Horizon 2020
European Union funding
for Research & Innovation

E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence :

toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D3.1

Co-design of LA solvers, Specification of characteristics and interfaces of the LA solvers for all target applications

Project and Deliverable Information Sheet

EoCoE-II	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence: towards exascale for energy
	Project Web Site:	http://www.eocoe2.eu
	Deliverable ID:	D3.1
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M8 31/08/2019
	Actual Date of Delivery:	M12 16/12/2019
	EC Project Officer:	Andréa Feltrin

* - 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 :	Co-design of LA solvers, Specification of characteristics and interfaces of the LA solvers for all target applications
	ID :	D3.1
	Available at:	http://www.eocoe2.eu
	Software tool:	LaTeX
Authorship	Written by:	Yvan Notay, Pasqua D'Ambra, Martin J. Kühn, Patrick Tamain
	Contributors:	Fabio Durastante, Salvatore Filippone, Herbert Owen
	Reviewed by:	Mathieu Lobet, Edoardo Di Napoli

Document Keywords: Linear algebra solvers, linear systems

Contents

List of Figures

List of Tables

1 Introduction

Solving Linear Algebra (LA) problems is a core task in four out of five EoCoE II Scientific Challenges (SC) and thus the availability of exascale-enabled LA solvers is fundamental in preparing the SC applications for the new exascale ecosystem. More specifically, "LA problem" refers here to the solution of systems of algebraic linear equations, with numbers of unknowns and equations that are increasingly larger as one is going towards exascale.

The goal of WP3 is to design and implement exascale-enabled LA solvers for the selected applications and to integrate them into the application codes.

This integration can take two different forms. Firstly, a specific solver can be designed to carry over the needs of a specific flagship code. This is mostly needed, e.g., when the flagship code operates in matrix-free mode, because general-purpose LA solvers require the system matrix as input data and can therefore not be used in such a context. Then, the solver is deeply integrated into the flagship code, which requires co-design decisions by solver developers and flagship codes developers.

Secondly, flagship codes may be interfaced with general-purpose solvers that have been developed as independent modules (in general, they are available as software libraries). Although the situation is here somewhat more simple, properly interfacing is often not a trivial task.

The goal of the present deliverable is to outline the co-design decisions and to report about the interfacing with LA libraries. These represent a preliminary step before getting the first results that will be presented in Deliverable D3.2. Because of this specialized topic, the present document has not to be confounded with a progress report on WP3 activities. On the one hand, some task have advanced significantly beyond the interfacing, but results which are sometimes mentioned, will be reported in detail only in Deliverable D3.2. On the other hand, some other tasks could only start with some delay, mainly because of personal issues [1]. This caused the delay of the whole deliverable, which was supposed to be complete with respect to co-design decisions and interfacing. Nevertheless, for reasons explained below (see Sections 3 and 4.2), two interfaces are still missing. The follow up of these tasks will be reported in subsequent deliverables. Apart from these two cases, the specific sub-tasks corresponding to co-design and interfacing may be considered as finished.

Finally, non-expert readers may find the present document difficult to read. This is due to the specific nature of Deliverable D3.1, which makes it a rather technical document likely less accessible than deliverables presenting syntheses of results to a wide audience.

[1] Most of the pertners involved in WP3 have significantly less than 36 PMs for this workpackage, and therefore could not start activities at month M1

2 Acronyms

Table 1: Acronyms for the partners and institutes therein.

Acronym	Partner and institute
BSC:	Barcelona Supercomputing Center
CEA:	Commissariat à l'énergie atomique et aux énergies alternatives
CNR:	Consiglio Nazionale Delle Ricerche
CNRS:	Centre Nationale de la Recherche Scientifique
CERFACS:	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
FZJ:	Forschungszentrum Jülich GmbH
INRIA:	Institut national de recherche en informatique et en automatique
IRIT:	Institut de Recherche en Informatique de Toulouse
IRFM:	Institute for Magnetic Fusion Research
MPG:	Max-Planck- Gesellschaft zur Förderung der Wissenschaften e.V
RWTH:	Rheinisch-Westfälische Technische Hochschule Aachen, Aachen University
ULB:	Université Libre de Bruxelles
UNITOV:	Università di Roma Tor Vergata

Table 2: Acronyms of software packages.

Acronym	Software and codes
	Application codes
Alya:	High Performance Computational Mechanics
libNEGF:	General library for Non Equilibrium Green's Functions
GyselaX:	GYrokinetic SEmi-LAgrangian
ParFlow:	PARallel Flow
SHEMAT:	Simulator of HEat and MAss Transport
TOKAM3X:	Transport and turbulence in the edge plasma of tokamaks
	LA software libraries
AGMG:	Iterative solution with AGgregation-based algebraic MultiGrid [9]
Fabulous:	Fast Accurate Block Linear krylOv Solver [3]
MaPHyS:	Massively Parallel Hybrid Solver [7]
MLD2P4:	MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS [2]
MUMPS:	MUltifrontal Massively Parallel sparse direct Solver [8]
PSBLAS:	Parallel Sparse Basic Linear Algebra Subroutines [4]
PaStiX:	Parallel Sparse matriX package [10]

3 Task 3.1: Linear Algebra solvers for Materials

Partners: INRIA
Software packages: libNEGF

It was planned to integrate the dense linear algebra library Chameleon developed at INRIA in the flagship PVnegf code from the Materials for Energy task T3.1. For reasons explained elsewhere, the PVnegf flagship has been replaced by the libNEGF code which has currently been benchmarked to identify shortcomings in terms of linear algebra. The results of the benchmarks will be available and analyzed early 2020. Consequently the linear algebra activity has been postponed and will probably be redefined accordingly.

4 Task 3.2: Linear Algebra solvers for Water

4.1 Interfacing ParFlow with PSBLAS/MLD2P4 Sparse Linear Algebra Libraries

Partners: CNR, FZJ, UNITOV
Software packages: ParFlow, MLD2P4, PSBLAS

The Parflow library contains different solver modules each representing a particular physical model, i.e., a set of Partial Differential Equations (PDEs), for either steady-state, groundwater flows (IMPES module) or variably saturated flows (RICHARDS module). The governing equations of the physical models are discretized in space by using Finite Volumes (FV) methods. All the functionalities to assemble the global matrix and right-hand-side (RHS) of the corresponding set of equations, including boundary conditions and material properties, are currently interfaced, together with all the functionalities needed to solve the nonlinear systems of equation associated to the discretization, to the KINSOL library. Therefore, we have started working on both data structures and solution algorithms to interface the PSBLAS/MLD2P4 suite of code with the KINSOL library so that the new functionalities can be called directly from Parflow by setting the corresponding KINSOL options.

The most critical point with this respect is the handling of the data structures for sparse matrices and dense vectors (RHS). At the present time, the KINSOL library can handle only parallel vectors and has no internal support for distributed sparse matrices. In this regard, we are currently implementing the three modules for handling parallel vectors, parallel matrices, and the iterative solution in a distributed setting of the linear systems; respectively, in the KINSOL nomenclature, new NVECTOR, NMATRIX, and SUNLINSOL modules.

The NVECTOR and NMATRIX modules require a predetermined set of linear algebra operations, some pertaining to the standard BLAS context, e.g., dot–products, vector norms, *etc.*, and some others relative to how the Newton solver is implemented, e.g., computing the minimum of the quotient obtained by term–wise division of two input vectors. While the first class of operations is readily available in the PSBLAS library, the second one needs to be directly implemented in it thus prompting some co-design aspects.

The integration of the parallel iterative solvers for solving algebraic linear systems arising from the computation of Newton steps contained in PSBLAS (rel. 3.6) and MLD2P4 (rel. 2.0), i.e., the new SUNLINSOL module, will permit to increase the number of solvers and preconditioners that are currently available in KINSOL, and thus in Parflow.

4.2 Interfacing the AGMG algebraic multigrid solver with PETSc

Partners: RWTH, ULB
Software packages: SHEMAT, AGMG

A feasibility study has been undertaken, yielding a positive answer. The concrete writing of the interface has however been delayed for two reasons. Firstly, because it will be written on the top of the standardized interface described in Section 7 below; the specifications of this standard have just been released, and additional time is needed to implement it. Secondly, some staff changes in RWTH delayed the support of this partner for this specific task.

5 Task 3.3: Linear Algebra solvers for Fusion

5.1 Co-Design of a Geometric Multigrid Solver for Plasma Fusion Simulations in GyselaX

Partners: CEA-IRFM, CERFACS, CNRS-IRIT, MPG
Software packages: GyselaX

The GyselaX code, mainly developed by CEA-IRFM, needs among other things a fast solver for an elliptic PDE defined on a stretched polar grid refined by patches on a collection of 2D slices of the Tokamak geometry. In order to satisfy particular requests from the application side and to design an optimally tailored PDE solver, several meetings between the MPG and CERFACS were already held between November 2018 and September 2019. Further meetings between CEA-IRFM and CERFACS took place in September 2019. An application-specific co-design is applied and we expect performance benefits of more than an order of magnitude as compared to a generic linear algebra library.

The project is currently at the stage of analysing the behavior of specific geometric multigrid solvers for the physical geometry and its representation in terms of polar coordinates and similar transformations.

A pre-solver routine is designated to obtain geometry information such as local refinements towards the edge or parameters to define more realistic, deformed geometries. These inputs are based on recent advantages in modeling realistic geometry in Tokamak simulations ; see, e.g., [11] or [1]. When this parameters are set to zero, the geometry reduces to standard polar coordinates.

Another routine is designated to obtain as input the corresponding right hand side for each slice to be solved by the geometric multigrid solver. This input is given from the pre-executed step of the Vlasov Boltzmann solver with a subsequent integration step as done in Gysela ; see [5].

Based on the previously described inputs, suited solver parameters are chosen and one solution per slice is returned. Prototype software has already been developed as a tool to study the mathematical and algorithmic foundations and how to solve the PDE with best possible efficiency. For further accelerating the PDE solver in order to save a substantial number of core-hours, the multigrid algorithm will be combined with extrapolation techniques to raise its order of convergence. In order to satisfy the requirement of a large number of degrees per freedom per computing node, matrix free methods are targeted.

5.2 Interfacing TOKAM3X with AGMG and PaStiX

Partners: CEA-IRFM, INRIA, ULB
Software packages: TOKAM3X, AGMG, Fabulous, PaStiX

The code TOKAM3X is a 3D fluid code solving mass, momentum, energy and charge balance equations to describe the dynamic of the edge plasma for tokamak applications. It is based on finite volume methods applied on a structured mesh where a domain decomposition is used to account for the topology of the magnetic field. The charge balance equation can be rewritten as a vorticity equation where the main operator takes the form of a 3D Laplacian. The magnetic field introduces a strong anisotropy and the diffusion in the direction parallel to the magnetic field can be more than times higher than the diffusion orthogonal to the magnetic field. This strong anisotropy implies a poor conditioning of the anisotropic 3D Laplacian operator. Moreover, the boundary conditions contribute also to the poor conditioning of the problem: they take the form of Robin boundary conditions (namely $a\phi + b\nabla\phi = c$) but the weight of the "Dirichlet part" (a) is small compared to the weight of the "Neumann part" (b) Boundary conditions are thus "nearly Neumann" and a Laplacian with Neumann boundary conditions is not invertible.

The inversion of the anisotropic Laplacian in TOKAM3X used to rely on direct solvers such as PASTIX. The drawback of direct solver is the numerical and memory cost when the size of the problem increases (scaling nearly as if is the size of the problem). Iterative solvers have the potential to scale better. However, their robustness and convergence is strongly dependent on the use of an effective preconditioner, especially for badly conditioned matrices such as the one found in TOKAM3X. In EoCoE, preliminary off-line tests made with the algebraic multigrid solver AGMG had demonstrated an interesting potential but the coupling of the solver with the code for production runs was hindered by the fact that the matrix produced by TOKAM3X required some heavy pre-treatments for it to be compatible with the algorithm used in AGMG.

Thanks to a good cooperation between ULB and CEA Cadarache, the discretization of the 3D anisotropic diffusion equations has been rewritten in order to facilitate the use of iterative solvers. In particular, the left scaling of the equations has been changed to reveal the natural symmetry of the system matrix, while boundary conditions have been imposed directly instead of via Lagrange multipliers. As a consequence, AGMG is more efficient and does not need anymore the specific preprocessing developed within the framework of EoCoE-I.

TOKAM3X is now interfaced with both AGMG and PaStiX and work is ongoing with both solvers. First 3D simulations have been done in circular geometry showing very encouraging results in terms of computation time. In particular, a simulation could be run with the largest resolution ever used in TOKAM3X, resolution which could not be reached with direct solvers due to memory limits (Figure 1). Results in more complex geometries are more contrasted with the aggregation step of AGMG failing at reducing significantly the size of the system. Further investigations are necessary to understand this behavior and possibly guide the solver towards a more effective aggregation strategy.

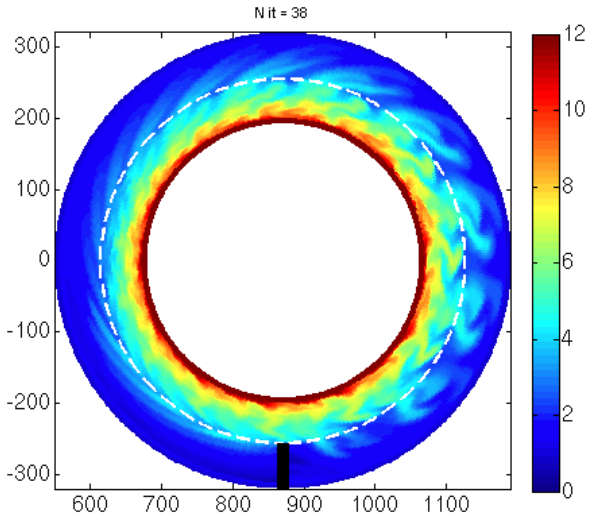

Figure 1: snapshot of turbulence (electron density field) in a TOKAM3X simulation in circular geometry with AGMG used as linear solver for the vorticity equation instead of PASTIX

6 Task 3.4: Linear Algebra solvers for Wind

Interfacing Alya with EoCoE Sparse Linear Algebra Libraries

Partners: BSC, CNR, CNRS-IRIT, INRIA, ULB, UNITOV
Software packages: Alya, AGMG, Fabulous, MaPHyS, MLD2P4, MUMPS, PaStiX, PSBLAS

The Alya code is organized in a modular way and his architecture is split in modules, kernel and services, which can be separately compiled and linked. Each module represents a physical model, i.e., a set of Partial Differential Equations (PDEs) which can interact for running a multi-physics simulation in a time-splitting approach, while Alya's kernel implements the functionalities for dealing with the discretization mesh, the solvers and the input-output functionalities. The governing equations of a physical model are discretized in space by using Finite Element (FE) methods and all the functionalities to assemble the global matrix and righ-hand-side (RHS) of the corresponding set of equations, including boundary conditions and material properties, are responsibility of the module. All the functionalities needed to solve the algebraic linear systems are instead implemented in the kernel. Some work on data structures and distribution of sparse matrices and RHS was needed in order to interface Alya with the available EoCoE scientific libraries for sparse linear algebra, as described in the following.

Currently, Alya has been successfully interfaced with five different Linear Algebra packages: MUMPS, PaStiX, MaPhyS, AGMG, and PSBLAS (and its preconditioners library MLD2P4). The first two packages implement direct solvers, while the last three provide iterative solvers. Main aims in our work is to verify if using external libraries outperforms, in terms of parallel efficiency and scalability, the Alya's internal solver.

The most critical point when interfacing with an external solver is the matrix storage scheme. There is a wide range of matrix storage schemes, and generally one may need to change from the scheme used within the application to the scheme accepted by the solver. All available solvers support either Compressed Sparse Row matrix scheme (CSR) or Coordinate scheme (COO) for the storage of the sparse matrix while Alya only uses

the CSR scheme. Transforming from CSR to COO is quite simple, and then there was no significant difficulty from this perspective. For vector unknowns, Alya stores all the components of the matrix related to a vector unknown in a block. This format was not available in any of the EoCoE solvers, therefore, the solution adopted was to treat each component of the vector unknown as a separate unknown. A subroutine was implemented inside the Alya's kernel that takes care of this.

Finally, the main difficulty in the interfacing process was how the data, i.e. the discretization mesh and the corresponding unknowns, are distributed among the parallel processes, and the way the related sparse matrix rows and RHS are locally assembled. The Alya code is based on a domain decomposition where the discretization mesh is partitioned into disjoint subsets of elements/nodes, referred to as subdomains. Then, each subdomain is assigned to a parallel process which carries out all the geometrical and algebraic operations corresponding to that part of the domain and the associated unknowns. The interface elements/nodes on the boundary between two subdomains are assigned to one of the subdomains (see Fig. 2).

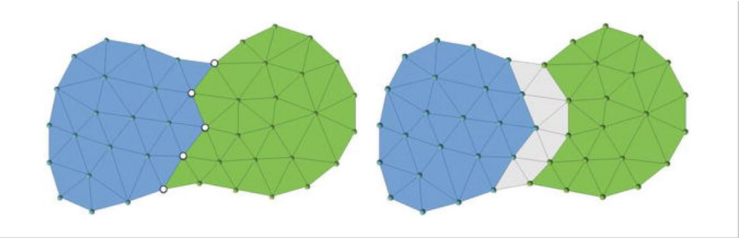

Figure 2: Mesh partitioning into: (left) disjoint sets of nodes. In white, interface nodes; and (right) disjoint sets of elements. In white, halo elements.

The sparse matrices expressing the linear couplings among the unknowns are distributed in such a way that each parallel process holds the entries associated with the couplings generated on its subdomain. In more details, two different options are possible for sparse matrix distribution: the Partial Row format and the Full Row format [6], respectively. In the Full Row format, if a mesh element/node and the corresponding unknown belongs to a process, all row entries related to that unknown are stored by that process. In the Partial Row format, the row of a matrix corresponding to an unknown is not full and it needs contributions from unknowns belonging to different processes (see Fig. 3). Alya uses a Partial Row format for storing the matrix.

General-purpose solvers, such as the EoCoE scientific libraries for sparse solvers, often support only one of the above formats for matrix distribution. In particular, Pastix and Maphys support the Partial Row format and then interfacing Alya with that solvers was immediate. On the other hand, interfacing with the other available libraries require some programming work, resulting in some new kernel's subroutines, to convert the Partial Row format into the Full Row format. This implies matrix data communication among different parallel processes to be applied, at each time step of a time-dependent simulation, before calling the solvers. The above data communication subroutines are needed for running Alya with AGMG, MUMPS and PSBLAS/MLD2P4.

The latest versions of PSBLAS (rel. 3.6) and MLD2P4 (rel. 2.0) were interfaced to Alya in order to use parallel iterative solvers for solving algebraic linear systems arising from simulations based on the NASTIN module, which deals with the incompressible Navier-Stokes equations for turbulent flows. A software module has been developed in the

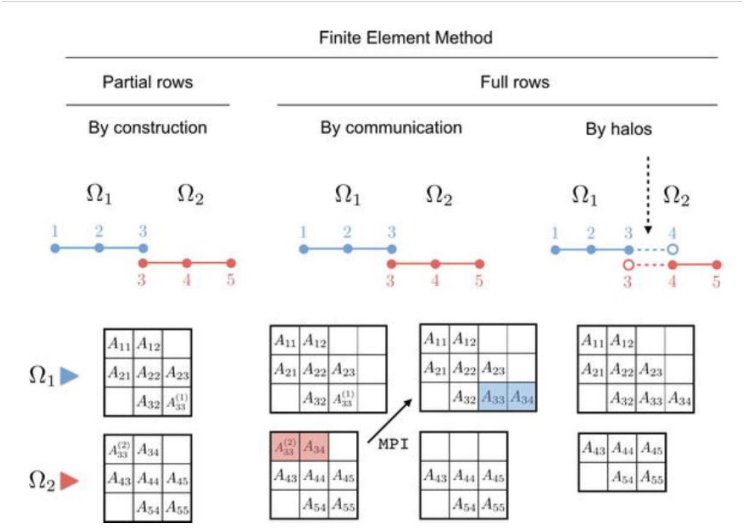

Figure 3: Matrix Distribution. From left to right: (1) FE: Partial rows, (2) FE: Full rows using communications, (3) FE: Full rows using halo elements.

Alya's kernel for declaration, allocation and initialization of the library's data structures as well as for using solvers and preconditioners. PSBLAS makes available some of the main and widely used iterative methods based on Krylov projection methods both for symmetric definite positive and for general unsymmetric linear systems, through a unique interface to a driver routine. The specification of that driver routine and a detailed description of its I/O parameters is provided in [4]. Preconditioners for PSBLAS Krylov solvers are available trough the MLD2P4 package. The latest version of the preconditioners package, improved and extended during the EoCoE I project, includes Algebraic MultiGrid (AMG) methods based on an aggregation scheme for pure algebraic coarsening. The package makes available many of the cycles and smoothers widely used in multigrid methods, including a version of a non-linear Krylov-based cycle. Either exact or approximate solvers can be used on the coarsest-level system. Specifically, different sparse LU factorizations from external packages, and native incomplete LU factorizations and Jacobi, hybrid Gauss-Seidel, and block-Jacobi solvers are also available. The main functionalities for selecting and building the chosen preconditioner are the responsibility of the software module included in the Alya's kernel, while the functionalities for applying it within the PSBLAS Krylov solver are completely transparent to the Alya code and are responsibility of the library. MLD2P4 is written in Fortran 2003, following an object-oriented design; therefore, all the functionalities for building and applying a preconditioner are written in terms of a method to be applied to the preconditioner object.

7 Task 3.5: Transversal activities

Standardized interface for parallel sparse linear system solvers

Partners: CNR, ULB
Software packages: All LA software libraries

7.1 Introduction

Sparse linear systems solvers are software libraries intended to solve linear systems of equations whose system matrix is stored in sparse mode. Although all libraries need essentially the same data and aim at delivering the same output (that is, the solution of the linear system), each has its specific interface and its own set of parameters. This makes the life of users difficult, especially when they want to test several solvers. For each of them, they often need to reconsider from scratch the build of a proper interface with the software code that needs to call the solver library.

In this document, we propose a standardized interface for sparse linear system solvers. It is restricted to "algebraic solver", that is, to solvers that do not require additional information besides the system matrix and the right hand side(s).

The idea is to facilitate users' life. If a program calls a solver from any participating library through an interface that is compliant with the proposed standard, exchanging this call for a call to another solver should require minimal effort. We also pay attention to avoid any specification that could be error prone.

On the other hand, solver libraries should be able to join the standardization with moderate effort. Also, design choices have been made in such a way that participating solvers keeps entire freedom to, e.g., distribute "native" interface besides the standardized one, or to add as many specific parameters to the common parameters defined in the standard. However, we point out here that the work for implementing the proposed common interface for libraries is not mandatory for every library exploited in the project, but each partner is free to decide if this could be done with the available man power.

The standard is presented below as a list of rules, sometimes preceded by the rationale behind these rules. A solver library has an interface compliant with the standard if and only if it comes with a set of interface functions or subroutines that agree with the given rules. Regarding optional features foreseen in the standard, they should not be seen as mandatory. All what is required is that

- a vanilla version of the solver is ready to work once mandatory parameters have been set up (setting whatever remaining parameters to default);
- if the user wants to activate an option foreseen in the standard that is not (yet) implemented, an error message is issued (the error can be fatal or not, depending on the case at hand).

7.2 General rules

- Each solver keeps its own, specific name (freely chosen) for the main function/ subroutine. To avoid confusion, this name may be different from the name used to call the library through its native interface.

  A prefix d will be added for double precision variants (non integer arguments are double precision) and a prefix z for complex variants (non integer arguments are double complex). On the other hand, a suffix 4 will be added when integer arguments are standard int (4 bytes), and a suffix 8 when they are long int (8 bytes).

  Examples: dmumps4, zagmg8

  This taxonomy is just to make clear the exact nature or the arguments. It does not mean that any package is committed to propose all versions. For instance, many solvers do not exist in complex mode. It is also possible to distribute additional versions, such as a variant working in real single precision.

- In MPI-based parallel mode, it is assumed that:
  - a library specific MPI communicator is available;
  - all MPI ranks in the library communicator simultaneously call the solver.
- In multithread mode, the calling is as in sequential mode; i.e., it is the responsibility of the solver to split the computation in multiple threads. (If the solver is called from inside a parallel region, it is expected that only the master thread calls the solver while the others are idle.)
- In hybrid MPI+multithread mode, the calling is the same as in pure MPI mode.
- The main function/subroutine will have the same name in all cases (Sequential/ Multithread/MPI/Hybrid). Which variant is requested will be indicated via some parameters (see below).

7.3 Main function/subroutine

Rationale

For flexibility, parameters controlling the solver are not passed to the main function or subroutine, but set up beforehand by calling auxiliary functions/subroutines (see below for details). In this way, it is easy to add parameters, and default are also set transparently: if a user does not call the auxiliary function or subroutine to set a given parameter, it is automatically set to default, while an error message is issued if the parameter is mandatory.

Rules

- Only five vector arguments that will contain the matrix, the right hand side and the solution, and one scalar argument specifying what action is requested:

  irow (integer), *jcol* (integer), *values* (double precision/complex),

  rhs (double precision/complex), *sol* (double precision/complex), *job_handle*(integer)

  (see below for details).

  Whether integer are standard or long, and whether other arguments are double precision or double complex should be transparent from the suffix and prefix (respectively) of the function name (see above).

- *job_handle* is an input/output integer argument.

  For standard usage, it should be set to 0 in input. Then, it is assumed that the calling program wants to solve one linear system at a time, and that each solve has to start from scratch with new data. Hence, memory is cleaned before returning to the calling program. In this case, *job_handle* is unmodified on output.

  If, however, the user wants to solve successively several linear systems with the same system matrix, it is advised to first call the main function or subroutine with *job_handle* set to 1 on input. Then, only setup is performed, that is all operations needed that do not depend on the right hand side. The result of these is stored in internal memory. Because no linear system is actually solved, the arrays *rhs* and sol are not accessed in this case. On the other hand, *job_handle* is set on output to a unique identifier, which has to be an integer not less than 2.

  To perform an actual solve with a matrix for which setup has been performed, *job_handle* has to be set on input to the identifier returned on output when calling

for setup. Then, arrays *irow*, *jcol* and *values* are normally not accessed. In this case, *job_handle* is unmodified on output.

To clean the memory associated with a particular instance of the solver, *job_handle* has to be set on input to a negative number which is the opposite of the identifier returned on output when calling for setup. Then, arrays *irow*, *jcol*, *values*, *rhs* and *sol* are not accessed. In this case, *job_handle* is unmodified on output.

Finally, setting *job_handle* to -1 will clean all memory. (This is equivalent to successive calls with the opposite of all valid identifiers.) In this case, *job_handle* is unmodified on output.

Note that it is not mandatory to have the above features fully implemented. For instance, if a solver has no setup phase, only *job_handle* equal to 0 may be allowed. A solver may also be able to deal with only one instance at a time. However, the solver documentation should inform about such limitations, and error messages issued when the calling program ignores them.

7.4 Sequential matrix format

Rationale

- There is no universal matrix format, but the coordinate (COO) format is easy to understand and used by many libraries as default. Hence, it is the default as well for the standardized interface.

  Many solvers use internally the compressed sparse row (CSR) format, or the compressed sparse column (CSC) format. These are also used by many software codes that need to call solver libraries. Hence these alternative formats are among the options foreseen by the standardized interface, and the documentation of participating libraries may advertise that using, say, CSR, will avoid some overhead in data handling. However, it is not mandatory to propose these options.

  It turns out that many users call solvers with "unassembled" matrices, in the sense that some entries appear several times in the input data, with the intent that the true value corresponds to the sum of these. Although this is not strictly part of COO (or CSR & CSC) standard, many software accept this and automatically make the assembly (e.g., Matlab).

  Hence, as it is better to manage properly what cannot be prevented, this assembly process should be part of the standard. Solvers may optionally propose alternative choices (such as raising an error if a repeated entry is found), controlled via appropriate parameters.

  However, for consistency, the assembly (i.e., repeated entries are summed) should be the default.

- When the matrix is symmetric, it seems also reasonable to allow the user to input only one of the triangular parts (hence to use symmetric storage).

- To avoid confusion among the users, it is important that the standard imposes a common indexing rule. Somehow arbitrarily, 1-based indexing has been chosen (see the rules below).

Rules

- Each solver should accept arrays *irow*, *jcol*, *values* corresponding to the COO format (default). Other input methods (CSR, CSC) can be accepted; these are requested by the calling program via some input parameters (see below).
- Via some parameter (see below), the user may also tell that input data use symmetric storage, and, more precisely, that either only the lower triangular part of input matrix is significant, or that only upper triangular part of input matrix is significant, or that all contributions from the strict lower and strict upper triangular parts are to be summed to define both the strict lower and strict upper triangular parts of the input matrix. If CSR and/or CSC are accepted, it is not mandatory that they are compatible with symmetric storage options.
- Repeated entries are summed to form the actual entry, except if an alternative behavior is explicitly asked via some (optional, solver specific) parameter.
- For the CSR format, it is not expected that entries in a row are sorted by increasing column index. (This is not part of the standard.) If a solver needs this sorting, it should foresee to process it internally, while, possibly, leaving the opportunity to the user to tell that the rows are sorted via some input parameter. Similarly, for the CSC format, it is not expected that entries in a column are sorted by increasing row index.
- For the COO format, entries may occur in any order.
- Arrays *irow*, *jcol* and *values* are input only: they are not modified on output.
- 1-based indexing is used. For all formats, it means that row and column indexes start at 1. For CSR and CSC formats, it means additionally that, for the pointer to the beginning of each row (array *irow* for the CRS format) or column (array *jcol* for the CSC format), a value of 1 points to the first entry in array *values* (that is, the first storage location associated with the corresponding pointer).

7.5 Distributed matrix format (MPI mode)

Rationale for the numbering of unknowns and their distribution among MPI ranks

This is the point where the solvers differ most between each other. But this is also here that users face most difficulties, and that a standardization effort is most welcome. The choices have been made on the basis of two principles. Firstly, the standard has to be clear and easy to understand: experience shows that subtle mechanisms lead frequently to errors which, in case of a standardized interface, could be on the side of the users' program, but also on the side of the implementers that need to convert transferred data to the native data structure of their solver. Secondly, such a transfer to about any kind of internal structure should not be difficult to implement and not entail scalability issues even at very large processor scale. (Although some overhead is unavoidable, and the solver documentation may recommend to use the native interface for best performance.)

Some solvers require that data are communicated with respect to a global numbering of the unknowns, while others only deal with local numberings and restore the global information in another way, for instance using unknowns that are shared by several MPI ranks. While this latter organization is appealing from the scalability viewpoint, it is, however, very hard to define a standard on such a basis satisfying the two requirements stated above. Opposite to this, it seems easy to transfer data provided with respect to a global numbering in any given local scheme. The same happens from the user viewpoint:

if a user program generates its data without referring to a global numbering, defining one from local numberings on each rank seems straightforward.

Hence, the standard assumes that a global numbering is in effect, and that row and column indexes of matrix entries passed to the solver refer to this global numbering.

Next the solver has to be informed on how the unknowns are distributed among the MPI ranks. Of course, it is not needed to have such a distribution in effect if the matrix is passed in COO format (only global indexes are transmitted), and if input and output vectors (arrays *rhs* and *sol*) are communicated in a centralized way. However, for scalability reasons, the default mode is to communicate vectors in a distributed way: each rank receives only the part of these vectors corresponding to local unknowns, according to a given distribution of these latter.

There is another reason why a distribution of the unknowns should be in effect. Many solvers use parallelization strategies that are driven by the distribution made by the calling program. With respect to this distribution, common requirements are: (1) matrix vector multiplication is dominated by computation associated with local unknowns, and require only relatively few communication between neighbor MPI ranks; (2) in the system matrix, the (global) weight of nonlocal connections (i.e., of entries connecting unknowns on different MPI ranks) is relatively small compared to the (global) weight of local connections (i.e., of entries connecting unknowns on a same MPI rank).

Hence, the standard assumes that some distribution of the unknowns is in effect.

Now, we need to discuss how MPI ranks are informed about this distribution. That is, how are related the global numbering and the local orderings according to which distributed input and output vectors are defined.

A flexible option would be to define a global list that maps each global index to a given MPI rank, but managing such a list would raise scalability issues at large processor scale. One could also foresee that each MPI rank receives a vector that maps its local numbers to global indexes, but, to execute a matrix vector product, one needs to be able to retrieve on any MPI rank the home rank corresponding to nonlocal unknowns involved in the product. Except if one requires from the user additional input data (likely error prone), this implies that these mappings should be shared among all MPI ranks. This would also lead to scalability issues.

Hence, although flexible schemes can be offered as an option (and should if the native interface has such options), the default is therefore to have no flexibility:

1. Each rank holds a block of consecutive unknowns with respect to the global numbering.

2. The global numbering is consistent with the MPI rank ordering: rank 0 holds the first block of unknowns, rank 1 the next, etc.

Then, to map the local ordering to a consistent global numbering, it suffices to propagate the number of local unknowns on each MPI rank. A global communication is still needed, but it is limited to a single integer per MPI rank.

From the implementers viewpoint, this choice is the most easy to deal with. If the solver requires by default mapping vectors, it is trivial to construct them from the above rules. If, opposite to this, it works without global numbering, retrieving local indexes from global ones is trivial as well. Further, if the internal structure is based on shared unknowns between MPI ranks, it is straightforward to deduce from matrix data which nonlocal unknowns should be shared unknowns in the internal structure, and attribute them further local numbers. Finally, if the solver foresees its own distribution of the work

and privileges centralized communication of input and output global vectors, conversion to such vectors from the default distributed input mode and vice versa is easily implemented with ALLGATHERV/ALLSCATTERV operations.

From the users viewpoint, this may require some efforts. However, this is the price to pay to have straightforward access to all participating libraries. Moreover, because the standard is simple, this should not be a major source of errors. Finally, the effort seems moderate whatever the organization of the calling program. If it is based on a global numbering with a distribution of the unknowns not matching the above rules, the best to do is to change the global ordering unknowns without changing their distribution. If no global numbering is in effect, the above rules actually define one that is not difficult to implement. Even if the calling program uses shared unknowns, constructing a global ordering should not be too difficult. Basically, it suffices to chose a rule to affect shared unknowns to a unique MPI rank (e.g., select, among the task sharing the node, the one with smallest rank). Further, the native communication mechanism associated with such structure should allow one to easily inform neighbor tasks about the indexes in global numbering that have been attributed to shared unknowns.

Rationale for the input matrix

The format for the input matrix is a copy-paste of the one used in sequential: COO is the default, CSR and CSC are possible (non mandatory) options. Because all MPI ranks simultaneously call the solver, it is natural that the input arrays *irow*, *jcol*, *values* contain each a portion of the global matrix.

When COO is used, arrays *irow* and *jcol* both contain indices in global numbering. It is not logical to introduce any constraint here; i.e., neither the row index nor the column index is enforced to correspond to a local unknown. This, in particular, is convenient for calling codes that use a structure based on the sharing of (boundary) unknowns.

The documentation of participating libraries may, however, warn users that some communication overhead will be avoided if on a given MPI ranks only entries corresponding to, say, local rows are input.

When CSR is used, because this format is row oriented, it is natural to assume that on each MPI rank the input data define the rows associated with the local unknowns. That is, *irow* (of size nrow+1, where nrow is the number of local unknowns) is still a pointer in *jcol* and *values*, for the block of matrix rows associated with local unknowns; *jcol* has to contain global column indexes. Global row indexes are not explicitly referred, but are deduced from the rules above, knowing just the global index of the first local unknown (via a computation based on the propagation of the number of local rows for each rank).

Similarly, when CSC is used, because this format is column oriented, it is assumed that on each MPI rank the input data define the columns associated with the local unknowns. That is, *jcol* (of size nrow+1, where nrow is the number of local unknowns) is still a pointer in *irow* and *values*, for the block of matrix rows associated with local unknowns; *irow* has to contain global row indexes. Global column indexes are not explicitly referred, but are deduced from the rules above, knowing just the global index of the first local unknown (via a computation based on the propagation of the number of local rows for each rank).

Because *jcol* and *irow* contain global indexes in possibly huge matrices, one should allow them to be 8 bytes length, motivating the variants with suffix 8. (It does not mean that these variants should be generated with compilers flags that turn all integers to long int, although it is of course possible to do so. In fact, we do expect that in most cases the indexes for the local parts of the matrices will fit into 4 bytes.)

Finally, regarding symmetric storage, it seems not wise to accept an option that ignores entries based on which triangular part they are in, because this might be confusing or

unpractical for entries connecting unknowns on different MPI ranks. Hence the only offered option sums all contributions dealing with the same pair of unknowns, regardless if it is input as a(i,j) or as a(j,i). Moreover, this option seems not very convenient when the CSR or the CSC format is used. Hence in the MPI case it is compatible with the COO format only.

Here again, users can be warned that a global symmetric storage implies additional communications, either at the matrix assembly or at the matrix-vector product stage.

Rules

- Each rank holds a set of consecutive unknowns with respect to the global numbering.
- The global numbering is consistent with the MPI rank ordering: rank 0 holds the first block of unknowns, rank 1 the next, etc.
- Rules for the input matrix in the sequential case apply in MPI mode as well, with the following restrictions or peculiarities:
  - When COO is used, arrays *irow* and *jcol* both contain indexes in global numbering. Neither the row index nor the column index is enforced to correspond to a local unknown, i.e., entries may still be input in any order on any MPI rank.
  - When CSR is used, the input matrix on each MPI rank defines the block of rows in the global matrix corresponding to the local unknowns. The full rows are to be provided, including non local connections. Indexes provided in *jcol* are to be global.

    Symmetric storage is not allowed.
  - When CSC is used, the input matrix on each MPI rank defines the block of columns in the global matrix corresponding to the local unknowns. The full columns are to be provided, including non local connections. Indexes provided in *irow* are to be global.

    Symmetric storage is not allowed.

7.6 Right-hand side and solution vectors

Rationale

The arguments *rhs* and *sol* do not need much additional comments. By default, in MPI mode, each rank receives only the part of these vectors corresponding to local unknowns. Solvers may propose optionally other input methods. For instance, they may accept that these vectors are communicated in a centralized way; that is, via a solver-specific parameter, the user may request that arrays *rhs* and *sol* are to be accessed only on a specific MPI rank, on which the full right and side is provided and the full solution has to be returned.

In case of several right hand sides, *rhs* and *sol* do in fact correspond to arrays with nrow x n*rhs* entries. A possible source of error is then the ordering used: row major order (C standard) or column major order (Fortran standard). To be on the safe side, the standard imposes that, if several right hand side are provided, the user explicitly indicates via some parameter which ordering is used (none of them can be used silently as default, even though only one of the two options would be offered to the users).

7.7 Auxiliary functions/subroutines

Rules

- Two should be provided for control parameters, one for integer parameters and one for double precision parameters. Suggested name: nameofthesolver_intparam, nameofthesolver_realparam.
- In each case, three arguments:
  keyword (character),
  value (integer for intparam / double precision for realparam),
  job_handle (integer).

  Examples: dmumps4_intparam('NROW',n,*job_handle*)
  zagmg8_realparam('TOL',1.0d-6,*job_handle*)
- *job_handle* is normally set to the same value that will be used for the next next call to the main function or subroutine, and then indicates that the calling program requests that the prescribed parameter setting be in effect for that call.

  More precisely, setting *job_handle* to 0 or 1 means that the parameter setting prescribed by the call to the auxiliary function/subroutine will be in effect for the next call to the main function or subroutine with *job_handle* set to 0 or 1.

  On the other hand, setting *job_handle* to a valid identifier returned by the main function/subroutine (when called with *job_handle* set to 1 on input, see above) allows the user to modify a parameter after setup or in between successive solves. Users are warned that some parameters cannot be changed after setup. Which parameters can be changed or not is freely chosen by the implementer. The only constraint is that an error message should be issued if a user wants to make a change that is not permitted.
- As a general rule, a parameter setting prescribed by calling auxiliary functions/ subroutines will be in effect for one instance of the solver only. That is, all relevant parameters need to be redefined in between two calls of the main function or subroutine with *job_handle* set to 0 or 1.
- For intparam, the type integer (standard or long) is in agreement with the suffix: standard if suffix 4 is used, long if suffix 8 is used.
- Implementers remain free to define additional functions of routines to pass other types of parameters, such as complex numbers or arrays.

7.8 Common parameters

Rationale

Since some parameters are obviously needed by most algebraic solvers, the standard imposes a unified syntax for these, and some rules to be followed strictly regarding their specific meaning.

It is harmless if some of these common parameters are meaningless for some solvers (example: tolerance is not relevant for direct solvers). If a user tries to set a parameter that is meaningless, this action should result in a warning message.

Besides, solver-specific parameters can be added freely. In fact, all parameters available via the native interface can (should) be available via the standardized interface as well. (The user guide should only be adapted to reflect the new way these are defined.)

On the other hand, it is not mandatory that all options foreseen by the list below are implemented (although it is expected that developers will consider this as an objective for the future releases).

Rules

- An interface will be considered as compliant with the standard as soon as:
- a vanilla version of the solver is ready to work once mandatory parameters have been set up (setting whatever remaining parameters to default);
- when setting any of the common parameters to non default value, either the corresponding option is implemented as described in the standard, or a (warning or fatal) error message is issued.

  Examples:

  if CSR format is not supported, requesting this format entails a fatal error;

  if multithreading is not implemented, requesting multithread mode entails a warning message. (Computation can be pursued in 1-thread mode.)
- For solver-specific parameters, keywords should follow a naming scheme that clearly identifies the underlying solver using them. Example: `'AGMG_NREST'`.
- To avoid confusion, if a solver has a "native" parameter that is close in some sense to one of the common parameters listed below, but with a slightly different meaning, implementers should treat it as a solver-specific parameter, and, possibly, issue a warning message when the user tries to set up the related common parameter. For instance, if a solver implements other stopping criteria than a test based on the relative residual error ($\|Ax-b\|/\|b\| < \mathrm{TOL}$) normally associated with the `'TOL'`parameter (see below), another keyword should be used for these alternative criteria, in such a way that `'TOL'` keeps a consistent meaning for all solvers.
- Parameters listed below that are flagged "GLOBAL" are global in MPI mode. That is, they are expected or need to be identical on all ranks. The standard imposes that they are set to a same value on all ranks by simultaneous calls to the appropriate input function/subroutine (this will in general not be checked by the solver).
- Keywords should be accepted in both upper case and lower case; e.g.: `'tol'` is equivalent to `'TOL'`. However, additional flexibility is discouraged (e.g., admitting `'Tol'`). It can indeed be a portability issue for users if a solver admits some input method that is not accepted by others.
- Via some optional, solver-specific parameters, the default rules below can be overwritten, in the sense that a mandatory parameter can be made optional (e.g., the number of local rows may become unnecessary if vectors are communicated in a centralized way) or that a global parameter can be made local (e.g., the number of threads can be processor dependent). However, to avoid confusion and errors by the users, it is important that such changes require explicit action.

List

Integer, Input

- `'NROW'` (number of local rows – mandatory)

- `'MPI'` (1 if MPI mode – default: 0)
  GLOBAL
- `'MTH'` (1 for multithread mode – default: 0)
  GLOBAL
- `'NRHS'` (number of right hand sides – default: 1)
  GLOBAL
- `'NTHREAD'` (number of threads in multithread mode – default: system default)
  GLOBAL
- `'MPICOMM'` (MPI communicator – mandatory in MPI mode, meaningless otherwise)
  GLOBAL
- `'INPUTFMT'` (Input format: 0 for COO (default); 1 for CSR; -1 for CSC)
  GLOBAL
- `'SYMSTO'` (symmetric storage:
  0: general nonsymmetric storage – default;
  -1: only lower triangular part of input matrix is significant – not compatible with MPI;
  1: only upper triangular part of input matrix is significant – not compatible with MPI;
  2: contributions from the strict lower and strict upper triangular parts are summed to form both the strict lower and the strict upper triangular parts of the input matrix; in MPI case, requires COO format)
  GLOBAL
- `'SPD'` (1 if the users guaranties that the input matrix is symmetric (hermitian in complex case) and positive definite; default: 0)
  GLOBAL
- `'INGUESS'` (1 if initial guess provided in array *sol*; default: 0)
  GLOBAL
- `'NVAL'` (mandatory for COO format, meaningless for other input formats: number of entries in array *irow*, *jcol* and *values*)
- `'MAXIT'` (maximum number of iterations – default: solver specific)
  GLOBAL
- `'MAJORD'` (Major order for the arrays *rhs* and *sol* – mandatory if more than 1 right hand side, and meaningless otherwise:
  1 for column major order (one vector after another in input/output arrays);
  -1 for row major order (interleave of components in input/output arrays).
  GLOBAL

Integer, Output

- `'FLAG'` (0 if successful, 1 if target accuracy not reached, 2 if other failure occurred)
- `'ITER'` (Number of performed iterations)

Double precision, Input

- `'TOL'` (Tolerance on the relative residual error: the solver exit when $\|Ax-b\|/\|b\| <$ TOL , where A is the system matrix, x the computed solution and b the right hand side — mandatory or meaningless)

  GLOBAL

Double precision, Output

- `'RELRES'` (Relative residual norm upon completion)

References

[1] Bouzat, Nicolas, Bressan, Camilla, Grandgirard, Virginie, Latu, Guillaume, and Mehrenberger, Michel. Targeting realistic geometry in tokamak code gysela. *ESAIM: ProcS*, 63:179–207, 2018.

[2] P. D'Ambra, D. di Serafino, and S. Filippone. MLD2P4 User's and Reference Guide, rel. 2.2, 2018. `https://github.com/sfilippone/mld2p4-2/`.

[3] Fabulous team. Fabulous software and documentation. `https://gitlab.inria.fr/solverstack/fabulous`.

[4] S. Filippone and A. Buttari. PSBLAS User's Guide, rel. 3.6, 2018. `https://github.com/sfilippone/psblas3`.

[5] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, Ch. Ehrlacher, D. Esteve, X. Garbet, Ph. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, Ch. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso. A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. *Computer Physics Communications*, 207:35 – 68, 2016.

[6] G. Houzeaux, R. Borrell, Y. Fournier, M. Garcia-Gasulla, J. Henrik Göbbert, E. Hachem, V. Mehta, Y. Mesri, H. Owen, and M. Vázquez. High-Performance Computing: Dos and Don'ts. In Adela Ionescu, editor, *Computational Fluid Dynamics - Basic Instruments and Applications in Science*, pages 3–41. IntechOpen, 2018.

[7] MaPHyS team. MaPHyS software and documentation. `https://gitlab.inria.fr/solverstack/maphys`.

[8] MUMPS team. MUMPS: Multifrontal massively parallel solver. `http://mumps-solver.org/`.

[9] Y. Notay. AGMG software and documentation. `http://agmg.eu`.

[10] PaStiX team. PaStiX software and documentation. `https://gitlab.inria.fr/solverstack/pastix`.

[11] Edoardo Zoni and Yaman Güçlü. Solving hyperbolic-elliptic problems on singular mapped disk-like domains with the method of characteristics and spline finite elements. *Journal of Computational Physics*, 398:108889, 2019.