BDDCML

solver library based on Multi-Level Balancing Domain Decomposition by Constraints

version 2.6

Jakub Šístek

Table of Contents

1 Introduction

The BDDCML (Balancing Domain Decomposition by Constraints - Multi-Level) is a library for solving large sparse linear systems resulting from computations by the finite element method (FEM). Domain decomposition technique is employed which allows distribution of the computations among processors.

The main goal of the package is to provide a scalable implementation of the (Adaptive) Multilevel BDDC method. Codes are written in Fortran 95 (using also some features of Fortran 2003) with MPI library. A library is provided, which is supposed to be called from users' applications. It provides a simple interface functions callable from Fortran and C.

Balancing Domain Decomposition by Constraints (BDDC) has quickly evolved into a very popular method. However, for very large numbers of subdomain, the coarse problem becomes a large problem to be solved in its own right. In Multilevel BDDC, the coarse problem is solved only approximately by recursive application of BDDC to higher levels.

The main web site of the BDDCML project is

`http://www.math.cas.cz/~sistek/software/bddcml.html`

In case of questions, reporting a bug, or just of interest, feel free to contact Jakub Šístek at `sistek@math.cas.cz`.

2 How to use BDDCML

The library provides a simple interface callable from Fortran and C. Although main parts of the solver are purely algebraic, the solver needs also to get some information of the computational mesh. This requirement is mainly motivated by selection of corners within the method, for which existing algorithms rely on geometry.

Two different modes are possible for input:

- user can provide division into subdomains on the first level and pass subdomain matrices for each subdomain to the routine (local loading),
- user can provide information about global mesh and a file with element matrices (global loading) (this feature will be removed in future release).

The solution process is divided into the following sequence of called functions. Their parameters are described in separate sections.

1. `bddcml_init` – initialization of the solver
2.
   - `bddcml_upload_subdomain_data` – loading data for one subdomain mesh and matrix (use for local loading)
   - `bddcml_upload_global_data` – loading global data about computational mesh and matrix (use for global loading)
3. `bddcml_setup_preconditioner` – prepare preconditioner
4. `bddcml_solve` – solve loaded system by a preconditioned Krylov subspace iterative method (PCG or BiCGstab) or steepest descent iterative method
5.
   - `bddcml_download_local_solution` – get the solution restricted to a subdomain from the solver (use for local loading)
   - `bddcml_download_global_solution` – get the global solution from the solver (use for global loading)
6. `bddcml_finalize` – clear solver data and deallocate memory

Examples are presented in the `examples` folder. The `poisson_on_cube.f90` is a self-standing application demonstrating the use of localized subdomain input. The `bddcml_global.f90` demonstrates the use of global input, while the `bddcml_local.f90` demonstrates the use of localized subdomain input. Optionally, one can ask for nodal reactions at Dirichlet boundary conditions by

- `bddcml_download_local_reactions` – get the reactions restricted to a subdomain from the solver (use for local loading)
- `bddcml_download_global_reactions` – get the global reactions from the solver (use for global loading)

The solver supports efficient handling of multiple-right hand side, for which the matrix as well as preconditioner are reused. Moreover, the Krylov method is accelerated by storing some of the Krylov basis. One can re-run the Krylov method for a new right-hand side and/or values of Dirichlet boundary conditions as

1.
   - `bddcml_change_global_data` – changing global data about right-hand side or Dirichlet b.c. (use for global loading)
   - `bddcml_change_subdomain_data` – changing data for one subdomain about right-hand side or Dirichlet b.c. (use for local loading)
2. `bddcml_setup_new_data` – after right-hand side or Dirichlet b.c. has changed, recreate related internal arrays
3. `bddcml_solve` – solve the system again for the new data

3 Notes for C/C++ users

Since BDDCML is written in Fortran 2003 language, users who want to use BDDCML from C/C++ need to overcome a few issues related to this fact.

3.1 Header file

There is a C interface provided in the file bddcml_interface_c.h that has to be included in applications. In C++, users should include this file using the extern directive as

```
extern "C" {
    #include <bddcml_interface_c.h>
}
```

3.2 Cross-compilation

The library provides interface for Fortran 90 (as a module) and C/C++. The names of the C routines have additional '_c' suffix. The interface for C is prepared using the iso_c_binding module of Fortran 2003, so these fanctions can be directly called from C.

3.3 Libraries of Fortran compiler

If C code is called from Fortran, no additional libraries are usually needed. However, in the opposite case, i.e. the one encountered if BDDCML is used from a C/C++ code, one needs to explicitly specify the libraries of Fortran compiler in the linking sequence when the final executable is build by a C++ linker. For example, if combining g++ and gfortran, it is necessary to add

```
-L/usr/lib -lmpi_f77 -lgfortran
```

when linking an executable by g++. Obviously, the path may be different from this example on a particular machine.

4 Description of interface functions

In this chapter, detailed description of the solver interface functions with explanation of individual arguments is given

4.1 bddcml_init

C interface

```
void bddcml_init_c( int *nl, int *nsublev, int *lnsublev, int *nsub_loc_1, int
*comm_init, int *verbose_level, int *numbase, int *just_direct_solve_int );
```

Fortran interface

```
subroutine bddcml_init(nl, nsublev,lnsublev, nsub_loc_1, comm_init,
verbose_level, numbase, just_direct_solve_int)

      integer, intent(in) :: nl
      integer, intent(in) :: lnsublev
      integer, intent(in) ::  nsublev(lnsublev)
      integer, intent(inout):: nsub_loc_1
      integer, intent(in):: comm_init
      integer, intent(in):: verbose_level
      integer, intent(in) :: numbase
      integer, intent(in)::  just_direct_solve_int
```

Description

Prepares internal data structures for the solver.

Parameters

`nl` given number of levels, `nl` >= 2, `nl` = 2 corresponds to standard (two-level) BDDC method

`nsublev` array with GLOBAL numbers of subdomains for each level. Need to be monotonically decreasing from `nsublev(1) = sum(nsub_loc_1)` to `nsublev(nl) = 1`

`lnsublev` length of array `nsublev` - should match `nl`

`nsub_loc_1`
LOCAL number of subdomains assigned to the process.

- >= 0 number of local subdomains - sum up across processes to nsublev[0]
- -1 let solver decide, the value is returned (determining linear partition)

`comm_init`
initial global communicator (possibly `MPI_COMM_WORLD`). This should be communicator in Fortran. When called from C, it should NOT be of type `MPI_Comm` but of type `int *`, and user should use the `MPI_Comm_c2f` function to convert the C `MPI_Comm` communicator to Fortran communicator of type `int *`.

`verbose_level`
: level of verbosity
  - 0 - only errors printed
  - 1 - some output
  - 2 - detailed output

`numbase` first index of arrays (0 for C, 1 for Fortran)

`just_direct_solve_int`
: Only perform a direct solve (0 - for a regular iterative solve with BDDC preconditioner, 1 - call parallel or serial MUMPS solver instead)

4.2 bddcml_upload_global_data

C interface

```
void bddcml_upload_global_data_c( int *nelem, int *nnod, int *ndof, int *ndim,
int *meshdim, int *inet, int *linet, int *nnet, int *lnnet, int *nndf, int
*lnndf, double *xyz, int *lxyz1, int *lxyz2, int *ifix, int *lifix, double
*fixv, int *lfixv, double *rhs, int *lrhs, double *sol, int *lsol, int *idelm,
int *neighbouring, int *load_division_int );
```

Fortran interface

```
subroutine bddcml_upload_global_data(nelem,nnod,ndof,ndim,meshdim,&
inet,linet,nnet,lnnet,nndf,lnndf,xyz,lxyz1,lxyz2,& ifix,lifix, fixv,lfixv,
rhs,lrhs, sol,lsol, idelm, & neighbouring, load_division_int)

      integer, intent(in):: nelem
      integer, intent(in):: nnod
      integer, intent(in):: ndof
      integer, intent(in) :: ndim
      integer, intent(in) :: meshdim
      integer, intent(in):: linet
      integer, intent(in)::  inet(linet)
      integer, intent(in):: lnnet
      integer, intent(in)::  nnet(lnnet)
      integer, intent(in):: lnndf
      integer, intent(in)::  nndf(lnndf)
      integer, intent(in):: lxyz1, lxyz2
      real(kr), intent(in):: xyz(lxyz1,lxyz2)
      integer, intent(in):: lifix
      integer, intent(in)::  ifix(lifix)
      integer, intent(in):: lfixv
      real(kr), intent(in):: fixv(lfixv)
      integer, intent(in):: lrhs
      real(kr), intent(in):: rhs(lrhs)
      integer, intent(in):: lsol
      real(kr), intent(in):: sol(lsol)
      integer, intent(in) :: idelm
      integer, intent(in) :: neighbouring
      integer, intent(in) :: load_division_int
```

Description

If no distribution of data exists in the user application, it may be left to the solver. This routine loads global information on mesh connectivity and coordinates. Matrix is passed as unassembled matrices of individual elements which will be read from opened file unit `idelm` and assembled within the solver. If partitionining into subdomains on the basic level exists in user's application, routine `bddcml_upload_subdomain_data` should be used instead.

Parameters

`nelem` GLOBAL number of elements

`nnod` GLOBAL number of nodes

`ndof` GLOBAL number of degrees of freedom, i.e. size of matrix

`ndim` number of space dimensions

`meshdim` mesh dimension. For 3D elements = `ndim`, for 3D shells = 2, for 3D beams = 1

`inet` GLOBAL array with Indices of Nodes on ElemenTs - this defines connectivity of the mesh.

`linet` length of array `inet`. It is given as a sum of entries in array `nnet`.

`nnet` GLOBAL array with Number of Nodes on ElemenTs. For each element, it gives number of nodes it is connected to. This is important to locate element entries in array `inet`

`lnnet` length of array `nnet`. It is equal to `nelem`.

`nndf` GLOBAL array with Number of Nodal Degrees of Freedom. For each node, it gives number of attached degrees of freedom.

`lnndf` length of array `nndf`. It is equal to `nnod`.

`xyz` GLOBAL Coordinates of nodes as one array (all X, all Y, all Z) or as two-dimensional array in Fortran (X `|` Y `|` Z). Rows are defined by nodes, columns are defined by dimension.

`lxyz1,lxyz2`
dimensions of array `xyz`. In C, lenght of `xyz` is defined as `lxyz1 * lxyz2`. In Fortran, dimension of `xyz` is given used as `xyz(lxyz1,lxyz2)`. The `lxyz1` is equal to `nnod`. The `lxyz2` is equal to `ndim`.

`ifix` GLOBAL array of Indices of FIXed variables - all degrees of freedom with Dirichlet boundary condition are marked by 1, degrees of freedom not prescribed are marked by 0, i.e. non-zero entries determine fixed degrees of freedom. The values of prescribed boundary conditions are given by corresponding entries of array `fixv`.

`lifix` length of array `ifix`, equal to `ndof`.

`fixv` GLOBAL array of FIXed Variables - where `ifix` is non-zero, `fixv` stores value of Dirichlet boundary condition. Where `ifix` is zero, corresponding value in `fixv` is meaningless.

`lfixv` length of array `fixv`, equal to `ndof`.

`rhs` GLOBAL array with Right-Hand Side

`lrhs` length of array `rhs`, equal to `ndof`.

`sol` GLOBAL array with initial SOLution guess. This is used as initial approximation for iterative method.

`lsol` length of array `sol`, equal to `ndof`.

`idelm` opened Fortran unit with unformatted file with element matrices

`neighbouring`
how many nodes should be shared by two elements to call them adjacent in graph. This parameter is used for division of mesh on the basic level by `ParMETIS` or `METIS`. Often, one gets better results if he specifies this number to define adjacency only if elements share a face in 3D or edge in 2D. E.g. for linear tetrahedra, the recommended value is 3.

`load_division_int`
Should division from file `partition_l1.ES` be used? (0 - partition is created in the solver, 1 - partition is read) If partition is read, the file contains for each element, number of subdomain it belongs to. Begins from 1.

4.3 bddcml_upload_subdomain_data

C interface

```
void bddcml_upload_subdomain_data_c( int *nelem, int *nnod, int *ndof, int
*ndim, int *meshdim, int *isub, int *nelems, int *nnods, int *ndofs, int *inet,
int *linet, int *nnet, int *lnnet, int *nndf, int *lnndf, int *isngn, int
*lisngn, int *isvgvn, int *lisvgvn, int *isegn, int *lisegn, double *xyz,
int *lxyz1, int *lxyz2, int *ifix, int *lifix, double *fixv, int *lfixv,
double *rhs, int *lrhs, int *is_rhs_complete, double *sol, int *lsol, int
*matrixtype, int *i_sparse, int *j_sparse, double *a_sparse, int *la, int
*is_assembled_int, double *user_constraints, int *luser_constraints1,
int *luser_constraints2, double *element_data, int *lelement_data1, int
*lelement_data2, double *dof_data, int *ldof_data, int *find_components_int,
int *use_dual_mesh_graph_int, int *neighbouring );
```

Fortran interface

```
subroutine bddcml_upload_subdomain_data(nelem, nnod, ndof, ndim, meshdim,
& isub, nelems, nnods, ndofs, & inet,linet, nnet,lnnet, nndf,lnndf, &
isngn,lisngn, isvgvn,lisvgvn, isegn,lisegn, & xyz,lxyz1,lxyz2, & ifix,lifix,
fixv,lfixv, & rhs,lrhs, is_rhs_complete_int, & sol,lsol, & matrixtype,
i_sparse, j_sparse, a_sparse, la, is_assembled_int,& user_constraints,luser_
constraints1,luser_constraints2,& element_data,lelement_data1,lelement_
data2, & dof_data,ldof_data, & find_components_int, use_dual_mesh_graph_int,
neighbouring)

      integer, intent(in):: nelem
      integer, intent(in):: nnod
      integer, intent(in):: ndof
      integer, intent(in):: ndim
      integer, intent(in):: meshdim
      integer, intent(in):: isub
      integer, intent(in):: nelems
      integer, intent(in):: nnods
      integer, intent(in):: ndofs
      integer, intent(in):: linet
      integer, intent(in)::  inet(linet)
      integer, intent(in):: lnnet
      integer, intent(in)::  nnet(lnnet)
      integer, intent(in):: lnndf
      integer, intent(in)::  nndf(lnndf)
      integer, intent(in):: lisngn
      integer, intent(in)::  isngn(lisngn)
      integer, intent(in):: lisvgvn
      integer, intent(in)::  isvgvn(lisvgvn)
      integer, intent(in):: lisegn
      integer, intent(in)::  isegn(lisegn)
```

```
integer, intent(in):: lxyz1, lxyz2
real(kr), intent(in):: xyz(lxyz1,lxyz2)
integer, intent(in):: lifix
integer, intent(in)::  ifix(lifix)
integer, intent(in):: lfixv
real(kr), intent(in):: fixv(lfixv)
integer, intent(in):: lrhs
real(kr), intent(in):: rhs(lrhs)
integer, intent(in):: lsol
real(kr), intent(in):: sol(lsol)
integer, intent(in)::  is_rhs_complete_int
integer, intent(in)::  matrixtype
integer, intent(in)::  i_sparse(la)
integer, intent(in)::  j_sparse(la)
real(kr), intent(in):: a_sparse(la)
integer, intent(in)::  la
integer, intent(in)::  is_assembled_int
integer, intent(in)::  luser_constraints1
integer, intent(in)::  luser_constraints2
real(kr), intent(in)::user_constraints(luser_constraints1*&
                                       luser_constraints2)
integer, intent(in)::  lelement_data1
integer, intent(in)::  lelement_data2
real(kr), intent(in):: element_data(lelement_data1*lelement_data2)
integer, intent(in)::  ldof_data
real(kr), intent(in):: dof_data(ldof_data)
integer, intent(in)::  find_components_int
integer, intent(in)::  use_dual_mesh_graph_int
integer, intent(in)::  neighbouring
```

Description

If distribution of data into subdomains exists already in the user application, data should be loaded into the solver using this routine. It may be called repeatedly by each process if more than one subdomain are assigned to that process. It loads the local mesh of the subdomain and assembled subdomain matrix in the coordinate format. Most data are localized to subdomain.

If partitionining into subdomains does not exist in user's application, routine `bddcml_upload_global_data` should be preferred.

Parameters

`nelem` GLOBAL number of elements

`nnod` GLOBAL number of nodes

`ndof` GLOBAL number of degrees of freedom, i.e. size of matrix

`ndim` number of space dimensions

`meshdim` mesh dimension. For 3D elements = `ndim`, for 3D shells = 2, for 3D beams = 1

`isub` GLOBAL index of subdomain which is loaded

`nelems` LOCAL number of elements in subdomain

`nnods` LOCAL number of nodes in subdomain mesh

`ndofs` LOCAL number of degrees of freedom in subdomain mesh

`inet` LOCAL array with Indices of Nodes on ElemenTs - this defines connectivity of the subdomain mesh.

`linet` length of array `inet`. It is given as a sum of entries in array `nnet`.

`nnet` LOCAL array with Number of Nodes on ElemenTs. For each element, it gives number of nodes it is connected to. This is important to locate element entries in array `inet`

`lnnet` length of array `nnet`. It is equal to `nelems`.

`nndf` LOCAL array with Number of Nodal Degrees of Freedom. For each node, it gives number of attached degrees of freedom.

`lnndf` length of array `nndf`. It is equal to `nnods`.

`isngn` array of Indices of Subdomain Nodes in Global Numbering (local to global map of nodes). For each local node gives the global index in original mesh.

`lisngn` length of array `isngn`. It is equal to `nnods`.

`isvgvn` array of Indices of Subdomain Variables in Global Variable Numbering (local to global map of variables). For each local degree of freedom gives the global index in original matrix.

`lisvgvn` length of array `isvgvn`. It is equal to `ndofs`.

`isegn` array of Indices of Subdomain Elements in Global Numbering (local to global map of elements). For each subdomain element gives global number in original mesh.

`lisegn` length of array `isegn`. It is equal to `nelems`.

`xyz` LOCAL array with coordinates of nodes as one array (all X, all Y, all Z) or as two-dimensional array in Fortran (X | Y | Z). Rows are defined by nodes, columns are defined by dimension.

`lxyz1,lxyz2`
dimensions of array `xyz`. In C, lenght of `xyz` is defined as `lxyz1 * lxyz2`. In Fortran, dimension of `xyz` is used as `xyz(lxyz1,lxyz2)`. The `lxyz1` is equal to `nnods`. The `lxyz2` is equal to `ndim`.

`ifix` LOCAL array of Indices of FIXed variables - all degrees of freedom with Dirichlet boundary condition are marked by 1, degrees of freedom not prescribed are marked by 0, i.e. non-zero entries determine fixed degrees of freedom. The values of prescribed boundary conditions are given by corresponding entries of array `fixv`.

`lifix` length of array `ifix`, equal to `ndofs`.

`fixv` LOCAL array of FIXed Variables - where `ifix` is non-zero, `fixv` stores value of Dirichlet boundary condition. Where `ifix` is zero, corresponding value in `fixv` is meaningless.

`lfixv` length of array `fixv`, equal to `ndofs`.

`rhs` LOCAL array with Right-Hand Side. Values at nodes repeated among subdomains are copied and not weighted.

`lrhs` length of array `rhs`, equal to `ndofs`.

`is_rhs_complete`
is the subdomain right-hand side complete?

- 0 - no, e.g. if only local subassembly of right-hand side was performed - interface values are not fully assembled, solver does not apply weights
- 1 - yes, e.g. if local right-hand side is a restriction of the global array to the subdomain - interface values are complete and repeated for more subdomains, solver applies weights to handle multiplicity of these entries

`sol` LOCAL array with initial SOLution guess. This is used as initial approximation for iterative method.

`lsol` length of array `sol`, equal to `ndofs`.

`matrixtype`
Type of the matrix. This parameter determines storage and underlying direct method of the `MUMPS` solver for factorizations. Matrix is loaded in coordinate format by three arrays described below. Options are

- 0 unsymmetric - whole matrix is loaded
- 1 symmetric positive definite - only upper triangle of the matrix is loaded
- 2 general symmetric - only upper triangle of the matrix is loaded

`i_sparse` array of row indices of non-zero entries in LOCAL numbering with respect to subdomain degrees of freedom, i.e. indices are in the range [`0`, `ndofs-1`] in C (and in [`1`, `ndofs`] in Fortran)

`j_sparse` array of column indices of non-zero entries in LOCAL numbering with respect to subdomain degrees of freedom, see `i_sparse` for their ranges

`a_sparse` array of values of non-zero entries

`la` length of previous arrays `i_sparse`, `j_sparse`, `a_sparse` (equal to number of non-zeros if the matrix is loaded already assembled)

`is_assembled_int`
is the matrix assembled? The solver comes with fast assembly routine so the users might want to pass just unassembled matrix for each subdomain (i.e. copy of element matrices equipped with global indexing), and let the solver assemble it.

- 0 - no, it can contain repeated entries, will be assembled by solver
- 1 - yes, it is sorted and does not contain repeated index pairs

`luser_contraints1, luser_constraints2`
: dimension of array with user constraints, `luser_constraints2` should equal to `nnods` if user wants to use this feature. Otherwise both can be set to 0.

`user_contraints`
: linearized matrix of user's constraints, one row per constraint, stored row-by-row. Each row has entries for all nodes (in columns) and may contain several constraints (at rows). It may be left empty if no additional contraints are demanded by call to `bddcml_setup_preconditioner`.

`lelement_data1, lelement_data2`
: dimension of array with data for elements, `lelement_data2` should equal to `nelems` if user wants to use this feature. Otherwise both can be set to 0.

`element_data`
: linearized matrix with element data, one row per field, stored row-by-row. Each row has entries for all elements (in columns). Single row is currently optionally used for generating interface weigths in the BDDC method, so called ρ-scaling.

`ldof_data`
: dimension of array with data for subdomain degrees of freedom, `ldof_data` should equal to `ndofs` if user wants to use this feature. Otherwise it can be set to 0.

`dof_data` coefficients for each local degree of freedom currently optionally used for generating interface weigths in the BDDC method. The array should be positive, `dof_data` > 0.

`find_components_int`
: Find components within the mesh of the subdomain? When using graph partitioners for creating subdomains, subdomains may be disconnected. For unstructured meshes, it is thus recommended to use this feature. The solver will detect connected components of the mesh and handle them as independent substructures in the selection of corners, edges and faces. This results in sufficient number of contraints while not affecting load balancing too much.

  - 0 - no, subdomain is considered as one connected component
  - 1 - yes, subdomain will be analysed for connected components. Recommended but can become expensive for elements of very high order.

`use_dual_mesh_graph_int`
: Should dual graph of the mesh be used for detecting components? Only accessed if find_components_int > 0.

  - 0 - no, primal graph of mesh will be used with vertices defined by nodes,
  - 1 - yes, dual graph of mesh will be used with vertices defined by elements and edge created between two elements if they share at least `neighbouring` nodes.

`neighbouring`
: How many nodes need two elements to share to define an edge between them in the dual mesh graph of subdomain? Only accessed if find_components_int > 0 and use_dual_mesh_graph_int > 0.

4.4 bddcml_setup_preconditioner

C interface

```
void bddcml_setup_preconditioner_c( int *matrixtype, int *use_defaults_
int, int *parallel_division_int, int *use_corner_constraints_int, int
*use_arithmetic_constraints_int, int *use_adaptive_constraints_int, int
*use_user_constraints_int, int *weights_type );
```

Fortran interface

```
subroutine bddcml_setup_preconditioner(matrixtype, use_defaults_int, &
parallel_division_int, & use_corner_constraints_int, & use_arithmetic_
constraints_int, & use_adaptive_constraints_int, & use_user_constraints_int,
& weights_type)
      integer,intent(in) :: matrixtype
      integer,intent(in) :: use_defaults_int
      integer,intent(in) :: parallel_division_int
      integer,intent(in) :: use_corner_constraints_int
      integer,intent(in) :: use_arithmetic_constraints_int
      integer,intent(in) :: use_adaptive_constraints_int
      integer,intent(in) :: use_user_constraints_int
      integer,intent(in) :: weights_type
```

Description

Calling this function prepares internal data of the preconditioner. Local factorizations are performed for each subdomain at each level and also the resulting coarse problem on the final level is factored. This might be quite costly routine. Once the preconditioner is set-up, it can be reused for new right hand sides (if the matrix is not changed) by calling `bddcml_upload_subdomain_data` followed by `bddcml_solve`.

Parameters

`matrixtype`
: Type of the matrix. This parameter determines storage and underlying direct method of the `MUMPS` solver for factorizations. Should keep the value inserted to `bddcml_upload_subdomain_data`. Options are
  - 0 unsymmetric - whole matrix is loaded
  - 1 symmetric positive definite - only upper triangle of the matrix is loaded
  - 2 general symmetric - only upper triangle of the matrix is loaded

`use_defaults_int`
: If `> 0`, other options are ignored and the solver uses default options.

`parallel_division_int`
: If `> 0`, solver will use `ParMETIS` to create division on first level. This option is only used for global input (`bddcml_upload_global_data`) and only applies to the first level. Otherwise, `METIS` is used. Default is `1`.

`use_corner_constraints_int`
: If `> 0`, solver will use continuity of coarse functions at corners to form the coarse space. These are the basic constraints in BDDC. Default is `1`.

`use_arithmetic_constraints_int`
: If `> 0`, solver will use continuity of arithmetic averages on faces in 2D and faces and edges in 3D to form the coarse space. Default is `1`.

`use_adaptive_constraints_int`
: If `> 0`, solver will use adaptive averages on faces in 2D and faces in 3D. This might be costly and should be used for very ill-conditioned problems. A generalized eigenvalue problem is solved at each face and weighted averages are derived from eigenvectors. For solving individual eigenproblems, `BLOPEX` package is used. Default is `0`.

`use_user_constraints_int`
: If `> 0`, solver will use array `USER_CONSTRAINTS` supplied in routine `bddcml_upload_subdomain_data` for construction of additional constraints on subdomain faces. This may be used e.g. for enforcing flux-based constraints for advection-diffusion problems. Default is `0`.

`weights_type`
: Type of weights to be used on interface (default 0). Possible choices

  - 0 – weights computed by cardinality, i.e. arithmetic average, the mean value of solution from neighbouring subdomains. Good choice for many problem types except those with large variations in material coefficients and/or sizes of elements.
  - 1 – weights from diagonal stiffness, corresponding diagonal entry in the matrix. Good choice for symmetric positive definite problems with large variations in material coefficients and/or sizes of elements. Not good for homogenous problems with irregular subdomains, such as those from graph partitioners, in combination with nodal elements. This option takes into account different sizes of elements and material coefficients.
  - 2 – weights based on loaded element data. For this choice, the first row of array `element_data` from function `bddcml_upload_subdomain_data` must contain element coefficients to be used here. This allows definition of the so called ρ-scaling. Good choice for problems with large variations in material coefficients if irregular subdomains from graph partitioners are used in combination with nodal finite elements.
  - 3 – weights based on loaded data for degees of freedom. For this choice, array `dof_data` from function `bddcml_upload_subdomain_data` must contain coefficients for each subdomain degree of freedom. This allows definition of general user-defined weigths on degrees of freedom tailored to particular applications.

4.5 bddcml_solve

C interface

```
void bddcml_solve_c( int *comm_all, int *method, double *tol, int *maxit,
int *ndecrmax, int *recycling_int, int *max_number_of_stored_vectors, int
*num_iter, int *converged_reason, double *condition_number);
```

Fortran interface

```
subroutine bddcml_solve(comm_all,method,tol,maxit,ndecrmax, & recycling_int,
max_number_of_stored_vectors, & num_iter,converged_reason,condition_number)
      integer, intent(in) :: comm_all
      integer, intent(in) :: method
      real(kr), intent(in) :: tol
      integer, intent(in) :: maxit
      integer, intent(in) :: ndecrmax
      integer, intent(in) :: recycling_int
      integer, intent(in) :: max_number_of_stored_vectors
      integer, intent(out) :: num_iter
      integer, intent(out) :: converged_reason
      real(kr), intent(out) :: condition_number
```

Description

This function launches the solution procedure for prepared data. System is solved either by preconditioned conjugate gradient (PCG) method or by preconditioned stabilized Bi-Conjugate Gradient (BiCGstab) method. Alternatively to Krylov subspace methods one can use steepest descent iteration.

Parameters

`comm_all` global communicator. Should be the same as `comm_init` for `bddcml_init` function.

`method` Krylov subspace iterative method

- -1 - use defaults - `tol`, `maxit`, and `ndecrmax` not accessed, BiCGstab method used by default,
- 0 - use PCG,
- 1 - use BiCGstab,
- 2 - use steepest descent method,
- 5 - use parallel direct solver instead of an iterative method.

`tol` desired accuracy of relative residual (default 1.e-6).

`maxit` limit on number of iterations (default 1000).

`ndecrmax` limit on number of iterations with non-decreasing residual (default 30) - used to stop a diverging process.

`recycling_int`
: If `> 0`, the Krylov subspace will be recycled in the Krylov subspace method (default 0). This can save time for solutions for many right-hand sides. The option triggers explicit orthogonalization to the stored Krylov basis. Therefore, even if this option is used with just one right-hand side solve, it can help the convergence for numerically complicated problems, where natural orthogonality of the basis may be lost. The approach to recycling is based on Farhat, C., Crivelli, L., Roux, X.: Extending substructure based iterative solvers to multiple load and repeated analyses. Comput. Methods in Appl. Mech. Engrg. 117 (1994), 195-209.

`max_number_of_stored_vectors`
: limit on the size of the Krylov subspace basis explicitly kept in the solver (default 100). Only meaningful if `recycling_int > 0`.

`num_iter` on output, resulting number of iterations.

`converged_reason`
: on output, contains reason for convergence/divergence

  - 0 - converged relative residual,
  - -1 - reached limit on number of iterations,
  - -2 - reached limit on number of iterations with non-decreasing residual.

`condition_number`
: on output, estimated condition number (for PCG only).

4.6 bddcml_download_local_solution

C interface

```
void bddcml_download_local_solution_c( int *isub, double *sols, int *lsols );
```

Fortran interface

```
subroutine bddcml_download_local_solution(isub, sols,lsols)

      integer, intent(in)::  isub
      integer, intent(in)::  lsols
      real(kr), intent(out):: sols(lsols)
```

Description

Subroutine for getting local solution, i.e. restriction of solution vector to subdomain (no weights are applied).

Parameters

`isub`	GLOBAL index of subdomain
`sols`	LOCAL array of solution restricted to subdomain
`lsols`	length of array `sols`, equal to `ndofs`.

4.7 bddcml_download_global_solution

C interface

```
void bddcml_download_global_solution_c( double *sol, int *lsol );
```

Fortran interface

```
subroutine bddcml_download_global_solution(sol, lsol)

      integer, intent(in)::  lsol
      real(kr), intent(out):: sol(lsol)
```

Description

This function downloads global solution of the system from the solver at the root process.

Parameters

`sol`	GLOBAL array of solution
`lsol`	length of array `sol`, equal to `ndof`

4.8 bddcml_download_local_reactions

C interface

```
void bddcml_download_local_reactions_c( int *isub, double *reas, int *lreas );
```

Fortran interface

```
subroutine bddcml_download_local_reactions(isub, reas,lreas)

      integer, intent(in)::  isub
      integer, intent(in)::  lreas
      real(kr), intent(out):: reas(lreas)
```

Description

Subroutine for getting local reactions at unknowns fixed by Dirichlet boundary conditions, i.e. restriction of vector of reactions to subdomain (no weights are applied). Global vector of reactions is given by `r = Ax - b`, where `A` is the matrix WITHOUT Dirichlet boundary conditions fixed, `x` is the solution, and `b` the original right-hand side.

Parameters

`isub`	GLOBAL index of subdomain
`reas`	LOCAL array of vector of reactions restricted to subdomain. It contains nonzeros only at unknowns marked in the IFIX array (see `bddcml_upload_local_data`).
`lreas`	length of array `reas`, equal to `ndofs`.

4.9 bddcml_download_global_reactions

C interface

```
void bddcml_download_global_reactions_c( double *rea, int *lrea );
```

Fortran interface

```
subroutine bddcml_download_global_reactions(rea, lrea)

      integer, intent(in)::  lrea
      real(kr), intent(out):: rea(lrea)
```

Description

Subroutine for getting global reactions at unknowns fixed by Dirichlet boundary conditions at the root process. Global vector of reactions is given by `r = Ax - b`, where `A` is the matrix WITHOUT Dirichlet boundary conditions fixed, `x` is the solution, and `b` the original right-hand side.

Parameters

`rea`	GLOBAL array of reactions. It contains nonzeros only at unknowns marked in the IFIX array (see `bddcml_upload_global_data`).
`lrea`	length of array `rea`, equal to `ndof`

4.10 bddcml_change_global_data

C interface

```
void bddcml_change_global_data_c( int *ifix, int *lifix, double *fixv, int
*lfixv, double *rhs, int *lrhs, double *sol, int *lsol);
```

Fortran interface

```
subroutine bddcml_change_global_data(ifix,lifix, fixv,lfixv, rhs,lrhs,
sol,lsol)

      integer, intent(in):: lifix
      integer, intent(in)::  ifix(lifix)
      integer, intent(in):: lfixv
      real(kr), intent(in):: fixv(lfixv)
      integer, intent(in):: lrhs
      real(kr), intent(in):: rhs(lrhs)
      integer, intent(in):: lsol
      real(kr), intent(in):: sol(lsol)
```

Description

A function for loading another right hand side, when global loading is used. Position of fixed variables, the `IFIX` array, is not allowed to change. Parameters correspond to routine `bddcml_upload_global_data`.

Parameters

`ifix` GLOBAL array of Indices of FIXed variables - all degrees of freedom with Dirichlet boundary condition are marked by 1, degrees of freedom not prescribed are marked by 0, i.e. non-zero entries determine fixed degrees of freedom. The values of prescribed boundary conditions are given by corresponding entries of array `fixv`.

`lifix` length of array `ifix`, equal to `ndof`.

`fixv` GLOBAL array of FIXed Variables - where `ifix` is non-zero, `fixv` stores value of Dirichlet boundary condition. Where `ifix` is zero, corresponding value in `fixv` is meaningless.

`lfixv` length of array `fixv`, equal to `ndof`.

`rhs` GLOBAL array with Right-Hand Side

`lrhs` length of array `rhs`, equal to `ndof`.

`sol` GLOBAL array with initial SOLution guess. This is used as initial approximation for iterative method.

`lsol` length of array `sol`, equal to `ndof`.

4.11 bddcml_change_subdomain_data

C interface

```
void bddcml_change_subdomain_data_c( int *isub, int *ifix, int *lifix, double
*fixv, int *lfixv, double *rhs, int *lrhs, int *is_rhs_complete, double *sol,
int *lsol );
```

Fortran interface

```
subroutine bddcml_change_subdomain_data(isub, & ifix,lifix, fixv,lfixv, &
rhs,lrhs, is_rhs_complete_int, & sol,lsol)
      integer, intent(in):: isub
      integer, intent(in):: lifix
      integer, intent(in)::  ifix(lifix)
      integer, intent(in):: lfixv
      real(kr), intent(in):: fixv(lfixv)
      integer, intent(in):: lrhs
      real(kr), intent(in):: rhs(lrhs)
      integer, intent(in)::  is_rhs_complete_int
      integer, intent(in):: lsol
      real(kr), intent(in):: sol(lsol)
```

Description

A function for loading another right hand side, when subdomain-by-subdomain loading is used. Position of fixed variables, the `IFIX` array, is not allowed to change. Parameters correspond to routine `bddcml_upload_subdomain_data`. It should be called repeatedly by each process if more than one subdomain are assigned to that process. This call should be followed by calling `bddcml_setup_new_data` before another call to `bddcml_solve`.

If partitionining into subdomains does not exist in user's application, routine `bddcml_change_global_data` should be preferred.

Parameters

`isub` GLOBAL index of subdomain which is loaded

`ifix` LOCAL array of Indices of FIXed variables - all degrees of freedom with Dirichlet boundary condition are marked by 1, degrees of freedom not prescribed are marked by 0, i.e. non-zero entries determine fixed degrees of freedom. The values of prescribed boundary conditions are given by corresponding entries of array `fixv`.

`lifix` length of array `ifix`, equal to `ndofs`.

`fixv` LOCAL array of FIXed Variables - where `ifix` is non-zero, `fixv` stores value of Dirichlet boundary condition. Where `ifix` is zero, corresponding value in `fixv` is meaningless.

`lfixv` length of array `fixv`, equal to `ndofs`.

`rhs` LOCAL array with Right-Hand Side. Values at nodes repeated among subdomains are copied and not weighted.

`lrhs` length of array `rhs`, equal to `ndofs`.

`is_rhs_complete`
is the subdomain right-hand side complete?

- 0 - no, e.g. if only local subassembly of right-hand side was performed - interface values are not fully assembled, solver does not apply weights
- 1 - yes, e.g. if local right-hand side is a restriction of the global array to the subdomain - interface values are complete and repeated for more subdomains, solver applies weights to handle multiplicity of these entries

`sol` LOCAL array with initial SOLution guess. This is used as initial approximation for iterative method.

`lsol` length of array `sol`, equal to `ndofs`.

4.12 bddcml_setup_new_data

C interface

```
void bddcml_setup_new_data_c( );
```

Fortran interface

```
subroutine bddcml_setup_new_data
```

Description

Setting up internal data-structures for handling new values of Dirichlet boundary conditions and right hand side — creating new reduced right-hand side.

Needs to be called after calling `bddcml_change_subdomain_data` or `bddcml_change_global_data` and before another call to `bddcml_solve`.

Parameters

This routine currently does not take any arguments.

4.13 bddcml_dotprod_subdomain

C interface

```
void bddcml_dotprod_subdomain_c( int *isub, double *vec1, int *lvec1, double
*vec2, int *lvec2, double *dotprod );
```

Fortran interface

```
subroutine bddcml_dotprod_subdomain( isub, vec1,lvec1, vec2,lvec2, dotprod )

      integer, intent(in) ::  isub
      integer, intent(in) ::  lvec1
      real(kr), intent(in) :: vec1(lvec1)
      integer, intent(in) ::  lvec2
      real(kr), intent(in) :: vec2(lvec2)
      real(kr), intent(out) :: dotprod
```

Description

Auxiliary subroutine to compute scalar product of two vectors of lenght of subdomain exploiting interface weights from the solver. This routine is useful if we want to compute global norm or dot-product based on vectors restricted to subdomains. Since interface values are contained in several vectors for several subdomains, this dot product or norm cannot be determined without weights.

Parameters

`isub`	GLOBAL index of subdomain
`vec1`	LOCAL first vector for dot-product
`lvec1`	length of `vec1`
`vec2`	LOCAL second vector for dot-product, may be same array as `vec1`
`lvec2`	length of `vec2`, should be same as `lvec1`
`dotprod`	on exit, returns vec1' * weights * vec2

4.14 bddcml_finalize

C interface

```
void bddcml_finalize_c( );
```

Fortran interface

```
subroutine bddcml_finalize
```

Description

Finalization of the solver. All internal data are deallocated.

Parameters

This routine currently does not take any arguments.