |

In this section we describe the design of the hybrid solver MaPHyS a non-overlapping domain decomposition. For the sake of simplicity, we assume that $\mathcal{A}$has a symmetric pattern. The MaPHyS package is available on the following git server:

https://gitlab.inria.fr/solverstack/maphys/maphys.

In this section, we present the design of the baseline MaPHyS hybrid solver. We aim at solving a sparse linear system of the form $\mathcal{A}x = b$, where $\mathcal{A}$ is a large, sparse, symmetric positive definite (SPD) matrix. We note $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ 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.

The governing idea behind substructuring or Schur complement methods is to split the unknowns into two categories: interior and interface vertices. We assume that the vertices of the graph $\mathcal{G}$ are partitioned into $N$ disconnected subgraphs $\mathcal{I}_1, \ldots, \mathcal{I}_N$ separated by the global vertex separator $\Gamma$. We also decompose the vertex separator $\Gamma$ into non-disjoint subsets $\Gamma_i$, where $\Gamma_i$ is the set of vertices in $\Gamma$ that disconnects $\mathcal{I}_i$ from other interior sets. 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 DDM 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 $\Gamma_i$. If we denote $\mathcal{I} = \cup \mathcal{I}_i$ and order vertices in $\mathcal{I}$ first, we obtain the following block reordered linear system

$$\begin{pmatrix} \mathcal{A}_{\mathcal{I}\mathcal{I}} & \mathcal{A}_{\mathcal{I}\Gamma} \\ \mathcal{A}_{\Gamma\mathcal{I}} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_{\mathcal{I}} \\ x_{\Gamma} \end{pmatrix} = \begin{pmatrix} b_{\mathcal{I}} \\ b_{\Gamma} \end{pmatrix} \tag{7}$$

where $x_\Gamma$ contains all unknowns associated with the separator and $x_\mathcal{I}$ contains the unknowns associated with the interiors.

Eliminating $x_\mathcal{I}$ from the second block row (with a direct method in our case, see below)

of Equation (7) leads to the reduced system

$$\mathcal{S}x_\Gamma = f \tag{8}$$

where

$$\mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma\mathcal{I}}\mathcal{A}_{\mathcal{I}\mathcal{I}}{}^{-1}\mathcal{A}_{\mathcal{I}\Gamma} \text{ and } f = b_\Gamma - \mathcal{A}_{\Gamma\mathcal{I}}\mathcal{A}_{\mathcal{I}\mathcal{I}}{}^{-1}b_\mathcal{I}. \tag{9}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix* and inherits the symmetric positive definite property of $\mathcal{A}$. This reformulation leads to a general strategy for solving (7). A Conjugate Gradient (CG) can be implemented to solve the reduced system (8). Once $x_\Gamma$ has been computed the interior variables $x_\mathcal{I}$ can be computed with one additional solve for the interior unknowns via

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

Because a direct solver is considered for this last step on can notice that

$$\frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|} \approx \frac{\|\mathcal{A}x - b\|}{\|b\|};$$

we use therefore the following normwise backward error stopping criterion for the PCG iterations

$$\frac{\|\mathcal{S}x_\Gamma - f\|}{\|b\|} \leq \varepsilon.$$

While the Schur complement system is significantly smaller and better conditioned than the original matrix $\mathcal{A}$, it is important to consider further preconditioning to accelerate the convergence of CG. We introduce the general form of the preconditioner considered in MaPHyS . To describe the main preconditioner in MaPHyS , we define $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i}\mathcal{S}\mathcal{R}_{\Gamma_i}^T$, where $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ is the canonical point-wise restriction which maps full vectors defined on $\Gamma$ into vectors defined on $\Gamma_i$. $\bar{\mathcal{S}}_i$ corresponds to the restriction of the Schur complement to the interface $\Gamma_i$ of each subdomain. If $\mathcal{I}_i$ is a fully connected subgraph of $\mathcal{G}$, and if for each $\gamma$ in $\Gamma_i$, there is an edge $(\gamma, v)$ in $\mathcal{G}$ with $v$ in $\mathcal{I}_i$, then the matrix $\bar{\mathcal{S}}_i$ is dense.

With these notations the algebraic Additive Schwarz preconditioner on the Schur system (AS/S) given by Equation (7) reads

$$\mathcal{M}_{AS/S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \bar{\mathcal{S}}_i^{-1} \mathcal{R}_{\Gamma_i}. \tag{10}$$

We notice that this preconditioner has a form similar to the Neumann-Neumann preconditioner [4, 8], but in the SPD case $\mathcal{M}_{AS/S}$ is always fully defined and SPD (as $\mathcal{S}$ is SPD [7]); which is not always the case for Neumann-Neumann. If we considered a planar graph partitioned into horizontal strips (1D decomposition) with $\Upsilon_k = \Omega_k \bigcap \Omega_{k+1}$ , the resulting Schur complement matrix has a block tridiagonal structure as depicted in Equation (11),

$$\mathcal{S} = \begin{pmatrix} \ddots & & & \\ & \mathcal{S}_{k,k} & \mathcal{S}_{k,k+1} & \\ & \mathcal{S}_{k+1,k} & \mathcal{S}_{k+1,k+1} & \mathcal{S}_{k+1,k+2} \\ & & \mathcal{S}_{k+1,k+2} & \mathcal{S}_{k+2,k+2} \\ & & & \ddots \end{pmatrix}. \tag{11}$$

For that particular structure of $\mathcal{S}$, the submatrices in boxes correspond to the $\bar{\mathcal{S}}_i$ that are the restriction of the Schur $\mathcal{S}$ to the interface of $\Omega_i$. Such diagonal blocks, which overlap with one another, are similar to the classical block overlap of the Schwarz method when writing in a matrix form for 1D decomposition. Similar ideas have been developed in a pure algebraic context in earlier papers (e.g., [5]) for the solution of general sparse linear systems.

***Parallelization strategy for distributed memory architectures.*** MAPHYS is based on an algebraic domain decomposition idea whose primary motivation is to naturally exploit a coarse grained parallelism between the computation performed on each sub-problem of the decomposition using MPI.

Based on the decomposition of $\mathcal{G}$ we can define a decomposition of the matrix $\mathcal{A}$ where each sub-matrix is associated with a subdomain and is allocated to one MPI process. The local interiors are disjoint and form a partition of the interior $\mathcal{I} = \sqcup \mathcal{I}_i$. Consequently the matrix $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ associated with the interior unknowns has a block diagonal structure; each diagonal block $\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}$ corresponds to the set of internal unknowns of $\Omega_i$.

Two subdomains $\Omega_i$ and $\Omega_j$ are defined as neighbor if their interfaces intercept that is $\Gamma_i \bigcap \Gamma_j \neq \emptyset$). The non disjoint union of the subdomain boundaries form the overall interface $\Gamma = \cup \Gamma_i$. This implies that a special attention has to be paid for the partitioning of $\mathcal{A}_{\Gamma\Gamma}$ as its entries are shared between different processes. In that respect the matrix entries of $\mathcal{A}_{\Gamma\Gamma}$ must be weighted so that the sum of the coefficients on the local interface submatrices are equal to one. For that, we introduce the *weighted local interface* matrix $\mathcal{A}^w_{\Gamma_i\Gamma_i}$ that satisfies $\mathcal{A}_{\Gamma\Gamma} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{A}^w_{\Gamma_i\Gamma_i} \mathcal{R}_{\Gamma_i}$, where we recall that $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ is the canonical point-wise restriction which maps full vectors defined on $\Gamma$ into vectors defined on $\Gamma_i$. In matrix terms, a subdomain $\Omega_i$ may then be represented by the *local matrix* $\mathcal{A}_i$ defined by

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{A}_{\Gamma_i\mathcal{I}_i} & \mathcal{A}^w_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{12}$$

The global Schur complement matrix $\mathcal{S}$ from (**??**) can then be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}^T_{\Gamma_i} \mathcal{S}_i \mathcal{R}_{\Gamma_i} \tag{13}$$

where

$$\mathcal{S}_i = \mathcal{A}^w_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i} {\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}}^{-1} \mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{14}$$

is the *local Schur complement* associated with subdomain $\Omega_i$. This local expression allows for computing local Schur complements independently from each other.

The $\bar{\mathcal{S}}_i$'s, involved in the definition of $\mathcal{M}_{AS/S}$, are the restriction of the global Schur complement to $\Gamma_i$ and can actually be built from this data distribution of the $\mathcal{S}_i$'s. To illustrate this construction, let us consider a sub-domain $\Omega_i$ with four neighbors and $\Gamma_i = E_m \cup E_g \cup E_k \cup E_m$ the union of the intersections of the boundary of $\Omega_i$ with each of its neighbors (assuming there is no cross-point, i.e., interface variables shared by more than two subdomains). The local Schur complement matrix associated with $\Omega_i$ has the

following $4 \times 4$ block structure

$$\mathcal{S}_i = \begin{pmatrix} \mathcal{S}_{m,m}^{(i)} & \mathcal{S}_{m,g} & \mathcal{S}_{m,k} & \mathcal{S}_{m,\ell} \\ \mathcal{S}_{g,m} & \mathcal{S}_{g,g}^{(i)} & \mathcal{S}_{g,k} & \mathcal{S}_{g,\ell} \\ \mathcal{S}_{k,m} & \mathcal{S}_{k,g} & \mathcal{S}_{k,k}^{(i)} & \mathcal{S}_{k,\ell} \\ \mathcal{S}_{\ell,m} & \mathcal{S}_{\ell,g} & \mathcal{S}_{\ell,k} & \mathcal{S}_{\ell,\ell}^{(i)} \end{pmatrix} \tag{15}$$

where each block is associated with each edge $E_j$, $j \in \{m, g, k, \ell\}$.

The matrix $\bar{\mathcal{S}}_i$ can be built from the local Schur complement $\mathcal{S}_i$ by collecting and summing (i.e., assembling in a finite element sense) its diagonal blocks thanks to a few neighbour to neighbour communications. For instance, the diagonal blocks of $\bar{\mathcal{S}}_i$ associated with the shared interface $E_k = \Gamma_i \cap \Gamma_j$ between $\Omega_i$ and $\Omega_j$ is $\mathcal{S}_{kk} = \mathcal{S}_{kk}^{(i)} + \mathcal{S}_{kk}^{(j)}$. Assembling each diagonal block of the local Schur complement matrices, we obtain the local assembled Schur complement, that is

$$\bar{\mathcal{S}}_i = \begin{pmatrix} \mathcal{S}_{m,m} & \mathcal{S}_{m,g} & \mathcal{S}_{m,k} & \mathcal{S}_{m,\ell} \\ \mathcal{S}_{g,m} & \mathcal{S}_{g,g} & \mathcal{S}_{g,k} & \mathcal{S}_{g,\ell} \\ \mathcal{S}_{k,m} & \mathcal{S}_{k,g} & \mathcal{S}_{k,k} & \mathcal{S}_{k,\ell} \\ \mathcal{S}_{\ell,m} & \mathcal{S}_{\ell,g} & \mathcal{S}_{\ell,k} & \mathcal{S}_{\ell,\ell} \end{pmatrix}.$$

**Algorithm 1:** MaPHyS `algorithm`

1 *partitioning step*
2 *factorization of the interiors*
3 *setup of the preconditioner*
4 *solve step*

Algorithm 1 summarizes how the classical parallel implementation of MaPHyS can be decomposed into four main phases:

- (1) the *partitioning step*, consisting of partitioning the adjacency graph $\mathcal{G}$ of $\mathcal{A}$ into several subdomains and distributing the $\mathcal{A}_i$ to different cores. This step is in practice often performed by the application, whose partitioning must match hypotheses 1;
- (2) the *factorization of the interiors* and the computation of the local Schur complement $\mathcal{S}_i$ using $\mathcal{A}_i$. This step is performed independently by each MPI process and is common whether or not the coarse space mechanism is applied and is thus not described further;
- (3) the *setup of the preconditioner* by assembling diagonal blocks of $\mathcal{S}_i$ via a few neighbour to neighbour communications and factorization of this one. In the 1-level baseline version of MaPHyS , this step corresponds to Algorithm 2;

**Algorithm 2:** `Baseline 1-level` *`setup of the preconditioner`* `(baseline step (3) of Algorithm 1)`

1 Compute $\bar{\mathcal{S}}_i$ from $\mathcal{S}_i$ by assembling diagonal blocks with neighbour to neighbour communications
2 Compute $\bar{\mathcal{S}}_i^{-1}$ (factorize $\bar{\mathcal{S}}_i$)

- (4) the *solve step*, where (4a) a parallel preconditioned Krylov method is performed on the reduced system (Equation (**??**)) to compute $x_{\Gamma_i}$, followed by (4b) independent back solves on the interiors to compute each $x_{\mathcal{I}_i}$. In the 1-level baseline version of MaPHyS , step (4a) corresponds to Algorithm 3. Step (4b) is common whether or not the coarse space mechanism is applied and is thus not described further.

**Algorithm 3:** `Baseline 1-level preconditioned Krylov solution on the reduced system (baseline step (4a) of Algorithm 1)`

```
for iteration ∈ {1, 2, ...} do
    Apply M_{AS/S} precond
    Apply matrix-vector product
end
```

When the coarse space mechanism is turned on, steps (3) (Algorithm 2) and (4a) (Algorithm 3) are enhanced to compute it and apply it, respectively, as further discussed below.

### 7.2 Improvement achieved

| | |
|---|---|
| Contributors | E. Agullo (Inria), G. Houzeaux (BSC), L. Giraud (Inria), M. Kuhn (Inria), G. Marait (Inria), L. Poirel (Inria). |

We made a few progresses from version 0.9.4.2, on various components of the software package addressing differents aspects:

1. New software deployment service on top of Spack to automatise the installation of the package and its numerous dependencies.
2. Replace the dedicated matrix partition by a more modular and flexible parallel partitioning/data distribution module.
3. Integrate a prototype of the new algebraic coarse space for SPD matrices to control the condition number [1]. A description of this feature is available below.
4. Design a new API to interface Maphys with newly developped block Krylov solvers for multiple right-hand sides [2].
5. Option to keep the same preconditionner when MaPHyS driver is called several time on different matrices. This is especially useful to solve non-linear simulation cases when the matrix changes little between iterations (this feature has been tested in AlyA).

These improvements have been tested and integrated in version 0.9.7 of MaPHyS.

***Coarse space correction mechanism for symmetric positive definite matrices.*** The goal of coarse space correction mechanisms is to improve the preconditioner's numerical quality to reduce the number of iterations by controling the condition number of the preconditioned system. A coarse space correction is defined by its coarse space $V_0$, and the way it is combined with the fist-level preconditioner that it improves. Within a purely algebraic solver, the construction of the solver can only rely on the information provided by the application, which is $\mathcal{A}$ and $b$. However, in MaPHyS , it is possible to provide the matrix $\mathcal{A}$ in a distributed fashion through the local matrices $\mathcal{A}_i$. If these local matrices

are symmetric positive semi-definite (SPSD), we can add a second level of preconditioning such that the condition number and the number of iterations to reach convergence can be bounded, as proved in [2] following a methodology closely related to the GenEO technique introduced in [12].

We have incorporated such a coarse space correction to the baseline (one-level) version of MaPHyS as follows. First, during the *setup of the preconditioner*, a *local coarse space* $V_i^0$ is computed in each domain $\Omega_i$; then, still during the *setup of the preconditioner*, a *coarse matrix* $\mathcal{S}_0$ computed from $\mathcal{S}$ and $V_i^0$ is computed and factorized; finally, each application of the $\mathcal{M}_{AS/S}$ preconditioner is modified to include a *coarse solve*, leading to the *two-level AS* preconditioner for the Schur problem denoted by $\mathcal{M}_{AS/S,2}$. The following subsections detail each of these operations.

***Construction of the local coarse space*** $V_i^0$**.** In each subdomain $\Omega_i$, the following generalized eigenproblem is solved to compute the $n_i$ smallest eigenvalues and its corresponding eigenvectors, thus including the kernel of $\mathcal{S}_i$ if $n_i \geq \text{rank}(\ker(\mathcal{S}_i))$

$$D_i^{-1}\mathcal{S}_i D_i^{-1}\ p_k^i \ = \ \lambda_k^i\ \bar{\mathcal{S}}_i\ p_k^i,$$

where $D_i$ is a *partition of unity*, such that $\sum_{i=1}^N \mathcal{R}_{\Gamma_i}^T D_i \mathcal{R}_{\Gamma_i} = I$, where $I$ the identity matrix. The local coarse space basis can be defined from these eigenvectors; in a matrix form it writes

$$V_i^0 = \left[p_1^i\ p_2^i\ \cdots\ p_{n_i}^i\right],$$

and the global coarse space basis can be formally defined as

$$V_0 = \left[(\mathcal{R}_{\Gamma_1}^T V_0^1)\ (\mathcal{R}_{\Gamma_2}^T V_0^2)\ \cdots\ (\mathcal{R}_{\Gamma_N}^T V_0^N)\right].$$

We notice that $V_0$ is never explicitly formed, and no communication is required for this first step. Solving the eigenproblems may take a lot of time, but it is purely local and consequently fully scalable.

***Computation of the coarse matrix*** $\mathcal{S}_0$**.** The coarse matrix $\mathcal{S}_0$ can be computed in parallel using Equation (13):

$$\mathcal{S}_0 = V_0^T \mathcal{S} V_0 = V_0^T \left(\sum_{i=1}^N \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}\right) V_0 = \sum_{i=1}^N \bar{V_i^0}^T \mathcal{S}_i \bar{V_i^0} = \sum_{i=1}^N \mathcal{S}_0^i,$$
$$\text{where } \bar{V_i^0} = \mathcal{R}_{\Gamma_i} V_0 = \left[(\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_1}^T V_0^1)\ (\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_2}^T V_0^2)\ \cdots\ (\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_N}^T V_0^N)\right].$$

Since $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T$ is zero if $\Gamma_i \cap \Gamma_j = \emptyset$, only neighbor-to-neighbor communications are needed to compute $\bar{V_i^0}$. The details of the calculation and the factorization of $\mathcal{S}_0$ are detailed below in Section 5.

***Application of the two-level preconditioner*** $\mathcal{M}_{AS/S,2}$**.** During each iteration of the Preconditioned Conjugate Gradient, a linear system that involves $\mathcal{S}_0$ needs to be performed. In its additive form, the preconditioner $\mathcal{M}_{AS/S}$ is enriched

$$\mathcal{M}_{AS/S,2} = \mathcal{M}_{AS/S} + \mathcal{M}_0 \tag{16}$$

where

$$\mathcal{M}_0 = V_0\ \mathcal{S}_0^{-1}\ V_0^T. \tag{17}$$

**Algorithm 4:** `2-level` *setup of the preconditioner* `(2-level step (3) of Algorithm 1)`

1 Compute $\bar{\mathcal{S}}_i$ from $\mathcal{S}_i$ by assembling diagonal blocks with neighbour to neighbour communications
2 Compute $\bar{\mathcal{S}}_i^{-1}$ (i.e., factorize $\bar{\mathcal{S}}_i$)
3 **Compute** $\mathcal{M}_0$

**Algorithm 5:** `2-level preconditioned Krylov solution on the reduced system (2-level step (4a) of Algorithm 1)`

1 **for** *iteration* $\in \{1, 2, \dots\}$ **do**
2 | Apply $\mathcal{M}_{AS/S}$ precond
3 | **Apply** $\mathcal{M}_0$ **precond**
4 | Apply matrix-vector product
5 **end**

To implement this second level of the preconditioner, we have to compute $z_0 = \mathcal{M}_0 r$ where the input vector $r$ is distributed according to the row partitioning of $\mathcal{A}$ on the different MPI processes that locally store $r_i = \mathcal{R}_{\Gamma_i} r$. The vector $z_0$ should also be distributed in output consistently with the input vector $r$, that is, each MPI process will have $z_0^i = \mathcal{R}_{\Gamma_i} z_0$. This calculation can be performed in three steps:

$$\begin{aligned} 1 - r_0 &= V_0^T r = \begin{pmatrix} {V_0^1}^T \mathcal{R}_{\Gamma_1} \\ \vdots \\ {V_0^N}^T \mathcal{R}_{\Gamma_N} \end{pmatrix} r = \begin{pmatrix} {V_0^1}^T r_1 \\ \vdots \\ {V_0^N}^T r_N \end{pmatrix}, \\ 2 - z_0 &= \mathcal{S}_0^{-1} r_0, \\ 3 - z_i &= \mathcal{R}_{\Gamma_i} V_0 z_0 = \bar{V}_i^0 z_0. \end{aligned} \tag{18}$$

Computing the products ${V_i^0}^T r_i$ and $\bar{V}_i^0 z_0$ can be done locally and do not present any particular challenge. The other computation steps deserve some attention and various implementation can be though to best exploit the computing resources depending on the problem size. We discuss next several implementations both for the coarse matrix factorization and the coarse solve.

***Coarse space correction parallel design.*** The coarse space of MaPHyS is built in the *setup of the preconditioner* step of the solver. Its application occurs in the *solve step* at each iteration of the PCG algorithm. As stated in Section 7.2, special care has to be taken in order to favor the parallel scalability when implementing the coarse correction mechanism. Four implementation strategies are available to build and apply the coarse space correction through the coarse preconditioner application.

Each of these implementations starts by first computing the local coarse space $V_i^0$ for each subdomain $\Omega_i$ as described in Section 7.2. This step is immediately followed by the calculation of the local coarse matrix $\mathcal{S}_0^i$ for each $\Omega_i$ as explained in Section 7.2. These two steps are performed in parallel across MPI processes and they only require neighbor-to-neighbor communications. They are implemented in the same way for all the implementations as detailed in Algorithm 6. However the factorization of $\mathcal{S}_0$ and the application of the resulting coarse preconditioner $\mathcal{M}_0$ involve different communication schemes depending on

the chosen implementation. These different implementation strategies are detailed in the following subsections and can be shortly introduced as follows:

- **Dense centralized sequential solution (DCS):** the coarse space matrix $\mathcal{S}_0$ is formed as a dense matrix on a single MPI process and is factorized sequentally using a dense factorization kernel. At each iteration when the coarse space component of the preconditioner needs to be computed, the residual is first gathered on the single MPI process, a sequential solve is performed and the solution is scattered back to all the MPI processes. See Section 5 for the algorithm description.
  Such an implementation might be effective for moderate size problems using also a moderate number of MPI processes so that it is not worth exploiting the sparsity of $\mathcal{S}_0$.
- **Sparse globally distributed parallel solution (SGDP):** this implementation allows us to exploit the sparsity of $\mathcal{S}_0$ and use all the processes. This is a similar to the previous one, but here a sparse solution technique is implemented using all the MPI processes to build and factorize $\mathcal{S}_0$. Compared to the previous one $\mathcal{S}_0$ should be large enough and the number of MPI processes moderated to allow for an efficient parallel sparse solution. See Section 5 for the algorithm description.

- **Sparse locally distributed parallel solution (SLDP):** This variant is similar to the previous one. The coarse space matrix is factorized and the solution involving the factors is performed on a subcommunicator of the communicator allocated to MaPHyS . At each iteration, the right-hand side of the coarse problem is computed by all MPI processes but gathered and summed only on the sub-communicator; after the local solve, the coarse solution is scattered back to all MPI processes. See Section 3 for the algorithm description.

- **Redundant sparse locally distributed parallel solution (RSLDP):** This variant is similar to the previous one, but all the entries of $\mathcal{S}_0$ are stored first on different MPI processes that will act as master of several sub-communicators to compute redundantly on each of them the coarse space correction kernels. This variant allows to express more parallelism with a better communication locality when diffusing the solution once the coarse problem is solved. See Section 5 for the algorithm description.

- **Hierarchical sparse distributed parallel solution (HSDP):** In this variant, all the MPI processes from the original MaPHyS communicator are first split into balanced sub-communicators. All the processes of a sub-communicator compute their contribution to $\mathcal{S}_0$ and make this contribution available on their master. All the masters of the sub-communicators are merged into a new communicator that is used to factorize $\mathcal{S}_0$. Every iteration, each MPI process provides its masters with its contribution to the right-hand side of the coarse problem. Then all the masters perform the solution of the coarse problem and scatter back the solution to their local communicator. See Section 6 for the algorithm description.

Before going into the different implementation details, we define here some useful notations for the MPI configurations:

- `Main_comm` is the MaPHyS MPI communicator provided by the user,
- `CSC_comm(g)` is one of the `n` MPI sub-communicator(s) of `Main_comm`,

- `CSC_master(k)` is the master process of `CSC_comm(k)`,
- `CSC_NP` is the number of MPI processes in charge of the factorization and solve of the coarse system,
- `CSC_comm_master` is the MPI communicator with all the `CSC_master(k)` processes, and is of interest only if =n=> 1.
- `CSC_comm_master_master` is the master of the sub-communicator `CSC_comm_master`.
- `i` is the index of the process (Algorithm 7)
- `g` is the index of the group (Algorithm 7)

**Algorithm 6:** Compute $V_0$ and $\mathcal{S}_0$

Solve $D_i^{-1}\mathcal{S}_i D_i^{-1}\ p_k^i = \lambda_k^i\ \bar{\mathcal{S}}_i\ p_k^i$ for the $n_i$ smallest eigenvalues
Compute $V_i^0 = [p_1^i\ p_2^i\ \cdots\ p_{n_i}^i]$
**for** $j \in \{1, \ldots, N\}$ **do**
  **if** $\Gamma_i \cap \Gamma_j \neq \emptyset$ **then**
    Send $\mathcal{R}_{\Gamma_j}\mathcal{R}_{\Gamma_i}^T V_i^0$ to process $j-1$
  **end**
**end**
**for** $j \in \{1, \ldots, N\}$ **do**
  **if** $\Gamma_i \cap \Gamma_j \neq \emptyset$ **then**
    Receive $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T V_0^j$ from process $j-1$
  **else**
    Set $\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_j}^T V_0^j = 0$
  **end**
**end**
Gather $\bar{V_i^0} = [\mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_1}^T V_0^1\ \cdots\ \mathcal{R}_{\Gamma_i}\mathcal{R}_{\Gamma_N}^T V_0^N]$
Compute $\mathcal{S}_0^i = {V_i^0}^T \mathcal{S}_i V_i^0$

**Algorithm 7:** Notations

Compute $i = \text{Comm\_rank}(\texttt{Main\_comm}) + 1$
Compute $g = \text{CSC\_group\_number}(i)$

The general algorithm for the application of the second level preconditioner $\mathcal{M}_0$ is given in algorithm 8.

**Algorithm 8:** $\mathcal{M}_{AS/S,2}$ `application`: general algorithm

Compute $r_0^i = {V_i^0}^T r_i$
Compute $z_i = \bar{\mathcal{S}}_i^{-1} r_i$
**Compute $z_0$ from $z_i$ (different methods implemented)**
Compute $z_0^i = \mathcal{R}_{\Gamma_i} V_0 z_0 = \bar{V_i^0} z_0$
Compute $z_i = z_i + z_0^i$

*Dense centralized sequential solution (DCS).* This implementation consists of using a dense sequential direct solver on a single MPI process (*e.g.* any Lapack implementation) to apply the coarse space correction All the processes compute the contribution

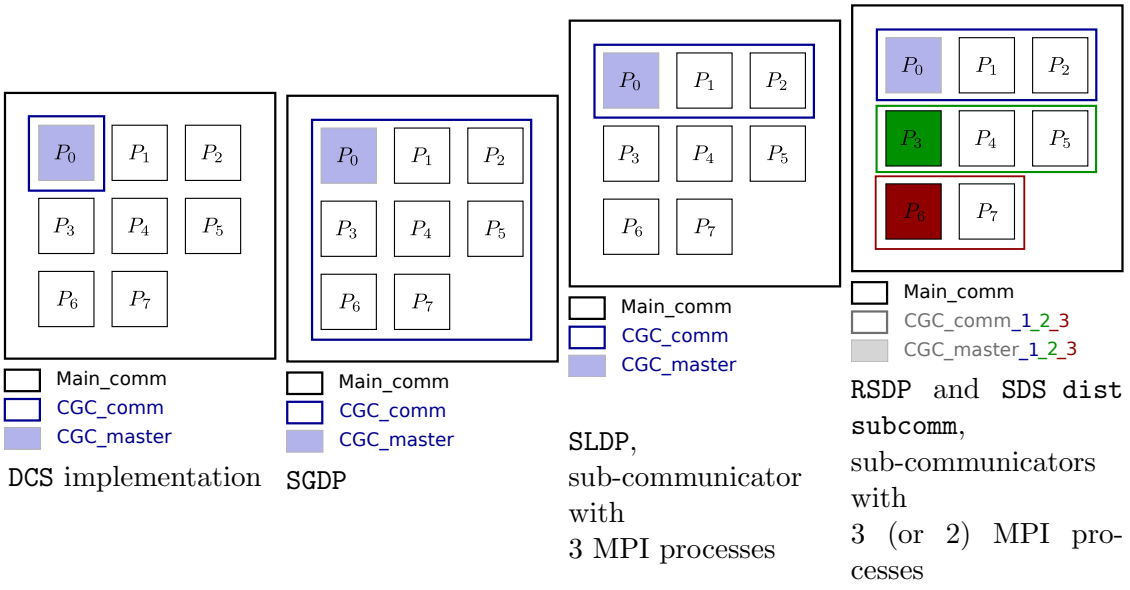



Figure 5: Examples of parallel implementation of the coarse space kernels

of their sub-domain to $\mathcal{S}_0$ and these contributions are then gathered and sum (*i.e.*, assemble) on a single process.

Figure 5 gives an example of MPI configuration for this implementation when using 8 MPI processes in `Main_comm` (in black). `CSC_NP` $= 1$ MPI process is in use to factorize the coarse matrix (in blue). The `CSC_master` MPI process of the `CSC_comm` communicator is colored in blue. Even if notified here for the sake of clarity, notice that `CSC_comm` is not created, and that `CSC_master` is set as the root process of `Main_comm`.

On the factorization of the coarse matrix side, Algorithm 9 shows at line 2 that a first communication is performed in order to centralize and assemble the coarse matrix. Then, the coarse matrix is factorized using a dense kernel on the `CSC_master` process (line 3,4 and 6).

On the preconditioner application side, at each iteration all the MPI processes compute the contribution of their sub-domain to the coarse right-hand side that is then gather and sum on the `CSC_master` MPI process. The solution can then be computed sequentially by this process and then scattered back to all the MPI processes that get their part of $z_0$. This implementation is depicted in Algorithm 10 at line 5,6 and 7.

This implementation strategy is efficient when considering small coarse matrices; consequently few MPI processes. However, as the order of the coarse matrices grows its parallel scalability degrades because of the bottleneck induces by the gather of the right-hand side/scatter of the solution required at each iteration.

**Algorithm 9:** `DCS`: compute $\mathcal{S}_0^{-1}$

1 Gather the $\mathcal{S}_0^i$ and sum them into $\mathcal{S}_0$ on `CSC_master`
2 **if** $i - 1 ==$ `CSC_master` **then**
3 | Factorize $\mathcal{S}_0$ using a dense direct solver on `CSC_master`
4 **end**

**Algorithm 10:** DCS: $\mathcal{M}_{AS/S,2}$ application

1 Gather the $r_0^i$ and sum them into $r_0$ on `CSC_master`
2 **if** $i - 1 ==$ `CSC_master` **then**
3 | Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with a dense direct solver on `CSC_master`
4 **end**
5 Broadcast the solution $z_0$ from `CSC_master` to all MPI processes in `Main_comm`

*Sparse globally distributed parallel solution (SGDP).* This implementation consists of using the a sparse direct solver with a distributed input mode for the matrix (*e.g.* the MUMPS solver) using all the processes of `Main_comm`. Each process computes its contribution $\mathcal{S}_0^i$ to $\mathcal{S}_0$ and calls the parallel sparse direct solver that first assembles $\mathcal{S}_0^i$ and factorizes $\mathcal{S}_0$ using the `Main_comm` communicator. Algorithm 11 shows the factorization call for this implementation strategy which, as can be seen, involves no additional communication on MAPHYS side in its *setup of the preconditioner* step. Indeed, the communications required to factorize the coarse matrix are performed by the sparse direct solver internally.

For the MPI configuration, Figure 5 gives an example when using 8 MPI processes in `Main_comm` (in black), implying the use of the same `CSC_NP` = 8 MPI processes for the SDS solver to factorize the coarse matrix (in blue). The `CSC_master` MPI process of the SDS is colored in blue.

The preconditioner application process is given by Algorithm 12. In this algorithm, we suppose the input mode for the right-hand side of the solver is centralized, so do we for the output of the solution[2], both provided and available on the `CSC_master` MPI process. Line 3 of the algorithm corresponds to the application of the baseline 1-level Additive Schwarz preconditioner. Line 5 corresponds to the 2-level/coarse preconditioner application. In terms of communication, this strategy implies to gather the coarse right-hand side on `CSC_master` MPI process (line 4) and to broadcast the coarse solution (line 6).

**Algorithm 11:** SGDP: compute $\mathcal{S}_0^{-1}$

1 Factorize $\mathcal{S}_0$ with the SDS in distributed input mode on `Main_comm`

**Algorithm 12:** SGDP: $\mathcal{M}_{AS/S,2}$ application

1 Gather $r_0^i$ and assembly into $r_0$ on `CSC_master`
2 Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with the SDS on `Main_comm`
3 Broadcast the solution $z_0$ from `CSC_master` to all MPI processes in `Main_comm`

The main advantage of this strategy is its ease of implementation as no extra MPI communicator and only 2 communications are required in the CSC preconditioner application step.

*Sparse locally distributed parallel solution (SLDP).* This implementation consists in using a sparse direct solver on a sub-communicator of `Main_comm` to perform all the calculation associated with the coarse problem but the initial calculation of the

[2]Notice that the MUMPS solver allows now to input the right-hand side in a distributed manner, which was not the case at the beginning of this study. Using this feature would remove this gather step.

$\mathcal{S}_0^i$ that are still computed by all the MPI processes for their sub-domain. For the sparse solvers we have considered in this study, it means that the coarse matrix $\mathcal{S}_0$ must first be gathered and sum on a single process that is the root of the sub-communicator used for its factorization.

Figure 5 gives an example of MPI configuration for this CSC mode when using 8 MPI processes in `Main_comm`. The CSC communicator responsible of factorizing and of solving the coarse problem is `CSC_comm`. It contains here `CSC_NP` = 3 MPI processes. The master process `CSC_master` is colored in blue.

For the factorization of the coarse matrix, Algorithm 13 shows that the coarse matrix is first centralized and assembled on the `CSC_master` process (line 2). Then, the assembled coarse matrix is factorized in a distributed way on the communicator `CSC_comm` with the SDS.

The application of $\mathcal{M}_0$ requires first the centralization and the assembly of the coarse RHS on the `CSC_master` MPI process (line 4). Then, the coarse problem is solved by the SDS on the `CSC_comm` communicator. After this solve, the coarse solution is broadcasted on the `Main_comm` communicator (line 8).

**Algorithm 13:** `SLDP`: compute $\mathcal{S}_0^{-1}$

```
1 Gather S_0^i and assembly into S_0 on CSC master process
2 if i − 1 ∈ CSC_comm then
3 |   Factorize S_0 a parallel sparse direct solver with centralized input mode on
  |    CSC_comm for S_0.
4 end
```

**Algorithm 14:** `SLDP`: $\mathcal{M}_{AS/S,2}$ application

```
1 Gather r_0^i and assembly into r_0 on CSC_master
2 if i − 1 ∈ CSC_comm then
3 |   Solve z_0 = S_0^{-1} r_0 with parallel sparse direct solver on CSC_comm
4 end
5 Broadcast the centralized solution z_0 from CSC_master to all MPI processes in
   Main_comm
```

In terms of communication scheme, this mode is very close to the DDS mode. On the performance side, using a SDS instead of a DDS becomes more interesting when the size of the coarse system increases, causing scaling issues with the centralized DDS strategy.

***Redundant sparse distributed parallel solution (RSDP).*** This implementation extends the previous implementation strategy by allowing to replicate the coarse problem on disjoint and equally sized sub-communicators `CSC_comm_n` of the `Main_comm` MPI communicator. This strategy allows to replace a global scather/gather at each iteration involving all processes of `Main_comm` by a more local one with each `CSC_comm_n`.

A parallel distributed parallel solver is used on each `CSC_comm_n` sub-communicator to factorize and to solve the coarse system. Figure 5 shows an example of this implementation strategy with 8 MPI processes in `Main_comm`. First, the `Main_comm` is split into sub-communicators with `CSC_NP` = 3 MPI processes, leading to 3 groups of processes:

CSC_comm_1 (in blue), CSC_comm_2 (in green) and CSC_comm_3 (in red). Notice CSC_comm_3 has one less MPI process because 3 does not divide 8. Each of these CSC_comm_n has its own master process CSC_master_n colored in blue, green and red on the figure, which are grouped in the CSC_master_comm communicator.

Algorithms 15 and 16 give respectively the implementation strategy for the factorization and the solve of the coarse system. Compared to the SDS centralized strategy, the major changes in these algorithms reside in the communication schemes for the centralization of the coarse matrix and of the coarse RHS; and for the broadcast of the coarse solution.

The centralization of the coarse matrix occurs here in two steps. The coarse matrix is first partially centralized and assembled into each CSC_comm_n group of MPI processes (see Algorithm 15 line 3). Then, the partially centralized coarse matrices are allgathered and sum on the CSC_comm_master communicator (at lines 4, 5 and 6). After these communications, the entire coarse system is duplicated on each CSC_master_n MPI process. Then, the duplicated coarse system is (redundantly) factorized concurrently on each CSC_comm_n communicator.

The centralization communication scheme of the coarse right-hand side and the solution inside Algorithm 15 is performed in a very similar way to the factorization scheme. Compared to the SLDP strategy, the broadcast of the solution is now performed inside each CSC_comm_n group as the entire coarse solution is duplicated on each of these groups (line 10).

This strategy was designed to enhance the scalability of the preconditioner application on large number of MPI processes. Despite the required allgather when centralizing and assembling the coarse right-hand side, replacing the scatter on Main_comm in the SLDP strategy (Algorithm 14 line 8) by a broadcast on each CSC_comm_n might lead to better parallel performances.

**Algorithm 15:** RSDP: compute $\mathcal{S}_0^{-1}$

1 Gather $\mathcal{S}_0^i$ and assembly into $\mathcal{S}_{00}^g$ on CSC_master_g process
2 **if** $i - 1 ==$ CSC_master_g **then**
3 | Allgather $\mathcal{S}_{00}^g$ and assembly into $\mathcal{S}_0$ on CSC_comm_master comunicator
4 **end**
5 Factorize $\mathcal{S}_0$ with the SDS in centralized input mode on CSC_comm_g

**Algorithm 16:** RSDP: $\mathcal{M}_{AS/S,2}$ application

1 Gather $r_0^i$ and assembly into $r_{00}^g$ on CSC_master_g process
2 **if** $i - 1 ==$ CSC_master_g **then**
3 | Allgather $r_{00}^g$ and assembly into $r_0$ on CSC_comm_master communicator
4 **end**
5 Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with the SDS on CSC_comm_g
6 Broadcast the solution $z_0$ from CSC_master_g to MPI processes in CSC_comm_g

*Hierarchical sparse distributed parallel solution (HSDP).* This last implementation is very similar to the parallel strategy presented in [10]. A parallel sparse direct solver with distributed matrix in input is used in order to factorize and solve the coarse problem on a sub-communicator of Main comm. To introduce the communicators involved

in this strategy, we consider the example given in Figure 5. Similarly to the `RSDP` strategy (see **??**), the `Main comm` with 8 MPI processes is split into sub-communicators.

The main difference here is that `CSC NP` corresponds now to the number of sub-communicators (or, equivalently, to the number of MPI processes in charge of the computation of the factorization and of the solve of the coarse system), that is equal to 3 on this example. Hence, `Main comm` is split into 3 disjoint communicators, namely `CSC_comm_1` (in blue), `CSC_comm_2` (in green) and `CSC_comm_3` (in red). Each of these `CSC_comm_n` has its own master process `CSC_master_n` colored in blue, green and red on the figure, which are grouped in the `CSC_master_comm` communicator. The MPI processes inside `CSC_master_comm` are in charge of the factorization and the solve of the coarse system.

Algorithms 17 and 18 give respectively the implementation strategy for the factorization and the solve of the coarse system. For the coarse factorization, see Algorithm 17, the coarse matrix is first partially gathered and sum on the master process `CSC_master_n` of each `CSC_comm_n` communicator (line 3). Then, the partially centralized coarse matrices are passed to a sparse direct solver with distributed input matrix run on the sub-communicator `CSC_master_comm` communicator (line 4) composed by all the masters `CSC_master_n`.

The solve of the coarse system is given by Algorithm 18. Similarly to the `RSDP` strategy, we suppose that the input mode for the right-hand side of the solver is centralized, so do we for the output of the solution. The centralization communication scheme of the coarse right-hand side and the coarse solution in Algorithm 17 is performed in a very similar way to the factorization scheme (lines 5 to 8). Once the coarse solution is computed, $z_0$ is scattered to all the MPI processes in `Main comm`.

Notice that considering this last strategy with `CSC_comm_n` sub-communicators of size 1 (or equivalently, considering `CSC NP` equal to the size of `Main comm`) is equivalent to employ the `SDS distributed` strategy.

**Algorithm 17:** `HSDP`: compute $\mathcal{S}_0^{-1}$

1 Partially gather $\mathcal{S}_0^i$ and assembly into $\mathcal{S}_0$ on `CSC_master_g` process
2 Factorize $\mathcal{S}_0$ with the SDS in distributed input mode on `CSC_master_comm`

**Algorithm 18:** `HSDP`: $\mathcal{M}_{AS/S,2}$ application

1 Gather $r_0^i$ and assembly into $r_{00}^g$ on `CSC_master_g` process
2 **if** $i - 1 ==$ `CSC_master_g` **then**
3 | Gather $r_{00}^g$ and assembly into $r_0$ on `CSC_comm_master_master` process
4 **end**
5 Solve $z_0 = \mathcal{S}_0^{-1} r_0$ with the SDS on `CSC_master_comm`
6 Broadcast the solution $z_0$ from `CSC_comm_master_master` to all MPI processes in `Main_comm`

***Parallel experiments platform.*** All the parallel experiments presented into this study were performed on the GENCI's OCCIGEN cluster, hosted by the CINES. The part of the cluster in use is composed of 2 Dodeca-core Haswell Intel Xeon E5-2690 v3 @ 2.6 GHz nodes with 64 and 128 Go RAM per node. The code was compiled with Intel compiler version 17.0.0, and linked with the multithreaded Intel MKL version 2017.0.0 and Intel MPI version 2017.0.0. All the runs are made such that the nodes of the cluster are

fully occupied (hence the number of cores is always a multiple of 24). Notice that on the OCCIGEN cluster, memory swapping is disabled by default. The simulation campaigns were realised with the help of JUBE Benchmarking Environment, allowing to explore parameters and analyse results comfortably.

***Alya simulation software.***

***Alya.*** Alya is a simulation software solving different physical problems [13]. The parallelization is hybrid MPI+OpenMP, including loop and task parallelisms at the shared memory level. The physics of concern in this paper are the incompressible Navier-Stokes equations. They are solved implicitly, using an algebraic fractional step based strategy described in [9]. At each time step, the momentum and continuity equations are solved repeatedly until the solution converges to the monolithic solution. On the one hand, an iterative solver for unsymmetric equations is required to solve the momentum equations, while the matrix coming from the continuity equation is SPD. To solve the algebraic system associated to this equation, the Deflated Conjugate Gradient (DCG) [3] with linelet preconditioner is considered [11]. This solver is going to be referred as Alya Internal Solver and is described into the next section.

***Test case presentation.*** The simulation of the airflow through the nose has been chosen to perform an evaluation of different coarse space implementations into MaPHyS . This test case simulates the airflow through the nose and large airways by solving the incompressible Navier-Stokes equations.

Three types of elements are in use for the mesh discretisation: TET04, PYR05 and PEN06, for a total of 17.7M elements and 6.9M nodes. The mesh is characterised by a very elongated geometry with small passages in the nasal cavity, leading to a pseudo-1D elongated domain decomposition when parallelising through partitioning the mesh, see Figure 6. This property makes this test case a very good candidate to evaluate the coarse space of MaPHyS in an applicative context.

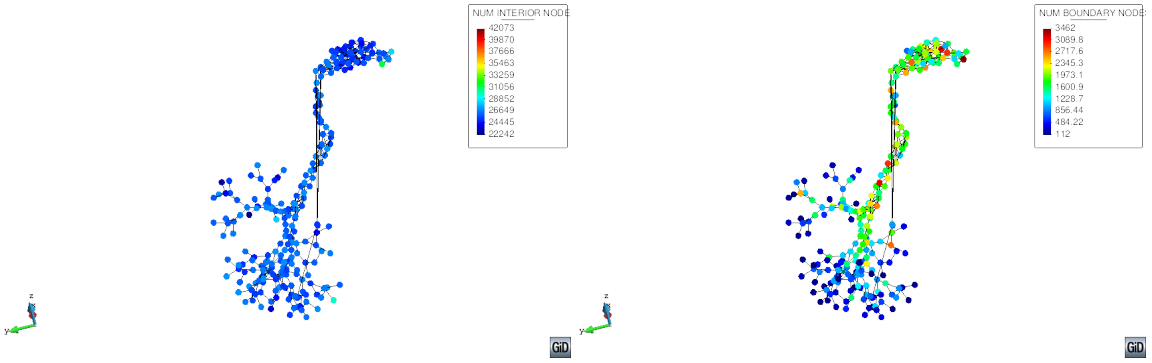
On the algebraic solver side, the discretisation of the problem leads to a coupled algebraic system to be solved at each time step. This algebraic system is split to solve independently the momentum and the continuity equations. Due to the splitting strategy, it is necessary to solve the momentum and the continuity equations twice per time step. As the problem is non-linear, the matrix changes between each time step.

The continuity equation is considered for the solver comparison study. This equation leads to the assembly of a SPD linear system. Due to the elongated geometry, low frequencies are hardly damped with a classical one-level DDM approach. Hence, coarse space or deflation mechanisms are investigated to solve the continuity equation.

For more details about this test case, please refer to [6].

***Performance results of the parallel coarse space implementations of*** MaPHyS **.** The parallel benchmarks have been performed in mono-threaded configuration, on 264, 528, 1056 and 2112 MPI processes, leading respectively to 265, 527, 1055 and 2111 subdomains in the domain decompositions (as Alya has a master process). The iterative solvers' stopping criterion is set to $10^{-6}$, to be reached in a maximum of 2000 iterations. For each experiment, 10 time steps are performed, each time step requiring two substeps.

Results are displayed in the next figures. The MaPHyS solver total time is given in Fig-




Number of interior vertices per domain  Number of interface vertices per domain

Figure 6: Respiratory test case: pseudo-1D domain decomposition into 255 subdomains.

ure 7, the global preconditioner application time for MaPHyS in Figure **??**, the speedups in Figure 9 and the efficiencies in Figure 10 of the MaPHyS solver depending on the implementation strategy. The several two level preconditioning techniques with coarse space correction described into 5 are considered for the iterative solution to the Schur system. The considered number of eigenvalues to build the coarse space for this test case is $n_v = $ 2, 3 and 5. For each coarse space mode, only the number of eigenvalue/eigenvector pairs $n_v$ leading to the lowest total computation time is displayed. For the `SDS centralized`, 12 MPI processes were in use to solve the coarse problem. For the `RSLDP (Redundant sparse locally distributed parallel solution)` mode, the coarse problem has been replicated on disjoint groups of 12 MPI processes. As the matrix changes between each time step, MaPHyS has to perform several times its factorization step in order to factorize the local interior problems and to compute the local Schur complements. The preconditioner (local and coarse) are set up to remain fixed through the time steps. If necessary, it could be set up to be recomputed at a predetermined fixed frequency.

By focusing on the first `SGDP (Sparse globally distributed parallel solution)` implementation of the coarse space, one can observe on Figure 7 (in blue), that MaPHyS coarse space performs poorly. Into this coarse space mode, MaPHyS was not able to scale beyond 528 cores, and did not give a solution for 2112 cores (Out Of Memory (OOM) event on the compute nodes). When having a look at the performances of `SGDP` coarse space mode concerning the global preconditioner application (still in blue), one can identify the required computation time for this part of the iterative process of MaPHyS increases with the number of processes, representing then an increasing ratio of the total computation time. The main reason of these results is that the coarse problem is solved with Mumps sparse direct solver with its distributed entry on too many MPI processes, leading to a too fine granularity hence implying poor performances.

In order to improve performances, two other parallel strategies for the coarse space have been implemented, namely `DCS (Dense centralized sequential solution)` and `SLDP (Sparse locally distributed parallel solution)`. These coarse space modes are displayed in greeny-yellow and in green. These two implementations allow to scale up to 1056 cores, leading to a surlinear speedup on 528 cores. Notice the results for the `SLDP` version become better than the `DCS` version when increasing the number of cores. This is due to the order of the coarse problem that increases with the number of domains in use

which makes it worth to exploit the sparsity pattern of the coarse matrix. However, these strategies do not scale beyond 1056 cores. This is mainly due to the global preconditioner application, whose computation time again increases with the number of processes, representing then an increasing ratio of the total computation time in greeny-yellow and in green.

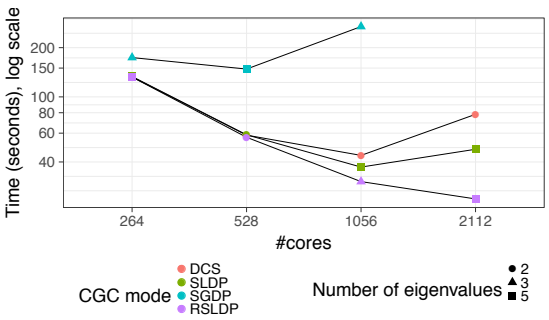



Figure 7: Total time to solve the continuity problem

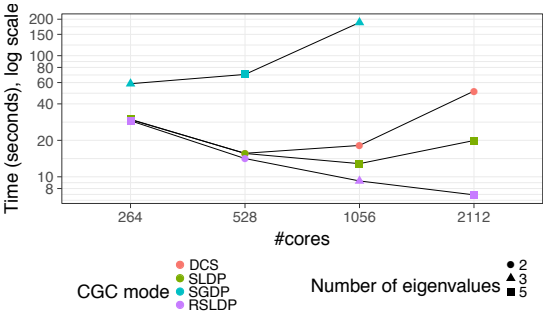



Figure 8: Global preconditioner application time

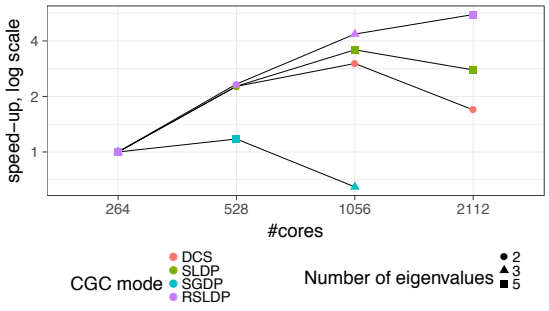



Figure 9: Speedup in solving the continuity equation

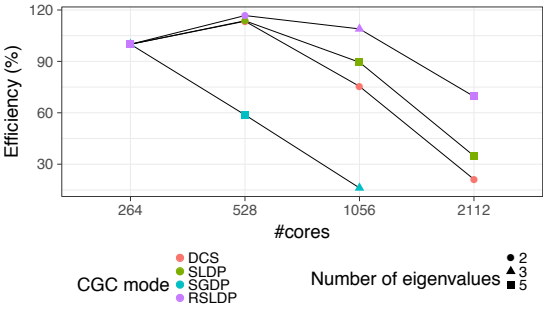



Figure 10: Efficiencies in solving the continuity equation

To go beyond the former limitation, another coarse space management has been implemented: `RSLDP`. This coarse space parallel implementation is closer to Alya's deflation implementation strategy, and allows to save one global MPI communication in the global preconditioner application process of MaPHyS ' iterative solve part as a comparison to the three former parallel algorithms. On Figure **??**, in purple, one can observe this last global communication bypass allows the global preconditioner application to scale up to the 2112 cores in use for these parallel experiments with this implementation strategy, leading to better performances in terms of execution time and of scaling potential.

## References

[1] Emmanuel Agullo, Luc Giraud, and Yan-Fei Jing. Block GMRES method with inexact breakdowns and deflated restarting. *SIAM Journal on Matrix Analysis and Applications*, 35(4):1625–1651, November 2014.

[2] Emmanuel Agullo, Luc Giraud, and Louis Poirel. Robust coarse spaces for Abstract Schwarz preconditioners via generalized eigenproblems. Research Report RR-8978, INRIA Bordeaux, November 2016.

[3] Romain Aubry, Fernando Mut, Rainald Löhner, and Juan R. Cebral. Deflated preconditioned conjugate gradient solvers for the pressure-poisson equation. *Journal of Computational Physics*, 227(24):10196–10208, 2008.

[4] J.-F. Bourgat, R. Glowinski, P. Le Tallec, and M. Vidrascu. Variational formulation and algorithm for trace operator in domain decomposition calculations. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Domain Decomposition Methods*, pages 3–16, Philadelphia, PA, 1989. SIAM.

[5] X.-C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical Linear Algebra with Applications*, 3:221–237, 1996.

[6] Hadrien Calmet, Alberto M. Gambaruto, Alister J. Bates, Mariano Vázquez, Guillaume Houzeaux, and Denis J. Doorly. Large-scale cfd simulations of the transitional and turbulent regime for the large human airways during rapid inhalation. *Computers in Biology and Medicine*, 69(nil):166–180, 2016.

[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] Y.-H. De Roeck and P. Le Tallec. Analysis and test of a local domain decomposition preconditioner. In Roland Glowinski, Yuri Kuznetsov, Gérard Meurant, Jacques Périaux, and Olof Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 112–128. SIAM, Philadelphia, PA, 1991.

[9] G. Houzeaux, R. Aubry, and M. Vázquez. Extension of fractional step techniques for incompressible flows: The preconditioned orthomin(1) for the pressure schur complement. *Computers & Fluids*, 44(1):297–313, 2011.

[10] 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.

[11] Orlando Soto, Rainald Löhner, and Fernando Camelli. A linelet preconditioner for incompressible flow solvers. *International Journal of Numerical Methods for Heat & Fluid Flow*, 13(1):133–147, 2003.

[12] 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.

[13] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira, Hadrien Calmet, Fernando Cucchietti, Herbert Owen, Ahmed Taha, Evan Dering Burness, José María Cela, and Mateo Valero. Alya: Multiphysics engineering simulation toward exascale. *Journal of Computational Science*, 14(nil):15–27, 2016.

# 8. MUMPS

## 8.1 Package ID card

| | |
|---|---|
| Package name | MUMPS |
| Functionalities offered | Parallel sparse direct solver |
| Description | MUMPS ("MUltifrontal Massively Parallel Solver") is a package for solving systems of linear equations of the form Ax = b, where A is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed memory computers. It was developped inside a consortium started around CERFACS, INPT, inria, ENS-Lyon and Bordeaux-Univeristy. |
| Number of users | 1-10 |
| Library dependencies | MPI, BLAS, LAPACK, ScaLAPACK |
| Package references | http://mumps.enseeiht.fr/ |
| Contact | • Fahreddin Sukru Torun (ftorun@enseeiht.fr)<br>• Philippe Leleux (leleux@cerfacs.fr)<br>• Mumps developers support (mumps-dev@listes.ens-lyon.fr) |

## 8.2 Improvement achieved

MUMPS was used for Linear Algebra support of the applications Alya, ParFLOW, SHEMAT-Suite and TOKAM3X. In this section, we present an overview of the solver as well as its latest feature: Block Low Rank approximation which we used extensively for support.

### Overview

MUMPS (MUltifrontal Massively Parallel direct Solver) is a package for solving systems of linear equations of the form Ax = b, where A is a sparse matrix. The solver has an Hybrid MPI/OpenMP model based on distributed dynamic scheduling, see [1] and [2] for more details.

MUMPS follows a multifrontal scheme, which is a direct method, composed of 3 steps:

- Analysis: preprocessing of the matrix (ordering, scaling, partitioning,...) and symbolic Factorisation. From the adjacency graph, this step allows the construction of an "elimination tree", decomposing the global system in smaller interconnected parts (fronts) for the factorisation. There exist 2 versions of this phase: one sequential and one parallel, we opted for the sequential option.
- Factorisation of the input matrix: this step makes use of 2 levels of parallelism, one introduced by the tree structure and the second is at node level where large fronts are solved by several processes.

- Solve: Forward/Backward substitution.

### Block Low Rank Approximation

*"Frontal matrices are not low-rank but in some applications they exhibit low-rank blocks. A block in the matrix represents the interaction between 2 subdomains. If they have a small diameter and are far away, their interaction is weak: the rank is low."*
The goal is to approximate blocks far from the diagonal with low rank products so that we do not lose much information. This is done on blocks distant enough via a truncated Pivoted QR decomposition with a threshold (BLR epsilon), see [3] for more details.

When increasing Block Low Rank threshold parameter, more blocks are approximated and:

- Factorisation timing decreases with corresponding operations (Flops),
- Accuracy of the solution matches the threshold used (Scaled Residual).

MUMPS group has worked on exploiting BLR compression to also reduce the memory usage, Preliminary results are available in the Phd Thesis of Theo Mary[3], Section 9.3. This feature should be available early 2018 before a consortium release mid 2018.

## References

[1] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Patrick R Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing*, 32(2):136–156, 2006.

[3] Théo Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability.* PhD thesis, UT3, 2017.

# 9. PSBLAS and MLD2P4

## 9.1 Package ID card

| | |
|---|---|
| Package name | PSBLAS |
| Functionalities offered | Parallel sparse linear algebra basic operators and iterative Krylov solvers |
| Description | PSBLAS (Parallel Sparse BLAS) is a library of Basic Linear Algebra Subroutines designed to handle the parallel implementation of iterative solvers for sparse linear systems. It includes functionalities for creating sparse matrices and handling their distribution and I/O, handling vectors associated with matrices, performing basic sparse matrix operations, and solving linear systems with a set of Krylov subspace methods. It is written in Fortran 2003, using MPI, and supports distributed sparse matrices in CSR, CSC, COO. Extensions for ELLPACK, JAD and GPU-enabled formats are also available. A plugin has been added to the library for efficient implementation of sparse matrix operations on GPUs. |
| Languages | Fortran 2003, interfaces to C and Octave in progress |
| Library dependencies | BLAS, MPI |
| Programing models | MPI, plugin for GPU available |
| Platforms | In the EoCoE project:<br>• CRESCO cluster (ENEA)<br>• IBM MareNostrum 4 (Barcelona Supercomputing Center)<br>• Yoda Cluster (ICAR-CNR) |
| Code distribution | Available from `https://github.com/sfilippone/psblas3` under a modified BSD licence. |
| Package references | [1] S. Filippone, M. Colajanni, PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices, ACM Trans. Math. Softw., 26, 2000, 527–550.<br>[2] S. Filippone, A. Buttari, Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003, ACM Trans. on Math Software, 38, 2012, Art. No. 23.<br>[3] V. Cardellini, S. Filippone, D. Rouson, Design Patterns for sparse-matrix computations on hybrid CPU/GPU platforms, Scientific Programming, 22, 2014, 1–19. |
| Contact | Salvatore Filippone (salvatore.filippone@cranfield.ac.uk) |

| | |
|---|---|
| Package name | MLD2P4 |
| Functionalities offered | Parallel Algebraic MultiGrid and Domain Decomposition preconditioners |
| Description | MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS) is a package of parallel Algebraic MultiGrid (AMG) and Domain Decomposition (multilevel additive and hybrid Schwarz) preconditioners. A decoupled version of the smoothed aggregation algorithm is applied to generate coarse-level corrections. MLD2P4 has been designed to provide scalable and easy-to-use preconditioners in the context of the PSBLAS computational framework and is used in conjuction with the PSBLAS Krylov solvers. MLD2P4 employs object-oriented design techniques in Fortran 2003, with interfaces to third party libraries such as MUMPS, UMFPACK, SuperLU, and SuperLU Dist, which can be exploited in building and applying AMG preconditioners. |
| Languages | Fortran 2003 |
| Library dependencies | BLAS, MPI, PSBLAS, UMFPACK (optional), MUMPS (optional), SuperLU (optional), SuperLu_Dist (optional) |
| Programing models | MPI; GPU through PBLAS plugin |
| Platforms | In the EoCoE project:<br>• CRESCO cluster (ENEA)<br>• IBM MareNostrum 4 (Barcelona Supercomputing Center)<br>• Yoda Cluster (ICAR-CNR) |
| Code distribution | Available from `https://github.com/sfilippone/mld2p4-2` under a modified BSD licence. |
| Package references | [1] P. D'Ambra, D. di Serafino, S. Filippone, MLD2P4: a Package of Parallel Algebraic Multilevel Domain Decomposition Preconditioners in Fortran 95, ACM Trans. Math. Softw., 37, 2010, Art. No. 30.<br>[2] P. D'Ambra, D. di Serafino, S. Filippone, MLD2P4 v. 2.1 User's and Reference Guide, July 31, 2017. Available from `https://github.com/sfilippone/mld2p4-2/tree/development/docs`. |
| Contact | • Salvatore Filippone (salvatore.filippone@cranfield.ac.uk)<br>• Pasqua D'Ambra (pasqua.dambra@cnr.it)<br>• Daniela di Serafino (daniela.diserafino@unicampania.it) |

### 9.2 Improvement achieved

| | |
|---|---|
| Contributors | Pasqua D'Ambra (National Research Council of Italy - CNR, Naples, Italy), Daniela di Serafino (University of Campania "L. Vanvitelli", Caserta, Italy), Salvatore Filippone (Cranfield University, Cranfield, UK), Ambra Abdullahi Hassan (University of Rome "Tor Vergata", Rome, Italy |

For each package, we first provide a short description of its status at the beginning of the EoCoE project and then outline the main improvements achieved. Results concerning the application of the current versions of MLD2P4 and PSBLAS to data sets from two different pillars of the EoCoE Project (Water for Energy - WP4, Meteorology for Energy - WP2) are described in Deliverable D1.3 (Application support outcome).

### PSBLAS: starting point

The Parallel Sparse Basic Linear Algebra Subroutines (PSBLAS) library was designed to provide the operators needed to build iterative methods for the solution of sparse linear systems on distributed memory parallel computers. Its development was started taking into account the discussions on the standardization of sparse matrix computations in the context of the BLAS Technical Forum [9]. The library revolves around a set of Krylov subspace solvers for both symmetric positive definite (spd) and general matrices, e.g, Conjugate Gradients (CG), GMRES and BiCGSTAB, and a set of simple preconditioners including ILU(0).

The library contains a significant amount of infrastructure code to handle data storage and distribution of sparse matrices. Matrices are distributed in general row-block fashion, consistent with common usage of graph partitioning heuristics embodied in libraries such as Metis and SCOTCH; the data distribution can be specified in multiple ways. The necessary data exchange patterns and the global-to-local index remapping are automatically extracted from the matrix data: the halo data exchange, a typical step in mesh based computations, is provided as a communication primitive, and it is built to work for arbitrary distributed mesh graphs.

The parallel implementation is based on a Single Program Multiple Data (SPMD) paradigm and internally uses MPI, but provides wrappers for most common operations: user code rarely needs to invoke MPI directly. Similarly, the internal matrix storage formats are handled automatically by the library, including support for common formats such as CSR and COO, while at the same time providing tools to easily extend the set of supported formats. A set of plugins provides support for additional data storage formats such as ELLPACK and JAD, including storage formats that interface computations on NVIDIA GPUs [4, 11]. The design of the library is object-oriented, and implemented in Fortran 2003 [4, 10].

### PSBLAS: improvement

The functionalities of PSBLAS have been extended during the EoCoE project, implementing a flexible version of the CG method (FCG) [12], and a variant of the Generalized Conjugate Residual method (GCR) [8, 13]. The former is equivalent to the standard Conjugate Gradient method when constant spd preconditioners are applied, and enhance the stability of the method when a variable preconditioner, such as the K-cycle available in MLD2P4 (see section 9.2), is employed. The latter applies to general linear systems and can be effectively used with variable preconditioning too.

We also included improvements needed when interfacing the GPU plugin [2, 4] with the MLD2P4 library described in the next section.

C and Octave interfaces to PSBLAS are under development and will be integrated in

future versions of the library.

The current stable version of PSBLAS (v. 3.5.2) is available from `https://github.com/sfilippone/psblas3`.

In the last period of the project we have started work on improving the handling of problems with very large index spaces, requiring 8-byte integers; in particular we did a complete overhaul of the configuration options and of the coupling between local and global indices. The initial design has already been tested with runs for linear systems of a global size in excess of $4 \times 10^9$ generated with a mini-app extracted from the ParFlow code, running on multi-GPU platforms; we are currently evaluating two design alternatives for some details of the implementation of the outermost matrix objects. In addition, we have improved the implementation of some methods used in MLD2P4 to set up the preconditioners. The corresponding development version of the software is available from the GitHub site, but is not expected to be merged into a stable release before the end of the project.

### MLD2P4: starting point

MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS was designed to provide scalable and easy-to-use algebraic multilevel domain decomposition preconditioners in the context of the PSBLAS (Parallel Sparse Basic Linear Algebra Subprograms) computational framework, for use with the Krylov solvers available from PSBLAS.

The release of MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS) available at the beginning of the EoCoE project provided multilevel additive and hybrid Schwarz preconditioners, as well as one-level additive Schwarz preconditioners [5]. A purely algebraic approach, based on the *smoothed aggregation* algorithm [3, 16], was implemented to generate coarse-level corrections, so that no geometric background was needed about the matrix to be preconditioned. A decoupled version of this algorithm was considered, where the smoothed aggregation is applied locally to each submatrix [15].

The package employs object-oriented design techniques in Fortran 2003, with interfaces to additional third party libraries such as MUMPS, UMFPACK, SuperLU, and SuperLU_Dist, which can be exploited in building multi-level preconditioners. The parallel implementation is based on a SPMD paradigm; the inter-process data communication is based on MPI and is managed through PSBLAS primitives.

Several extensions and improvements have been introduced in MLD2P4 as a part of the EoCoE project.

### MLD2P4: improvement

Several extensions and improvements have been introduced in MLD2P4 as a part of the EoCoE project, as specified next.

The package functionalities have been extended including multilevel cycles and smoothers widely used in multigrid methods. The classical V-cycle and W-cycle have been included in

MLD2P4; furthermore, a K-cycle for both spd and general matrices has been implemented, where the coarse systems are solved by FCG(1) or GCR iterations at each level but the coarsest one [14, 13], in order to improve convergence when using unsmoothed constant piecewise prolongators. To enhance implementation scalability on linear systems coming from elliptic PDEs on regular grids, classical parallel pointwise smoothers have been added to the original additive Schwarz ones.

The user interface has been modified, in order to separate the construction of the multilevel hierarchy from the construction of the smoothers and solvers, and to allow for more flexibility at each level.

The software architecture has significantly evolved, in order to fully exploit the Fortran 2003 features implemented in PSBLAS 3.

Internal changes have been applied to MLD2P4 to guarantee optimal use of the GPU plugin available from PSBLAS.

MLD2P4 has also been interfaced with the compatible weighted matching aggregation algorithm implemented in the BootCMatch (Bootstrap AMG based on Compatible Weighted Matching) sequential code [7], obtaining a parallel decoupled version of this aggregation algorithm to be used within MLD2P4 for improving robustness and efficiency on sparse systems coming from anisotropic PDE problems on general grids [1]. Actually, this last issue is part of longer-term applied research work carried out within Task 2 of Workpackage 1. This work concerns the investigation of coarsening algorithms based on graph matching approaches in the AMG framework, and is motivated by the observation that the AMG preconditioners implemented in MLD2P4 may lose their robustness and parallel efficiency when applied to systems arising from highly anisotropic problems from EoCoE. A detailed description of this activity is provided in Deliverable D1.11.

The current stable version of MLD2P4 (v. 2.1.1) is available from `https://github.com/sfilippone/mld2p4-2`. (see [6] for a description of its functionalities). It includes all the previous improvements, except for the interface with BootCMatch.

In the last period of the project we have worked on the interface of the latest development version of PSBLAS with the new handling of 8-bytes integers for large index spaces; the initial design has already been tested with runs for linear systems of a global size in excess of $4 \times 10^9$ generated with a mini-app extracted from the ParFlow code, on multi-GPU platforms. Work is currently under way to improve interfacing of aggregation algorithms, increasing the ease of experimentation, including a full integration of the BootCMatch software. The corresponding development version of the software is available from the GitHub site, but is not expected to be merged into a stable release before the end of the project.

## References

[1] A. Abdullahi Hassan, P. D'Ambra, D. di Serafino, S. Filippone, *Parallel Aggregation Based on Compatible Weighted Matching for AMG*, in "Large-Scale Scientific Computing", I. Lirkov and S. Margenov eds., Lecture Notes in Computer Science, vol. 10665, Springer, 2018, pp. 563-571.

[2] D. Bertaccini and S. Filippone, *Sparse approximate inverse preconditioners on high performance GPU platforms*, Comput. Math. Appl., 71, 2016, 693–711.

[3] M. Brezina, P. Vaněk, *A Black-Box Iterative Solver Based on a Two-Level Schwarz Method*, Computing, 63, 1999, 233–263.

[4] V. Cardellini, S. Filippone, D. Rouson, *Design Patterns for sparse-matrix computations on hybrid CPU/GPU platforms*, Scientific Programming, 22, 2014, 1–19.

[5] P. D'Ambra, D. di Serafino, S. Filippone, *MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95*, ACM Trans. Math. Softw., 37, 2010, Art. No. 30.

[6] P. D'Ambra, D. di Serafino, S. Filippone, *MLD2P4 v. 2.1 User's and Reference Guide*, July 31, 2017.

[7] P. D'Ambra, S. Filippone, P. S. Vassilevski, *BootCMatch: a Software Package for Bootstrap AMG based on Graph Weighted Matching*, ACM Trans. on Math Software, 44, 2018, Art. No. 39.

[8] S. C. Eisenstat, H. C. Elman, M. H. Schultz, *Variational iterative methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20, 1983, 345-–357.

[9] S. Filippone, M. Colajanni, *PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices*, ACM Trans. Math. Softw., 26, 2000, 527–550.

[10] S. Filippone, A. Buttari, *Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003*, ACM Trans. on Math Software, 38, 2012, Art. No. 23.

[11] S. Filippone, V. Cardellini, D. Barbieri, A. Fanfarillo, *Sparse Matrix-Vector Multiplication on GPGPUs*, ACM Trans. Math. Softw., 43, 2016, Art. No. 30.

[12] Y. Notay, *Flexible conjugate gradients*, SIAM J. Sci. Comput., 22, 2000, 1444–1460.

[13] Y. Notay, *An Aggregation-based Algebraic Multigrid Method*, Electron. Trans. Numer. Anal., 37, 2010, 123–146.

[14] Y. Notay, P. S. Vassilevski, *Recursive Krylov-based multigrid cycles*, Numer. Lin. Alg. Appl., 15, 2008, 473–487.

[15] R. S. Tuminaro, C. Tong, *Parallel Smoothed Aggregation Multigrid: Aggregation Strategies on Massively Parallel Machines*, in Proceedings of SuperComputing 2000 ( J. Donnelley, Ed.), Dallas, 2000.

[16] P. Vaněk, J. Mandel and M. Brezina, *Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems*, Computing, 56, 1996, 179–196.