MAPHYS USER'S GUIDE 0.9.4

This document explains the Fortran 90 interface of the package MAPHYS, the Massively Parallel Hybrid Solver.

Author : HiePaCS Team, INRIA Bordeaux, France

Table des matières

1 General overview **4**

2 General introduction **5**
2.1 Parallel hierarchical implementation 6
2.2 Dropping strategy to sparsify the preconditioner 9
2.3 Sparse approximation based on partial $ILU(t,p)$ 9
2.4 Exact vs. approximated Schur algorithm to build the precond 10

3 Installation **13**
3.1 Requirements 13
3.2 Compilation and installation 13

4 Sequence in which routines are called **14**

5 Application Programming Interface **16**
5.1 General 16
5.1.1 Maphys instance 16
5.1.2 Arithmetic versions 16
5.1.3 Version number 16
5.1.4 Control of the four main phases 16
5.1.5 Control of parallelism 16
5.2 Input matrix, right-hand side and solution vectors 17
5.2.1 Matrix type 17
5.2.2 Matrix format 17
5.3 Writing a matrix to a file 21
5.4 Using two levels of parallelism inside MAPHYS 21

6 List of control parameters **23**
6.1 ICNTL 23
6.2 RCNTL 28
6.3 Compatibility table 28

7 List of information parameters **30**
7.1 Information local to each processor 30
7.1.1 IINFO 30
7.1.2 RINFO 32
7.2 Information available on all processes 34
7.2.1 IINFOG 34
7.2.2 RINFOG 34

8 Error diagnostics **35**

9 Examples of use of MaPHyS **36**
9.1 A matrix problem 36
9.1.1 How to write an input file 36
9.1.2 System description 37
9.1.3 System description 38
9.2 A cube generation problem 38
9.3 How to use MAPHYS in an other code 39
9.3.1 Fortran example 39
9.3.2 C example 42

10 Notes on MaPHyS distribution **45**
10.1 License 45
10.2 How to refer to MAPHYS 45
10.3 Authors 45

1 General overview

MaPHyS is a software package for the solution of sparse linear system

$$\mathcal{A}x = b$$

where $\mathcal{A}$ is a square real or complex non singular general matrix, b is a given right-hand side vector, and x is the solution vector to be computed. It follows a non-overlapping algebraic domain decomposition method that first reorders the linear system into a 2×2 block system

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

where $\mathcal{A}_{II}$ and $\mathcal{A}_{\Gamma\Gamma}$ respectively represent interior subdomains and separators, and $\mathcal{A}_{I\Gamma}$ and $\mathcal{A}_{\Gamma I}$ are the coupling between interior and separators. By eliminating the unknowns associated with the interior subdomains $\mathcal{A}_{II}$ we get

$$\begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ 0 & \mathcal{S} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ f \end{pmatrix}, \quad (2)$$

with

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

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix*. Because most of the fill-in appears in the Schur complement, the Schur complement system is solved using a preconditioned Krylov subspace method while the interior subdomain systems are solved using a sparse direct solver. Although, the Schur complement system is significantly better conditioned than the original matrix $\mathcal{A}$, it is important to consider further preconditioning when employing a Krylov method.

In MaPHyS, several overlapping block diagonal preconditioning techniques are implemented, where each diagonal block is associated with the interface of a subdomain :

1. dense block diagonal factorization : each diagonal block is factorized using the appropriated LAPACKsubroutine,
2. sparsified block diagonal factorization : the dense diagonal block is first sparsified by droppring entry $s_{i,j}$ if it is lower than $\xi(|s_{i,i}| + |s_{j,j}|)$. The sparse factorization is performed by a sparse direct solver.
3. sparse approximation of the diagonal block : a sparse approximation of the diagonal block is computed by replacing $\mathcal{A}_{II}^{-1}$ by an incomplete $ILU(t,p)$ factorization. The computed approximation of the Schur complement is further sparsified.

Because of its diagonal nature (consequently local), the preconditioner tends to be less efficient when the number of subdomains is increased. The efficient solution provided by MaPHyS results from a trade-off between the two contradictory ideas that are increasing the number of domains to reduce the cost of the sparse factorization of the interior domain on one hand ; and reducing the number of domains to make easier the iterative solution for the interface solution on the other hand.

2 General introduction

Solving large sparse linear systems $\mathcal{A}x = b$, where $\mathcal{A}$ is a given matrix, b is a given vector, and x is an unknown vector to be computed, appears often in the inner-most loops of intensive simulation codes. It is consequently the most time-consuming computation in many large-scale computer simulations in science and engineering, such as computational incompressible fluid dynamics, structural analysis, wave propagation, and design of new materials in nanoscience, to name a few. Over the past decade or so, several teams have been developing innovative numerical algorithms to exploit advanced high performance, large-scale parallel computers to solve these equations efficiently. There are two basic approaches for solving linear systems of equations : direct methods and iterative methods. Those two large classes of methods have somehow opposite features with respect to their numerical and parallel implementation efficiencies.

Direct methods are based on the Gaussian elimination that is probably among the oldest method for solving linear systems. Tremendous effort has been devoted to the design of sparse Gaussian eliminations that efficiently exploit the sparsity of the matrices. These methods indeed aim at exhibiting dense submatrices that can then be processed with computational intensive standard dense linear algebra kernels such as BLAS, LAPACKand SCALAPACK. Sparse direct solvers have been for years the methods of choice for solving linear systems of equations because of their reliable numerical behavior [13]. There are on-going efforts in further improving existing parallel packages. However, it is admitted that such approaches are intrinsically not scalable in terms of computational complexity or memory for large problems such as those arising from the discretization of large 3-dimensional partial differential equations (PDEs). Furthermore, the linear systems involved in the numerical simulation of complex phenomena result from some modeling and discretization which contains some uncertainties and approximation errors. Consequently, the highly accurate but costly solution provided by stable Gaussian eliminations might not be mandatory.

Iterative methods, on the other hand, generate sequences of approximations to the solution either through fixed point schemes or via search in Krylov subspaces [20]. The best known representatives of these latter numerical techniques are the conjugate gradient [12] and the GMRES [21] methods. These methods have the advantage that the memory requirements are small. Also, they tend to be easier to parallelize than direct methods. However, the main problem with this class of methods is the rate of convergence, which depends on the properties of the matrix. Very effective techniques have been designed for classes of problems such as the multigrid methods [10], that are well suited for specific problems such as those arising from the discretization of elliptic PDEs. One way to improve the convergence rate of Krylov subspace solver is through preconditioning and fixed point iteration schemes are often used as preconditioner. In many computational science areas, highly accurate solutions are not required as long as the quality of the computed solution can be assessed against measurements or data uncertainties. In such a framework, the iterative schemes play a central role as they might be stopped as soon as an accurate enough solution is found. In our work, we consider stopping criteria based on the backward error analysis [1, 7, 9].

Our approach to high-performance, scalable solution of large sparse linear systems in parallel scientific computing is to combine direct and iterative methods. Such

a hybrid approach exploits the advantages of both direct and iterative methods. The iterative component allows us to use a small amount of memory and provides a natural way for parallelization. The direct part provides its favorable numerical properties. Furthermore, this combination enables us to exploit naturally several levels of parallelism that logically match the hardware feature of emerging many-core platforms as stated in Section 1. In particular, we can use parallel multi-threaded sparse direct solvers within the many-core nodes of the machine and message passing among the nodes to implement the gluing parallel iterative scheme.

The general underlying ideas are not new. They have been used to design domain decomposition techniques for the numerical solution of PDEs [22, 17, 16]. Domain decomposition refers to the splitting of the computational domain into sub-domains with or without overlap. The splitting strategies are generally governed by various constraints/objectives but the main one is to enhance parallelism. The numerical properties of the PDEs to be solved are usually extensively exploited at the continuous or discrete levels to design the numerical algorithms. Consequently, the resulting specialized technique will only work for the class of linear systems associated with the targeted PDEs. In our work, we develop domain decomposition techniques for general unstructured linear systems. More precisely, we consider numerical techniques based on a non-overlapping decomposition of the graph associated with the sparse matrices. The vertex separator, constructed using graph partitioning [14, 5], defines the interface variables that will be solved iteratively using a Schur complement approach, while the variables associated with the interior sub-graphs will be handled by a sparse direct solver. Although the Schur complement system is usually more tractable than the original problem by an iterative technique, preconditioning treatment is still required. For that purpose, we developed parallel preconditioners and designed a hierarchical parallel implementation. Linear systems with a few tens of millions unknowns have been solved on a few thousand of processors using our designed software.

2.1 Parallel hierarchical implementation

In this section, methods based on non-overlapping regions are described. Such domain decomposition algorithms are often referred to as substructuring schemes. This terminology comes from the structural mechanics discipline where non-overlapping ideas were first developed. The structural analysis finite element community has been heavily involved in the design and development of these techniques. Not only is their definition fairly natural in a finite element framework but their implementation can preserve data structures and concepts already present in large engineering software packages.

Let us now further describe this technique and let $\mathcal{A}x = b$ be the linear problem. For the sake of simplicity, we assume that $\mathcal{A}$ is symmetric in pattern and we denote $G = \{V, E\}$ the adjacency graph associated with $\mathcal{A}$. In this graph, each vertex is associated with a row or column of the matrix $\mathcal{A}$ and it exists an edge between the vertices i and j if the entry $a_{i,j}$ is non zero.

We assume that the graph G is partitioned into N non-overlapping sub-graphs G_1, ..., G_N with boundaries Γ_1, ..., Γ_N. The governing idea behind substructuring or Schur complement methods is to split the unknowns in two subsets. This induces the follo-

wing block reordered linear system :

$$\begin{pmatrix} \mathcal{A}_{II} & \mathcal{A}_{I\Gamma} \\ \mathcal{A}_{\Gamma I} & \mathcal{A}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} x_I \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} b_I \\ b_\Gamma \end{pmatrix}, \tag{4}$$

where x_Γ contains all unknowns associated with sub-graph interfaces and x_I contains the remaining unknowns associated with sub-graph interiors. Because the interior points are only connected to either interior points in the same sub-graph or with points on the boundary of the sub-graphs, the matrix $\mathcal{A}_{II}$ has a block diagonal structure, where each diagonal block corresponds to one sub-graph. Eliminating x_I from the second block row of Equation (4) leads to the reduced system

$$\mathcal{S}x_\Gamma = f, \tag{5}$$

where

$$\mathcal{S} = \mathcal{A}_{\Gamma\Gamma} - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}\mathcal{A}_{I\Gamma} \text{ and } f = b_\Gamma - \mathcal{A}_{\Gamma I}\mathcal{A}_{II}^{-1}b_I. \tag{6}$$

The matrix $\mathcal{S}$ is referred to as the *Schur complement matrix*. This reformulation leads to a general strategy for solving Equation (4). Specifically, an iterative method can be applied to Equation (5). Once x_Γ is known, x_I can be computed with one additional solve on the sub-graph interiors.

Let Γ denote the entire interface defined by $\Gamma = \cup\,\Gamma_i$; we notice that if two sub-graphs G_i and G_j share an interface then $\Gamma_i \bigcap \Gamma_j \neq \emptyset$. As interior unknowns are no longer considered, new restriction operators must be defined as follows. Let $\mathcal{R}_{\Gamma_i} : \Gamma \to \Gamma_i$ be the canonical point-wise restriction which maps full vectors defined on Γ into vectors defined on Γ_i. Thus, in the case of many sub-graphs, the fully assembled global Schur $\mathcal{S}$ is obtained by summing the contributions over the sub-graphs. The global Schur complement matrix, Equation (6), can be written as the sum of elementary matrices

$$\mathcal{S} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \mathcal{S}_i \mathcal{R}_{\Gamma_i}, \tag{7}$$

where

$$\mathcal{S}_i = \mathcal{A}_{\Gamma_i\Gamma_i} - \mathcal{A}_{\Gamma_i\mathcal{I}_i}\mathcal{A}_{\mathcal{I}_i\mathcal{I}_i}^{-1}\mathcal{A}_{\mathcal{I}_i\Gamma_i} \tag{8}$$

is a local Schur complement associated with G_i and is in general a dense matrix (eg : if $\mathcal{A}_{\mathcal{I}\mathcal{I}}$ is irreducible, $\mathcal{S}_i$ is dense). It can be defined in terms of sub-matrices from the local matrix $\mathcal{A}_i$ defined by

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ A_{\Gamma_i\mathcal{I}_i} & A_{\Gamma_i\Gamma_i} \end{pmatrix}. \tag{9}$$

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

sub-graphs effectively reduces the condition number of the Schur complement matrix to $\kappa(S) = \mathcal{O}(h^{-1})$. While improved, preconditioning can significantly lower this condition number further.

We introduce the general form of the preconditioner considered in this work. The preconditioner presented below was originally proposed in [4] in two dimensions and successfully applied to large three dimensional problems and real life applications in [8, 11]. To describe this preconditioner we define the local assembled Schur complement, $\bar{\mathcal{S}}_i = \mathcal{R}_{\Gamma_i} S \mathcal{R}_{\Gamma_i}^T$, that corresponds to the restriction of the Schur complement to the interface Γ_i. This local assembled preconditioner can be built from the local Schur complements $\mathcal{S}_i$ by assembling their diagonal blocks.

With these notations the preconditioner reads

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

If we considered a planar graph partitioned into horizontal strips (1D decomposition), the resulting Schur complement matrix has a block tridiagonal structure as depicted in Equation (11)

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

For that particular structure of $\mathcal{S}$ the submatrices in boxes correspond to the $\bar{\mathcal{S}}_i$. Such diagonal blocks, that overlap, are similar to the classical block overlap of the Schwarz method when written in a matrix form for 1D decomposition. Similar ideas have been developed in a pure algebraic context in earlier papers [3, 18] for the solution of general sparse linear systems. Because of this link, the preconditioner defined by Equation (10) is referred to as algebraic additive Schwarz for the Schur complement. One advantage of using the assembled local Schur complements instead of the local Schur complements (like in the Neumann-Neumann [2, 6]) is that in the SPD case the assembled Schur complements cannot be singular (as $\mathcal{S}$ is SPD [4]).

The original idea of non-overlapping domain decomposition method consists into subdividing the graph into sub-graphs that are individually mapped to one processor. With this data distribution, each processor P_i can concurrently partially factorize it to compute its local Schur complement $\mathcal{S}_i$. This is the first computational phase that is performed concurrently and independently by all the processors. The second step corresponds to the construction of the preconditioner. Each processor communicates with its neighbors (in the graph partitioning sense) to assemble its local Schur complement $\bar{\mathcal{S}}_i$ and performs its factorization. This step only requires a few point-to-point communications. Finally, the last step is the iterative solution of the interface problem, Equation (5). For that purpose, parallel matrix-vector product involving $\mathcal{S}$, the preconditioner $M_\star$ and dot-product calculation must be performed. For the matrix-vector

product each processor P_i performs its local matrix-vector product involving its local Schur complement and communicates with its neighbors to assemble the computed vector to comply with Equation (7). Because the preconditioner, Equation (10), has a similar form as the Schur complement, Equation (7), its parallel application to a vector is implemented similarly. Finally, the dot products are performed as follows : each processor P_i performs its local dot-product and a global reduction is used to assemble the result. In this way, the hybrid implementation can be summarized by the above main three phases.

2.2 Dropping strategy to sparsify the preconditioner

The construction of the proposed local preconditioners can be computationally expensive because the dense matrices $\mathcal{S}_i$ should be factorized. We intend to reduce the storage and the computational cost to form and apply the preconditioner by using sparse approximation of $\bar{\mathcal{S}}_i$ in M_d following the strategy described by Equation (12). The approximation $\hat{\mathcal{S}}_i$ can be constructed by dropping the elements of $\bar{S}_i$ that are smaller than a given threshold. More precisely, the following symmetric dropping formula can be applied :

$$\tilde{\bar{s}}_{\ell j} = \begin{cases} 0, & \text{if } |\bar{s}_{\ell j}| \leq \xi(|\bar{s}_{\ell\ell}| + |\bar{s}_{jj}|), \\ \bar{s}_{\ell j}, & \text{otherwise}, \end{cases} \tag{12}$$

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

$$M_{sp} = \sum_{i=1}^{N} \mathcal{R}_{\Gamma_i}^T \left(\tilde{\bar{S}}_i\right)^{-1} \mathcal{R}_{\Gamma_i}. \tag{13}$$

We notice that such a dropping strategy preserves the symmetry in the symmetric case but it requires to first assemble $\bar{S}_i$ before sparsifying it.

Note : see ICNTL(21)=2 or 3 and RCNTL(11) for more details on how to control this dropping strategy

2.3 Sparse approximation based on partial $ILU(t,p)$

One can design a computationally and memory cheaper alternative to approximate the local Schur complements $S^{(i)}$. Among the possibilities, we consider in this report a variant based on the $ILU(t,p)$ [19] that is also implemented in pARMS [15].

The approach consists in applying a partial incomplete factorisation to the matrix $A^{(i)}$. The incomplete factorisation is only run on A_{ii} and it computes its ILU factors $\tilde{L}_i$ and $\tilde{U}_i$ using to the dropping parameter threshold t_{factor}.

$$pILU\ (A^{(i)}) \equiv pILU\ \begin{pmatrix} A_{ii} & A_{i\Gamma_i} \\ A_{\Gamma_i i} & A^{(i)}_{\Gamma_i\Gamma_i} \end{pmatrix} \equiv \begin{pmatrix} \tilde{L}_i & 0 \\ A_{\Gamma i}\tilde{U}_i^{-1} & I \end{pmatrix} \begin{pmatrix} \tilde{U}_i & \tilde{L}_i^{-1}A_{i\Gamma} \\ 0 & \tilde{S}^{(i)} \end{pmatrix}$$

where

$$\tilde{S}^{(i)} = A^{(i)}_{\Gamma_i\Gamma_i} - A_{\Gamma_i i}\tilde{U}_i^{-1}\tilde{L}_i^{-1}A_{i\Gamma_i}$$

The incomplete factors are then used to compute an approximation of the local Schur complement. Because our main objective is to get an approximation of the local Schur complement we switch to another less stringent threshold parameter t_{Schur} to compute the sparse approximation of the local Schur complement.

Such a calculation can be performed using a IKJ-variant of the Gaussian elimination [20], where the $\tilde{L}$ factor is computed but not stored as we are only interested in an approximation of $\tilde{S}_i$. This further alleviate the memory cost.

Contrary to Equation (13) (see Section 2.2), the assembly step can only be performed after the sparsification, which directly results from the ILU process. as a consequence, the assembly step is performed on sparse data structures thanks to a few neighbour to neighbour communications to form $\overline{\tilde{S}}^{(i)}$ from $\tilde{S}^{(i)}$. The preconditionner finally reads :

$$M_{sp} = \sum_{i=1}^{N} R_{\Gamma_i}^T \left(\overline{\tilde{S}}^{(i)} \right)^{-1} R_{\Gamma_i}. \tag{14}$$

Note : To use this method, you should put ICNTL(30)=2, the control parameters related to the ILUT method are : ICNTL(8), ICNTL(9), RCNTL(8) and RCNTL(9).

2.4 Exact vs. approximated Schur algorithm to build the precond

In a full MPI implementation, the domain decomposition strategy is followed to assign each local PDE problem (sub-domain) to one MPi process that works independently of other processes and exchange data using message passing. In that computational framework, the implementation of the algorithms based on preconditioners built from the exact or approximated local Schur complement only differ in the preliminary phases. The parallel SPMD algorithm is as follow :

1. Initialization phase :
   - Exact Schur : using the sparse direct solver, we compute at once the LU factorization of A_{ii} and the local Schur complement $S^{(i)}$ using $A^{(i)}$ as input matrix ;
   - Approximated Schur : using the sparse direct solver we only compute the LU factorization of A_{ii}, then we compute the approximation of the local Schur complement $\tilde{S}^{(i)}$ by performing a partial ILU factorization of $A^{(i)}$.
2. Set-up of the preconditioner :
   - Exact Schur : we first assemble the diagonal problem thanks to few neighbour to neighbour communications (computation of $\bar{S}^{(i)}$), we sparsify the assembled local Schur (i.e., $\hat{S}^{(i)}$) that is then factorized.
   - Approximated Schur : we assemble the sparse approximation also thanks to few neighbour to neighbour communications and we factorize the resulting sparse approximation of the assembled local Schur.
3. Krylov subspace iteration : the same numerical kernels are used. The only difference is the sparse factors that are considered in the preconditioning step dependent on the selected strategy (exact v.s. approximated).

From a high performance computing point of view, the main difference relies in the computation of the local Schur complement. In the exact situation, this calculation is performed using sparse direct techniques which make intensive use of BLAS-3 technology as most of the data structure and computation schedule are performed in a symbolic analysis phase when fill-in is analyzed. For partial incomplete factorization, because fill-in entries might be dropped depending on their numerical values, no prescription of the structure of the pattern of the factors can be symbolically computed. Consequently this calculation is mainly based on sparse BLAS-1 operations that are much less favorable to an efficient use of the memory hierarchy and therefore less effective in terms of their floating point operation rate. In short, the second case leads to fewer operations but at a lower computing rate, which might result in higher overall elapsed time in some situations. Nevertheless, in all cases the approximated Schur approach consumes much less memory as illustrated later on in this report.

Note : To choose between each strategy you should use ICNTL(30).

Références

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[2] 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.

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

[4] 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.

[5] C. Chevalier and F. Pellegrini. PT-SCOTCH : a tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8), 2008.

[6] 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.

[7] J. Drkošová, M. Rozložník, Z. Strakoš, and A. Greenbaum. Numerical stability of the GMRES method. *BIT*, 35 :309–330, 1995.

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

[9] A. Greenbaum. *Iterative methods for solving linear systems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[10] W. Hackbusch. *Multigrid methods and applications.* Springer, 1985.

[11] A. Haidar. *On the parallel scalability of hybrid solvers for large 3D problems*. Ph.D. dissertation, INPT, June 2008. TH/PA/08/57.

[12] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear system. *J. Res. Nat. Bur. Stds.*, B49 :409–436, 1952.

[13] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

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

[15] Z. Li, Y. Saad, and M. Sosonkina. pARMS : A parallel version of the algebraic recursive multilevel solver. Technical Report Technical Report UMSI-2001-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, 2001.

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

[17] A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations*. Numerical mathematics and scientific computation. Oxford science publications, Oxford, 1999.

[18] G. Radicati and Y. Robert. Parallel conjugate gradient-like algorithms for solving nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11 :223–239, 1989.

[19] Y. Saad. ILUT : a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1 :387–402, 1994.

[20] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2003. Second edition.

[21] Y. Saad and M. H. Schultz. GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comp.*, 7 :856–869, 1986.

[22] B. F. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1st edition, 1996.

3 Installation

3.1 Requirements

MAPHYS needs some libraries in order to be installed :

— A MPI communication library (INTELMPI MPICH2, OPENMPI, ...) ;
— A BLAS and LAPACK library (MKL of preference)
— The SCOTCH library for the partitioning of the adjacency graph of the sparse matrix ;
— The MUMPSor PASTIX library as sparse solver ;
— Optionnally the HWLOC library for a better fit with the hardware topology..

3.2 Compilation and installation

To compile and install MAPHYS, follow the six following steps :

— Install SCOTCH 5.xx or higher ;
— Install HWLOC ;
— Install MUMPS 4.7 or higher ;
— Uncompress the MAPHYS archives, copy the file Makefile.inc.example into Makefile.inc ;
— Set the variable of the Makefile.inc depending of your architecture and library (such as MPI and BLAS) available ;
— Choose the user's compiling options : arithmetic (real or complex..), solvers choice, in the Makefile.inc file ;
— Compile : **make all** in order to get the library and test program.

4 Sequence in which routines are called

The MAPHYS package provides one single static library *libmaphys.a*, which contains all the artihmetic versions specified in WITH_ARITHS (see *Makefile.inc* in the top directory of MAPHYS).

The calling sequence of MAPHYS is similar for all arithmetics, and may look like this for the DOUBLE PRECISION case :

```
...
Use DMPH_maphys_mod
Implicit None
Include "mpif.h"
...
Integer :: ierr
Type(DMPH_maphys_t) :: mphs ! one per linear system to solve
...
Call MPI_Init(ierr)
...
mphs%comm = ...  ! set communicator
mphs%job  = -1   ! request initialisation
Call DMPH_maphys_driver(mphs)
...
mphs%sym    = ... ! set matrix input
mphs%n      = ... ! set matrix input
mphs%nnz    = ... ! set matrix input
mphs%rows   = ... ! set matrix input
mphs%cols   = ... ! set matrix input
mphs%values = ... ! set matrix input
mphs%rhs    = ... ! set the right-hand side

...
mphs%icntl(10) = ... ! set the arguments (components of the structure)
mphs%job  = 6     ! request to perform all steps at once
Call DMPH_maphys_driver(mphs)
...
mphs%job  = -2     ! request finalisation
Call DMPH_maphys_driver(mphs)

Call MPI_Finalize(ierr)
```

To select another arithmetic, simply change the prefix **DMPH_** by **CMPH_** for the **COMPLEX** arithmetic ; or **SMPH_** for the **REAL** arithmetic ; or **ZMPH_** for the **DOUBLE COMPLEX** arithmetic.

For convenience purpose, MAPHYS also generates static libraries specific to one arithmetic only, namely :

— libcmaphys for the **COMPLEX** arithmetic ;
— libdmaphys for the **DOUBLE PRECISION** arithmetic ;
— libsmaphys for the **REAL** arithmetic ;
— libzmaphys for the **DOUBLE COMPLEX** arithmetic.

5 Application Programming Interface

5.1 General

5.1.1 Maphys instance

mphs is the structure that will describe one linear system to solve. You have to use one structure for each linear system to solve.

5.1.2 Arithmetic versions

The package is available in the four main arithmetics : REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX.

5.1.3 Version number

mphs%VERSION (string) gives information about the current MAPHYS version. It is set during the intialization step (mphs%JOB=-1)

5.1.4 Control of the four main phases

The main action performed by MAPHYS (*i.e.* when calling `*MPH_maphys_driver(mph)`) is specified by the integer mphs%JOB. The user must set it on all processes before each call to MAPHYS. Possible values of mphs%JOB are :

- **-1** initializes the instance. It must be called before any other call to MAPHYS. MAPHYS will nullify all pointers of the structure, and fill the controls arrays (mphs%ICNTL and mphs%RCNTL) with default values.
- **-2** destroys the instance. MAPHYS will free its internal memory.
- **1** perfoms the analysis. In this step, MAPHYS will pretreat the input matrix, compute how to distribute the global system, and distribute the global system into local systems.
- **2** performs the factorization assuming the data is distributed in the form of local systems. In this step, MAPHYS will factorize the interior of the local matrices, and compute their local Schur complements.
- **3** performs the preconditioning. In this step, MAPHYS will compute the preconditioner of the Schur complement system.
- **4** performs the solution. In this step, MAPHYS will solve the linear system. Namely, MAPHYS will reformulate the right-hand-side of the input linear system (mphs%RHS) into local system right-hand-sides, solve the Schur complement system(s), solve the local sytem
- **5** combines the actions from JOB=1 to JOB=3.
- **6** combines the actions from JOB=1 to JOB=4.

5.1.5 Control of parallelism

- **mphs%COMM** (integer) is the MPI communicator used by MAPHYS. It must be set to a valid MPI communicator by the user before the initialization step (mphs%JOB = -1) , and must not changed later.

— *In the current version, when the input system is initially centralized on the host (ICNTL(43) = 1), the MPI communicator must have a number of process that is a power of 2.*
— *In the current version, the working host process is fixed to the process with MPI rank 0 according to the chosen communicator.*

5.2 Input matrix, right-hand side and solution vectors

MAPHYS aims at solving a linear system of the form $\mathcal{A}x = b$, where $\mathcal{A}$ is a square real or complex non singular sparse matrix, b is a given right-hand side (RHS) vector, and x is the solution vector to be computed.

Section 5.2.1 presents the types of matrices MAPHYS can handle (through **mphs%SYM** control parameter) while Section 5.2.2 details the matrix (and subsequent RHS and solution vectors) formats MAPHYS can handle.

Note that the current version does not support multiple RHS.

5.2.1 Matrix type

— **mphs%SYM** (integer) specifies the input matrix type. It is accessed during the analysis and the solve steps (mphs%JOB == -1 and mphs%JOB == 4 respectively). MAPHYS does not modify its value during its execution. Possible values of mphs%SYM are :
  — **0 or SM_SYM_IsGeneral** means the input matrix is general, *i.e.* no assumption is made on his pattern.
  — **1 or SM_SYM_IsSPD** means the input matrix is symmetric positive definite (SPD) in real arithmetic (s/d). In complex arithmetic (z/c), this is an invalid value.
  — **2 or SM_SYM_IsSymmetric** means the input matrix is symmetric (even in complex).

In centralized input mode (Section 5.2.2.1), be aware that in the case of symmetric matrices (**mphs%SYM=1 or 2**), only “half” of the matrix (including diagonal) should be provided (because MAPHYS automatically fills the other off-diagonal half).

Unsymmetric matrices must be processed as general matrices (**mphs%SYM=0**). Although not recommended, symmetric matrices can also be processed in this mode but all coefficients must then be filled (lower and upper coefficients) in that case.

In the present version, MAPHYS does not support Hermitian or Hermitian Positive Definite matrices. Such matrices have to be processed as general matrices (**mphs%SYM=0**).

5.2.2 Matrix format

The matrix format is controlled by ICNTL(43). MAPHYS handles either a centralized assembled matrix format (ICNTL(43)=1) or a distributed subdomain format (ICNTL(43)=2) :

— ICNTL(43)=1 : the input matrix and RHS are supplied *centrally* on the host process (MPI process 0) in an *assembled form*. In this case, during the analysis step, MAPHYS performs a domain decomposition and distributes the matrix

on the MPI processes according to internal MaPHyS parallelization strategies. Section 5.2.2.1 further describes the expected input.
- ICNTL(43)=2 : the input matrix and RHS are supplied in a *distributed* way (each MPI process provides the entries of the matrix and RHS it has computed) in a *subdomain form* (multiple MPI processes may compute part of a coefficient – at their interface – that are then summed up by MAPHYS). This interface shall be very convenient for users having a parallel distributed application that already performs a domain decomposition. Indeed, each MPI process can provide the local matrices and RHS it has assembled on the subdomain it is associated to. Section 5.2.2.2 further describes the expected input.

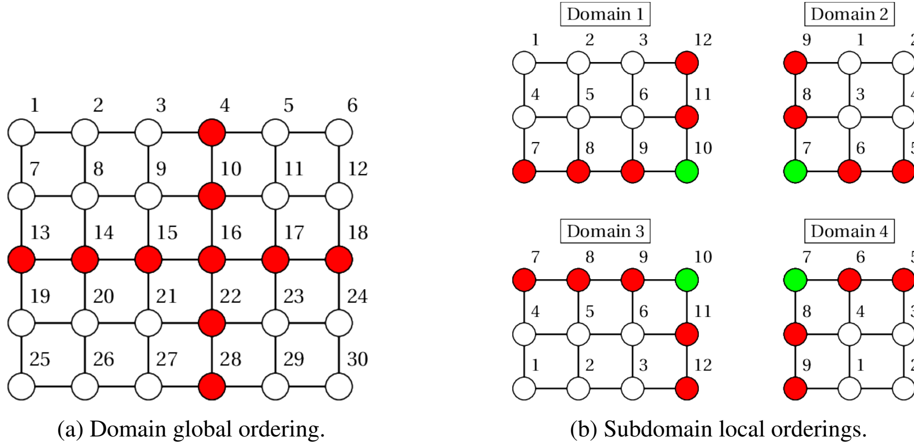
5.2.2.1 Centralized assembled matrix input In the centralized assembled matrix input mode (ICNTL(43)=1), the matrix is provided in coordinate format through mphs%SYM, mphs%N, mphs%NNZ, mphs%ROWS, mphs%COLS and mphs%VALUES. These components are accessed during the analysis step (until the matrix gets distributed) and the solve step (to compute statistics) by the host process. They are not altered by MAPHYS. In details :

- **mphs%SYM** (integer) specifies the type of symmetry (see Section 5.2.1)
- **mphs%N** (integer) specifies the order of the input matrix (hence $N > 0$)
- **mphs%NNZ** (integer) specifies the number of the entries in the input matrix (hence $NNZ > 0$)
- **mphs%ROWS** (integer array of size NNZ) lists the row indices of the matrix entries
- **mphs%COLS** (integer array of size NNZ) lists the column indices of the matrix entries
- **mphs%VALUES** (Real/Complex array of size NNZ) lists the values of the matrix entries

Note that, in the case of symmetric matrices (mphs%SYM=1 or 2), only half of the matrix should be provided. For example, only the lower triangular part of the matrix (including the diagonal) or only the upper triangular part of the matrix (including the diagonal) can be provided in ROWS, COLS, and VALUES. More precisely, a diagonal nonzero $\mathcal{A}_{ii}$ must be provided as $VALUES(k) = \mathcal{A}_{ii}$, $ROWS(k) = COLS(k) = i$, and a pair of off-diagonal nonzeros $\mathcal{A}_{ij} = \mathcal{A}_{ji}$ must be provided either as $\mathcal{A}(k) = \mathcal{A}_{ij}$ and $ROWS(k) = i$, $COLS(K) = j$ or vice-versa. Duplicate entries are summed. In particular, this means that if both $\mathcal{A}_{ij}$ and $\mathcal{A}_{ji}$ are provided, they will be summed.

The RHS and solution vector are handled as follows :

- **mphs%RHS** (Real/Complex array of lenght mphs%N) is the centralized right-hand-side of the linear system. It holds the right-hand side (RHS) allocated and filled by the user, on the host, before the solve step (mphs%JOB == 4,6). MAPHYS does not alterate its value.
- **mphs%SOL** (Real/Complex array of lenght mphs%N) is the centralized initial guess and/or the solution of the linear system. It must be allocated by the user before the solve step (mphs%JOB == 4,6).
  - Before the solve step, on the host, it holds the initial guess if the initial guess option is activated (mphs%icntl(23) == 1).



(a) Domain global ordering.

(b) Subdomain local orderings.

FIGURE 1 – Domain decomposition of a 6x5 grid performed by the application.

— After the solve step, on the host, it holds the solution.

5.2.2.2 Distributed subdomain interface (experimental and succeptible to change)

The distributed subdomain input mode (ICNTL(43)=2) has been especially designed for users who have a parallel distributed application that already performs a domain decomposition before calling MAPHYS. At this stage, it is functional, but the API is not stabilized and hence succeptible to change in future versions. We make the following assumptions :

1. the user has performed a domain decomposition with vertex separators such as in the example of Figure 1a ;
2. one MPI process is associated to one domain and assembles the coefficients of the matrix and the RHS arising from its domain (4 processes in the example of Figure 1) ;
3. each MPI process orders contiguously its local interior vertices prior to its interface vertices ranging from 1 to the local number of degrees of freedom (12 for subdomain 1 of Figure 1b).

In this context, instead of providing $\mathcal{A}$, x and b global data to MAPHYS, the user provides :

1. $\mathcal{A}_i$, x_i and b_i local data associated to domain i, where

$$\mathcal{A}_i = \begin{pmatrix} \mathcal{A}_{\mathcal{I}_i\mathcal{I}_i} & \mathcal{A}_{\mathcal{I}_i\Gamma_i} \\ \mathcal{A}_{\Gamma_i\mathcal{I}_i} & \mathcal{A}_{\Gamma_i\Gamma_i} \end{pmatrix}, b_i = \begin{pmatrix} b_{\mathcal{I}_i} \\ b_{\Gamma_i} \end{pmatrix}$$

consistently with Section 2.1. Each process hence fills the mphs%SYM, mphs%N, mphs%NNZ, mphs%ROWS, mphs%COLS, mphs%VALUES, mphs%RHS and possibly mphs%SOL as in 5.2.2.1 but with respect to the $\mathcal{A}_i$, x_i and b_i local

data it has assembled (*e.g.*, mphs%N = 12 for Domain 1 in Figure 1b) ; in particular, the numbering of rows mphs%ROWS and columns mphs%COLS are local consistently with the example provided in Figure 1b.

2. the interconnectivity pattern.

In the current MAPHYS API, the interconnectivity pattern is provided to MAPHYS in two steps. First we provide some data structures (allocated by the user) and then we call the dedicated `XMPH_distributed_interface` MAPHYS helper routine that provides those data structures to MAPHYS so that the solver can keep track of them (and rearrange them according to some internal constraints).

These data structures, which the user must specify on each process, are divided into the three categories :

1. related to the local matrix :
   — `myndof` : number of rows (columns) in the local matrix.
2. related to the interface :
   — `mysizeIntrf` : size of the local interface (:= size of local Schur).
   — `myinterface` : global numbering of the local interface
   Warning : the ordering of the interface nodes into `myinterface` has to match the ordering of the interface nodes into the local matrix.
3. related to the subdomain's neighbors :
   — `mynbvi` : number of subdomains neighbor of the current subdomain,
   — `myindexVi(1:mynbvi)` : array of indices of the neighbor MPI processes, holding a neighbor subdomain,
   — `myptrindexVi(1:mynbvi+1)` : index of the first vertex on the interface shared with neighbor sudomains in `mypindexintrf(:)`,
   — `myindexintrf(1:lenindintrf)` : list of the interface vertices shared with neighbors (in the same order than in `myindexVi(:)`).
   ```
   Example: for a neighbor v  in myindexVi(:), the node  shared are
   given by myindexintrf(myptrindexVi(v):myptrindexVi(v+1)-1)
   ```

In the example of Figure 1b, assuming each domain number i is mapped on the MPI process of rank $i-1$, the resulting data structures are as follows (arrays are written between square brackets, and elements within arrays are separated with commas) :

Data structure (user input)	Domain 1	Domain 2
`myndof`	12	9
`mysizeIntrf`	6	5
`myinterface`	[13, 14, 15, 16, 10, 4]	[18, 17, 16, 10, 4]
`mynbvi`	3	3
`myindexVi(:)`	[1, 2, 3]	[0, 2, 3]
`myptrindexVi(:)`	[1, 4, 8, 9]	[1, 4, 5, 8]
`myindexintrf(:)`	[4, 5, 6, 1, 2, 3, 4, 4]	[3, 4, 5, 3, 1, 2, 3]

Data structure (user input)	Domain 3	Domain 4
`myndof`	12	9
`mysizeIntrf`	6	5
`myinterface`	[13, 14, 15, 16, 22, 28]	[18, 17, 16, 22, 28]
`mynbvi`	3	3
`myindexVi(:)`	[0, 1, 3]	[0, 1, 2]
`myptrindexVi(:)`	[1, 5, 6, 9]	[1, 2, 5, 8]
`myindexintrf(:)`	[1, 2, 3, 4, 4, 4, 5, 6]	[3, 1, 2, 3, 3, 4, 5]

Once these data structures have been filled on each process, the `XMPH_distributed_interface` MAPHYS helper routine must be called. In Fortran, the call is performed as follows :

```
CALL XMPH_distributed_interface(mphs, myndof,
    mysizeintrf, myinterface, mynbvi, myindexvi,
    myptrindexvi, myindexintrf)
```

where the X character of the subroutine has to be replaced by S,D,C or Z, depending on the chosen arithmetic, and mphs is the MaPHyS instance.

The function to be called in C is :

```
xmph_distributed_interface_c(XMPH_maphys_t_c*
    maphys_par, int myndof, int mysizeintrf, int*
    myinterface, int mynbvi, int* myindexvi, int*
    myptrindexvi, int* myindexintrf)
```

5.3 Writing a matrix to a file

The user may get the preprocessed input matrix (assembled and eventually symmetrised in structure) by setting the string mphs%**WRITE_MATRIX** (string) on process 0 before the analysis step (mphs%JOB=1). This matrix will be written during the analysis step on the file mphs%**WRITE_MATRIX** in matrix market format.

5.4 Using two levels of parallelism inside MAPHYS

The current release of MAPHYS supports two levels of parallelism (MPI+threads). The MPI level is inherent to MAPHYS and cannot be disabled. By default, only the MPI level of parallelism is enabled and the user can optionnally furthermore activate multithreading using the following control parameters :

- ICNTL(42) = 1 : activate the multithreading inside MAPHYS ;
- ICNTL(37) = number of nodes ;
- ICNTL(38) = number of CPU cores on each nodes ;
- ICNTL(39) = number of MPI processes ;
- ICNTL(40) = number of Threads that you want to use inside each MPI process.

You can also set ICNTL(36) in order to specify the binding you want to use inside MAPHYS as follow :

— 0 = do not bind ;
— 1 = thread to core binding ;
— 2 = group binding.

note 1 : Be carreful that if you use more threads than cores available no binding will be perform.

note 2 : Depending on the problem you want to solve it could be better to use more MPI processes than using threads you can found a study on some problems in Stojce Nakov PHD thesis.

6 List of control parameters

The execution of MAPHYS can be controlled by the user through the 2 arrays **mphs%ICNTL** and **mphs%RCNTL**. When an execution control should be expressed as an integer (or an equivalent), this control is a entry of **mphs%ICNTL** ; and when an execution control should be expressed as a float (or an equivalent), this control is a entry of **mphs%RCNTL**.

6.1 ICNTL

mphs%ICNTL is an integer array of size **MAPHYS_ICNTL_SIZE**.

ICNTL(1) Controls where are written error messages. Fortran unit used to write error messages. Negative value means do not print error messages.
— Default = 0 (stderr)

ICNTL(2) Controls where are written warning messages. Fortran unit used to write warning messages. Negative value means do not print warning messages.
— Default = 0 (stderr)

ICNTL(3) Controls where are written statistics messages. Fortran unit used to write statistics messages. Negative value means do not print statistics.
— Default = 6 (stdout)

ICNTL(4) Controls the verbosity of maphys.
— ≤ 0 = do not print output
— 1 = print errors.
— 2 = print errors, warnings & main statistics
— 3 = print errors, warnings & several detailled statistics
— 4 = print errors, warnings & verbose on detailled statistics
— > 4 = debug level
— Default = 3

ICNTL(5) Controls when to print list of controls (?cntl). This option is useful to check if a control is taken into account.
— 0 = never print
— 1 = print once at the begining of analysis
— 2 = print at the begining of each step.
— Default = 0

ICNTL(6) Controls when to print list of informations (?info*). This option is useful while performing benchmarks.
— 0 = never print
— 1 = print once (at the end of solve)
— 2 = print at the end of each step
— Default = 0

ICNTL(7) UNUSED in current version

ICNTL(8) controls the Partial ILUT solver. level of filling for L and U in the ILUT method. (that is the maximun number of entries per row in the LU factorization.) This parameter is only relevant while using ILUT and is accessed

during factorisation by each process. User must set this parameter while using ILUT before the factorisation. **See also :** ICNTL(30), ICNTL(9), RCNTL(8) and RCNTL(9). **Warning : the default value is a bad value.**
— Default = -1
Note : For more details on ILU method, you may refer to Section 2.3

ICNTL(9) controls the Partial ILUT solver. level of filling for the Schur complement in the ILUT method (that is the maximun number of entries per row in the Schur complement). This parameter is only relevant while using ILUT and is accessed during factorisation by each process. User must set this parameter while using ILUT before the factorisation. **See also :** ICNTL(30),ICNTL(8), RCNTL(8) and RCNTL(9). **Warning : the default value is a bad value.**
— Default = -1
Note : For more details on ILU method, you may refer to Section 2.3

ICNTL(10 :12) UNUSED in current version

ICNTL(13) the sparse direct solver package to be used by MAPHYS. It is accessed during the factorization and the the preconditionning step by each MPI process.
— 1 = use MUMPS
— 2 = use PaSTiX
— 3 = use multiple sparse direct solvers. See ICNTL(15) and ICNTL(32)
— Default = the first available starting from index 1.

ICNTL(14) UNUSED in current version

ICNTL(15) the sparse direct solver package to be used by MAPHYS for the computing the preconditioner. It is accessed during the preconditioning step, by each MPI process.
— 1 = use MUMPS
— 2 = use PaSTiX
— Default = value of ICNTL(13)

ICNTL(16 :19) UNUSED in current version

ICNTL(20) controls the 3rd party iterative solver used.
— 0 = Unset
— 1 = use Pack GMRES
— 2 = use Pack CG
— 3 = use Pack CG if matrix is SPD, Pack GMRES otherwise
— Default = 3

ICNTL(21) The strategy for preconditionning the Schur. **Warning : If (ICNTL(30)=2) and (ICNTL(21)/=4) then ICNTL(21) must be 3**.
— 1 = Preconditioner is the dense factorization of the assembled local schur.
— 2 = Preconditioner is the sparse factorization of the assembled local schur, you have to specify the threshold using RCNTL(11).
— 3 = Preconditioner is the factorization of a sparse matrix. Where the sparse matrix is the assembly of the approximated local schur computed with the Partial Incomplete LU (PILUT), you have to specify the threshold using RCNTL(9).

— 4 = No Preconditioner.
— 5 = autodetect, order of preferences is $1 > 2 > 3 > 4$.
— 10 = Preconditioner is the dense factorization of the assembled local schur with additive coarse grid correction. You have to set the number of of eigenvalues per subdomain to keep with ICNTL(33). **Warning : only works for a symmetric positive definite matrix (mphs%SYM = 2)**. **Warning : MUMPS is required to use this preconditioner**.
— Default = 2

Note : For more details on dropping strategy (ICNTL(21) = 2 or 3), you may refer to Section 2.2, the parameter RCNTL(8)/RCNTL(9) if ICNTL(21)=2/3 respectivly

ICNTL(22) controls the iterative solver. Determines which orthogonalisation scheme to apply.
— 0 = modified Gram-Schmidt orthogonalization (MGS)
— 1 = iterative selective modified Gram-Schmidt orthogonalization (IMGS)
— 2 = classical Gram-Schmidt orthogonalization (CGS)
— 3 = iterative classical Gram-Schmidt orthogonalisation (ICGS)
— Default = 3

ICNTL(23) controls the iterative solver. Controls whether the user wishes to supply an initial guess of solution vector in (mphs%SOL). It must equal to 0 or 1. It is accessed by all processes during the solve step.
— 0 = No initilization guess (start from null vector)
— 1 = User defined initialization guess
— Default = 0

ICNTL(24) controls the iterative solver. It define the maximumn number of iterations (cumulated over restarts) allowed. It must be equal or larger than 0 and be the same on each process.
— 0 = means the parameter is unset, MAPHYS will performs at most 100 iterations
— > 0 = means the parameter is set, MAPHYS will performs at most ICNTL(24) iterations
— Default = 0

ICNTL(25) controls the iterative solver. Controls the strategy to compute the residual at the restart, it must be the same on each process. It is irrelevant while the iterative solver is CG (ICNTL(20) = 2,3)
— 0 = A recurrence formula is used to compute the residual at each restart, except if the convergence was detected using the Arnoldi residual during the previous restart.
— 1 = The residual is explicitly computed using a matrix vector product
— Default = 0

ICNTL(26) controls the iterative solver, it is the restart parameter of GMRES. Controls the number of iterations between each restart. It is irrelevant while the iterative solver is CG (ICNTL(20) = 2,3).
— 0 = means the parameter is unset, MAPHYS will restart every 100 iterations

— > 0 = means the parameter is set, MAPHYS will restart every ICNTL(24) iterations
— Default = 0

ICNTL(27) controls the iterative solver, it is the method used to perform multiplication between the Schur complement matrix and a vector. It should be set before the solving step on each process. It can be set on each process before the preconditioning step to allow possible memory footprint reduction (peak and usage).
— -1 = Value unset (error)
— 0 = automatic (1 if possible, 2 if not.)
— 1 = Product of schur complement with a vector is performed explicitly. That is, the Schur complement is stored in a dense matrix, and a BLAS routine is called to perform the product.
— 2 = Product of schur complement with a vector is performed implicitly. That is, we do not use a schur complement stored in a dense matrix, to perform the product. Indeed, a "solve" and several sparse matrix/vector products are used to perform the product. This is done by using the definition schur complement, which is : $S = A_{i,\Gamma}.A_{i,i}^{-1}.A_{i,\Gamma} - A_{\Gamma,\Gamma}$ This option reduce the memory footprint by substituing in memory the schur complement by the preconditioner.
— Default = 0

ICNTL(29) UNUSED in current version

ICNTL(30) controls how to compute the schur complement matrix or its approximation. It is accessed during the factorisation step by each process. Each processor must have the same value (current version do not check it). Valid values are :
— 0 = use the schur complement returned by the sparse direct solver package.
— 2 = approximate computation done using a modifyied version of partial ILUT (PILUT). In this case, user must set ICNTL(8), ICNTL(9), RCNTL(8) and RCNTL(9) the parameters of PILUT. This option force the preconditioner choice (ICNTL(21) = 3). *Note : For more details on ILU method, you may refer to Section 2.3*
— Default = 0

Note : For more details on exact and approximated Schur, you may refer to Section 2.4

ICNTL(31) UNUSED in current version

ICNTL(32) the sparse direct solver package to be used by MAPHYS for the computing the local schur and perform the local factorisation. It is accessed during the factorization step, by each MPI process.
— 1 = use MUMPS
— 2 = use PaSTiX
— Default = value of ICNTL(13)

ICNTL(33) Number of eigenvalues per subdomain to keep for the coarse grid correction. Ignored if ICNTL(21) $\neq 10$.

— Default = 10

ICNTL(36) Is the control paramater that specify how to bind thread inside MAPHYS.
- 0 = Do not bind ;
- 1 = Thread to core binding ;
- 2 = Grouped bind.
- Default = 2

ICNTL(37) In 2 level of parallelism version, specifies the number of nodes. It is only useful if ICNTL(42) > 0.
- Default = 0

ICNTL(38) In 2 level of parallelism version, specifies the number of cores per node. It is only useful if ICNTL(42) > 0.
- Default = 0

ICNTL(39) In 2 level of parallelism version, specifies the number of threads per domains. It is only useful if ICNTL(42) > 0.
- Default = 0

ICNTL(40) In 2 level of parallelism version, specifies the number domains. It is only useful if ICNTL(42) > 0.
- Default = 0

ICNTL(42) Is the control paramater that active the 2 level of parallelism version.
- 0 = Only MPI will be use.
- 1 = Activate the Multithreading+MPI ;
- Default = 0.

Note : If the 2 level of parallelism are activated you have to set the parameters ICNTL(36), ICNTL(37), ICNTL(38), ICNTL(39) and ICNTL(40)

ICNTL(43) specifies the input system (centralized on the host, distributed, ...).
- 1 = The input system is centralized on host. the input matrix and the right-hand member are centralized on the host. User must provides the global system by setting on the host :
  - the global matrix before the analysis step : (mphs%rows,mphs%cols, etc.)
  - the global right-hand side before the solve step : (mphs%rhs)
- 2 = The input system is distributed the input matrix and the right-hand side are distributed. User must provides on each MPI process the local system by setting :
  - the local matrix before the analysis step : (mphs%rows,mphs%cols, etc.)
  - the local right-hand side before the solve step : (mphs%rhs)
  - the data distribution before the analysis step : mphs%lc_domain

  Warning : see directory gendistsys/*testdistsys* for an example.
- Default = 1

ICNTL(45) controls the local output after analysis. If set to 1, it allows to perform a dump of the local matrices, rhs and domain data after analysis. The files will be outputed in the working directory (*i.e. where the code is called*). This option is unactivated by default.

6.2 RCNTL

mphs%RCNTL is an integer array of size **MAPHYS_RCNTL_SIZE**.

RCNTL(8) specifies the threshold used to sparsify the LU factors while using PILUT. Namely, all entries (i,j) of the U part such as : $|U(i,j)|/(|U(i,i)|) \leq RCNTL(8)$ are deleted. It is only relevant and must be set by the user before the factorisation step while using PILUT. **See also :** ICNTL(30), ICNTL(8), ICNTL(9) andRCNTL(9). **Warning : the default value is a bad value.**
— Default = 0.e0
Note : For more details on ILU method, you may refer to Section 2.3

RCNTL(9) specifies the threshold used to sparsify the schur complement while computing it with PILUT. Namely, all entries (i,j) of the schur S such as : $|S(i,j)|/(|S(i,i)|) \leq RCNTL(9)$ are deleted. It is only relevant and must be set by the user before the factorisation step while using PILUT. See also ICNTL(30), ICNTL(8), ICNTL(9) and RCNTL(9) **Warning : the default value is a bad value.**
— Default = 0.e0
Note : For more details on ILU method, you may refer to Section 2.3

RCNTL(11) specifies the threshold used to sparsify the Schur complement while computing a sparse preconditioner. It is only useful while selecting a sparse preconditioner. (ie ICNTL(21) = 2 or 3) It must be above 0. All entries (i,j) of the assembled schur S such as : $|S(i,j)|/(|S(i,i)|+|S(j,j)|) \leq RCNTL(11)$ are deleted.
— Default = 1.e-4
Note : For more details on dropping strategy (ICNTL(21) = 2 or 3), you may refer to Section 2.2

RCNTL (15) specifies the imbalance tolerance used in Scotch partitionner to create the subdomains. more high it is more Scotch will allow to have imbalance between domains in order to reduce the interface (so the Schur)
— Default = 0.2

RCNTL (21) specifies the threshold τ for the stopping criterion within the iterative solve step so that $\frac{||A.x-b||_2}{||b||_2} \leq \tau$. It must be above 0. Note that the iterative solver actually ensures that $\frac{||S.x_\Gamma-b_\Gamma||_2}{||b||_2} \leq \tau$. Because the solution computed on the interiors (x_I) is computed with a direct solver, we shall have $||S.x-b_\Gamma||_2 = ||A.x-b||_2$ (at least in exact arithmetics), hence, $\frac{||S.x_\Gamma-b_\Gamma||_2}{||b||_2} \leq \tau$ shall be equivalent to $\frac{||A.x-b||_2}{||b||_2} \leq \tau$ (at least in exact arithmetics). Note that MAPHYS also explicitly computes $\frac{||A.x-b||_2}{||b||_2}$ after the solve step (see RINFOG(4) and RINFOG(2)).
— Default = 1.e-5

6.3 Compatibility table

Some values of the control parameters cannot be used at the same time, here is a table that summarizes the conflict between the various control parameters :

Control parameters values	uncompatibility parameters
ICNTL(30) = 0	ICNTL(8), ICNTL(9), RCNTL(8) and RCNTL(9) are unused
ICNTL(13) /= 3	ICNTL(15) and ICNTL(32) are unused
ICNTL(30) = 2 and ICNTL(21) /= 4	ICNTL(21) must be forced to 3
ICNTL(20) = 3 or ICNTL(20) = 2	ICNTL(25) is unused
ICNTL(21) = 2 or ICNTL(21) = 3	RCNTL(11) is unused

7 List of information parameters

Values return to the user, so that he/she had feed-back on execution.

7.1 Information local to each processor

7.1.1 IINFO

IINFO(1) The status of the instance.
— 0 = means success
— <0 = means failure
— >0 = means warning

IINFO(2) UNUSED in current version

IINFO(3) flag indicating the strategy used to partition the input linear system. It is available after the analysis step.

IINFO(4) Local matrix order. It is available after the analysis step.

IINFO(5) Size in MegaBytes used to store the local matrice and its subblocs. It is available after the analysis step.

IINFO(6) Number of entries in the local matrix order. It is available after the analysis step.

IINFO(7) flag indicating which sparse direct solver was used to compute the factorization. It is available after the factorisation step.

IINFO(8) flag indicating how the Schur was computed. It is available after the factorisation step

IINFO(9) Size in MegaBytes used to stores the factors of the local matrix (if available). It is available after the factorisation step.
— -1 means the data is unvailable
— >0 is the size in megabits

IINFO(10) the number of static pivots during the factorization. It is available after the factorisation step.
— -1 means the data is unvailable
— >0 is the number of pivots

IINFO(11) Local Schur complement order. It is available after the analysis step.

IINFO(12) in MegaBytes used to stores the Schur complement or it approximation. It is available after the factorisation step.

IINFO(13) Number of entries in the approximation of the local Schur, only available with ILU. It is available after the factorisation step.

IINFO(14) Preconditioner strategy . The flag indicating which strategy was used to compute the preconditioner. available after the preconditioning step. (preconditioner is local dense, local sparse, etc.)]

IINFO(15) . indicates which sparse direct solver was used to compute the preconditioner. It is available after the preconditioning step.

IINFO(16) The order of the preconditioner. It is available after the preconditioning step.

IINFO(17) Size in megabits used to stores the preconditioner (if available). It is available after the preconditioning step.

IINFO(18) the number of static pivots done while computing the preconditioner . It is available after the preconditioner step.
— -1 means the data was unvailable
— > 0 is the number of pivots

IINFO(19) the percentage of kept entries in the assembled Schur complement to build the local part of the preconditionner. It is available after the preconditioner step with sparse local preconditioners (ICNTL(21) == 2).
— -1 means the data was unvailable
— >0 is the value

IINFO(20) specifies the iterative solver. The flag indicating which iterative solver was used to solve the Schur complement system. It is available after the solution step.

IINFO(21) specifies the Schur matrix vector product. The flag indicating how was performed the schur matrix vector product. (Implicit/Explicit) It is available after the solution step.

IINFO(22) On the local system, size in megabytes, of internal data allocated by the sparse direct solver. For MUMPS it is IINFOG(19). It is available after the factorization step.

IINFO(23) On the local Schur system, size in megabytes, of internal data allocated by the sparse direct solver to compute the preconditioner. For MUMPS it is IINFOG(19). It is available after the preconditioning step with sparse preconditioners.

IINFO(24) estimation in MegaBytes of the peak of the internal data allocated by the MaPHyS Process. It only takes into account the allocated data with the heaviest memory cost (Schur, preconditioner, factors of local system and solve).

IINFO(25) estimation in MegaBytes of the internal data allocated by the MaPHyS Process. It only takes into account the allocated data with the heaviest memory cost (Schur, preconditioner, factors of local system and solve).

IINFO(26) Upper bound for memory usage of the iterative solver. It is only available after the solution step.

IINFO(27) Upper bound for memory peak of the iterative solver. It is only available after the solution step.

IINFO(28) memory used in MAPHYSbuffers. It is available after the analysis step.

IINFO(29) memory peak in MAPHYSbuffers. It is available after the analysis step.

IINFO(30) The internal memory usage of the sparse direct solver library (MUMPS, PaSTiX,etc.). It is available after the factorisation step. Note that an external PaSTiX instance may influence this data.

IINFO(31) The internal memory peak of the sparse direct solver library (MUMPS, PaSTiX,etc.). It is available after the factorisation step. Note that an external PaSTiX instance may influence this data.

IINFO(32) The identifier of the node (from 1 to the number of nodes) It is available after the initialization step.

IINFO(33) The number of domains associated to the node It is available after the initialization step.

IINFO(34) The amount of memory used on the node in MegaBytes. It is updated at the end of each step. Note that an external PaSTiX instance may influence this data.

IINFO(35) The amount of memory used on the node in MegaBytes. It is updated at the end of each step. Note that an external PaSTiX instance may influence this data.

IINFO(36) The number of non zero in the schur after the factorisation of the local matrix. It is available after the preconditionning step. Note that is only available for the PaSTiX version.

IINFO(37) Effective memory usage of the iterative solver. It is only available after the solution step.

7.1.2 RINFO

RINFO(1) is UNUSED in current version.

RINFO(2) Norm 1 of the Schur stored in dense matrix. It is only available is the dense local preconditioner was selected (ICNTL(21) == 1). It is computed during the preconditioning step and is available on each processor.

RINFO(3) Reciproque of the condition number of the local assembled dense Schur matrix. It is only available is the dense local preconditioner was selected (ICNTL(21) == 1). It is computed during the preconditioning step and is available on each processor.

RINFO(4) Execution Time of the analysis step. It is available after the analysis step.

RINFO(5) Execution Time of the factorisation step. It is available after the factorisation step.

RINFO(6) Execution Time of the precond step. It is available after the precond step.

RINFO(7) Execution Time of the solution step (include iterative solution of the Schur and backsolve to compute interior). It is available after the solution step.

RINFO(8) Time to preprocess the input matrix. It is only available on the host after the analysis step.

RINFO(9) Time to convert the input system into local system. It is available after the analysis step.

RINFO(10) Time to extract subblocks of the local matrix. It is available after the analysis step.

RINFO(11) Time to factorize the local matrix. It is available after the factorisation step. If the Schur complement is computed by the sparse direct solver, It includes the time to compute the Schur complement matrix.

RINFO(12) Time to compute the Schur complement or its approximation. It is available after the factorisation step. If the Schur complement is computed by the sparse direct solver, this is zero. Otherwise, it is the time to compute the Schur complement from factorization Or to estimate the Schur complement with ILUT.

RINFO(13) Time to assemble the Schur complement or its approximation. It is available after the preconditioning step.

RINFO(14) Time to factorize the assembled Schur complement (the preconditioner). It is available after the preconditioning step.

RINFO(15) the solve step, execution time to distribute the global right-hand side

RINFO(16) In the solve step, execution time to generate right-hand side of the iterative solver.

RINFO(17) In the solution step, execution time of the iterative solver (to solve the interface)

RINFO(18) In the solution step, execution time of the sparse direct solver (to solve the interiors)

RINFO(19) In the solution step, execution time to gather the solution

RINFO(20) Sum of the execution times of the 4 steps. It is available after the solution step.

RINFO(21) Starting/Total execution time from the begining of the analysis to the end of the solve. Before the solve step, it contains the starting time of execution. After the solve step, it contains the total execution time. Warning : it includes the time between each steps (idle time due to synchronisations).

RINFO(22) The estimated floating operations for the elimination process on the local system. It is only available after the factorisation.

RINFO(23) The floating operations for the assembly process on the local system. It is only available after the factorisation.

RINFO(24) The floating operations for the elimination process on the local system. It is only available after the factorisation.

RINFO(25) The estimated floating operations for the elimination process to obtain the sparse preconditioner. It is only available after the preconditioning step with a sparse preconditioner.

RINFO(26) The floating point operations for the assembly process to obtain the sparse preconditioner. It is only available after the preconditioning step with a sparse preconditioner.

RINFO(27) The floating point operations for the elimination process to obtain the factorisation of the sparse preconditioner. It is only available after the preconditioning step with a sparse preconditioner.

RINFO(28) The time spent to perform communications while calling the iterative solver. It is cumulated over the iterations.

RINFO(29) Total time spent to perform the matrix-vector products in the iterative solver (where the matrix is the Schur complement). It includes the time to synchronise the vector.

RINFO(30) Total time spent to apply the preconditioner in the iterative solver. This includes the time to synchronise the vector.

RINFO(31) Total time spent to perform scalar products in the iterative solver. It includes the time to reduce the result.

RINFO(32) to RINFO(35) Fields unused in current version.

7.2 Information available on all processes

7.2.1 IINFOG

IINFOG (1) the instance status. **Warning** : In current version, there is no guarantee that on failure, IINFOG(1) is properly set (see section 8)
- < 0 = an Error
- 0 = instance is in a correct state
- -1 = an error occured on processor IINFOG(2)

IINFOG (2) the additionnal information if an error occured. **Warning** : In current version, there is no guarantee that on failure, IINFOG(2) is properly set (see section 8)
- If IINFOG(1) = -1, specifies which processor failed (from 1 to np)

IINFOG (3) order of the input matrix.

IINFOG (4) input matrix's number of entries. It is only available on processor 0.

IINFOG (5) number of iterations performed by the iterative solver. It is set during the solving step.

7.2.2 RINFOG

RINFOG (1) UNUSED in current version.

RINFOG (2) Value of $\frac{||A.x-b||_2}{||b||_2}$ estimated in centralized form, where $||.||_2$ is the euclidean norm, A the input matrix, x the computed solution, b the given right-hand-side. Note that this computation is performed explicitly by MAPHYS(out of the iterative solver) on the global system. It is available after the *solve* step.

RINFOG (3) the backward error of the Schur system. It is computed during the solve step and is available on each processor.

RINFOG (4) Value of $\frac{||A.x-b||_2}{||b||_2}$ estimated in distributed form. Note that this computation is performed explicitly by MAPHYS(out of the iterative solver) on the global system. Note also that it is the same computation as the one performed for RINFOG(2), except that is is performed in parallel.

8 Error diagnostics

In this current version, on errors MAPHYS will :

1. print a trace if it was compiled with the option flag **-DMAPHYS_BACKTRACE_ERROR**
2. exit by calling MPI_Abort on the MPI communicator mphs%COMM

Messages in the trace look like :

```
[00002] ERROR: dmph\_maphys\_mod.F90:line   278. Check Failed
```

Here it means that a check failed on process rank 2 in file 'dmph_maphys_mod.F90' at line 278.

9 Examples of use of MaPHyS

The is some driver that we provide to use some examples of maphys

9.1 A matrix problem

Those examples of MAPHYSare stored in the directory *examples*. There is the drivers using MAPHYS :

`<Arithms>_examplekv` is a drivers that use input.in file.

Where `<Arithms>` is one of the different arithmetic that could be used.

— **smph** for real arithmetic
— **dmph** for double precision arithmetic
— **cmph** for complex arithmetic
— **zmph** for complex double arithmetic

Only the test with the arithmetic you have choose during the compilation will be available.

You can see some examples of this files in the template available in the same directory *examples* such as :

— **real_bcsstk17.in** is the input file to solve $A.x = b$ with A the bcsstk17 matrix (real, SPD, from Matrix Market), b is stored in file, x the solution should be the vector ones.
— **real_bcsstk17_noRHS.in** is the input file to solve $A.x = b$ with A the bcsstk17 matrix (real, SPD, from Matrix Market), b generated such as solution x is a pseudo random vector.
— **complex_young1c.in** is the input file to solve $A.x = b$ with A the young1c matrix (complex,unsymmetric, from Matrix Market), b generated such as solution x is a pseudo random vector.

In order to launch these tests, you need to use the command :

```
mpirun -np <nbproc> ./<Arithms>_examplekv <ExampleFile>
```

Where `ExampleFile` is a string that contains the link to the input file you want to use and `<nbproc>` the number of processus you want to use.

For example you can use the following command :

```
mpirun -np 4 ./smph_examplekv real_bcsstk17.in
```

Warning : Make sure that <nbproc> is a power of two because maphys is based on dissection method.

9.1.1 How to write an input file

The input.in file are file in free format. An example can be found in `template.in`. It should be written as follow.

All lines begining with # are a comment line.
Empty lines are possible.
Setting parameter is based on key/value in the following format :
`KEY = VALUE`

9.1.2 System description

- Key : `MATFILE`
  Fields : `[I,J,V,N,NNZ]`
  Type : `string`
  Description : Path to the matrix file Constrains : supported file extension are :
  - .mtx or .mm (Matrix Market)
  - .ijv (coordinate format)
  - .rsa (Harwell Boeing)

- Key : `SYM`
  Fields : `[SYM]`
  Type : `Integer(kind=4)`
  Description : Specifies the matrix symmetry, if given, it supperseed the symmertry read in "MATFILE".
  Constrains : supported value are :
  - 0 (General)
  - 1 (SPD)
  - 2 (symmetric)

- Key : `RHSFILE`
  Fields : `[RHS]`
  Type : `string`
  Description : Path to the file containing the right-hand-side.
  Constrains : supported file extension are :
  - .ijv (coordinate format)

  Note 1 : If file is invalid, or not set, the right-hand-side is generated from a pseudo random solution

- Key : `INITGESS`
  Fields : `[SOL]`
  Type : `string`
  Description : Path to the file containing the initial guess.
  Constrains : supported file extension are :
  - .ijv (coordinate format)

  Note 1 : User must also set ICNTL(23) to 1 to activate this option
  Note 2 : If not set, initial guess is vector Null

- Key : `OUTRHSFILE`
  Fields : `[RHS]`
  Type : `string`
  Description : Path to the file where to write the rhs.
  Constrains : must be non empty, writtent only by process 0.

- Key : `OUTSOLFILE`
  Fields : `[SOL]`

Type : `string`
Description : Path to the file where to write the solution.
Constrains : must be non empty, writtent only by process 0.

9.1.3 System description

— Key : `JOB`
Fields : `[JOB]`
Type : `Integer(kind=4)`
Description : set the job to perform
Contraints : are those of JOB.

— Key : `ICNTL([0-9]+)`
Fields : `[ICNTL([0-9]+)]`
Type : `Integer(kind=4)`
Description : set an ICNTL parameter
Contraints : are those of ICNTL.

— Key :`RCNTL([0-9]+)`
Fields : `[RCNTL([0-9]+)]`
Type : `Real(kind=8)`
Description : set an RCNTL parameter
Contraints : are those of RCNTL.

Note : For more details on possible value for ICNTL and RCNTL, you may refer to Section 6

9.2 A cube generation problem

An other example that we provide is a cubes generations problem that you can have using the command line :

```
make testdistsys
```

Those examples of MAPHYSare stored in the directory *examples*. There is the drivers using MAPHYS :

`<Arithms>_testdistsys` is a drivers that use input.in file.

Where `<Arithms>` is one of the different arithmetic that could be used.

— **smph** for real arithmetic
— **dmph** for double precision arithmetic
— **cmph** for complex arithmetic
— **zmph** for complex double arithmetic

Only the test with the arithmetic you have choose during the compilation will be available.

In order to launch these tests, you need to use the command :

```
mpirun -np <nbproc> ./<Arithms>_testdistsys <CubeFile> <InputFile>
```

Where `CubeFile` is a string that contains the link to the cube description file you want to use and `InputFile` is a string that contains the link to the input file you want to use and `<nbproc>` the number of processus you want to use.

For example you can use the following command :

```
mpirun -np 4 ./smph_testdistsys cube.in dmph.in
```

Warning : Make sure that <nbproc> is a power of two because maphys is based on dissection method.

The `InputFile` is a file that is describe as in section xxx

The `CubeFile` is a file that is describe as follow :

```
10   # size_x
10   # size_y
10   # size_z
0    # problem
0    # an
0.d0 # p1
0.d0 # p2
```

Where `size_x`, `size_y` and `size_z` are the cube dimensions.

`problem` is ..

`an` is ..

`p1` and `p2` are..

9.3 How to use MAPHYS in an other code

9.3.1 Fortran example

Here is a fortran example of a driver using MAPHYS :

```
#include "mph_defs_f.h"
#include "mph_macros_f.h"
#define JOB_INIT -1
#define JOB_END -2
#define USE_COMM_WORLD -987654


Program Call_maphys

  !* Modules & co. *!
  Use DMPH_maphys_mod
  Implicit None
  Include 'mpif.h'
  Integer        :: ierr
```

```
Integer        :: myid
Integer        :: job
Integer        :: nbdom
Integer        :: i

! Arrays
Real   (KIND=8) , pointer :: sol(:)
Real   (KIND=8) , pointer :: rhs(:)
Real   (KIND=8) , pointer :: a(:)
Integer, pointer :: irn(:)
Integer, pointer :: jcn(:)

! Matrix description
Integer             :: n
Integer             :: nnz
Integer             :: sym

! Derived types
Type(DMPH_maphys_t) :: id

Call MPI_INIT( ierr )
if (ierr /= MPI_SUCCESS) ierr = -1
if (ierr < 0) goto 9999

Call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
if (ierr /= MPI_SUCCESS) ierr = -1
if (ierr < 0) goto 9999

! init instance
id%job = JOB_INIT ! init job
id%comm = MPI_COMM_WORLD
Call  dmph_maphys_driver(id)
ierr = id%iinfog(1)
if (ierr < 0) goto 9999

! init matrix
id%sym = 2
n = 1000
nnz = 5998
Allocate( irn( nnz ))
Allocate( jcn( nnz ))
Allocate( a( nnz ))
Allocate( rhs( n ))
Allocate( sol( n ))

Do i=1,1000
```

```
  rhs(i)=1.0
  a(i)=(i)*1.0
  irn(i)=i
  jcn(i)=i
  sol(i)=0.0
End Do
Do i=1,999
  a(1000+i)=(i)*1.0
  irn(1000+i)=i+1
  jcn(1000+i)=i
End Do
Do i=1,999
  a(1999+i)=(i)*1.01
  irn(1999+i)=i
  jcn(1999+i)=i+1
End Do
Do i=1,1000
  a(2998+i)=(i)*1.0
  irn(2998+i)=i
  jcn(2998+i)=499
  a(3998+i)=(i)*1.0
  irn(3998+i)=i
  jcn(3998+i)=500
  a(4998+i)=(i)*1.0
  irn(4998+i)=i
  jcn(4998+i)=501
End Do
id%sym=2



! define the problem on the host
If (myid == 0) then
  id%n = n
  id%nnz = nnz
  id%rows => irn
  id%cols => jcn
  id%sol => sol
  id%values => a
  id%rhs => rhs
End If

! Call the Maphys package.
id%ICNTL(6) = 2
id%ICNTL(24) = 500
id%ICNTL(26) = 500
```

```
  id%ICNTL(27) = 1
  id%ICNTL(22) = 3
  id%RCNTL(21) = 1.0e-5
  id%RCNTL(11) = 1.0e-6
  id%job=6
  call dmph_maphys_driver(id)
  id%job=JOB_END
  Call dmph_maphys_driver(id)
  If (myid == 0) Then
    print *, "Ending maphys ok"
  End If
  Call MPI_Finalize(ierr)
  If (ierr /= 0) Then
    Print *, "Problem with mpi finalise"
  End If

9999 Continue

  If (ierr /=0) then
    Print *, "error"
  End If

End Program Call_maphys
```

9.3.2 C example

Here is a C example of a driver using MAPHYS :

```
/* Example program using the C interface to the
* double precision version of Maphys, dmph_maphys_driver_c. */
#include <stdio.h>
#include "mpi.h"
#include "dmph_maphys_type_c.h"
#define JOB_INIT -1
#define JOB_END -2
#define USE_COMM_WORLD -987654
int main(int argc, char ** argv) {
  DMPH_maphys_t_c id;
  int n = 1000;
  int nnz = 5998;
  int i;
  int irn[nnz];
  int jcn[nnz];
  double a[nnz];
  double rhs[n];
```

```
  double sol[n];
  int myid, ierr;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /* Initialize a Maphys instance. Use MPI_COMM_WORLD. */
  id.job=JOB_INIT; id.sym=2;id.comm=USE_COMM_WORLD;
  dmph_maphys_driver_c(&id);

  /* Define A and rhs */
  for (i=0; i<1000; i++) {
    rhs[i]=1.0;
    a[i]=(i+1)*1.0;
    irn[i]=i+1;
    jcn[i]=i+1;
    sol[i]=0.0;
  };
  for (i=0; i<999; i++) {
    a[1000+i]=(i+1)*1.0;
    irn[1000+i]=i+2;
    jcn[1000+i]=i+1;
  };
  for (i=0; i<999; i++) {
    a[1999+i]=(i+1)*1.01;
    irn[1999+i]=i+1;
    jcn[1999+i]=i+2;
  };
  for (i=0; i<1000; i++) {
    a[2998+i]=(i+1)*1.0;
    irn[2998+i]=i+1;
    jcn[2998+i]=499;
    a[3998+i]=(i+1)*1.0;
    irn[3998+i]=i+1;
    jcn[3998+i]=500;
    a[4998+i]=(i+1)*1.0;
    irn[4998+i]=i+1;
    jcn[4998+i]=501;
  };
  id.sym=2;

  /* Define the problem on the host */
  if (myid == 0) {
    id.n = n; id.nnz =nnz; id.rows=irn; id.cols=jcn; id.sol=sol;
    id.values = a; id.rhs = rhs;
  }
#define ICNTL(I) icntl[(I)-1] /* macro s.t. indices match documentation */
```

```
#define RCNTL(I) rcntl[(I)-1] /* macro s.t. indices match documentation */
  /* Call the Maphys package. */
  id.ICNTL(6) = 2;
  id.ICNTL(24) = 500;
  id.ICNTL(26) = 500;
  id.ICNTL(27) = 1;
  id.ICNTL(22) = 3;
  id.RCNTL(21) = 1.0e-5;
  id.RCNTL(11) = 1.0e-6;
  id.job=6;
  dmph_maphys_driver_c(&id);
  id.job=JOB_END; dmph_maphys_driver_c(&id); /* Terminate instance */
  if (myid == 0) {
    printf("Ending maphys ok");
  }
  ierr = MPI_Finalize();
  if (ierr != 0) {
    fprintf(stderr, "Problem with mpi finalise");
  };
  return 0;

}
```

10 Notes on MaPHyS distribution

10.1 License

MAPHYSis under the CeCILL-C license, you can find the description of the licence in file *doc/CeCILL-C_V1-en.txt*.

10.2 How to refer to MAPHYS

In order to refer to the MAPHYSsolver, you can refer to the article called : .....

10.3 Authors

People involved in MAPHYS development are :

- Azzam Haidar
- Luc Giraud
- Emmanuel Agullo
- Yohan Lee-tin-yien
- Julien Pedron
- Stojce Nakov