hypre Documentation

Release 2.24.0

Lawrence Livermore National Laboratory

February 11, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

This manual describes hypre, a software library of high performance preconditioners and solvers for the solution of large, sparse linear systems of equations on massively parallel computers [FaJY2004]. The hypre library was created with the primary goal of providing users with advanced parallel preconditioners. The library features parallel multigrid solvers for both structured and unstructured grid problems. For ease of use, these solvers are accessed from the application code via hypre's conceptual linear system interfaces [FaJY2005] (abbreviated to *conceptual interfaces* throughout much of this manual), which allow a variety of natural problem descriptions.

This introductory chapter provides an overview of the various features in hypre, discusses further sources of information on hypre, and offers suggestions on how to get started.

1.1 Overview of Features

Scalable preconditioners provide efficient solution on today's and tomorrow's systems: hypre contains several families of preconditioner algorithms focused on the scalable solution of *very large* sparse linear systems. (Note that small linear systems, systems that are solvable on a sequential computer, and dense systems are all better addressed by other libraries that are designed specifically for them.) hypre includes "grey box" algorithms that use more than just the matrix to solve certain classes of problems more efficiently than general-purpose libraries. This includes algorithms such as structured multigrid.

Suite of common iterative methods provides options for a spectrum of problems: hypre provides several of the most commonly used Krylov-based iterative methods to be used in conjunction with its scalable preconditioners. This includes methods for nonsymmetric systems such as GMRES and methods for symmetric matrices such as Conjugate Gradient.

Intuitive grid-centric interfaces obviate need for complicated data structures and provide access to advanced solvers: hypre has made a major step forward in usability from earlier generations of sparse linear solver libraries in that users do not have to learn complicated sparse matrix data structures. Instead, hypre does the work of building these data structures for the user through a variety of conceptual interfaces, each appropriate to different classes of users. These include stencil-based structured/semi-structured interfaces most appropriate for finite-difference applications; a finite-element based unstructured interface; and a linear-algebra based interface. Each conceptual interface provides access to several solvers without the need to write new interface code.

User options accommodate beginners through experts: hypre allows a spectrum of expertise to be applied by users. The beginning user can get up and running with a minimal amount of effort. More expert users can take further control of the solution process through various parameters.

Configuration options to suit your computing system: hypre allows a simple and flexible installation on a wide variety of computing systems. Users can tailor the installation to match their computing system. Options include debug and optimized modes, the ability to change required libraries such as MPI and BLAS, a sequential mode, and modes enabling threads for certain solvers. On most systems, however, hypre can be built by simply typing `configure` followed by `make`, or by using CMake [CMakeWeb].

Interfaces in multiple languages provide greater flexibility for applications: hypre is written in C (with the exception of the FEI interface, which is written in C++) and provides an interface for Fortran users.

1.2 Getting More Information

This user's manual consists of chapters describing each conceptual interface, a chapter detailing the various linear solver options available, and detailed installation information. In addition to this manual, a number of other information sources for hypre are available.

- **Reference Manual:** The reference manual comprehensively lists all of the interface and solver functions available in hypre. The reference manual is ideal for determining the various options available for a particular solver or for viewing the functions provided to describe a problem for a particular interface.
- **Example Problems:** A suite of example problems is provided with the hypre installation. These examples reside in the `examples` subdirectory and demonstrate various features of the hypre library. Associated documentation may be accessed by viewing the `README.html` file in that same directory.
- **Papers, Presentations, etc.:** Articles and presentations related to the hypre software library and the solvers available in the library are available from the hypre web page at http://www.llnl.gov/CASC/hypre/.
- **Mailing List:** The mailing list `hypre-announce` can be subscribed to through the hypre web page at http://www.llnl.gov/CASC/hypre/. The development team uses this list to announce new releases of hypre. It cannot be posted to by users.

1.3 How to get started

1.3.1 Installing hypre

As previously noted, on most systems hypre can be built by simply typing `configure` followed by `make` in the top-level source directory. Alternatively, the CMake system [CMakeWeb] can be used, and is the best approach for building hypre on Windows systems in particular. For more detailed instructions, read the `INSTALL` file provided with the hypre distribution or refer to the last chapter in this manual. Note the following requirements:

- To run in parallel, hypre requires an installation of MPI.
- Configuration of hypre with threads requires an implementation of OpenMP. Currently, only a subset of hypre is threaded.
- The hypre library currently does not directly support complex-valued systems.

1.3.2 Choosing a conceptual interface

An important decision to make before writing any code is to choose an appropriate conceptual interface. These conceptual interfaces are intended to represent the way that applications developers naturally think of their linear problem and to provide natural interfaces for them to pass the data that defines their linear system into hypre. Essentially, these conceptual interfaces can be considered convenient utilities for helping a user build a matrix data structure for hypre solvers and preconditioners. The top row of *Figure 1* illustrates a number of conceptual interfaces. Generally, the conceptual interfaces are denoted by different types of computational grids, but other application features might also be used, such as geometrical information. For example, applications that use structured grids (such as in the left-most interface in *Figure 1*) typically view their linear problems in terms of stencils and grids. On the other hand, applications that use unstructured grids and finite elements typically view their linear problems in terms of elements and element stiffness matrices. Finally, the right-most interface is the standard linear-algebraic (matrix rows/columns) way of viewing the linear problem.

The hypre library currently supports four conceptual interfaces, and typically the appropriate choice for a given problem is fairly obvious, e.g. a structured-grid interface is clearly inappropriate for an unstructured-grid application.

- **Structured-Grid System Interface (Struct):** This interface is appropriate for applications whose grids consist of unions of logically rectangular grids with a fixed stencil pattern of nonzeros at each grid point. This interface supports only a single unknown per grid point. See Chapter *Structured-Grid System Interface (Struct)* for details.
- **Semi-Structured-Grid System Interface (SStruct):** This interface is appropriate for applications whose grids are mostly structured, but with some unstructured features. Examples include block-structured grids, composite grids in structured adaptive mesh refinement (AMR) applications, and overset grids. This interface supports multiple unknowns per cell. See Chapter *Semi-Structured-Grid System Interface (SStruct)* for details.
- **Finite Element Interface (FEI):** This is appropriate for users who form their linear systems from a finite element discretization. The interface mirrors typical finite element data structures, including element stiffness matrices. Though this interface is provided in hypre, its definition was determined elsewhere (please send email to Alan Williams william@sandia.gov for more information). See Chapter *Finite Element Interface* for details.
- **Linear-Algebraic System Interface (IJ):** This is the traditional linear-algebraic interface. It can be used as a last resort by users for whom the other grid-based interfaces are not appropriate. It requires more work on the user's part, though still less than building parallel sparse data structures. General solvers and preconditioners are available through this interface, but not specialized solvers which need more information. Our experience is that users with legacy codes, in which they already have code for building matrices in particular formats, find the IJ interface relatively easy to use. See Chapter *Linear-Algebraic System Interface (IJ)* for details.

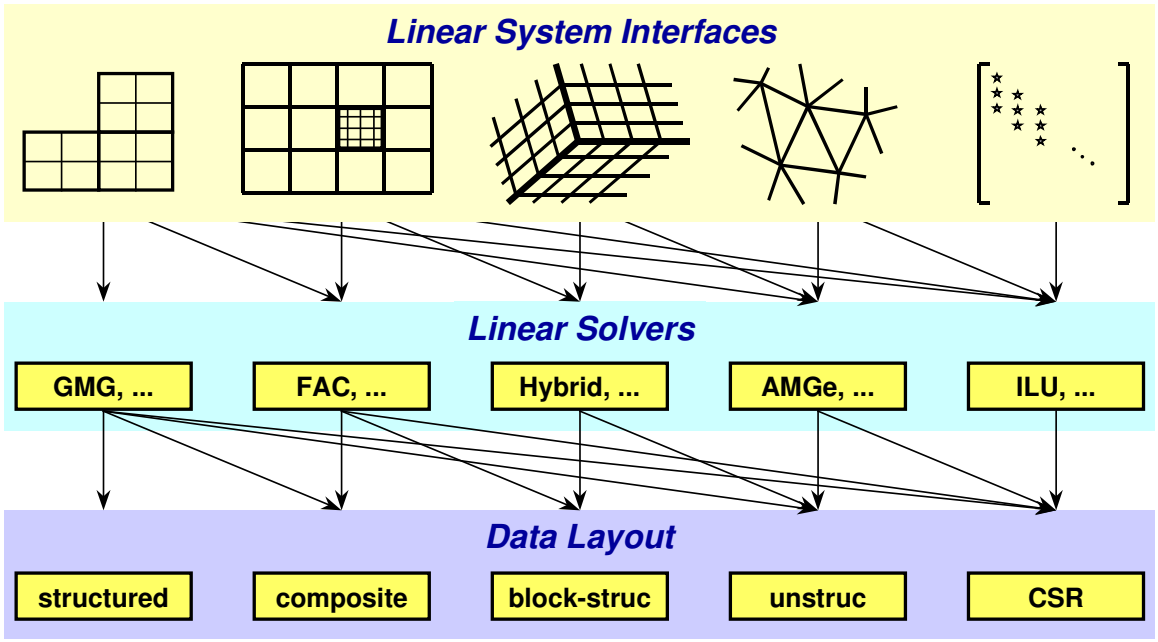


Fig. 1: Figure 1
Graphic illustrating the notion of conceptual linear system interfaces.

Generally, a user should choose the most specific interface that matches their application, because this will allow them to use specialized and more efficient solvers and preconditioners without losing access to more general solvers. For example, the second row of Figure *Figure 1* is a set of linear solver algorithms. Each linear solver group requires different information from the user through the conceptual interfaces. So, the geometric multigrid algorithm (GMG) listed in the left-most box, for example, can only be used with the left-most conceptual interface. On the other hand, the ILU algorithm in the right-most box may be used with any conceptual interface. Matrix requirements for each solver and preconditioner are provided in Chapter *Solvers and Preconditioners* and in the hypre Reference Manual. Your desired solver strategy may influence your choice of conceptual interface. A typical user will select a single Krylov

method and a single preconditioner to solve their system.

The third row of Figure *Figure 1* is a list of data layouts or matrix/vector storage schemes. The relationship between linear solver and storage scheme is similar to that of the conceptual interface and linear solver. Note that some of the interfaces in hypre currently only support one matrix/vector storage scheme choice. The conceptual interface, the desired solvers and preconditioners, and the matrix storage class must all be compatible.

1.3.3 Writing your code

As discussed in the previous section, the following decisions should be made before writing any code:

- Choose a conceptual interface.
- Choose your desired solver strategy.
- Look up matrix requirements for each solver and preconditioner.
- Choose a matrix storage class that is compatible with your solvers and preconditioners and your conceptual interface.

Once the previous decisions have been made, it is time to code your application to call hypre. At this point, reviewing the previously mentioned example codes provided with the hypre library may prove very helpful. The example codes demonstrate the following general structure of the application calls to hypre:

- **Build any necessary auxiliary structures for your chosen conceptual interface.** This includes, e.g., the grid and stencil structures if you are using the structured-grid interface.
- **Build the matrix, solution vector, and right-hand-side vector through your chosen conceptual interface.** Each conceptual interface provides a series of calls for entering information about your problem into hypre.
- **Build solvers and preconditioners and set solver parameters (optional).** Some parameters like convergence tolerance are the same across solvers, while others are solver specific.
- **Call the solve function for the solver.**
- **Retrieve desired information from solver.** Depending on your application, there may be different things you may want to do with the solution vector. Also, performance information such as number of iterations is typically available, though it may differ from solver to solver.

The subsequent chapters of this User's Manual provide the details needed to more fully understand the function of each conceptual interface and each solver. Remember that a comprehensive list of all available functions is provided in the hypre Reference Manual, and the provided example codes may prove helpful as templates for your specific application.

CHAPTER

TWO

STRUCTURED-GRID SYSTEM INTERFACE (STRUCT)

In order to get access to the most efficient and scalable solvers for scalar structured-grid applications, users should use the `Struct` interface described in this chapter. This interface will also provide access (this is not yet supported) to solvers in hypre that were designed for unstructured-grid applications and sparse linear systems in general. These additional solvers are usually provided via the unstructured-grid interface (`FEI`) or the linear-algebraic interface (`IJ`) described in Chapters *Finite Element Interface* and *Linear-Algebraic System Interface (IJ)*.

Figure *Structured Grid Example* gives an example of the type of grid currently supported by the `Struct` interface. The interface uses a finite-difference or finite-volume style, and currently supports only scalar PDEs (i.e., one unknown per gridpoint).

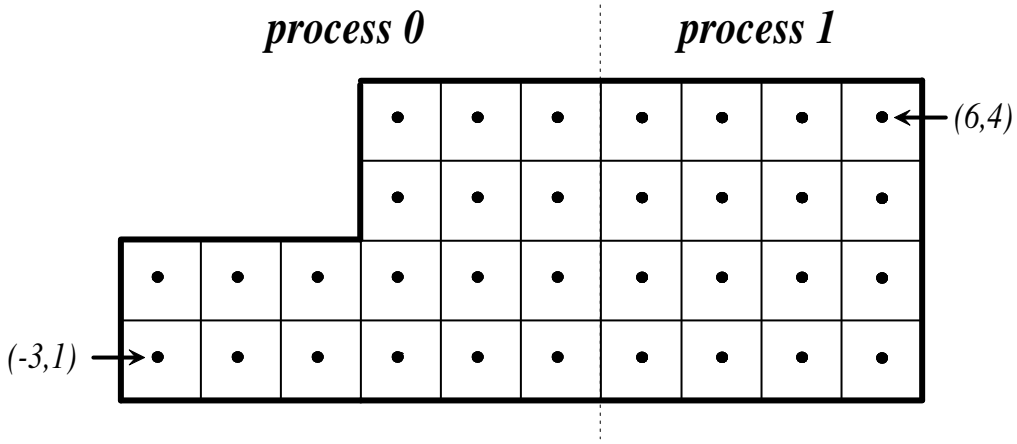


Fig. 1: Structured Grid Example
An example 2D structured grid, distributed accross two processors.

There are four basic steps involved in setting up the linear system to be solved:

1. set up the grid,
2. set up the stencil,
3. set up the matrix,
4. set up the right-hand-side vector.

To describe each of these steps in more detail, consider solving the 2D Laplacian problem

$$\begin{cases} \nabla^2 u = f, & \text{in the domain,} \\ u = 0, & \text{on the boundary.} \end{cases} \tag{2.1}$$

Assume (2.1) is discretized using standard 5-pt finite-volumes on the uniform grid pictured in *Structured Grid Example*, and assume that the problem data is distributed across two processes as depicted.

2.1 Setting Up the Struct Grid

The grid is described via a global *index space*, i.e., via integer singles in 1D, tuples in 2D, or triples in 3D (see Figure *Boxes in Index Space*).

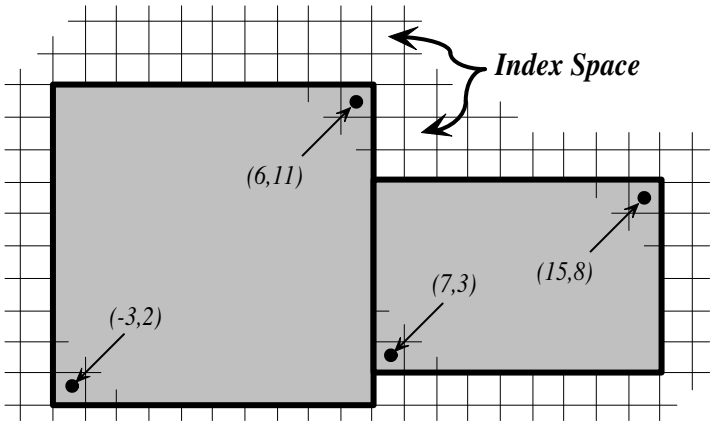
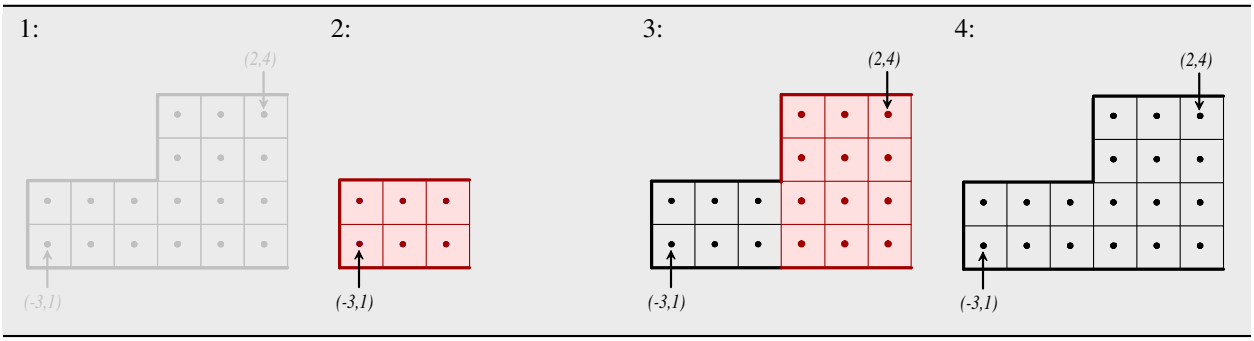


Fig. 2: Boxes in Index Space
A box is a collection of abstract cell-centered indices, described by its minimum and maximum indices. Here, two boxes are illustrated.

The integers may have any value, negative or positive. The global indexes allow hypre to discern how data is related spatially, and how it is distributed across the parallel machine. The basic component of the grid is a *box*: a collection of abstract cell-centered indices in index space, described by its "lower" and "upper" corner indices. The scalar grid data is always associated with cell centers, unlike the more general `SStruct` interface which allows data to be associated with box indices in several different ways.

Each process describes that portion of the grid that it "owns", one box at a time. For example, the global grid in Figure *Structured Grid Example* can be described in terms of three boxes, two owned by process 0, and one owned by process 1. The following is the code (with visual annotations) for setting up the grid on process 0 (the code for process 1 is similar).




```
   HYPRE_StructGrid grid;
   int ndim        = 2;
   int ilower[][2] = {{-3,1}, {0,1}};
   int iupper[][2] = {{-1,2}, {2,4}};

   /* Create the grid object */
1: HYPRE_StructGridCreate(MPI_COMM_WORLD, ndim, &grid);
```

(continues on next page)

(continued from previous page)

```
    /* Set grid extents for the first box */
2:  HYPRE_StructGridSetExtents(grid, ilower[0], iupper[0]);

    /* Set grid extents for the second box */
3:  HYPRE_StructGridSetExtents(grid, ilower[1], iupper[1]);

    /* Assemble the grid */
4:  HYPRE_StructGridAssemble(grid);
```

The images along the top illustrate the result of the numbered lines of code. The `Create()` routine creates an empty 2D grid object that lives on the `MPI_COMM_WORLD` communicator. The `SetExtents()` routine adds a new box to the grid. The `Assemble()` routine is a collective call (i.e., must be called on all processes from a common synchronization point), and finalizes the grid assembly, making the grid "ready to use".

2.2 Setting Up the Struct Stencil

The geometry of the discretization stencil is described by an array of indexes, each representing a relative offset from any given gridpoint on the grid. For example, the geometry of the 5-pt stencil for the example problem being considered can be represented by the list of index offsets shown in Figure *Figure 4a*.

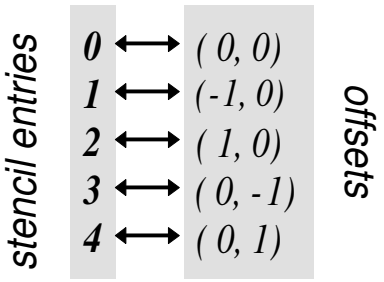


Fig. 3: Figure 4a
Representation of the 5-point discretization stencil for the example problem.

Here, the $(0, 0)$ entry represents the "center" coefficient, and is the 0th stencil entry. The $(0, -1)$ entry represents the "south" coefficient, and is the 3rd stencil entry. And so on.

On process 0 or 1, the following code (with visual annotations) will set up the stencil in Figure *Figure 4a*. The stencil must be the same on all processes.

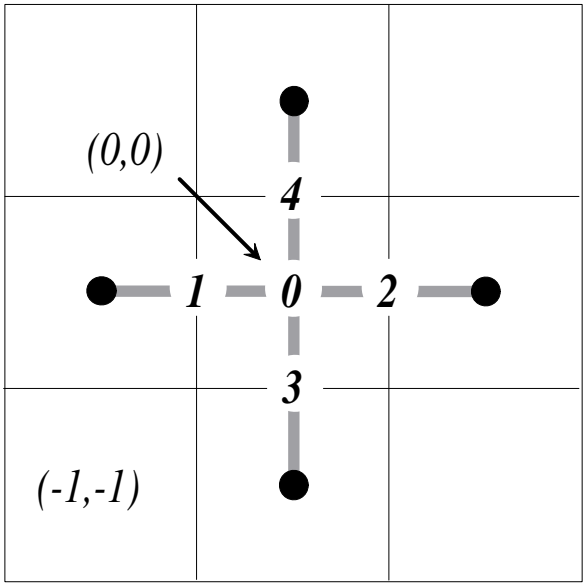
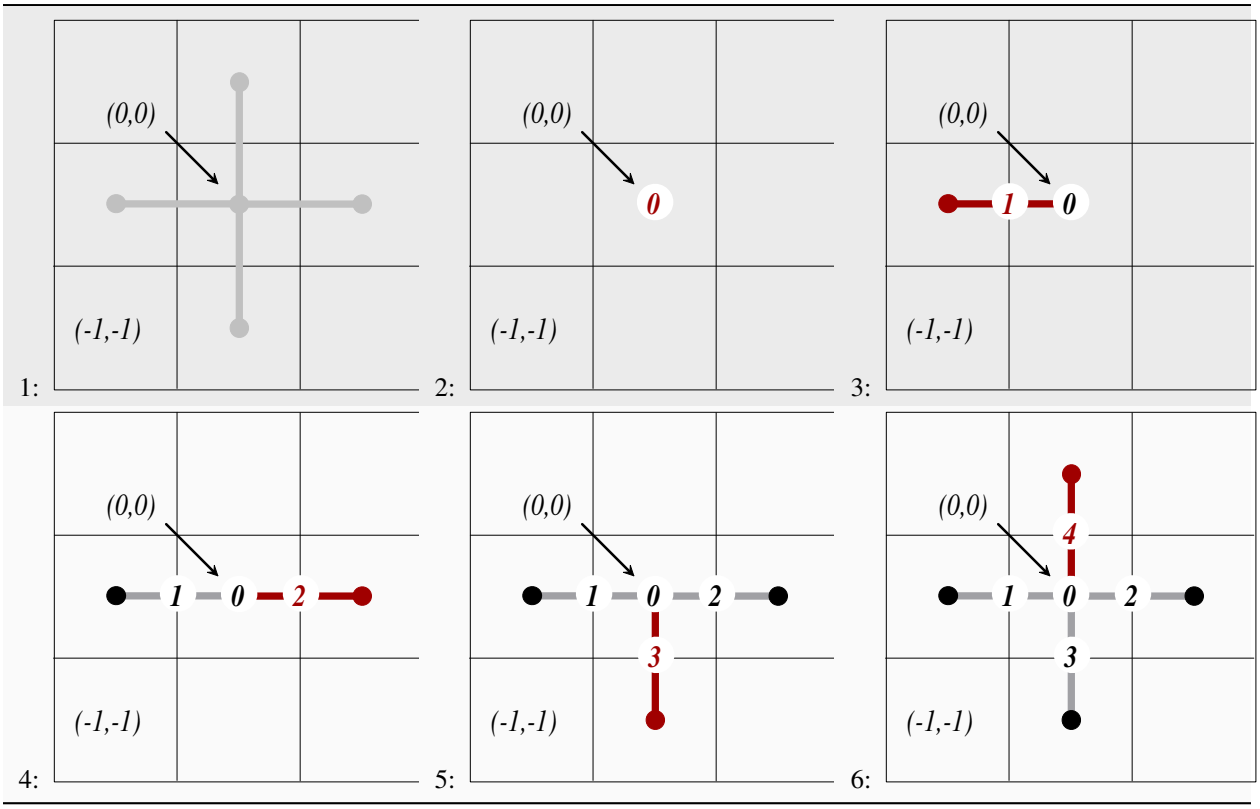


Fig. 4: Figure 4b
Need to combine this with 4a.




```
    HYPRE_StructStencil stencil;
    int ndim          = 2;
    int size          = 5;
```

(continues on next page)

(continued from previous page)

```
      int entry;
      int offsets[][2] = {{0,0}, {-1,0}, {1,0}, {0,-1}, {0,1}};

      /* Create the stencil object */
  1:  HYPRE_StructStencilCreate(ndim, size, &stencil);

      /* Set stencil entries */
      for (entry = 0; entry < size; entry++)
      {
2-6:     HYPRE_StructStencilSetElement(stencil, entry, offsets[entry]);
      }

      /* Thats it!  There is no assemble routine */
```

The `Create()` routine creates an empty 2D, 5-pt stencil object. The `SetElement()` routine defines the geometry of the stencil and assigns the stencil numbers for each of the stencil entries. None of the calls are collective calls.

2.3 Setting Up the Struct Matrix

The matrix is set up in terms of the grid and stencil objects described in Sections *Setting Up the Struct Grid* and *Setting Up the Struct Stencil*. The coefficients associated with each stencil entry will typically vary from gridpoint to gridpoint, but in the example problem being considered, they are as follows over the entire grid (except at boundaries; see below):

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}. \tag{2.2}$$

On process 0, the following code sets up matrix values associated with the center (entry 0) and south (entry 3) stencil entries as given by (2.2) and Figure *Figure 4a* (boundaries are ignored here temporarily).

```
HYPRE_StructMatrix  A;
double              values[36];
int                 stencil_indices[2] = {0,3};
int                 i;

HYPRE_StructMatrixCreate(MPI_COMM_WORLD, grid, stencil, &A);
HYPRE_StructMatrixInitialize(A);

for (i = 0; i < 36; i += 2)
{
   values[i]   =  4.0;
   values[i+1] = -1.0;
}

HYPRE_StructMatrixSetBoxValues(A, ilower[0], iupper[0], 2,
                               stencil_indices, values);
HYPRE_StructMatrixSetBoxValues(A, ilower[1], iupper[1], 2,
                               stencil_indices, values);

/* set boundary conditions */
...
```

(continues on next page)

(continued from previous page)

```
HYPRE_StructMatrixAssemble(A);
```

The `Create()` routine creates an empty matrix object. The `Initialize()` routine indicates that the matrix coefficients (or values) are ready to be set. This routine may or may not involve the allocation of memory for the coefficient data, depending on the implementation. The optional `Set` routines mentioned later in this chapter and in the Reference Manual, should be called before this step. The `SetBoxValues()` routine sets the matrix coefficients for some set of stencil entries over the gridpoints in some box. Note that the box need not correspond to any of the boxes used to create the grid, but values should be set for all gridpoints that this process "owns". The `Assemble()` routine is a collective call, and finalizes the matrix assembly, making the matrix "ready to use".

Matrix coefficients that reach outside of the boundary should be set to zero. For efficiency reasons, hypre does not do this automatically. The most natural time to insure this is when the boundary conditions are being set, and this is most naturally done after the coefficients on the grid's interior have been set. For example, during the implementation of the Dirichlet boundary condition on the lower boundary of the grid in Figure *Structured Grid Example*, the south coefficient must be set to zero. To do this on process 0, the following code could be used:

```
int  ilower[2] = {-3, 1};
int  iupper[2] = { 2, 1};

/* create matrix and set interior coefficients */
...

/* implement boundary conditions */
...

for (i = 0; i < 12; i++)
{
   values[i] =  0.0;
}

i = 3;
HYPRE_StructMatrixSetBoxValues(A, ilower, iupper, 1, &i, values);

/* complete implementation of boundary conditions */
...
```

2.4 Setting Up the Struct Right-Hand-Side Vector

The right-hand-side vector is set up similarly to the matrix set up described in Section *Setting Up the Struct Matrix* above. The main difference is that there is no stencil (note that a stencil currently does appear in the interface, but this will eventually be removed).

On process 0, the following code sets up the right-hand-side vector values.

```
HYPRE_StructVector  b;
double              values[18];
int                 i;

HYPRE_StructVectorCreate(MPI_COMM_WORLD, grid, &b);
HYPRE_StructVectorInitialize(b);
```

(continues on next page)

(continued from previous page)

```
for (i = 0; i < 18; i++)
{
   values[i]   =  0.0;
}

HYPRE_StructVectorSetBoxValues(b, ilower[0], iupper[0], values);
HYPRE_StructVectorSetBoxValues(b, ilower[1], iupper[1], values);

HYPRE_StructVectorAssemble(b);
```

The `Create()` routine creates an empty vector object. The `Initialize()` routine indicates that the vector coefficients (or values) are ready to be set. This routine follows the same rules as its corresponding `Matrix` routine. The `SetBoxValues()` routine sets the vector coefficients over the gridpoints in some box, and again, follows the same rules as its corresponding `Matrix` routine. The `Assemble()` routine is a collective call, and finalizes the vector assembly, making the vector "ready to use".

2.5 Symmetric Matrices

Some solvers and matrix storage schemes provide capabilities for significantly reducing memory usage when the coefficient matrix is symmetric. In this situation, each off-diagonal coefficient appears twice in the matrix, but only one copy needs to be stored. The `Struct` interface provides support for matrix and solver implementations that use symmetric storage via the `SetSymmetric()` routine.

To describe this in more detail, consider again the 5-pt finite-volume discretization of (2.1) on the grid pictured in Figure *Structured Grid Example*. Because the discretization is symmetric, only half of the off-diagonal coefficients need to be stored. To turn symmetric storage on, the following line of code needs to be inserted somewhere between the `Create()` and `Initialize()` calls.

```
HYPRE_StructMatrixSetSymmetric(A, 1);
```

The coefficients for the entire stencil can be passed in as before. Note that symmetric storage may or may not actually be used, depending on the underlying storage scheme. Currently in hypre, the `Struct` interface always uses symmetric storage.

To most efficiently utilize the `Struct` interface for symmetric matrices, notice that only half of the off-diagonal coefficients need to be set. To do this for the example being considered, we simply need to redefine the 5-pt stencil of Section *Setting Up the Struct Stencil* to an "appropriate" 3-pt stencil, then set matrix coefficients (as in Section *Setting Up the Struct Matrix*) for these three stencil elements *only*. For example, we could use the following stencil

$$\begin{bmatrix} & (0,1) & \\ & (0,0) & (1,0) \\ & & \end{bmatrix}. \tag{2.3}$$

This 3-pt stencil provides enough information to recover the full 5-pt stencil geometry and associated matrix coefficients.

CHAPTER

THREE

SEMI-STRUCTURED-GRID SYSTEM INTERFACE (SSTRUCT)

The `SStruct` interface is appropriate for applications with grids that are mostly—but not entirely—structured, e.g. block-structured grids (see Figure *Figure 6a*), composite grids in structured adaptive mesh refinement (AMR) applications (see Figure *Figure 9*), and overset grids. In addition, it supports more general PDEs than the `Struct` interface by allowing multiple variables (system PDEs) and multiple variable types (e.g. cell-centered, face-centered, etc.). The interface provides access to data structures and linear solvers in hypre that are designed for semi-structured grid problems, but also to the most general data structures and solvers.

The `SStruct` grid is composed out of a number of structured grid *parts*, where the physical inter-relationship between the parts is arbitrary. Each part is constructed out of two basic components: boxes (see Figure *Boxes in Index Space*) and *variables*. Variables represent the actual unknown quantities in the grid, and are associated with the box indices in a variety of ways, depending on their types. In hypre, variables may be cell-centered, node-centered, face-centered, or edge-centered. Face-centered variables are split into x-face, y-face, and z-face, and edge-centered variables are split into x-edge, y-edge, and z-edge. See Figure *Figure 5* for an illustration in 2D.

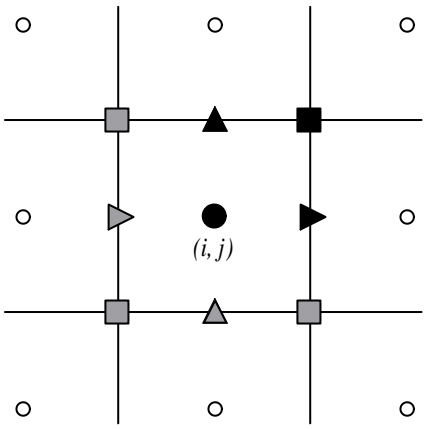


Fig. 1: Figure 5
Grid variables in hypre are referenced by the abstract cell-centered index to the left and down in 2D (analogously in 3D). In the figure, index (i, j) is used to reference the variables in black. The variables in grey—although contained in the pictured cell—are not referenced by the (i, j) index.

The `SStruct` interface uses a *graph* to allow nearly arbitrary relationships between part data. The graph is constructed from stencils or finite element stiffness matrices plus some additional data-coupling information set by the `GraphAddEntries()` routine. Two other methods for relating part data are the `GridSetNeighborPart()` and `GridSetSharedPart()` routines, which are particularly well suited for block-structured grid problems. The latter is useful for finite element codes.

There are five basic steps involved in setting up the linear system to be solved:

1. set up the grid,

2. set up the stencils (if needed),
3. set up the graph,
4. set up the matrix,
5. set up the right-hand-side vector.

3.1 Block-Structured Grids with Stencils

In this section, we describe how to use the `SStruct` interface to define block-structured grid problems. We do this primarily by example, paying particular attention to the construction of stencils and the use of the `GridSetNeighborPart()` interface routine.

Consider the solution of the diffusion equation

$$-\nabla \cdot (D\nabla u) + \sigma u = f \tag{3.1}$$

on the block-structured grid in Figure *Figure 6a*, where D is a scalar diffusion coefficient, and $\sigma \geq 0$. The discretization [MoRS1998] introduces three different types of variables: cell-centered, x-face, and y-face. The three discretization stencils that couple these variables are also given in the figure. The information in this figure is essentially all that is needed to describe the nonzero structure of the linear system we wish to solve.

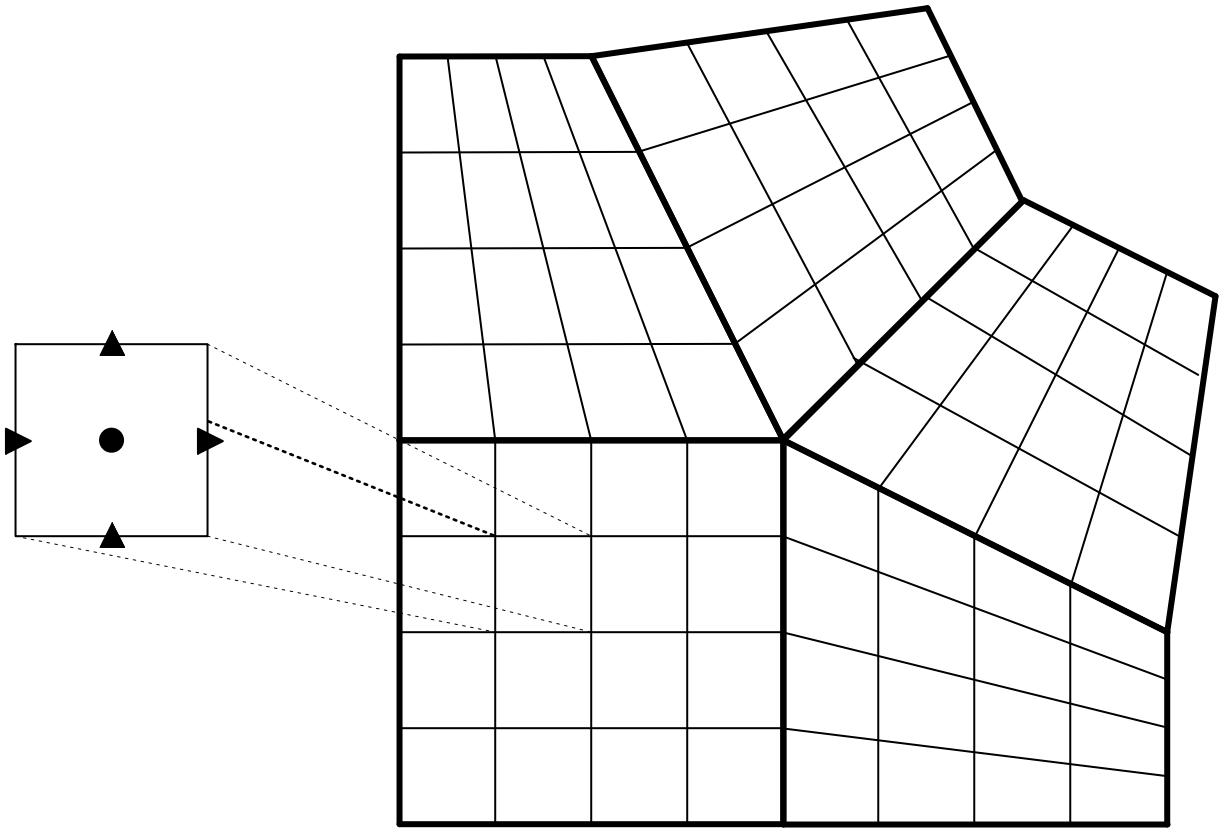


Fig. 2: Figure 6a
Example of a block-structured grid with five logically-rectangular blocks and three variables types: cell-centered, x-face, and y-face. Discretization stencils for the cell-centered (left), x-face (middle), and y-face (right) variables are also pictured.

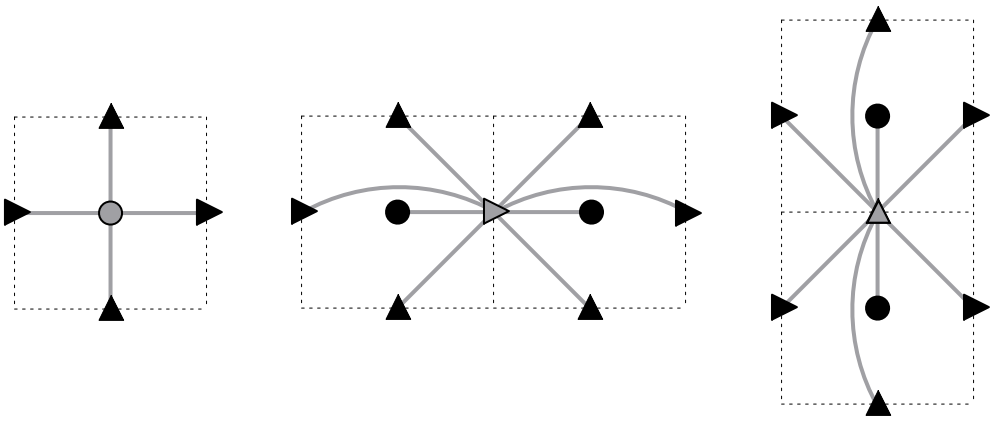


Fig. 3: Figure 6b
Need to combine this with 6a.

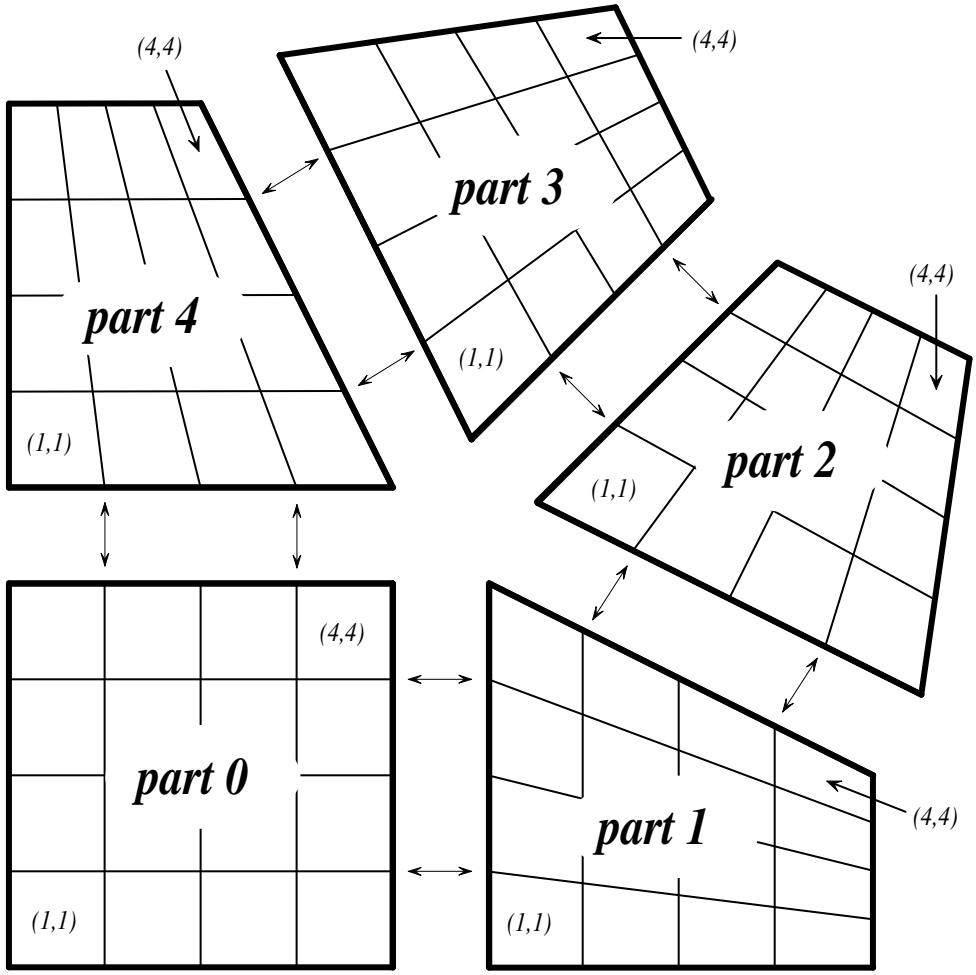
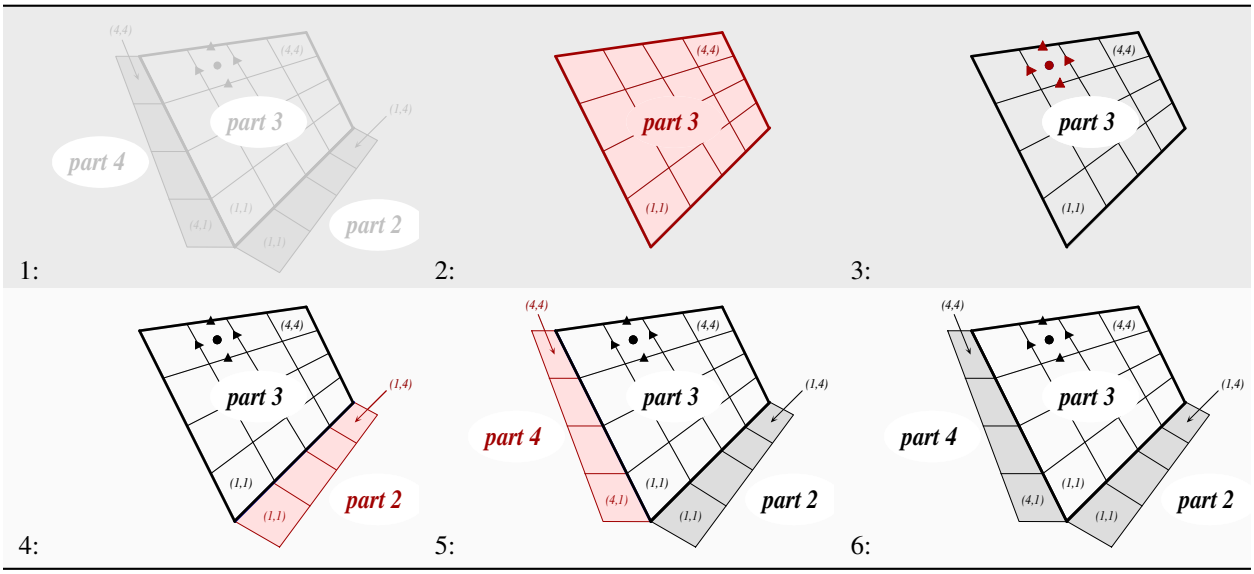


Fig. 4: Figure 7
One possible labeling of the grid in Figure *Figure 6a*.

The grid in Figure *Figure 6a* is defined in terms of five separate logically-rectangular parts as shown in Figure *Figure 7*, and each part is given a unique label between 0 and 4. Each part consists of a single box with lower index $(1, 1)$ and upper index $(4, 4)$ (see Section *Setting Up the Struct Grid*), and the grid data is distributed on five processes such that data associated with part p lives on process p. Note that in general, parts may be composed out of arbitrary unions of boxes, and indices may consist of non-positive integers (see Figure *Boxes in Index Space*). Also note that the `SStruct` interface expects a domain-based data distribution by boxes, but the actual distribution is determined by the user and simply described (in parallel) through the interface.




```
   HYPRE_SStructGrid grid;
   int ndim = 2, nparts = 5, nvars = 3, part = 3;
   int extents[][2] = {{1,1}, {4,4}};
   int vartypes[]   = {HYPRE_SSTRUCT_VARIABLE_CELL,
                       HYPRE_SSTRUCT_VARIABLE_XFACE,
                       HYPRE_SSTRUCT_VARIABLE_YFACE};
   int nb2_n_part      = 2,              nb4_n_part      = 4;
   int nb2_exts[][2]   = {{1,0}, {4,0}}, nb4_exts[][2]   = {{0,1}, {0,4}};
   int nb2_n_exts[][2] = {{1,1}, {1,4}}, nb4_n_exts[][2] = {{4,1}, {4,4}};
   int nb2_map[2]      = {1,0},          nb4_map[2]      = {0,1};
   int nb2_dir[2]      = {1,-1},         nb4_dir[2]      = {1,1};

1: HYPRE_SStructGridCreate(MPI_COMM_WORLD, ndim, nparts, &grid);

   /* Set grid extents and grid variables for part 3 */
2: HYPRE_SStructGridSetExtents(grid, part, extents[0], extents[1]);
3: HYPRE_SStructGridSetVariables(grid, part, nvars, vartypes);

   /* Set spatial relationship between parts 3 and 2, then parts 3 and 4 */
4: HYPRE_SStructGridSetNeighborPart(grid, part, nb2_exts[0], nb2_exts[1],
      nb2_n_part, nb2_n_exts[0], nb2_n_exts[1], nb2_map, nb2_dir);
5: HYPRE_SStructGridSetNeighborPart(grid, part, nb4_exts[0], nb4_exts[1],
      nb4_n_part, nb4_n_exts[0], nb4_n_exts[1], nb4_map, nb4_dir);

6: HYPRE_SStructGridAssemble(grid);
```

Code on process 3 for setting up the grid in Figure fig-sstruct-example}.

As with the `Struct` interface, each process describes that portion of the grid that it "owns", one box at a time. Figure fig-sstruct-grid shows the code for setting up the grid on process 3 (the code for the other processes is similar). The "icons" at the top of the figure illustrate the result of the numbered lines of code. Process 3 needs to describe the data pictured in the bottom-right of the figure. That is, it needs to describe part 3 plus some additional neighbor information that ties part 3 together with the rest of the grid. The `Create()` routine creates an empty 2D grid object with five parts that lives on the `MPI_COMM_WORLD` communicator. The `SetExtents()` routine adds a new box to the grid. The `SetVariables()` routine associates three variables of type cell-centered, x-face, and y-face with part 3.

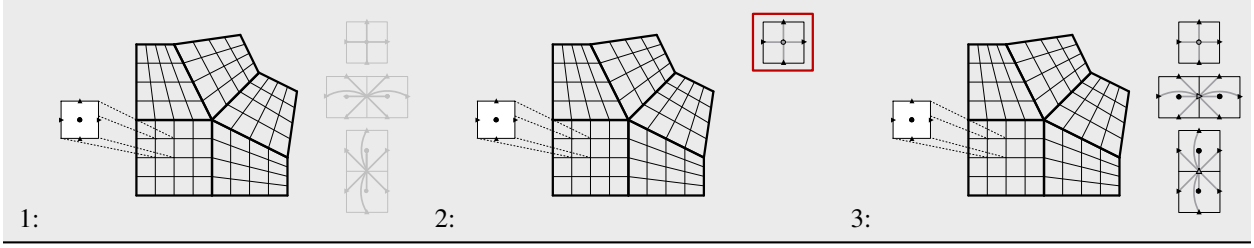
At this stage, the description of the data on part 3 is complete. However, the spatial relationship between this data and the data on neighboring parts is not yet defined. To do this, we need to relate the index space for part 3 with the index spaces of parts 2 and 4. More specifically, we need to tell the interface that the two grey boxes neighboring part 3 in the bottom-right of Figure fig-sstruct-grid also correspond to boxes on parts 2 and 4. This is done through the two calls to the `SetNeighborPart()` routine. We discuss only the first call, which describes the grey box on the right of the figure. Note that this grey box lives outside of the box extents for the grid on part 3, but it can still be described using the index-space for part 3 (recall Figure *Boxes in Index Space*). That is, the grey box has extents $(1, 0)$ and $(4, 0)$ on part 3's index-space, which is outside of part 3's grid. The arguments for the `SetNeighborPart()` call are simply the lower and upper indices on part 3 and the corresponding indices on part 2. The final two arguments to the routine indicate that the positive x-direction on part 3 (i.e., the i component of the tuple (i, j)) corresponds to the positive y-direction on part 2 and that the positive y-direction on part 3 corresponds to the positive x-direction on part 2.

The `Assemble()` routine is a collective call (i.e., must be called on all processes from a common synchronization point), and finalizes the grid assembly, making the grid "ready to use".

With the neighbor information, it is now possible to determine where off-part stencil entries couple. Take, for example, any shared part boundary such as the boundary between parts 2 and 3. Along these boundaries, some stencil entries reach outside of the part. If no neighbor information is given, these entries are effectively zeroed out, i.e., they don't participate in the discretization. However, with the additional neighbor information, when a stencil entry reaches into a neighbor box it is then coupled to the part described by that neighbor box information.

Another important consequence of the use of the `SetNeighborPart()` routine is that it can declare variables on different parts as being the same. For example, the face variables on the boundary of parts 2 and 3 are recognized as being shared by both parts (prior to the `SetNeighborPart()` call, there were two distinct sets of variables). Note also that these variables are of different types on the two parts; on part 2 they are x-face variables, but on part 3 they are y-face variables.

For brevity, we consider only the description of the y-face stencil in Figure *Figure 6a*, i.e. the third stencil in the figure. To do this, the stencil entries are assigned unique labels between 0 and 8 and their "offsets" are described relative to the "center" of the stencil. This process is illustrated in Figure *Figure 7a*. Nine calls are made to the routine `HYPRE_SStructStencilSetEntry()`. As an example, the call that describes stencil entry 5 in the figure is given the entry number 5, the offset $(-1, 0)$, and the identifier for the x-face variable (the variable to which this entry couples). Recall from Figure *Figure 5* the convention used for referencing variables of different types. The geometry description uses the same convention, but with indices numbered relative to the referencing index $(0, 0)$ for the stencil's center. Figure fig-sstruct-graph shows the code for setting up the graph .




```
HYPRE_SStructGraph graph;
HYPRE_SStructStencil c_stencil, x_stencil, y_stencil;
```

(continues on next page)

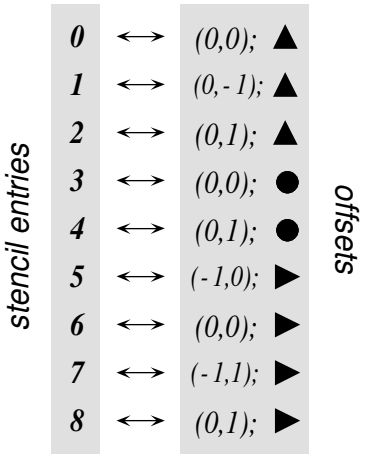


Fig. 5: Figure 7a
Assignment of labels and geometries to the y-face stencil in Figure fig-sstruct-example}.

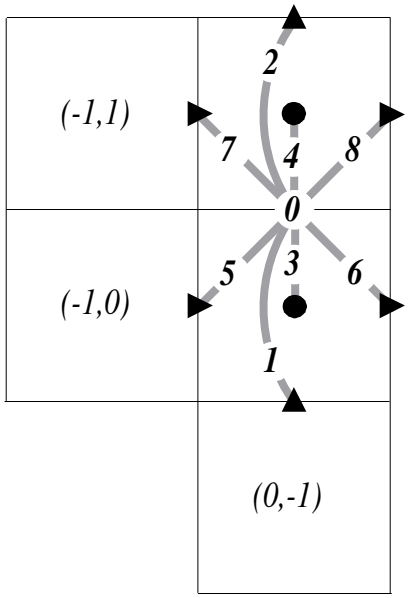


Fig. 6: Figure 7b
Need to combine this with 7a.

(continued from previous page)

```
   int c_var = 0, x_var = 1, y_var = 2;
   int part;

1: HYPRE_SStructGraphCreate(MPI_COMM_WORLD, grid, &graph);

   /* Set the cell-centered, x-face, and y-face stencils for each part */
   for (part = 0; part < 5; part++)
   {
2:    HYPRE_SStructGraphSetStencil(graph, part, c_var, c_stencil);
      HYPRE_SStructGraphSetStencil(graph, part, x_var, x_stencil);
      HYPRE_SStructGraphSetStencil(graph, part, y_var, y_stencil);
   }

3: HYPRE_SStructGraphAssemble(graph);
```

Code on process 3 for setting up the graph for Figure fig-sstruct-example}.

With the above, we now have a complete description of the nonzero structure for the matrix. The matrix coefficients are then easily set in a manner similar to what is described in Section *Setting Up the Struct Matrix* using routines `MatrixSetValues()` and `MatrixSetBoxValues()` in the `SStruct` interface. As before, there are also `AddTo` variants of these routines. Likewise, setting up the right-hand-side is similar to what is described in Section *Setting Up the Struct Right-Hand-Side Vector*. See the hypre reference manual for details.

An alternative approach for describing the above problem through the interface is to use the `GraphAddEntries()` routine instead of the `GridSetNeighborPart()` routine. In this approach, the five parts would be explicitly "sewn" together by adding non-stencil couplings to the matrix graph. The main downside to this approach for block-structured grid problems is that variables along block boundaries are no longer considered to be the same variables on the corresponding parts that share these boundaries. For example, any face variable along the boundary between parts 2 and 3 in Figure *Figure 6a* would represent two different variables that live on different parts. To "sew" the parts together correctly, we would need to explicitly select one of these variables as the representative that participates in the discretization, and make the other variable a dummy variable that is decoupled from the discretization by zeroing out appropriate entries in the matrix. All of these complications are avoided by using the `GridSetNeighborPart()` for this example.

3.2 Block-Structured Grids with Finite Elements

In this section, we describe how to use the `SStruct` interface to define block-structured grid problems with finite elements. We again do this by example, paying particular attention to the use of the `FEM` interface routines and the `GridSetSharedPart()` routine. See example code `ex14.c` for a complete implementation.

Consider a nodal finite element (FEM) discretization of the Laplace equation on the star-shaped grid in Figure *Figure 8a*. The local FEM stiffness matrix in the figure describes the coupling between the grid variables. Although we could still describe this problem using stencils as in Section *Block-Structured Grids with Stencils*, an FEM-based approach (available in hypre version `2.6.0b` and later) is a more natural alternative.

The grid in Figure *Figure 8a* is defined in terms of six separate logically-rectangular parts, and each part is given a unique label between 0 and 5. Each part consists of a single box with lower index $(1, 1)$ and upper index $(9, 9)$, and the grid data is distributed on six processes such that data associated with part p lives on process p.

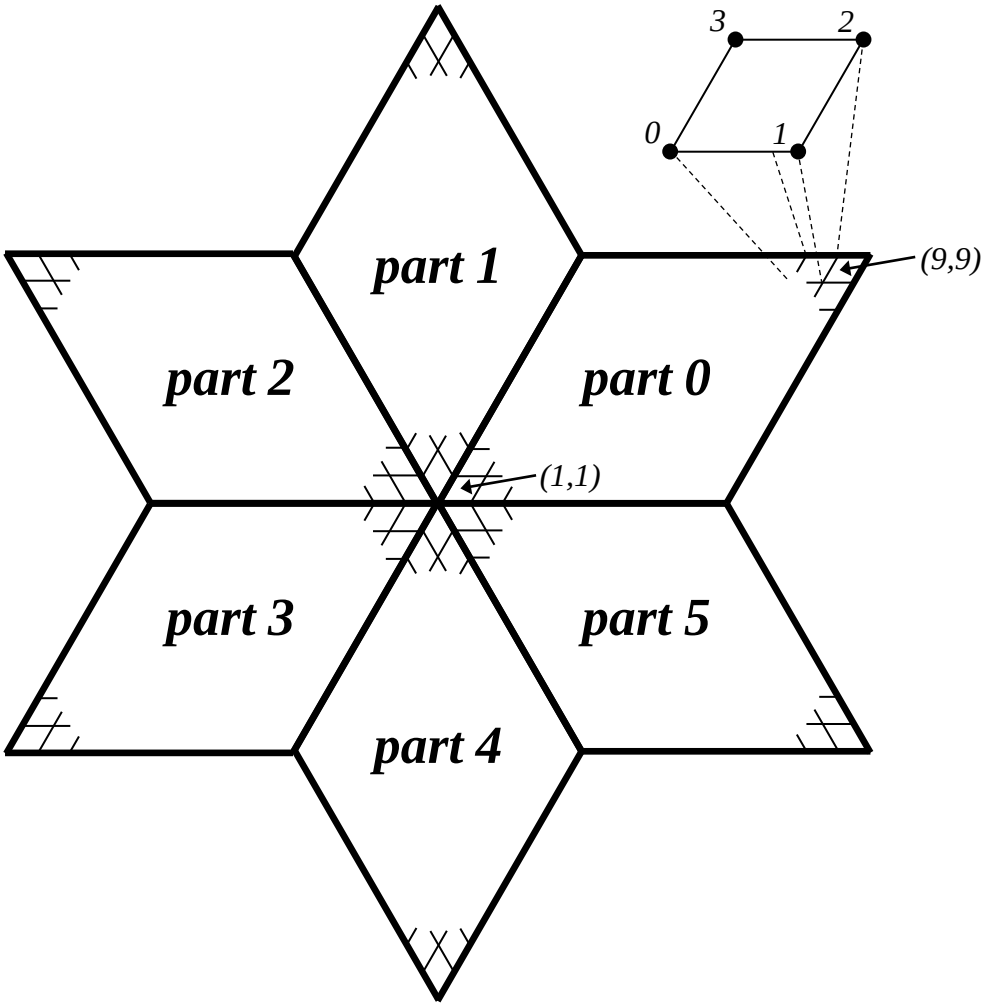
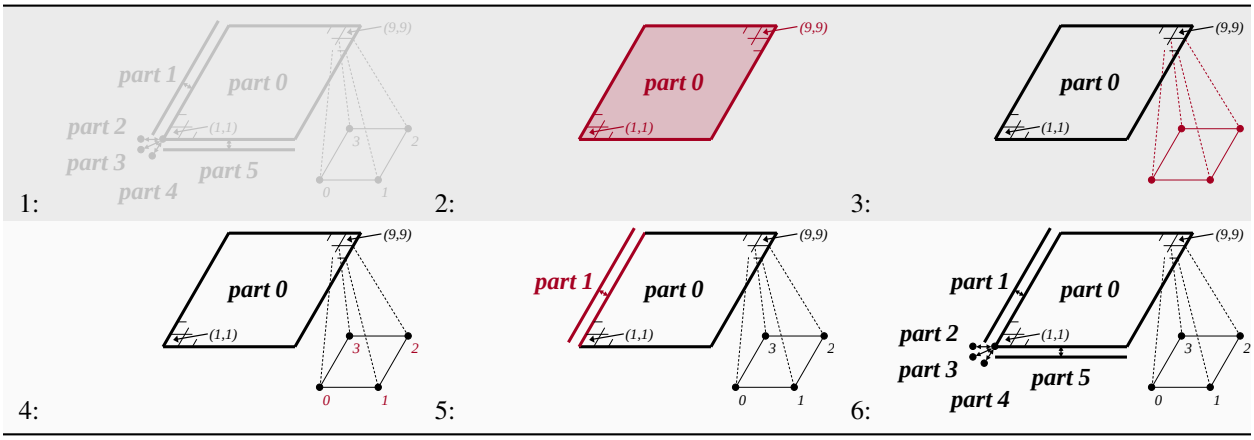


Fig. 7: Figure 8a
Example of a star-shaped grid with six logically-rectangular blocks and one nodal variable. Each block has an angle at the origin given by $\gamma = \pi/3$. The finite element stiffness matrix (right) is given in terms of the pictured variable ordering (left).

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 4-k & -1 & -2+k & -1 \\ -1 & 4+k & -1 & -2-k \\ -2+k & -1 & 4-k & -1 \\ -1 & -2-k & -1 & 4+k \end{pmatrix} \end{array} \alpha$$

$$\alpha = (6\sin(\gamma))^{-1}, \quad k = 3\cos(\gamma), \quad \gamma = \pi/3$$

Fig. 8: Figure 8b
Need to combine this with 8a.




```
    HYPRE_SStructGrid grid;
    int ndim = 2, nparts = 6, nvars = 1, part = 0;
    int ilower[2]    = {1,1}, iupper[2] = {9,9};
    int vartypes[]   = {HYPRE_SSTRUCT_VARIABLE_NODE};
    int ordering[12] = {0,-1,-1,  0,+1,-1,  0,+1,+1,  0,-1,+1};

    int s_part   = 2;
    int ilo[2]   = {1,1}, iup[2]   = {1,9}, offset[2]   = {-1,0};
    int s_ilo[2] = {1,1}, s_iup[2] = {9,1}, s_offset[2] = {0,-1};
    int map[2]   = {1,0};
    int dir[2]   = {-1,1};

1:  HYPRE_SStructGridCreate(MPI_COMM_WORLD, ndim, nparts, &grid);

    /* Set grid extents, grid variables, and FEM ordering for part 0 */
2:  HYPRE_SStructGridSetExtents(grid, part, ilower, iupper);
3:  HYPRE_SStructGridSetVariables(grid, part, nvars, vartypes);
4:  HYPRE_SStructGridSetFEMOrdering(grid, part, ordering);

    /* Set shared variables for parts 0 and 1 (0 and 2/3/4/5 not shown) */
5:  HYPRE_SStructGridSetSharedPart(grid, part, ilo, iup, offset,
       s_part, s_ilo, s_iup, s_offset, map, dir);

6:  HYPRE_SStructGridAssemble(grid);
```

Code on process 0 for setting up the grid in Figure *Figure 8a*.

As in Section *Block-Structured Grids with Stencils*, each process describes that portion of the grid that it "owns", one box at a time. Figure fig-sstruct-fem-grid shows the code for setting up the grid on process 0 (the code for the other processes is similar). The "icons" at the top of the figure illustrate the result of the numbered lines of code. Process 0 needs to describe the data pictured in the bottom-right of the figure. That is, it needs to describe part 0 plus some additional information about shared data with other parts on the grid. The `SetFEMOrdering()` routine sets the ordering of the unknowns in an element (an element is always a grid cell in hypre). This determines the ordering of the data passed into the routines `MatrixAddFEMValues()` and `VectorAddFEMValues()` discussed later.

At this point, the layout of the data on part 0 is complete, but there is no relationship to the rest of the grid. To couple the parts, we need to tell hypre that some of the boundary variables on part 0 are shared with other parts, i.e., they are the same as some of the variables on other parts. This is done through five calls to the `SetSharedPart()` routine. Only the first call is shown in the figure; the other four calls are similar. The arguments to this routine are the same as

`SetNeighborPart()` with the addition of two new offset arguments, named `offset` and `s_offset` in the figure. Each offset represents a pointer from the cell center to one of the following: all variables in the cell (no nonzeros in offset); all variables on a face (only 1 nonzero); all variables on an edge (2 nonzeros); all variables at a point (3 nonzeros). The two offsets must be consistent with each other.

The graph is set up similarly to Figure fig-sstruct-graph, except that the stencil calls are replaced by calls to `GraphSetFEM()`. The nonzero pattern of the stiffness matrix can also be set by calling the optional routine `GraphSetFEMSparsity()`.

Matrix and vector values are set one element at a time. For the example in this section, calls on part 0 would have the following form:

```
int part = 0;
int index[2] = {i,j};
double m_values[16] = {...};
double v_values[4]  = {...};

HYPRE_SStructMatrixAddFEMValues(A, part, index, m_values);
HYPRE_SStructVectorAddFEMValues(v, part, index, v_values);
```

Here, `m_values` contains local stiffness matrix values and `v_values` contains local variable values. The global matrix and vector are assembled internally by hypre, using the shared variables to couple the parts.

3.3 Structured Adaptive Mesh Refinement

We now briefly discuss how to use the `SStruct` interface in a structured AMR application. Consider Poisson's equation on the simple cell-centered example grid illustrated in Figure *Figure 9*. For structured AMR applications, each refinement level should be defined as a unique part. There are two parts in this example: part 0 is the global coarse grid and part 1 is the single refinement patch. Note that the coarse unknowns underneath the refinement patch (gray dots in Figure *Figure 9*) are not real physical unknowns; the solution in this region is given by the values on the refinement patch. In setting up the composite grid matrix [McCo1989] for hypre the equations for these "dummy" unknowns should be uncoupled from the other unknowns (this can easily be done by setting all off-diagonal couplings to zero in this region).

In the example, parts are distributed across the same two processes with process 0 having the "left" half of both parts. The composite grid is then set up part-by-part by making calls to `GridSetExtents()` just as was done in Section *Block-Structured Grids with Stencils* and Figure fig-sstruct-grid (no `SetNeighborPart` calls are made in this example). Note that in the interface there is no required rule relating the indexing on the refinement patch to that on the global coarse grid; they are separate parts and thus each has its own index space. In this example, we have chosen the indexing such that refinement cell $(2i, 2j)$ lies in the lower left quadrant of coarse cell (i, j). Then the stencil is set up. In this example we are using a finite volume approach resulting in the standard 5-point stencil in Section *Setting Up the Struct Grid* in both parts.

The grid and stencil are used to define all intra-part coupling in the graph, the non-zero pattern of the composite grid matrix. The inter-part coupling at the coarse-fine interface is described by `GraphAddEntries()` calls. This coupling in the composite grid matrix is typically the composition of an interpolation rule and a discretization formula. In this example, we use a simple piecewise constant interpolation, i.e. the solution value in a coarse cell is equal to the solution value at the cell center. Then the flux across a portion of the coarse-fine interface is approximated by a difference of the solution values on each side. As an example, consider approximating the flux across the left interface of cell $(6, 6)$ in Figure *Figure 2*. Let h be the coarse grid mesh size, and consider a local coordinate system with the origin at the

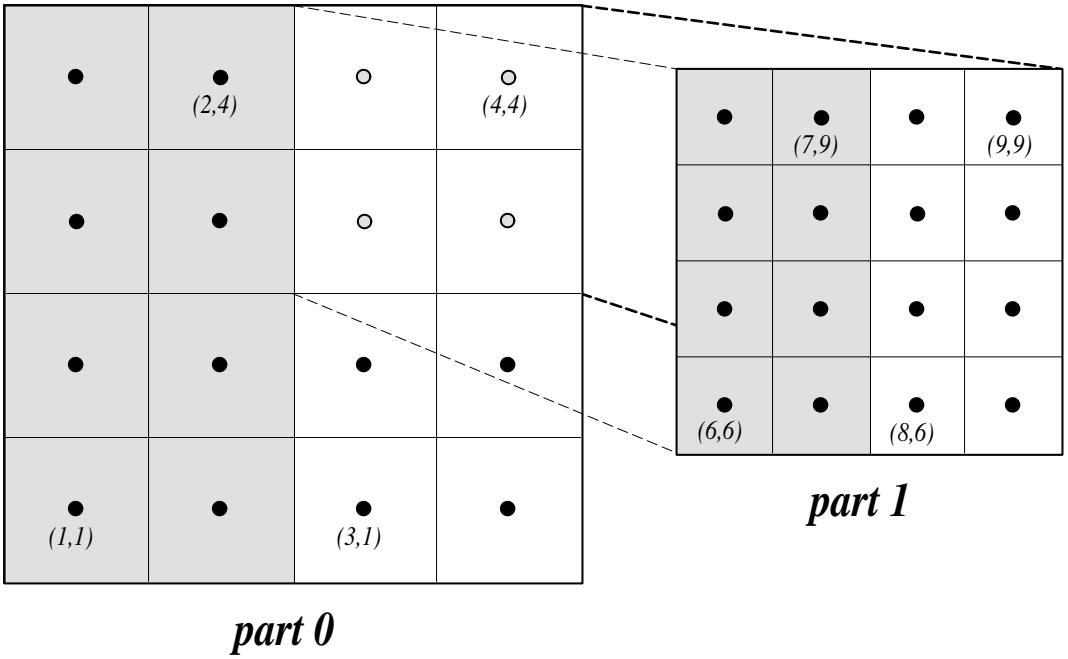


Fig. 9: Figure 9
Structured AMR grid example. Shaded regions correspond to process 0, unshaded to process 1. The grey dots are dummy variables.

center of cell $(6, 6)$. We approximate the flux as follows

$$\int_{-h/4}^{h/4} u_x(-h/4, s)ds \approx \frac{h}{2} u_x(-h/4, 0) \approx \frac{h}{2} \frac{u(0,0) - u(-3h/4, 0)}{3h/4}$$
$$\approx \frac{2}{3}(u_{6,6} - u_{2,3}).$$

The first approximation uses the midpoint rule for the edge integral, the second uses a finite difference formula for the derivative, and the third the piecewise constant interpolation to the solution in the coarse cell. This means that the equation for the variable at cell $(6, 6)$ involves not only the stencil couplings to $(6, 7)$ and $(7, 6)$ on part 1 but also non-stencil couplings to $(2, 3)$ and $(3, 2)$ on part 0. These non-stencil couplings are described by `GraphAddEntries()` calls. The syntax for this call is simply the part and index for both the variable whose equation is being defined and the variable to which it couples. After these calls, the non-zero pattern of the matrix (and the graph) is complete. Note that the "west" and "south" stencil couplings simply "drop off" the part, and are effectively zeroed out (currently, this is only supported for the `HYPRE_PARCSR` object type, and these values must be manually zeroed out for other object types; see `MatrixSetObjectType()` in the reference manual).

The remaining step is to define the actual numerical values for the composite grid matrix. This can be done by either `MatrixSetValues()` calls to set entries in a single equation, or by `MatrixSetBoxValues()` calls to set entries for a box of equations in a single call. The syntax for the `MatrixSetValues()` call is a part and index for the variable whose equation is being set and an array of entry numbers identifying which entries in that equation are being set. The entry numbers may correspond to stencil entries or non-stencil entries.

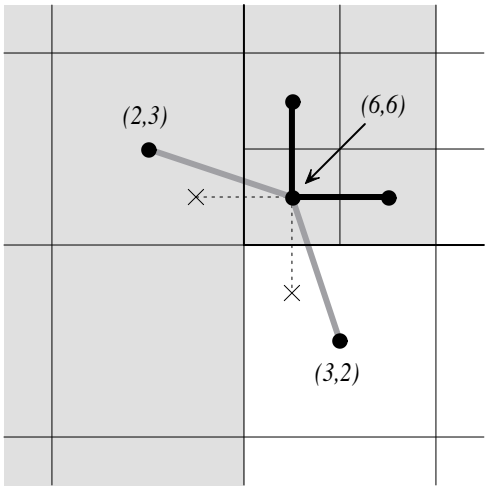


Fig. 10: Figure 2
Coupling for equation at corner of refinement patch. Black lines (solid and broken) are stencil couplings. Gray line are non-stencil couplings.

CHAPTER

FOUR

FINITE ELEMENT INTERFACE

4.1 Introduction

Many application codes use unstructured finite element meshes. This section describes an interface for finite element problems, called the FEI, which is supported in hypre.

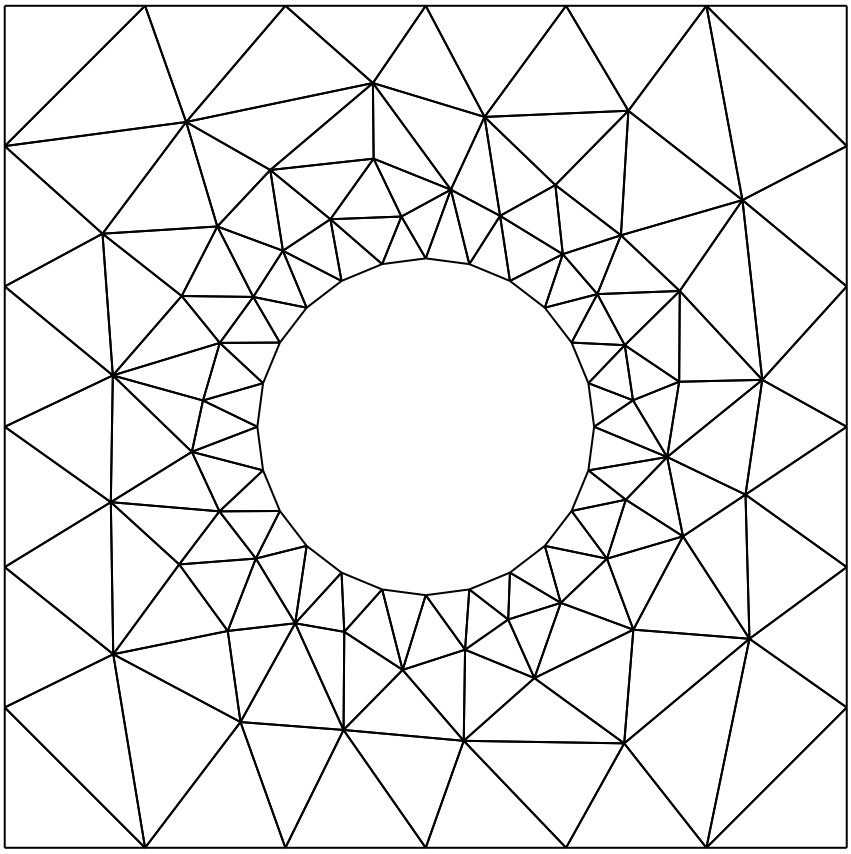


Fig. 1: Example of an unstructured mesh.

FEI refers to a specific interface for black-box finite element solvers, originally developed in Sandia National Lab, see [ClEA1999]. It differs from the rest of the conceptual interfaces in hypre in two important aspects: it is written in C++, and it does not separate the construction of the linear system matrix from the solution process. A complete description of Sandia's FEI implementation can be obtained by contacting Alan Williams at Sandia (william@sandia.gov).

A simplified version of the FEI has been implemented at LLNL and is included in hypre. More details about this implementation can be found in the header files of the `FEI_mv/fei-base` and `FEI_mv/fei-hypre` directories.

4.2 A Brief Description of the Finite Element Interface

Typically, finite element codes contain data structures storing element connectivities, element stiffness matrices, element loads, boundary conditions, nodal coordinates, etc. One of the purposes of the FEI is to assemble the global linear system in parallel based on such local element data. We illustrate this in the rest of the section and refer to example 10 (in the `examples` directory) for more implementation details.

In hypre, one creates an instance of the FEI as follows:

```
LLNL_FEI_Impl *feiPtr = new LLNL_FEI_Impl(mpiComm);
```

Here `mpiComm` is an MPI communicator (e.g. `MPI\_COMM\_WORLD`). If Sandia's FEI package is to be used, one needs to define a hypre solver object first:

```
LinearSystemCore   *solver = HYPRE_base_create(mpiComm);
FEI_Implementation *feiPtr = FEI_Implementation(solver,mpiComm,rank);
```

where `rank` is the number of the master processor (used only to identify which processor will produce the screen outputs). The `LinearSystemCore` class is the part of the FEI that interfaces with the linear solver library. It will be discussed later in Sections *FEI Solvers* and *Using HYPRE in External FEI Implementations*.

Local finite element information is passed to the FEI using several methods of the `feiPtr` object. The first entity to be submitted is the *field* information. A *field* has an identifier called `fieldID` and a rank or `fieldSize` (number of degree of freedom). For example, a discretization of the Navier Stokes equations in 3D can consist of velocity vector having 3 degrees of freedom in every node (vertex) of the mesh and a scalar pressure variable, which is constant over each element. If these are the only variables, and if we assign `fieldID` 7 and 8 to them, respectively, then the finite element field information can be set up by

```
nFields   = 2;                /* number of unknown fields */
fieldID   = new int[nFields]; /* field identifiers */
fieldSize = new int[nFields]; /* vector dimension of each field */

/* velocity (a 3D vector) */
fieldID[0]   = 7;
fieldSize[0] = 3;

/* pressure (a scalar function) */
fieldID[1]   = 8;
fieldSize[1] = 1;

feiPtr -> initFields(nFields, fieldSize, fieldID);
```

Once the field information has been established, we are ready to initialize an element block. An element block is characterized by the block identifier, the number of elements, the number of nodes per element, the nodal fields and the element fields (fields that have been defined previously). Suppose we use 1000 hexahedral elements in the element block 0, the setup consists of

```
elemBlkID  = 0;     /* identifier for a block of elements */
nElems     = 1000;  /* number of elements in the block */
elemNNodes = 8;     /* number of nodes per element */
```

(continues on next page)

(continued from previous page)

```
/* nodal-based field for the velocity */
nodeNFields     = 1;
nodeFieldIDs    = new[nodeNFields];
nodeFieldIDs[0] = fieldID[0];

/* element-based field for the pressure */
elemNFields     = 1;
elemFieldIDs    = new[elemNFields];
elemFieldIDs[0] = fieldID[1];

feiPtr -> initElemBlock(elemBlkID, nElems, elemNNodes, nodeNFields,
                        nodeFieldIDs, elemNFields, elemFieldIDs, 0);
```

The last argument above specifies how the dependent variables are arranged in the element matrices. A value of 0 indicates that each variable is to be arranged in a separate block (as opposed to interleaving).

In a parallel environment, each processor has one or more element blocks. Unless the element blocks are all disjoint, some of them share a common set of nodes on the subdomain boundaries. To facilitate setting up interprocessor communications, shared nodes between subdomains on different processors are to be identified and sent to the FEI. Hence, each node in the whole domain is assigned a unique global identifier. The shared node list on each processor contains a subset of the global node list corresponding to the local nodes that are shared with the other processors. The syntax for setting up the shared nodes is

```
feiPtr -> initSharedNodes(nShared, sharedIDs, sharedLengs, sharedProcs);
```

This completes the initialization phase, and a completion signal is sent to the FEI via

```
feiPtr -> initComplete();
```

Next, we begin the *load* phase. The first entity for loading is the nodal boundary conditions. Here we need to specify the number of boundary equations and the boundary values given by `alpha`, `beta`, and `gamma`. Depending on whether the boundary conditions are Dirichlet, Neumann, or mixed, the three values should be passed into the FEI accordingly.

```
feiPtr -> loadNodeBCs(nBCs, BCEqn, fieldID, alpha, beta, gamma);
```

The element stiffness matrices are to be loaded in the next step. We need to specify the element number i, the element block to which element i belongs, the element connectivity information, the element load, and the element matrix format. The element connectivity specifies a set of 8 node global IDs (for hexahedral elements), and the element load is the load or force for each degree of freedom. The element format specifies how the equations are arranged (similar to the interleaving scheme mentioned above). The calling sequence for loading element stiffness matrices is

```
for (i = 0; i < nElems; i++)
   feiPtr -> sumInElem(elemBlkID, elemID, elemConn[i], elemStiff[i],
                       elemLoads[i], elemFormat);
```

To complete the assembling of the global stiffness matrix and the corresponding right hand side, a signal is sent to the FEI via

```
feiPtr -> loadComplete();
```

CHAPTER

FIVE

LINEAR-ALGEBRAIC SYSTEM INTERFACE (IJ)

The IJ interface described in this chapter is the lowest common denominator for specifying linear systems in hypre. This interface provides access to general sparse-matrix solvers in hypre, not to the specialized solvers that require more problem information.

5.1 IJ Matrix Interface

As with the other interfaces in hypre, the IJ interface expects to get data in distributed form because this is the only scalable approach for assembling matrices on thousands of processes. Matrices are assumed to be distributed by blocks of rows as follows:

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{P-1} \end{bmatrix}$$

In the above example, the matrix is distributed across the P processes, $0, 1, ..., P-1$ by blocks of rows. Each submatrix A_p is "owned" by a single process and its first and last row numbers are given by the global indices `ilower` and `iupper` in the `Create()` call below.

The following example code illustrates the basic usage of the IJ interface for building matrices:

```
MPI_Comm            comm;
HYPRE_IJMatrix      ij_matrix;
HYPRE_ParCSRMatrix  parcsr_matrix;
int                 ilower, iupper;
int                 jlower, jupper;
int                 nrows;
int                *ncols;
int                *rows;
int                *cols;
double             *values;

HYPRE_IJMatrixCreate(comm, ilower, iupper, jlower, jupper, &ij_matrix);
HYPRE_IJMatrixSetObjectType(ij_matrix, HYPRE_PARCSR);
HYPRE_IJMatrixInitialize(ij_matrix);

/* set matrix coefficients */
HYPRE_IJMatrixSetValues(ij_matrix, nrows, ncols, rows, cols, values);
...
/* add-to matrix cofficients, if desired */
```

(continues on next page)

(continued from previous page)

```
HYPRE_IJMatrixAddToValues(ij_matrix, nrows, ncols, rows, cols, values);
...

HYPRE_IJMatrixAssemble(ij_matrix);
HYPRE_IJMatrixGetObject(ij_matrix, (void **) &parcsr_matrix);
```

The `Create()` routine creates an empty matrix object that lives on the `comm` communicator. This is a collective call (i.e., must be called on all processes from a common synchronization point), with each process passing its own row extents, `ilower` and `iupper`. The row partitioning must be contiguous, i.e., `iupper` for process `i` must equal `ilower`-1 for process `i`$+1$. Note that this allows matrices to have 0- or 1-based indexing. The parameters `jlower` and `jupper` define a column partitioning, and should match `ilower` and `iupper` when solving square linear systems. See the Reference Manual for more information.

The `SetObjectType()` routine sets the underlying matrix object type to `HYPRE_PARCSR` (this is the only object type currently supported). The `Initialize()` routine indicates that the matrix coefficients (or values) are ready to be set. This routine may or may not involve the allocation of memory for the coefficient data, depending on the implementation. The optional `SetRowSizes()` and `SetDiagOffdSizes()` routines mentioned later in this chapter and in the Reference Manual, should be called before this step.

The `SetValues()` routine sets matrix values for some number of rows (`nrows`) and some number of columns in each row (`ncols`). The actual row and column numbers of the matrix `values` to be set are given by `rows` and `cols`. The coefficients can be modified with the `AddToValues()` routine. If `AddToValues()` is used to add to a value that previously didn't exist, it will set this value. Note that while `AddToValues()` will add to values on other processors, `SetValues()` does not set values on other processors. Instead if a user calls `SetValues()` on processor i to set a matrix coefficient belonging to processor j, processor i will erase all previous occurrences of this matrix coefficient, so they will not contribute to this coefficient on processor j. The actual coefficient has to be set on processor j.

The `Assemble()` routine is a collective call, and finalizes the matrix assembly, making the matrix "ready to use". The `GetObject()` routine retrieves the built matrix object so that it can be passed on to hypre solvers that use the `ParCSR` internal storage format. Note that this is not an expensive routine; the matrix already exists in `ParCSR` storage format, and the routine simply returns a "handle" or pointer to it. Although we currently only support one underlying data storage format, in the future several different formats may be supported.

One can preset the row sizes of the matrix in order to reduce the execution time for the matrix specification. One can specify the total number of coefficients for each row, the number of coefficients in the row that couple the diagonal unknown to (`Diag`) unknowns in the same processor domain, and the number of coefficients in the row that couple the diagonal unknown to (`Offd`) unknowns in other processor domains:

```
HYPRE_IJMatrixSetRowSizes(ij_matrix, sizes);
HYPRE_IJMatrixSetDiagOffdSizes(matrix, diag_sizes, offdiag_sizes);
```

Once the matrix has been assembled, the sparsity pattern cannot be altered without completely destroying the matrix object and starting from scratch. However, one can modify the matrix values of an already assembled matrix. To do this, first call the `Initialize()` routine to re-initialize the matrix, then set or add-to values as before, and call the `Assemble()` routine to re-assemble before using the matrix. Re-initialization and re-assembly are very cheap, essentially a no-op in the current implementation of the code.

5.2 IJ Vector Interface

The following example code illustrates the basic usage of the `IJ` interface for building vectors:

```
MPI_Comm         comm;
HYPRE_IJVector   ij_vector;
HYPRE_ParVector  par_vector;
int              jlower, jupper;
int              nvalues;
int             *indices;
double          *values;

HYPRE_IJVectorCreate(comm, jlower, jupper, &ij_vector);
HYPRE_IJVectorSetObjectType(ij_vector, HYPRE_PARCSR);
HYPRE_IJVectorInitialize(ij_vector);

/* set vector values */
HYPRE_IJVectorSetValues(ij_vector, nvalues, indices, values);
...

HYPRE_IJVectorAssemble(ij_vector);
HYPRE_IJVectorGetObject(ij_vector, (void **) &par_vector);
```

The `Create()` routine creates an empty vector object that lives on the `comm` communicator. This is a collective call, with each process passing its own index extents, `jlower` and `jupper`. The names of these extent parameters begin with a `j` because we typically think of matrix-vector multiplies as the fundamental operation involving both matrices and vectors. For matrix-vector multiplies, the vector partitioning should match the column partitioning of the matrix (which also uses the `j` notation). For linear system solves, these extents will typically match the row partitioning of the matrix as well.

The `SetObjectType()` routine sets the underlying vector storage type to `HYPRE_PARCSR` (this is the only storage type currently supported). The `Initialize()` routine indicates that the vector coefficients (or values) are ready to be set. This routine may or may not involve the allocation of memory for the coefficient data, depending on the implementation.

The `SetValues()` routine sets the vector `values` for some number (`nvalues`) of `indices`. The values can be modified with the `AddToValues()` routine. Note that while `AddToValues()` will add to values on other processors, `SetValues()` does not set values on other processors. Instead if a user calls `SetValues()` on processor i to set a value belonging to processor j, processor i will erase all previous occurrences of this matrix coefficient, so they will not contribute to this value on processor j. The actual value has to be set on processor j.

The `Assemble()` routine is a trivial collective call, and finalizes the vector assembly, making the vector "ready to use". The `GetObject()` routine retrieves the built vector object so that it can be passed on to hypre solvers that use the `ParVector` internal storage format.

Vector values can be modified in much the same way as with matrices by first re-initializing the vector with the `Initialize()` routine.

5.3 A Scalable Interface

As explained in the previous sections, problem data is passed to the hypre library in its distributed form. However, as is typically the case for a parallel software library, some information regarding the global distribution of the data will be needed for hypre to perform its function. In particular, a solver algorithm requires that a processor obtain "nearby" data from other processors in order to complete the solve. While a processor may easily determine what data it needs from other processors, it may not know which processor owns the data it needs. Therefore, processors must determine their communication partners, or neighbors.

The straightforward approach to determining neighbors involves constructing a global partition of the data. This approach, however, requires $O(P)$ storage and computations and is not scalable for machines with tens of thousands of processors. The *assumed partition* algorithm was developed to address this problem [BaFY2006]. It is the approach used in hypre.

CHAPTER

SIX

SOLVERS AND PRECONDITIONERS

There are several solvers available in hypre via different conceptual interfaces:

	System Interfaces			
Solvers	Struct	SStruct	FEI	IJ
Jacobi	X	X		
SMG	X	X		
PFMG	X	X		
Split		X		
SysPFMG		X		
FAC		X		
Maxwell		X		
BoomerAMG		X	X	X
AMS		X	X	X
ADS		X	X	X
MLI		X	X	X
MGR				X
ParaSails		X	X	X
hypre-ILU				X
Euclid		X	X	X
PILUT		X	X	X
PCG	X	X	X	X
GMRES	X	X	X	X
FlexGMRES	X	X	X	X
LGMRES	X	X		X
BiCGSTAB	X	X	X	X
Hybrid	X	X	X	X
LOBPCG	X	X		X

Note that there are a few additional solvers and preconditioners not mentioned in the table that can be used only through the FEI interface and are described in Paragraph 6.14. The procedure for setup and use of solvers and preconditioners is largely the same. We will refer to them both as solvers in the sequel except when noted. In normal usage, the preconditioner is chosen and constructed before the solver, and then handed to the solver as part of the solver's setup. In the following, we assume the most common usage pattern in which a single linear system is set up and then solved with a single righthand side. We comment later on considerations for other usage patterns.

Setup:

1. **Pass to the solver the information defining the problem.** In the typical user cycle, the user has passed this information into a matrix through one of the conceptual interfaces prior to setting up the solver. In this situation, the problem definition information is then passed to the solver by passing the constructed matrix into the solver. As described before, the matrix and solver must be compatible, in that the matrix must provide the services needed

by the solver. Krylov solvers, for example, need only a matrix-vector multiplication. Most preconditioners, on the other hand, have additional requirements such as access to the matrix coefficients.

2. **Create the solver/preconditioner** via the `Create()` routine.
3. **Choose parameters for the preconditioner and/or solver.** Parameters are chosen through the `Set()` calls provided by the solver. Throughout hypre, we have made our best effort to give all parameters reasonable defaults if not chosen. However, for some preconditioners/solvers the best choices for parameters depend on the problem to be solved. We give recommendations in the individual sections on how to choose these parameters. Note that in hypre, convergence criteria can be chosen after the preconditioner/solver has been setup. For a complete set of all available parameters see the Reference Manual.
4. **Pass the preconditioner to the solver.** For solvers that are not preconditioned, this step is omitted. The preconditioner is passed through the `SetPrecond()` call.
5. **Set up the solver.** This is just the `Setup()` routine. At this point the matrix and right hand side is passed into the solver or preconditioner. Note that the actual right hand side is not used until the actual solve is performed.

At this point, the solver/preconditioner is fully constructed and ready for use.

Use:

1. **Set convergence criteria.** Convergence can be controlled by the number of iterations, as well as various tolerances such as relative residual, preconditioned residual, etc. Like all parameters, reasonable defaults are used. Users are free to change these, though care must be taken. For example, if an iterative method is used as a preconditioner for a Krylov method, a constant number of iterations is usually required.
2. **Solve the system.** This is just the `Solve()` routine.

Finalize:

1. **Free the solver or preconditioner.** This is done using the `Destroy()` routine.

Synopsis

In general, a solver (let's call it `SOLVER`) is set up and run using the following routines, where `A` is the matrix, `b` the right hand side and `x` the solution vector of the linear system to be solved:

```
/* Create Solver */
int HYPRE_SOLVERCreate(MPI_COMM_WORLD, &solver);

/* Set certain parameters if desired */
HYPRE_SOLVERSetTol(solver, 1.e-8);
...

/* Set up Solver */
HYPRE_SOLVERSetup(solver, A, b, x);

/* Solve the system */
HYPRE_SOLVERSolve(solver, A, b, x);

/* Destroy the solver */
HYPRE_SOLVERDestroy(solver);
```

In the following sections, we will give brief descriptions of the available hypre solvers with some suggestions on how to choose the parameters as well as references for users who are interested in a more detailed description and analysis of the solvers. A complete list of all routines that are available can be found in the reference manual.

6.1 SMG

SMG is a parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation,

$$\nabla \cdot (D \nabla u) + \sigma u = f$$

on logically rectangular grids. The code solves both 2D and 3D problems with discretization stencils of up to 9-point in 2D and up to 27-point in 3D. See [Scha1998], [BrFJ2000], [FaJo2000] for details on the algorithm and its parallel implementation/performance.

SMG is a particularly robust method. The algorithm semicoarsens in the z-direction and uses plane smoothing. The xy plane-solves are effected by one V-cycle of the 2D SMG algorithm, which semicoarsens in the y-direction and uses line smoothing.

6.2 PFMG

PFMG is a parallel semicoarsening multigrid solver similar to SMG. See [AsFa1996], [FaJo2000] for details on the algorithm and its parallel implementation/performance.

The main difference between the two methods is in the smoother: PFMG uses simple pointwise smoothing. As a result, PFMG is not as robust as SMG, but is much more efficient per V-cycle.

6.3 SysPFMG

SysPFMG is a parallel semicoarsening multigrid solver for systems of elliptic PDEs. It is a generalization of PFMG, with the interpolation defined only within the same variable. The relaxation is of nodal type- all variables at a given point location are simultaneously solved for in the relaxation.

Although SysPFMG is implemented through the SStruct interface, it can be used only for problems with one grid part. Ideally, the solver should handle any of the seven variable types (cell-, node-, xface-, yface-, zface-, xedge-, yedge-, and zedge-based). However, it has been completed only for cell-based variables.

6.4 SplitSolve

SplitSolve is a parallel block Gauss-Seidel solver for semi-structured problems with multiple parts. For problems with only one variable, it can be viewed as a domain-decomposition solver with no grid overlapping.

Consider a multiple part problem given by the linear system $Ax = b$. Matrix A can be decomposed into a structured intra-variable block diagonal component M and a component N consisting of the inter-variable blocks and any unstructured connections between the parts. SplitSolve performs the iteration

$$x_{k+1} = \tilde{M}^{-1}(b + Nx_k),$$

where $\tilde{M}^{-1}$ is a decoupled block-diagonal V(1,1) cycle, a separate cycle for each part and variable type. There are two V-cycle options, SMG and PFMG.

6.5 FAC

FAC is a parallel fast adaptive composite grid solver for finite volume, cell-centred discretizations of smooth diffusion coefficient problems. To be precise, it is a FACx algorithm since the patch solves consist of only relaxation sweeps. For details of the basic overall algorithms, see [McCo1989]. Algorithmic particularities include formation of non-Galerkin coarse-grid operators (i.e., coarse-grid operators underlying refinement patches are automatically generated) and non-stored linear/constant interpolation/restriction operators. Implementation particularities include a processor redistribution of the generated coarse-grid operators so that intra-level communication between adaptive mesh refinement (AMR) levels during the solve phase is kept to a minimum. This redistribution is hidden from the user.

The user input is essentially a linear system describing the *composite* operator, and the refinement factors between the AMR levels. To form this composite linear system, the AMR grid is described using semi-structured grid parts. Each AMR level grid corresponds to a separate part so that this level grid is simply a collection of boxes, all with the same refinement factor, i.e., it is a struct grid. However, several restrictions are imposed on the patch (box) refinements. First, a refinement box must cover all underlying coarse cells- i.e., refinement of a partial coarse cell is not permitted. Also, the refined/coarse indices must follow a mapping: with $[r_1, r_2, r_3]$ denoting the refinement factor and $[a_1, a_2, a_3] \times [b_1, b_2, b_3]$ denoting the coarse subbox to be refined, the mapping to the refined patch is

$$[r_1 * a_1, r_2 * a_2, r_3 * a_3] \times [r_1 * b_1 + r_1 - 1, r_2 * b_2 + r_2 - 1, r_3 * b_3 + r_3 - 1].$$

With the AMR grid constructed under these restrictions, the composite matrix can be formed. Since the AMR levels correspond to semi-structured grid parts, the composite matrix is a semi-structured matrix consisting of structured components within each part, and unstructured components describing the coarse-to-fine/fine-to-coarse connections. The structured and unstructured components can be set using stencils and the `HYPRE_SStructGraphAddEntries` routine, respectively. The matrix coefficients can be filled after setting these non-zero patterns. Between each pair of successive AMR levels, the coarse matrix underlying the refinement patch must be the identity and the corresponding rows of the rhs must be zero. These can performed using routines `HYPRE_SStructFACZeroCFSten` (to zero off the stencil values reaching from coarse boxes into refinement boxes), `HYPRE_SStructFACZeroFCSten` (to zero off the stencil values reaching from refinement boxes into coarse boxes), `HYPRE_SStructFACZeroAMRMatrixData` (to set the identity at coarse grid points underlying a refinement patch), and `HYPRE_SStructFACZeroAMRVectorData` (to zero off a vector at coarse grid points underlying a refinement patch). These routines can simplify the user's matrix setup. For example, consider two successive AMR levels with the coarser level consisting of one box and the finer level consisting of a collection of boxes. Rather than distinguishly setting the stencil values and the identity in the appropriate locations, the user can set the stencil values on the whole coarse grid using the `HYPRE_SStructMatrixSetBoxValues` routine and then zero off the appropriate values using the above zeroing routines.

The coarse matrix underlying these patches are algebraically generated by operator-collapsing the refinement patch operator and the fine-to-coarse coefficients (this is why stencil values reaching out of a part must be zeroed). This matrix is re-distributed so that each processor has all of its coarse-grid operator.

To solve the coarsest AMR level, a PFMG V cycle is used. Note that a minimum of two AMR levels are needed for this solver.

6.6 Maxwell

Maxwell is a parallel solver for edge finite element discretization of the curl-curl formulation of the Maxwell equation

$$\nabla \times \alpha \nabla \times E + \beta E = f, \beta > 0$$

on semi-structured grids. Details of the algorithm can be found in [JoLe2006]. The solver can be viewed as an operator-dependent multiple-coarsening algorithm for the Helmholtz decomposition of the error correction. Input to this solver consist of only the linear system and a gradient operator. In fact, if the orientation of the edge elements conforms to a lexicographical ordering of the nodes of the grid, then the gradient operator can be generated with the routine `HYPRE_MaxwellGrad`: at grid points (i, j, k) and $(i - 1, j, k)$, the produced gradient operator takes values 1 and -1

respectively, which is the correct gradient operator for the appropriate edge orientation. Since the gradient operator is normalized (i.e., h independent) the edge finite element must also be normalized in the discretization.

This solver is currently developed for perfectly conducting boundary condition (Dirichlet). Hence, the rows and columns of the matrix that corresponding to the grid boundary must be set to the identity or zeroed off. This can be achieved with the routines `HYPRE_SStructMaxwellPhysBdy` and `HYPRE_SStructMaxwellEliminateRowsCols`. The former identifies the ranks of the rows that are located on the grid boundary, and the latter adjusts the boundary rows and cols. As usual, the rhs of the linear system must also be zeroed off at the boundary rows. This can be done using `HYPRE_SStructMaxwellZeroVector`.

With the adjusted linear system and a gradient operator, the user can form the Maxwell multigrid solver using several different edge interpolation schemes. For problems with smooth coefficients, the natural Nedelec interpolation operator can be used. This is formed by calling `HYPRE_SStructMaxwellSetConstantCoef` with the flag > 0 before setting up the solver, otherwise the default edge interpolation is an operator-collapsing/element-agglomeration scheme. This is suitable for variable coefficients. Also, before setting up the solver, the user must pass the gradient operator, whether user or `HYPRE_MaxwellGrad` generated, with `HYPRE_SStructMaxwellSetGrad`. After these preliminary calls, the Maxwell solver can be setup by calling `HYPRE_SStructMaxwellSetup`.

There are two solver cycling schemes that can be used to solve the linear system. To describe these, one needs to consider the augmented system operator

$$\mathbf{A} = \begin{bmatrix} A_{ee} & A_{en} \\ A_{ne} & A_{nn} \end{bmatrix},$$

where A_{ee} is the stiffness matrix corresponding to the above curl-curl formulation, A_{nn} is the nodal Poisson operator created by taking the Galerkin product of A_{ee} and the gradient operator, and A_{ne} and A_{en} are the nodal-edge coupling operators (see [JoLe2006]). The algorithm for this Maxwell solver is based on forming a multigrid hierarchy to this augmented system using the block-diagonal interpolation operator

$$\mathbf{P} = \begin{bmatrix} P_e & 0 \\ 0 & P_n \end{bmatrix},$$

where P_e and P_n are respectively the edge and nodal interpolation operators determined individually from A_{ee} and A_{nn}. Taking a Galerkin product between $\mathbf{A}$ and $\mathbf{P}$ produces the next coarse augmented operator, which also has the nodal-edge coupling operators. Applying this procedure recursively produces nodal-edge coupling operators at all levels. Now, the first solver cycling scheme, `HYPRE_SStructMaxwellSolve`, keeps these coupling operators on all levels of the V-cycle. The second, cheaper scheme, `HYPRE_SStructMaxwellSolve2`, keeps the coupling operators only on the finest level, i.e., separate edge and nodal V-cycles that couple only on the finest level.

6.7 Hybrid

The hybrid solver is designed to detect whether a multigrid preconditioner is needed when solving a linear system and possibly avoid the expensive setup of a preconditioner if a system can be solved efficiently with a diagonally scaled Krylov solver, e.g. a strongly diagonally dominant system. It first uses a diagonally scaled Krylov solver, which can be chosen by the user (the default is conjugate gradient, but one should use GMRES if the matrix of the linear system to be solved is nonsymmetric). It monitors how fast the Krylov solver converges. If there is not sufficient progress, the algorithm switches to a preconditioned Krylov solver.

If used through the `Struct` interface, the solver is called StructHybrid and can be used with the preconditioners SMG and PFMG (default). It is called ParCSRHybrid, if used through the `IJ` interface and is used here with BoomerAMG. The user can determine the average convergence speed by setting a convergence tolerance $0 \leq \theta < 1$ via the routine `HYPRE_StructHybridSetConvergenceTol` or `HYPRE_StructParCSRHybridSetConvergenceTol`. The default setting is 0.9.

The average convergence factor $\rho_i = \left(\frac{\|r_i\|}{\|r_0\|}\right)^{1/i}$ is monitored within the chosen Krylov solver, where $r_i = b - Ax_i$ is the i-th residual. Convergence is considered too slow when

$$\left(1 - \frac{|\rho_i - \rho_{i-1}|}{\max(\rho_i, \rho_{i-1})}\right) \rho_i > \theta.$$

When this condition is fulfilled the hybrid solver switches from a diagonally scaled Krylov solver to a preconditioned solver.

6.8 BoomerAMG

BoomerAMG is a parallel implementation of the algebraic multigrid method [RuSt1987]. It can be used both as a solver or as a preconditioner. The user can choose between various different parallel coarsening techniques, interpolation and relaxation schemes. While the default settings work fairly well for two-dimensional diffusion problems, for three-dimensional diffusion problems, it is recommended to choose a lower complexity coarsening like HMIS or PMIS (coarsening 10 or 8) and combine it with a distance-two interpolation (interpolation 6 or 7), that is also truncated to 4 or 5 elements per row. Additional reduction in complexity and increased scalability can often be achieved using one or two levels of aggressive coarsening.

6.8.1 Parameter Options

Various BoomerAMG functions and options are mentioned below. However, for a complete listing and description of all available functions, see the reference manual.

BoomerAMG's Create function differs from the synopsis in that it has only one parameter `HYPRE_BoomerAMGCreate(HYPRE_Solver *solver)`. It uses the communicator of the matrix A.

6.8.2 Coarsening Options

Coarsening can be set by the user using the function `HYPRE_BoomerAMGSetCoarsenType`. A detailed description of various coarsening techniques can be found in [HeYa2002], [Yang2005].

Various coarsening techniques are available:

- the Cleary-Luby-Jones-Plassman (CLJP) coarsening,
- the Falgout coarsening which is a combination of CLJP and the classical RS coarsening algorithm,
- CGC and CGC-E coarsenings [GrMS2006a], [GrMS2006b],
- PMIS and HMIS coarsening algorithms which lead to coarsenings with lower complexities [DeYH2004] as well as
- aggressive coarsening, which can be applied to any of the coarsening techniques mentioned above a nd thus achieving much lower complexities and lower memory use [Stue1999].

To use aggressive coarsening the user has to set the number of levels to which he wants to apply aggressive coarsening (starting with the finest level) via `HYPRE_BoomerAMGSetAggNumLevels`. Since aggressive coarsening requires long range interpolation, multipass interpolation is always used on levels with aggressive coarsening, unless the user specifies another long-range interpolation suitable for aggressive coarsening.

Note that the default coarsening is HMIS [DeYH2004].

6.8.3 Interpolation Options

Various interpolation techniques can be set using `HYPRE_BoomerAMGSetInterpType`:

- the "classical" interpolation as defined in [RuSt1987],
- direct interpolation [Stue1999],
- standard interpolation [Stue1999],
- an extended "classical" interpolation, which is a long range interpolation and is recommended to be used with PMIS and HMIS coarsening for harder problems [DFNY2008],
- multipass interpolation [Stue1999],
- two-stage interpolation [Yang2010],
- Jacobi interpolation [Stue1999],
- the "classical" interpolation modified for hyperbolic PDEs.

Jacobi interpolation is only use to improve certain interpolation operators and can be used with `HYPRE_BoomerAMGSetPostInterpType`. Since some of the interpolation operators might generate large stencils, it is often possible and recommended to control complexity and truncate the interpolation operators using `HYPRE_BoomerAMGSetTruncFactor` and/or `HYPRE_BoomerAMGSetPMaxElmts`, or `HYPRE_BoomerAMGSetJacobiTruncTheshold` (for Jacobi interpolation only).

Note that the default interpolation is extended+i interpolation [DFNY2008] truncated to 4 elements per row.

6.8.4 Non-Galerkin Options

In order to reduce communication, there is a non-Galerkin coarse grid sparsification option available [FaSc2014]. This option can be used by itself or with existing strategies to reduce communication such as aggressive coarsening and HMIS coarsening. To use, call `HYPRE_BoomerAMGSetNonGalerkTol`, which gives BoomerAMG a list of level specific non-Galerkin drop tolerances. It is common to drop more aggressively on coarser levels. A common choice of drop-tolerances is $[0.0, 0.01, 0.05]$ where the value of 0.0 will skip the non-Galerkin process on the first coarse level (level 1), use a drop-tolerance of 0.01 on the second coarse level (level 2) and then use 0.05 on all subsequent coarse levels. While still experimental, this capability has significantly improved performance on a variety of problems. See the `ij` driver for an example usage and the reference manual for more details.

6.8.5 Smoother Options

A good overview of parallel smoothers and their properties can be found in [BFKY2011]. Various of the described relaxation techniques are available:

- weighted Jacobi relaxation,
- a hybrid Gauss-Seidel / Jacobi relaxation scheme,
- a symmetric hybrid Gauss-Seidel / Jacobi relaxation scheme,
- l1-Gauss-Seidel or Jacobi,
- Chebyshev smoothers,
- hybrid block and Schwarz smoothers [Yang2004],
- ILU and approximate inverse smoothers.

Point relaxation schemes can be set using `HYPRE_BoomerAMGSetRelaxType` or, if one wants to specifically set the up cycle, down cycle or the coarsest grid, with `HYPRE_BoomerAMGSetCycleRelaxType`. To use the more complicated smoothers, e.g. block, Schwarz, ILU smoothers, it is necessary to use `HYPRE_BoomerAMGSetSmoothType` and `HYPRE_BoomerAMGSetSmoothNumLevels`. There are further parameter choices for the individual smoothers, which are described in the reference manual. The default relaxation type is l1-Gauss-Seidel, using a forward solve on the down cycle and a backward solve on the up-cycle, to keep symmetry. Note that if BoomerAMG is used as a preconditioner for conjugate gradient, it is necessary to use a symmetric smoother. Other symmetric options are weighted Jacobi or hybrid symmetric Gauss-Seidel.

6.8.6 AMG for systems of PDEs

If the users wants to solve systems of PDEs and can provide information on which variables belong to which function, BoomerAMG's systems AMG version can also be used. Functions that enable the user to access the systems AMG version are `HYPRE_BoomerAMGSetNumFunctions`, `HYPRE_BoomerAMGSetDofFunc` and `HYPRE_BoomerAMGSetNodal`.

If the user can provide the near null-space vectors, such as the rigid body modes for linear elasticity problems, an interpolation is available that will incorporate these vectors with `HYPRE_BoomerAMGSetInterpVectors` and `HYPRE_BoomerAMGSetInterpVecVariant`. This can lead to improved convergence and scalability [BaKY2010].

6.8.7 Special AMG Cycles

The default cycle is a V(1,1)-cycle, however it is possible to change the number of sweeps of the up- and down-cycle as well as the coare grid. One can also choose a W-cycle, however for parallel processing this is not recommended, since it is not scalable.

BoomerAMG also provides an additive V(1,1)-cycle as well as a mult-additive V(1,1)-cycle and a simplified versioni [VaYa2014]. The additive variants can only be used with weighted Jacobi or l1-Jacobi smoothing.

6.8.8 GPU-supported Options

In general, CUDA unified memory is required for running BoomerAMG solvers on GPUs. However, hypre can also be built without `--enable-unified-memory` if all the selected parameters have GPU-support. The currently available GPU-supported BoomerAMG options include:

- Coarsening: PMIS (8)
- Interpolation: direct (3), BAMG-direct (15), extended (14), extended+i (6) and extended+e (18)
- Aggressive coarsening
- Second-stage interpolation with aggressive coarsening: extended (5) and extended+e (7)
- Smoother: Jacobi (7), l1-Jacobi (18), hybrid Gauss Seidel/SRROR (3 4 6), two-stage Gauss-Seidel (11,12) [BKRHSMTY2021]
- Relaxation order: must be 0, i.e., lexicographic order

A sample code of setting up IJ matrix A and solve $Ax = b$ using AMG-preconditioned CG on GPUs is shown below.

```
cudaSetDevice(device_id); /* GPU binding */
...
HYPRE_Init(); /* must be the first HYPRE function call */
...
/* AMG in GPU memory (default) */
HYPRE_SetMemoryLocation(HYPRE_MEMORY_DEVICE);
```

(continues on next page)

(continued from previous page)

```
/* setup AMG on GPUs */
HYPRE_SetExecutionPolicy(HYPRE_EXEC_DEVICE);
/* use hypre's SpGEMM instead of cuSPARSE */
HYPRE_SetSpGemmUseCusparse(FALSE);
/* use GPU RNG */
HYPRE_SetUseGpuRand(TRUE);
if (useHypreGpuMemPool)
{
   /* use hypre's GPU memory pool */
   HYPRE_SetGPUMemoryPoolSize(bin_growth, min_bin, max_bin, max_bytes);
}
else if (useUmpireGpuMemPool)
{
   /* or use Umpire GPU memory pool */
   HYPRE_SetUmpireUMPoolName("HYPRE_UM_POOL_TEST");
   HYPRE_SetUmpireDevicePoolName("HYPRE_DEVICE_POOL_TEST");
}
...
/* setup IJ matrix A */
HYPRE_IJMatrixCreate(comm, first_row, last_row, first_col, last_col, &ij_A);
HYPRE_IJMatrixSetObjectType(ij_A, HYPRE_PARCSR);
/* GPU pointers; efficient in large chunks */
HYPRE_IJMatrixAddToValues(ij_A, num_rows, num_cols, rows, cols, data);
HYPRE_IJMatrixAssemble(ij_A);
HYPRE_IJMatrixGetObject(ij_A, (void **) &parcsr_A);
...
/* setup AMG */
HYPRE_ParCSRPCGCreate(comm, &solver);
HYPRE_BoomerAMGCreate(&precon);
HYPRE_BoomerAMGSetRelaxType(precon, rlx_type); /* 3, 4, 6, 7, 18, 11, 12 */
HYPRE_BoomerAMGSetRelaxOrder(precon, FALSE); /* must be false */
HYPRE_BoomerAMGSetCoarsenType(precon, coarsen_type); /* 8 */
HYPRE_BoomerAMGSetInterpType(precon, interp_type); /* 3, 15, 6, 14, 18 */
HYPRE_BoomerAMGSetAggNumLevels(precon, agg_num_levels);
HYPRE_BoomerAMGSetAggInterpType(precon, agg_interp_type); /* 5 or 7 */
HYPRE_BoomerAMGSetKeepTranspose(precon, TRUE); /* keep transpose to avoid SpMTV */
HYPRE_BoomerAMGSetRAP2(precon, FALSE); /* RAP in two multiplications
                                          (default: FALSE) */
HYPRE_ParCSRPCGSetPrecond(solver, HYPRE_BoomerAMGSolve, HYPRE_BoomerAMGSetup,
                          precon);
HYPRE_PCGSetup(solver, parcsr_A, b, x);
...
/* solve */
HYPRE_PCGSolve(solver, parcsr_A, b, x);
...
HYPRE_Finalize(); /* must be the last HYPRE function call */
```

`HYPRE_Init()` must be called and precede all the other `HYPRE_` functions, and `HYPRE_Finalize()` must be called before exiting.

6.8.9 Miscellaneous

For best performance, it might be necessary to set certain parameters, which will affect both coarsening and interpolation. One important parameter is the strong threshold, which can be set using the function `HYPRE_BoomerAMGSetStrongThreshold`. The default value is 0.25, which appears to be a good choice for 2-dimensional problems and the low complexity coarsening algorithms. For 3-dimensional problems a better choice appears to be 0.5, when using the default coarsening algorithm. However, the choice of the strength threshold is problem dependent and therefore there could be better choices than the two suggested ones.

6.9 AMS

AMS (the Auxiliary-space Maxwell Solver) is a parallel unstructured Maxwell solver for edge finite element discretizations of the variational problem

$$\text{Find } \mathbf{u} \in \mathbf{V}_h \ : \qquad (\alpha \nabla \times \mathbf{u}, \nabla \times \mathbf{v}) + (\beta \mathbf{u}, \mathbf{v}) = (\mathbf{f}, \mathbf{v}) \,, \qquad \text{for all } \mathbf{v} \in \mathbf{V}_h \,. \tag{6.1}$$

Here $\mathbf{V}_h$ is the lowest order Nedelec (edge) finite element space, and $\alpha > 0$ and $\beta \geq 0$ are scalar, or SPD matrix coefficients. AMS was designed to be scalable on problems with variable coefficients, and allows for β to be zero in part or the whole domain. In either case the resulting problem is only semidefinite, and for solvability the right-hand side should be chosen to satisfy compatibility conditions.

AMS is based on the auxiliary space methods for definite Maxwell problems proposed in [HiXu2006]. For more details, see [KoVa2009].

6.9.1 Overview

Let $\mathbf{A}$ and $\mathbf{b}$ be the stiffness matrix and the load vector corresponding to (6.1). Then the resulting linear system of interest reads,

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \,. \tag{6.2}$$

The coefficients α and β are naturally associated with the "stiffness" and "mass" terms of $\mathbf{A}$. Besides $\mathbf{A}$ and $\mathbf{b}$, AMS requires the following additional user input:

1. The discrete gradient matrix G representing the edges of the mesh in terms of its vertices. G has as many rows as the number of edges in the mesh, with each row having two nonzero entries: $+1$ and -1 in the columns corresponding to the vertices composing the edge. The sign is determined based on the orientation of the edge. We require that G includes all (interior and boundary) edges and vertices.
2. The representations of the constant vector fields $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ in the $\mathbf{V}_h$ basis, given as three vectors: G_x, G_y, and G_z. Note that since no boundary conditions are imposed on G, the above vectors can be computed as $G_x = Gx$, $G_y = Gy$ and $G_z = Gz$, where x, y, and z are vectors representing the coordinates of the vertices of the mesh.

In addition to the above quantities, AMS can utilize the following (optional) information:

- The Poisson matrices A_α and A_β, corresponding to assembling of the forms $(\alpha \nabla u, \nabla v) + (\beta \nabla u, \nabla v)$ and $(\beta \nabla u, \nabla v)$ using standard linear finite elements on the same mesh.

Internally, AMS proceeds with the construction of the following additional objects:

- A_G – a matrix associated with the mass term which is either $G^T \mathbf{A} G$ or the Poisson matrix A_β (if given).
- $\mathbf{\Pi}$ – the matrix representation of the interpolation operator from vector linear to edge finite elements.
- $\mathbf{A}_{\mathbf{\Pi}}$ – a matrix associated with the stiffness term which is either $\mathbf{\Pi}^T \mathbf{A} \mathbf{\Pi}$ or a block-diagonal matrix with diagonal blocks A_α (if given).

- B_G and $\mathbf{B_\Pi}$ – efficient (AMG) solvers for A_G and $\mathbf{A_\Pi}$.

The solution procedure then is a three-level method using smoothing in the original edge space and subspace corrections based on B_G and $\mathbf{B_\Pi}$. We can employ a number of options here utilizing various combinations of the smoother and solvers in additive or multiplicative fashion. If β is identically zero one can skip the subspace correction associated with G, in which case the solver is a two-level method.

6.9.2 Sample Usage

AMS can be used either as a solver or as a preconditioner. Below we list the sequence of hypre calls needed to create and use it as a solver. See example code `ex15.c` for a complete implementation. We start with the allocation of the `HYPRE_Solver` object:

```
HYPRE_Solver solver;
HYPRE_AMSCreate(&solver);
```

Next, we set a number of solver parameters. Some of them are optional, while others are necessary in order to perform the solver setup.

AMS offers the option to set the space dimension. By default we consider the dimension to be 3. The only other option is 2, and it can be set with the function given below. We note that a 3D solver will still work for a 2D problem, but it will be slower and will require more memory than necessary.

```
HYPRE_AMSSetDimension(solver, dim);
```

The user is required to provide the discrete gradient matrix G. AMS expects a matrix defined on the whole mesh with no boundary edges/nodes excluded. It is essential to **not** impose any boundary conditions on G. Regardless of which hypre conceptual interface was used to construct G, one can obtain a ParCSR version of it. This is the expected format in the following function.

```
HYPRE_AMSSetDiscreteGradient(solver, G);
```

In addition to G, we need one additional piece of information in order to construct the solver. The user has the option to choose either the coordinates of the vertices in the mesh or the representations of the constant vector fields in the edge element basis. In both cases three hypre parallel vectors should be provided. For 2D problems, the user can set the third vector to NULL. The corresponding function calls read:

```
HYPRE_AMSSetCoordinateVectors(solver,x,y,z);
```

or

```
HYPRE_AMSSetEdgeConstantVectors(solver, one_zero_zero, zero_one_zero, zero_zero_one);
```

The vectors `one_zero_zero`, `zero_one_zero` and `zero_zero_one` above correspond to the constant vector fields $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$.

The remaining solver parameters are optional. For example, the user can choose a different cycle type by calling

```
HYPRE_AMSSetCycleType(solver, cycle_type); /* default value: 1 */
```

The available cycle types in AMS are:

- `cycle_type=1`: multiplicative solver (01210)
- `cycle_type=2`: additive solver $(0 + 1 + 2)$
- `cycle_type=3`: multiplicative solver (02120)

- `cycle_type=4`: additive solver $(010+2)$
- `cycle_type=5`: multiplicative solver (0102010)
- `cycle_type=6`: additive solver $(1+020)$
- `cycle_type=7`: multiplicative solver (0201020)
- `cycle_type=8`: additive solver $(0(1+2)0)$
- `cycle_type=11`: multiplicative solver (013454310)
- `cycle_type=12`: additive solver $(0+1+3+4+5)$
- `cycle_type=13`: multiplicative solver (034515430)
- `cycle_type=14`: additive solver $(01(3+4+5)10)$

Here we use the following convention for the three subspace correction methods: 0 refers to smoothing, 1 stands for BoomerAMG based on B_G, and 2 refers to a call to BoomerAMG for $\mathbf{B_\Pi}$. The values 3, 4 and 5 refer to the scalar subspaces corresponding to the x, y and z components of $\mathbf{\Pi}$.

The abbreviation $xyyz$ for $x, y, z \in \{0, 1, 2, 3, 4, 5\}$ refers to a multiplicative subspace correction based on solvers x, y, y, and z (in that order). The abbreviation $x+y+z$ stands for an additive subspace correction method based on x, y and z solvers. The additive cycles are meant to be used only when AMS is called as a preconditioner. In our experience the choices `cycle_type=1,5,8,11,13` often produced fastest solution times, while `cycle_type=7` resulted in smallest number of iterations.

Additional solver parameters, such as the maximum number of iterations, the convergence tolerance and the output level, can be set with

```
HYPRE_AMSSetMaxIter(solver, maxit);     /* default value: 20 */
HYPRE_AMSSetTol(solver, tol);           /* default value: 1e-6 */
HYPRE_AMSSetPrintLevel(solver, print);  /* default value: 1 */
```

More advanced parameters, affecting the smoothing and the internal AMG solvers, can be set with the following three functions:

```
HYPRE_AMSSetSmoothingOptions(solver, 2, 1, 1.0, 1.0);
HYPRE_AMSSetAlphaAMGOptions(solver, 10, 1, 3, 0.25, 0, 0);
HYPRE_AMSSetBetaAMGOptions(solver, 10, 1, 3, 0.25, 0, 0);
```

For (singular) problems where $\beta = 0$ in the whole domain, different (in fact simpler) version of the AMS solver is offered. To allow for this simplification, use the following hypre call

```
HYPRE_AMSSetBetaPoissonMatrix(solver, NULL);
```

If β is zero only in parts of the domain, the problem is still singular, but the AMS solver will try to detect this and construct a non-singular preconditioner. Though this often works well in practice, AMS also provides a more robust version for solving such singular problems to very low convergence tolerances. This version takes advantage of additional information: the list of nodes which are interior to a zero-conductivity region provided by the function

```
HYPRE_AMSSetInteriorNodes(solver, HYPRE_ParVector interior_nodes);
```

A node is interior, if its entry in the `interior_nodes` array is 1.0. Based on this array, a restricted discrete gradient operator G_0 is constructed, and AMS is then defined based on the matrix $\mathbf{A} + \delta G_0^T G_0$ which is non-singular, and a small $\delta > 0$ perturbation of $\mathbf{A}$. When iterating with this preconditioner, it is advantageous to project on the compatible subspace $Ker(G_0^T)$. This can be done periodically, or manually through the functions

```
HYPRE_AMSSetProjectionFrequency(solver, int projection_frequency);
HYPRE_AMSProjectOutGradients(solver, HYPRE_ParVector x);
```

Two additional matrices are constructed in the setup of the AMS method—one corresponding to the coefficient α and another corresponding to β. This may lead to prohibitively high memory requirements, and the next two function calls may help to save some memory. For example, if the Poisson matrix with coefficient β (denoted by `Abeta`) is available then one can avoid one matrix construction by calling

```
HYPRE_AMSSetBetaPoissonMatrix(solver, Abeta);
```

Similarly, if the Poisson matrix with coefficient α is available (denoted by `Aalpha`) the second matrix construction can also be avoided by calling

```
HYPRE_AMSSetAlphaPoissonMatrix(solver, Aalpha);
```

Note the following regarding these functions:

- Both of them change their input. More specifically, the diagonal entries of the input matrix corresponding to eliminated degrees of freedom (due to essential boundary conditions) are penalized.
- It is assumed that their essential boundary conditions of $\mathbf{A}$, `Abeta` and `Aalpha` are on the same part of the boundary.
- `HYPRE_AMSSetAlphaPoissonMatrix` forces the AMS method to use a simpler, but weaker (in terms of convergence) method. With this option, the multiplicative AMS cycle is not guaranteed to converge with the default parameters. The reason for this is the fact the solver is not variationally obtained from the original matrix (it utilizes the auxiliary Poisson–like matrices `Abeta` and `Aalpha`). Therefore, it is recommended in this case to use AMS as preconditioner only.

After the above calls, the solver is ready to be constructed. The user has to provide the stiffness matrix $\mathbf{A}$ (in ParCSR format) and the hypre parallel vectors $\mathbf{b}$ and $\mathbf{x}$. (The vectors are actually not used in the current AMS setup.) The setup call reads,

```
HYPRE_AMSSetup(solver, A, b, x);
```

It is important to note the order of the calling sequence. For example, do **not** call `HYPRE_AMSSetup` before calling `HYPRE_AMSSetDiscreteGradient` and one of the functions `HYPRE_AMSSetCoordinateVectors` or `HYPRE_AMSSetEdgeConstantVectors`.

Once the setup has completed, we can solve the linear system by calling

```
HYPRE_AMSSolve(solver, A, b, x);
```

Finally, the solver can be destroyed with

```
HYPRE_AMSDestroy(&solver);
```

More details can be found in the files `ams.h` and `ams.c` located in the `parcsr_ls` directory.

6.9.3 High-order Discretizations

In addition to the interface for the lowest-order Nedelec elements described in the previous subsections, AMS also provides support for (arbitrary) high-order Nedelec element discretizations. Since the robustness of AMS depends on the performance of BoomerAMG on the associated (high-order) auxiliary subspace problems, we note that the convergence may not be optimal for large polynomial degrees $k \geq 1$.

In the high-order AMS interface, the user does not need to provide the coordinates of the vertices (or the representations of the constant vector fields in the edge basis), but instead should construct and pass the Nedelec interpolation matrix $\mathbf{\Pi}$ which maps (high-order) vector nodal finite elements into the (high-order) Nedelec space. In other words, $\mathbf{\Pi}$ is the (parallel) matrix representation of the interpolation mapping from $\mathrm{P}_k^3/\mathrm{Q}_k^3$ into ND_k, see [HiXu2006], [KoVa2009]. We require this matrix as an input, since in the high-order case its entries very much depend on the particular choice of the basis functions in the edge and nodal spaces, as well as on the geometry of the mesh elements. The columns of $\mathbf{\Pi}$ should use a node-based numbering, where the $x/y/z$ components of the first node (vertex or high-order degree of freedom) should be listed first, followed by the $x/y/z$ components of the second node and so on (see the documentation of `HYPRE_BoomerAMGSetDofFunc`).

Similarly to the Nedelec interpolation, the discrete gradient matrix G should correspond to the mapping $\varphi \in \mathrm{P}_k^3/\mathrm{Q}_k^3 \mapsto \nabla\varphi \in \mathrm{ND}_k$, so even though its values are still independent of the mesh coordinates, they will not be ± 1, but will be determined by the particular form of the high-order basis functions and degrees of freedom.

With these matrices, the high-order setup procedure is simply

```
HYPRE_AMSSetDimension(solver, dim);
HYPRE_AMSSetDiscreteGradient(solver, G);
HYPRE_AMSSetInterpolations(solver, Pi, NULL, NULL, NULL);
```

We remark that the above interface calls can also be used in the lowest-order case (or even other types of discretizations such as those based on the second family of Nedelec elements), but we recommend calling the previously described `HYPRE_AMSSetCoordinateVectors` instead, since this allows AMS to handle the construction and use of $\mathbf{\Pi}$ internally.

Specifying the monolithic $\mathbf{\Pi}$ limits the AMS cycle type options to those less than 10. Alternatively one can separately specify the x, y and z components of $\mathbf{\Pi}$:

```
HYPRE_AMSSetInterpolations(solver, NULL, Pix, Piy, Piz);
```

which enables the use of AMS cycle types with index greater than 10. By definition, $\mathbf{\Pi}^x\varphi = \mathbf{\Pi}(\varphi, 0, 0)$, and similarly for $\mathbf{\Pi}^y$ and $\mathbf{\Pi}^z$. Each of these matrices has the same sparsity pattern as G, but their entries depend on the coordinates of the mesh vertices.

Finally, both $\mathbf{\Pi}$ and its components can be passed to the solver:

```
HYPRE_AMSSetInterpolations(solver, Pi, Pix, Piy, Piz);
```

which will duplicate some memory, but allows for experimentation with all available AMS cycle types.

6.9.4 Non-conforming AMR Grids

AMS could also be applied to problems with adaptive mesh refinement (AMR) posed on non-conforming quadrilateral/hexahedral meshes, see [GrKo2015] for more details.

On non-conforming grids (assuming also arbitrarily high-order elements), each finite element space has two versions: a conforming one, e.g. $\mathrm{Q}_k^c/\mathrm{ND}_k^c$, where the *hanging* degrees of freedom are constrained by the conforming (*real*) degrees of freedom, and a non-conforming one, e.g. $\mathrm{Q}_k^{nc}/\mathrm{ND}_k^{nc}$ where the non-conforming degrees of freedom (hanging and real) are unconstrained. These spaces are related with the conforming prolongation and the pure restriction operators P and R, as well as the conforming and non-conforming version of the discrete gradient operator as follows:

$$\begin{array}{ccc} \mathrm{Q}_k^c & \xrightarrow{G_c} & \mathrm{ND}_k^c \\ {\scriptstyle P_{\mathrm{Q}}}\downarrow\uparrow{\scriptstyle R_{\mathrm{Q}}} & & {\scriptstyle P_{\mathrm{ND}}}\downarrow\uparrow{\scriptstyle R_{\mathrm{ND}}} \\ \mathrm{Q}_k^{nc} & \xrightarrow{G_{nc}} & \mathrm{ND}_k^{nc} \end{array}$$

Since the linear system is posed on ND_k^c, the user needs to provide the conforming discrete gradient matrix G_c to AMS, using `HYPRE_AMSSetDiscreteGradient`. This matrix is defined by the requirement that the above diagram commutes from Q_k^c to ND_k^{nc}, corresponding to the definition

$$G_c = R_{\mathrm{ND}}\, G_{nc}\, P_{\mathrm{Q}}\,,$$

i.e. the conforming gradient is computed by starting with a conforming nodal Q_k function, interpolating it in the hanging nodes, computing the gradient locally and representing it in the Nedelec space on each element (the non-conforming discrete gradient G_{nc} in the above formula), and disregarding the values in the hanging ND_k degrees of freedom.

Similar considerations imply that the conforming Nedelec interpolation matrix $\mathbf{\Pi}_c$ should be defined as

$$\mathbf{\Pi}_c = R_{\mathrm{ND}}\, \mathbf{\Pi}_{nc}\, P_{\mathrm{Q}^3}\,,$$

with $\mathbf{\Pi}_{nc}$ computed element-wise as in the previous subsection. Note that in the low-order case, $\mathbf{\Pi}_c$ can be computed internally in AMS based only G_c and the conforming coordinates of the vertices $x_c/y_c/z_c$, see [GrKo2015].

6.10 ADS

The Auxiliary-space Divergence Solver (ADS) is a parallel unstructured solver similar to AMS, but targeting $H(div)$ instead of $H(curl)$ problems. Its usage and options are very similar to those of AMS, and in general the relationship between ADS and AMS is analogous to that between AMS and AMG.

Specifically ADS was designed for the scalable solution of linear systems arising in the finite element discretization of the variational problem

$$\text{Find } \mathbf{u} \in \mathbf{W}_h \;:\qquad (\alpha\, \nabla\cdot\mathbf{u}, \nabla\cdot\mathbf{v}) + (\beta\,\mathbf{u},\mathbf{v}) = (\mathbf{f},\mathbf{v})\,,\qquad \text{for all } \mathbf{v}\in\mathbf{W}_h\,, \tag{6.3}$$

where $\mathbf{W}_h$ is the lowest order Raviart-Thomas (face) finite element space, and $\alpha > 0$ and $\beta > 0$ are scalar, or SPD matrix variable coefficients. It is based on the auxiliary space methods for $H(div)$ problems proposed in [HiXu2006].

6.10.1 Overview

Let $\mathbf{A}$ and $\mathbf{b}$ be the stiffness matrix and the load vector corresponding to (6.3). Then the resulting linear system of interest reads,

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}\,. \tag{6.4}$$

The coefficients α and β are naturally associated with the "stiffness" and "mass" terms of $\mathbf{A}$. Besides $\mathbf{A}$ and $\mathbf{b}$, ADS requires the following additional user input:

1. The discrete curl matrix C representing the faces of the mesh in terms of its edges. C has as many rows as the number of faces in the mesh, with each row having nonzero entries $+1$ and -1 in the columns corresponding to the edges composing the face. The sign is determined based on the orientation of the edges relative to the face. We require that C includes all (interior and boundary) faces and edges.
2. The discrete gradient matrix G representing the edges of the mesh in terms of its vertices. G has as many rows as the number of edges in the mesh, with each row having two nonzero entries: $+1$ and -1 in the columns corresponding to the vertices composing the edge. The sign is determined based on the orientation of the edge. We require that G includes all (interior and boundary) edges and vertices.
3. Vectors x, y, and z representing the coordinates of the vertices of the mesh.

Internally, ADS proceeds with the construction of the following additional objects:

- A_C – the curl-curl matrix $C^T\mathbf{A}C$.
- $\mathbf{\Pi}$ – the matrix representation of the interpolation operator from vector linear to face finite elements.
- $\mathbf{A_\Pi}$ – the vector nodal matrix $\mathbf{\Pi}^T\mathbf{A}\mathbf{\Pi}$.
- B_C and $\mathbf{B_\Pi}$ – efficient (AMS/AMG) solvers for A_C and $\mathbf{A_\Pi}$.

The solution procedure then is a three-level method using smoothing in the original face space and subspace corrections based on B_C and $\mathbf{B_\Pi}$. We can employ a number of options here utilizing various combinations of the smoother and solvers in additive or multiplicative fashion.

6.10.2 Sample Usage

ADS can be used either as a solver or as a preconditioner. Below we list the sequence of hypre calls needed to create and use it as a solver. We start with the allocation of the `HYPRE_Solver` object:

```
HYPRE_Solver solver;
HYPRE_ADSCreate(&solver);
```

Next, we set a number of solver parameters. Some of them are optional, while others are necessary in order to perform the solver setup.

The user is required to provide the discrete curl and gradient matrices C and G. ADS expects a matrix defined on the whole mesh with no boundary faces, edges or nodes excluded. It is essential to **not** impose any boundary conditions on C or G. Regardless of which hypre conceptual interface was used to construct the matrices, one can always obtain a ParCSR version of them. This is the expected format in the following functions.

```
HYPRE_ADSSetDiscreteCurl(solver, C);
HYPRE_ADSSetDiscreteGradient(solver, G);
```

Next, ADS requires the coordinates of the vertices in the mesh as three hypre parallel vectors. The corresponding function call reads:

```
HYPRE_ADSSetCoordinateVectors(solver, x, y, z);
```

The remaining solver parameters are optional. For example, the user can choose a different cycle type by calling

```
HYPRE_ADSSetCycleType(solver, cycle_type); /* default value: 1 */
```

The available cycle types in ADS are:

- `cycle_type=1`: multiplicative solver (01210)
- `cycle_type=2`: additive solver $(0+1+2)$
- `cycle_type=3`: multiplicative solver (02120)
- `cycle_type=4`: additive solver $(010+2)$
- `cycle_type=5`: multiplicative solver (0102010)
- `cycle_type=6`: additive solver $(1+020)$
- `cycle_type=7`: multiplicative solver (0201020)
- `cycle_type=8`: additive solver $(0(1+2)0)$
- `cycle_type=11`: multiplicative solver (013454310)
- `cycle_type=12`: additive solver $(0+1+3+4+5)$
- `cycle_type=13`: multiplicative solver (034515430)
- `cycle_type=14`: additive solver $(01(3+4+5)10)$

Here we use the following convention for the three subspace correction methods: 0 refers to smoothing, 1 stands for AMS based on B_C, and 2 refers to a call to BoomerAMG for $\mathbf{B}_{\mathbf{\Pi}}$. The values 3, 4 and 5 refer to the scalar subspaces corresponding to the x, y and z components of $\mathbf{\Pi}$.

The abbreviation $xyyz$ for $x, y, z \in \{0, 1, 2, 3, 4, 5\}$ refers to a multiplicative subspace correction based on solvers x, y, y, and z (in that order). The abbreviation $x+y+z$ stands for an additive subspace correction method based on x, y and z solvers. The additive cycles are meant to be used only when ADS is called as a preconditioner. In our experience the choices `cycle_type=1,5,8,11,13` often produced fastest solution times, while `cycle_type=7` resulted in smallest number of iterations.

Additional solver parameters, such as the maximum number of iterations, the convergence tolerance and the output level, can be set with

```
HYPRE_ADSSetMaxIter(solver, maxit);     /* default value: 20 */
HYPRE_ADSSetTol(solver, tol);           /* default value: 1e-6 */
HYPRE_ADSSetPrintLevel(solver, print);  /* default value: 1 */
```

More advanced parameters, affecting the smoothing and the internal AMS and AMG solvers, can be set with the following three functions:

```
HYPRE_ADSSetSmoothingOptions(solver, 2, 1, 1.0, 1.0);
HYPRE_ADSSetAMSOptions(solver, 11, 10, 1, 3, 0.25, 0, 0);
HYPRE_ADSSetAMGOptions(solver, 10, 1, 3, 0.25, 0, 0);
```

We note that the AMS cycle type, which is the second parameter of `HYPRE_ADSSetAMSOptions` should be greater than 10, unless the high-order interface of `HYPRE_ADSSetInterpolations` described in the next subsection is being used.

After the above calls, the solver is ready to be constructed. The user has to provide the stiffness matrix $\mathbf{A}$ (in ParCSR format) and the hypre parallel vectors $\mathbf{b}$ and $\mathbf{x}$. (The vectors are actually not used in the current ADS setup.) The setup call reads,

```
HYPRE_ADSSetup(solver, A, b, x);
```

It is important to note the order of the calling sequence. For example, do **not** call `HYPRE_ADSSetup` before calling each of the functions `HYPRE_ADSSetDiscreteCurl`, `HYPRE_ADSSetDiscreteGradient` and `HYPRE_ADSSetCoordinateVectors`.

Once the setup has completed, we can solve the linear system by calling

```
HYPRE_ADSSolve(solver, A, b, x);
```

Finally, the solver can be destroyed with

```
HYPRE_ADSDestroy(&solver);
```

More details can be found in the files `ads.h` and `ads.c` located in the `parcsr_ls` directory.

6.10.3 High-order Discretizations

Similarly to AMS, ADS also provides support for (arbitrary) high-order $H(div)$ discretizations. Since the robustness of ADS depends on the performance of AMS and BoomerAMG on the associated (high-order) auxiliary subspace problems, we note that the convergence may not be optimal for large polynomial degrees $k \geq 1$.

In the high-order ADS interface, the user does not need to provide the coordinates of the vertices, but instead should construct and pass the Raviart-Thomas and Nedelec interpolation matrices $\mathbf{\Pi}_{RT}$ and $\mathbf{\Pi}_{ND}$ which map (high-order) vector nodal finite elements into the (high-order) Raviart-Thomas and Nedelec space. In other words, these are the (parallel) matrix representation of the interpolation mappings from $\mathrm{P}_k^3/\mathrm{Q}_k^3$ into RT_{k-1} and ND_k, see [HiXu2006], [KoVa2009]. We require these matrices as inputs, since in the high-order case their entries very much depend on the particular choice of the basis functions in the finite element spaces, as well as on the geometry of the mesh elements. The columns of the $\mathbf{\Pi}$ matrices should use a node-based numbering, where the x/y/z components of the first node (vertex or high-order degree of freedom) should be listed first, followed by the x/y/z components of the second node and so on (see the documentation of `HYPRE_BoomerAMGSetDofFunc`). Furthermore, each interpolation matrix can be split into x, y and z components by defining $\mathbf{\Pi}^x\varphi = \mathbf{\Pi}(\varphi, 0, 0)$, and similarly for $\mathbf{\Pi}^y$ and $\mathbf{\Pi}^z$.

The discrete gradient and curl matrices G and C should correspond to the mappings $\varphi \in \mathrm{P}_k^3/\mathrm{Q}_k^3 \mapsto \nabla\varphi \in \mathrm{ND}_k$ and $\mathbf{u} \in \mathrm{ND}_k \mapsto \nabla \times \mathbf{u} \in \mathrm{RT}_{k-1}$, so even though their values are still independent of the mesh coordinates, they will not be ± 1, but will be determined by the particular form of the high-order basis functions and degrees of freedom.

With these matrices, the high-order setup procedure is simply

```
HYPRE_ADSSetDiscreteCurl(solver, C);
HYPRE_ADSSetDiscreteGradient(solver, G);
HYPRE_ADSSetInterpolations(solver, RT_Pi, NULL, NULL, NULL,
                                   ND_Pi, NULL, NULL, NULL);
```

We remark that the above interface calls can also be used in the lowest-order case (or even other types of discretizations), but we recommend calling the previously described `HYPRE_ADSSetCoordinateVectors` instead, since this allows ADS to handle the construction and use of the interpolations internally.

Specifying the monolithic $\mathbf{\Pi}_{RT}$ limits the ADS cycle type options to those less than 10. Alternatively one can separately specify the x, y and z components of $\mathbf{\Pi}_{RT}$.

```
HYPRE_ADSSetInterpolations(solver, NULL, RT_Pix, RT_Piy, RT_Piz,
                                   ND_Pi, NULL, NULL, NULL);
```

which enables the use of ADS cycle types with index greater than 10. The same holds for $\mathbf{\Pi}_{ND}$ and its components, e.g. to enable the subspace AMS cycle type greater then 10 we need to call

```
HYPRE_ADSSetInterpolations(solver, NULL, RT_Pix, RT_Piy, RT_Piz,
                                   NULL, ND_Pix, ND_Piy, ND_Piz);
```

Finally, both $\mathbf{\Pi}$ and their components can be passed to the solver:

```
HYPRE_ADSSetInterpolations(solver, RT_Pi, RT_Pix, RT_Piy, RT_Piz
                                   ND_Pi, ND_Pix, ND_Piy, ND_Piz);
```

which will duplicate some memory, but allows for experimentation with all available ADS and AMS cycle types.

6.11 The MLI Package

MLI is an object-oriented module that implements the class of algebraic multigrid algorithms based on Vanek and Brezina's smoothed aggregation method [VaMB1996], [VaBM2001]. There are two main algorithms in this module - the original smoothed aggregation algorithm and the modified version that uses the finite element substructure matrices to construct the prolongation operators. As such, the later algorithm can only be used in the finite element context via the finite element interface. In addition, the nodal coordinates obtained via the finite element interface can be used to construct a better prolongation operator than the pure translation modes.

Below is an example on how to set up MLI as a preconditioner for conjugate gradient.

```
HYPRE_LSI_MLICreate(MPI_COMM_WORLD, &pcg_precond);

HYPRE_LSI_MLISetParams(pcg_precond, "MLI strengthThreshold 0.08");
...

HYPRE_PCGSetPrecond(pcg_solver,
                    (HYPRE_PtrToSolverFcn) HYPRE_LSI_MLISolve,
                    (HYPRE_PtrToSolverFcn) HYPRE_LSI_MLISetup,
                    pcg_precond);
```

Note that parameters are set via `HYPRE_LSI_MLISetParams`. A list of valid parameters that can be set using this routine can be found in the FEI section of the reference manual.

6.12 Multigrid Reduction (MGR)

MGR is a parallel multigrid reduction solver and preconditioner designed to take advantage of use-provided information to solve systems of equations with multiple vatiable types. The algorithm is similar to two-stage preconditioner strategies and other reduction techniques like ARMS, but in a standard multigrid framework.

The MGR algorithm accepts information about the variables in block form from the user and uses it to define the appropriate C/F splitting for the multigrid scheme. The linear system solve proceeds with an F-relaxation solve on the F points, folowed by a coarse grid correction. The coarse grid solve is handled by scalar AMG (BoomerAMG). MGR provides users with more control over the coarsening process, and can potentially be a starting point for designing multigrid-based physics-based preconditioners.

The following represents a minimal set of functions, and some optional functions, to call to use the MGR solver. For simplicity, we ignore the function parameters here, and refer the reader to the reference manual for more details on the parameters and their defaults.

- `HYPRE_MGRCreate:` Create the MGR solver object.
- `HYPRE_MGRSetCpointsByBlock:` Set up block data with information about coarse indexes for reduction. Here, the user specifies the number of reduction levels, as well as the the coarse nodes for each level of the reduction. These coarse nodes are indexed by their index in the block of unknowns. This is used internally to tag the appropriate indexes of the linear system matrix as coarse nodes.
- (Optional) `HYPRE_MGRSetReservedCoarseNodes:` Prescribe a subset of nodes to be kept as coarse nodes until the coarsest level. These nodes are transferred onto the coarsest grid of the BoomerAMG coarse grid solver.
- (Optional) `HYPRE_MGRSetNonCpointsToFpoints:` Set points not prescribed as C points to be fixed as F points for intermediate levels. Setting this to 1 uses the user input to define the C/F splitting. Otherwise, a BoomerAMG coarsening routine is used to determine the C/F splitting for intermediate levels.
- (Optional) `HYPRE_MGRSetCoarseSolver:` This function sets the BoomerAMG solver to be used for the solve on the coarse grid. The user can define their own BoomerAMG solver with their preferred options and pass this to the MGR solver. Otherwise, an internal BoomerAMG solver is used as the coarse grid solver instead.
- `HYPRE_MGRSetup:` Setup and MGR solver object.
- `HYPRE_MGRSolve:` Solve the linear system.
- `HYPRE_MGRDestroy:` Destroy the MGR solver object

For more details about additional solver options and parameters, please refer to the reference manual. NOTE: The MGR solver is currently only supported by the IJ interface.

6.13 ParaSails

ParaSails is a parallel implementation of a sparse approximate inverse preconditioner, using *a priori* sparsity patterns and least-squares (Frobenius norm) minimization. Symmetric positive definite (SPD) problems are handled using a factored SPD sparse approximate inverse. General (nonsymmetric and/or indefinite) problems are handled with an unfactored sparse approximate inverse. It is also possible to precondition nonsymmetric but definite matrices with a factored, SPD preconditioner.

ParaSails uses *a priori* sparsity patterns that are patterns of powers of sparsified matrices. ParaSails also uses a post-filtering technique to reduce the cost of applying the preconditioner. In advanced usage not described here, the pattern of the preconditioner can also be reused to generate preconditioners for different matrices in a sequence of linear solves.

For more details about the ParaSails algorithm, see [Chow2000].

6.13.1 Parameter Settings

The accuracy and cost of ParaSails are parametrized by the real `thresh` and integer `nlevels` parameters, $0 \leq$ `thresh` ≤ 1, $0 \leq$ `nlevels`. Lower values of `thresh` and higher values of `nlevels` lead to more accurate, but more expensive preconditioners. More accurate preconditioners are also more expensive per iteration. The default values are `thresh` $= 0.1$ and `nlevels` $= 1$. The parameters are set using `HYPRE_ParaSailsSetParams`.

Mathematically, given a symmetric matrix A, the pattern of the approximate inverse is the pattern of $\tilde{A}^m$ where $\tilde{A}$ is a matrix that has been sparsified from A. The sparsification is performed by dropping all entries in a symmetrically diagonally scaled A whose values are less than `thresh` in magnitude. The parameter `nlevel` is equivalent to $m - 1$. Filtering is a post-thresholding procedure. For more details about the algorithm, see [Chow2000].

The storage required for the ParaSails preconditioner depends on the parameters `thresh` and `nlevels`. The default parameters often produce a preconditioner that can be stored in less than the space required to store the original matrix. ParaSails does not need a large amount of intermediate storage in order to construct the preconditioner.

ParaSail's Create function differs from the synopsis in the following way:

```
int HYPRE_ParaSailsCreate(MPI_Comm comm, HYPRE_Solver *solver, int symmetry);
```

where `comm` is the MPI communicator.

The value of `symmetry` has the following meanings, to indicate the symmetry and definiteness of the problem, and to specify the type of preconditioner to construct:

value	meaning
0	nonsymmetric and/or indefinite problem, and nonsymmetric preconditioner
1	SPD problem, and SPD (factored) preconditioner
2	nonsymmetric, definite problem, and SPD (factored) preconditioner

For more information about the final case, see section *Preconditioning Nearly Symmetric Matrices*.

Parameters for setting up the preconditioner are specified using

```
int HYPRE_ParaSailsSetParams(HYPRE_Solver solver, double thresh,
                             int nlevel, double filter);
```

The parameters are used to specify the sparsity pattern and filtering value (see above), and are described with suggested values as follows:

param	type	range	sug. values	default	meaning
`nlevel`	integer	≥ 0	0, 1, 2	1	$m = 1+$ `nlevel`
`thresh`	real	≥ 0	0, 0.1, 0.01	0.1	thresh $=$ `thresh`
		< 0	-0.75, -0.90		thresh auto-selected
`filter`	real	≥ 0	0, 0.05, 0.001	0.05	filter $=$ `filter`
		< 0	-0.90		filter auto-selected

When `thresh` < 0, then a threshold is selected such that `thresh` represents the negative of the fraction of nonzero elements that are dropped. For example, if `thresh` $= -0.9$ then $\tilde{A}$ will contain approximately ten percent of the nonzeros in A.

When `filter` < 0, then a filter value is selected such that `filter` represents the negative of the fraction of nonzero elements that are dropped. For example, if `filter` $= -0.9$ then approximately 90 percent of the entries in the computed approximate inverse are dropped.

6.13.2 Preconditioning Nearly Symmetric Matrices

A nonsymmetric, but definite and nearly symmetric matrix A may be preconditioned with a symmetric preconditioner M. Using a symmetric preconditioner has a few advantages, such as guaranteeing positive definiteness of the preconditioner, as well as being less expensive to construct.

The nonsymmetric matrix A must be definite, i.e., $(A + A^T)/2$ is SPD, and the *a priori* sparsity pattern to be used must be symmetric. The latter may be guaranteed by 1) constructing the sparsity pattern with a symmetric matrix, or 2) if the matrix is structurally symmetric (has symmetric pattern), then thresholding to construct the pattern is not used (i.e., zero value of the `thresh` parameter is used).

6.14 hypre-ILU

The hypre-ILU solver is a parallel ILU solver based on a domain decomposition framework. It may be use iteratively as a standalone solver or smoother, as well as a preconditioner for accelerators like Krylov subspace methods. This solver implements various parallel variants of the dual threshold (truncation) incomplete LU factorization - ILUT, and the level-based incomplete LU factorization - ILUk.

6.14.1 Overview

The parallel hypre-ILU solver follows a domain decomposition approach for solving distributed sparse linear systems of equations. The strategy is to decompose the domain into interior and interface nodes, where an interface node separates two interior nodes from adjacent domains. In the purely algebraic setting, this is equivalent to partitioning the matrix row data into local (processor-owned) data and external (off-processor-owned) data. The resulting global view of the partitioned matrix has (diagonal) blocks corresponding to local data, and off-diagonal blocks corresponding to non-local data. The resulting parallel ILU strategy is composed of a (local) block factorization and a (global) Schur complement solve. Several strategies provided to efficiently solve the Schur complement system.

The following represents a minimal set of functions, and some optional functions, to call to use the hypre_ILU solver. For simplicity, we ignore the function parameters here, and refer the reader to the reference manual for more details on the parameters and their defaults.

- `HYPRE_ILUCreate:` Create the hypre_ILU solver object.
- `HYPRE_ILUSetType:` Set the type of ILU factorization to do. Here, the user specifies one of several flavors of parallel ILU based on the different combinations of local factorizations and global Schur complement solves. Please refer to the reference manual for more details about the different options available to the user.
- (Optional) `HYPRE_ILUSetLevelOfFill:` Set the level of fill used by the level-based ILUk strategy.
- (Optional) `HYPRE_ILUSetSchurMaxIter:` Set the maximum number of iterations for solving the Schur complement system.
- (Optional) `HYPRE_ILUSetMaxIter:` Set the maximum number of iterations when used as a solver or smoother.
- `HYPRE_ILUSetup:` Setup and hypre_ILU solver object.
- `HYPRE_ILUSolve:` Solve the linear system.
- `HYPRE_ILUDestroy:` Destroy the hypre_ILU solver object

For more details about additional solver options and parameters, please refer to the reference manual. NOTE: The hypre_ILU solver is currently only supported by the IJ interface.

6.15 Euclid

The Euclid library is a scalable implementation of the Parallel ILU algorithm that was presented at SC99 [HyPo1999], and published in expanded form in the SIAM Journal on Scientific Computing [HyPo2001]. By *scalable* we mean that the factorization (setup) and application (triangular solve) timings remain nearly constant when the global problem size is scaled in proportion to the number of processors. As with all ILU preconditioning methods, the number of iterations is expected to increase with global problem size.

Experimental results have shown that PILU preconditioning is in general more effective than Block Jacobi preconditioning for minimizing total solution time. For scaled problems, the relative advantage appears to increase as the number of processors is scaled upwards. Euclid may also be used to good advantage as a smoother within multigrid methods.

6.15.1 Overview

Euclid is best thought of as an "extensible ILU preconditioning framework." *Extensible* means that Euclid can (and eventually will, time and contributing agencies permitting) support many variants of ILU(k) and ILUT preconditioning. (The current release includes Block Jacobi ILU(k) and Parallel ILU(k) methods.) Due to this extensibility, and also because Euclid was developed independently of the hypre project, the methods by which one passes runtime parameters to Euclid preconditioners differ in some respects from the hypre norm. While users can directly set options within their code, options can also be passed to Euclid preconditioners via command line switches and/or small text-based configuration files. The latter strategies have the advantage that users will not need to alter their codes as Euclid's capabilities are extended.

The following fragment illustrates the minimum coding required to invoke Euclid preconditioning within hypre application contexts. The next subsection provides examples of the various ways in which Euclid's options can be set. The final subsection lists the options, and provides guidance as to the settings that (in our experience) will likely prove effective for minimizing execution time.

```
#include "HYPRE_parcsr_ls.h"

HYPRE_Solver eu;
HYPRE_Solver pcg_solver;
HYPRE_ParVector b, x;
HYPRE_ParCSRMatrix A;

//Instantiate the preconditioner.
HYPRE_EuclidCreate(comm, &eu);

//Optionally use the following three methods to set runtime options.
// 1. pass options from command line or string array.
HYPRE_EuclidSetParams(eu, argc, argv);

// 2. pass options from a configuration file.
HYPRE_EuclidSetParamsFromFile(eu, "filename");

// 3. pass options using interface functions.
HYPRE_EuclidSetLevel(eu, 3);
...

//Set Euclid as the preconditioning method for some
//other solver, using the function calls HYPRE_EuclidSetup
//and HYPRE_EuclidSolve.  We assume that the pcg_solver
```

(continues on next page)

(continued from previous page)

```
//has been properly initialized.
HYPRE_PCGSetPrecond(pcg_solver,
                    (HYPRE_PtrToSolverFcn) HYPRE_EuclidSolve,
                    (HYPRE_PtrToSolverFcn) HYPRE_EuclidSetup,
                    eu);

//Solve the system by calling the Setup and Solve methods for,
//in this case, the HYPRE_PCG solver.  We assume that A, b, and x
//have been properly initialized.
HYPRE_PCGSetup(pcg_solver, (HYPRE_Matrix)A, (HYPRE_Vector)b, (HYPRE_Vector)x);
HYPRE_PCGSolve(pcg_solver, (HYPRE_Matrix)parcsr_A, (HYPRE_Vector)b, (HYPRE_Vector)x);

//Destroy the Euclid preconditioning object.
HYPRE_EuclidDestroy(eu);
```

6.15.2 Setting Options: Examples

For expositional purposes, assume you wish to set the ILU(k) factorization level to the value $k = 3$. There are several methods of accomplishing this. Internal to Euclid, options are stored in a simple database that contains (name, value) pairs. Various of Euclid's internal (private) functions query this database to determine, at runtime, what action the user has requested. If you enter the option `-eu_stats 1`, a report will be printed when Euclid's destructor is called; this report lists (among other statistics) the options that were in effect during the factorization phase.

Method 1. By default, Euclid always looks for a file titled `database` in the working directory. If it finds such a file, it opens it and attempts to parse it as a configuration file. Configuration files should be formatted as follows.

```
>cat database
#this is an optional comment
-level 3
```

Any line in a configuration file that contains a "#" character in the first column is ignored. All other lines should begin with an option *name*, followed by one or more blanks, followed by the option *value*. Note that option names always begin with a - character. If you include an option name that is not recognized by Euclid, no harm should ensue.

Method 2. To pass options on the command line, call

```
HYPRE_EuclidSetParams(HYPRE_Solver solver, int argc, char *argv[]);
```

where `argc` and `argv` carry the usual connotation: `main(int argc, char *argv[])`. If your hypre application is called `phoo`, you can then pass options on the command line per the following example.

```
mpirun -np 2 phoo -level 3
```

Since Euclid looks for the `database` file when `HYPRE_EuclidCreate` is called, and parses the command line when `HYPRE_EuclidSetParams` is called, option values passed on the command line will override any similar settings that may be contained in the `database` file. Also, if same option name appears more than once on the command line, the final appearance determines the setting.

Some options, such as `-bj` (see next subsection) are boolean. Euclid always treats these options as the value `1` (true) or `0` (false). When passing boolean options from the command line the value may be committed, in which case it assumed to be `1`. Note, however, that when boolean options are contained in a configuration file, either the `1` or `0` must stated explicitly.

Method 3. There are two ways in which you can read in options from a file whose name is other than `database`. First, you can call `HYPRE_EuclidSetParamsFromFile` to specify a configuration filename. Second, if you have passed the command line arguments as described above in Method 2, you can then specify the configuration filename on the command line using the `-db_filename filename` option, e.g.,

```
mpirun -np 2 phoo -db_filename ../myConfigFile
```

Method 4. One can also set parameters via interface functions, e.g

```
int HYPRE_EuclidSetLevel(HYPRE_Solver solver, int level);
```

For a full set of functions, see the reference manual.

6.15.3 Options Summary

- **-level** $\langle int \rangle$ Factorization level for ILU(k). Default: 1. Guidance: for 2D convection-diffusion and similar problems, fastest solution time is typically obtained with levels 4 through 8. For 3D problems fastest solution time is typically obtained with level 1.
- **-bj** Use Block Jacobi ILU preconditioning instead of PILU. Default: 0 (false). Guidance: if subdomains contain relatively few nodes (less than 1,000), or the problem is not well partitioned, Block Jacobi ILU may give faster solution time than PILU.
- **-eu_stats** When Euclid's destructor is called a summary of runtime settings and timing information is printed to stdout. Default: 0 (false). The timing marks in the report are the maximum over all processors in the MPI communicator.
- **-eu_mem** When Euclid's destructor is called a summary of Euclid's memory usage is printed to stdout. Default: 0 (false). The statistics are for the processor whose rank in `MPI_COMM_WORLD` is 0.
- **-printTestData** This option is used in our autotest procedures, and should not normally be invoked by users.
- **-sparseA** $\langle float \rangle$ Drop-tolerance for ILU(k) factorization. Default: 0 (no dropping). Entries are treated as zero if their absolute value is less than `sparseA * max`, where `max` is the largest absolute value of any entry in the row. Guidance: try this in conjunction with -rowScale. CAUTION: If the coefficient matrix A is symmetric, this setting is likely to cause the filled matrix, $F = L + U - I$, to be unsymmetric. This setting has no effect when ILUT factorization is selected.
- **-rowScale** Scale values prior to factorization such that the largest value in any row is +1 or -1. Default: 0 (false). CAUTION: If the coefficient matrix A is symmetric, this setting is likely to cause the filled matrix, $F = L+U-I$, to be unsymmetric. Guidance: if the matrix is poorly scaled, turning on row scaling may help convergence.
- **-ilut** $\langle float \rangle$ Use ILUT factorization instead of the default, ILU(k). Here, $\langle float \rangle$ is the drop tolerance, which is relative to the largest absolute value of any entry in the row being factored. CAUTION: If the coefficient matrix A is symmetric, this setting is likely to cause the filled matrix, $F = L + U - I$, to be unsymmetric. NOTE: this option can only be used sequentially!

6.16 PILUT: Parallel Incomplete Factorization

Note: this code is no longer supported by the hypre team. We recommend to use Euclid instead, which is more versatile and in general more efficient, especially when used with many processors.

PILUT is a parallel preconditioner based on Saad's dual-threshold incomplete factorization algorithm. The original version of PILUT was done by Karypis and Kumar [KaKu1998] in terms of the Cray SHMEM library. The code was subsequently modified by the hypre team: SHMEM was replaced by MPI; some algorithmic changes were made; and it was software engineered to be interoperable with several matrix implementations, including hypre's ParCSR format, PETSc's matrices, and ISIS++ RowMatrix. The algorithm produces an approximate factorization LU, with the preconditioner M defined by $M = LU$.

Note: PILUT produces a nonsymmetric preconditioner even when the original matrix is symmetric. Thus, it is generally inappropriate for preconditioning symmetric methods such as Conjugate Gradient.

6.16.1 Parameters:

- `SetMaxNonzerosPerRow( int LFIL ); (Default: 20)` Set the maximum number of nonzeros to be retained in each row of L and U. This parameter can be used to control the amount of memory that L and U occupy. Generally, the larger the value of `LFIL`, the longer it takes to calculate the preconditioner and to apply the preconditioner and the larger the storage requirements, but this trades off versus a higher quality preconditioner that reduces the number of iterations.
- `SetDropTolerance( double tol ); (Default: 0.0001)` Set the tolerance (relative to the 2-norm of the row) below which entries in L and U are automatically dropped. PILUT first drops entries based on the drop tolerance, and then retains the largest LFIL elements in each row that remain. Smaller values of `tol` lead to more accurate preconditioners, but can also lead to increases in the time to calculate the preconditioner.

6.17 LOBPCG Eigensolver

LOBPCG (Locally Optimal Block Preconditioned Conjugate Gradient) is a simple, yet very efficient, algorithm suggested in [Knya2001], [KLAO2007], [BLOPEWeb] for computing several smallest eigenpairs of the symmetric generalized eigenvalue problem $Ax = \lambda Bx$ with large, possibly sparse, symmetric matrix A and symmetric positive definite matrix B. The matrix A is not assumed to be positive, which also allows one to use LOBPCG to compute the largest eigenpairs of $Ax = \lambda Bx$ simply by solving $-Ax = \mu Bx$ for the smallest eigenvalues $\mu = -\lambda$.

LOBPCG simultaneously computes several eigenpairs together, which is controlled by the `blockSize` parameter, see example `ex11.c`. The LOBCPG also allows one to impose constraints on the eigenvectors of the form $x^T B y_i = 0$ for a set of vectors y_i given to LOBPCG as input parameters. This makes it possible to compute, e.g., 50 eigenpairs by 5 subsequent calls to LOBPCG with the `blockSize=10`, using deflation. LOBPCG can use preconditioning in two different ways: by running an inner preconditioned PCG linear solver, or by applying the preconditioner directly to the eigenvector residual (option `-pcgitr 0`). In all other respects, LOBPCG is similar to the PCG linear solver.

The LOBPCG code is available for system interfaces: Struct, SStruct, and IJ. It is also used in the Auxiliary-space Maxwell Eigensolver (AME). The LOBPCG setup is similar to the setup for PCG.

6.18 FEI Solvers

After the FEI has been used to assemble the global linear system (as described in Chapter *Finite Element Interface*), a number of hypre solvers can be called to perform the solution. This is straightforward, if hypre's FEI has been used. If an external FEI is employed, the user needs to link with hypre's implementation of the `LinearSystemCore` class, as described in Section *Using HYPRE in External FEI Implementations*.

Solver parameters are specified as an array of strings, and a complete list of the available options can be found in the FEI section of the reference manual. They are passed to the FEI as in the following example:

```
nParams = 5;
paramStrings = new char*[nParams];
for (i = 0; i < nParams; i++) }
   paramStrings[i] = new char[100];

strcpy(paramStrings[0], "solver cg");
strcpy(paramStrings[1], "preconditioner diag");
strcpy(paramStrings[2], "maxiterations 100");
strcpy(paramStrings[3], "tolerance 1.0e-6");
strcpy(paramStrings[4], "outputLevel 1");

feiPtr -> parameters(nParams, paramStrings);
```

To solve the linear system of equations, we call

```
feiPtr -> solve(&status);
```

where the returned value `status` indicates whether the solve was successful.

Finally, the solution can be retrieved by the following function call:

```
feiPtr -> getBlockNodeSolution(elemBlkID, nNodes, nodeIDList,
                               solnOffsets, solnValues);
```

where `nodeIDList` is a list of nodes in element block `elemBlkID`, and `solnOffsets[i]` is the index pointing to the first location where the variables at node i is returned in `solnValues`.

6.18.1 Solvers Available Only through the FEI

While most of the solvers from the previous sections are available through the FEI interface, there are number of additional solvers and preconditioners that are accessible only through the FEI. These solvers are briefly described in this section (see also the reference manual).

Sequential and Parallel Solvers

hypre currently has many iterative solvers. There is also internally a version of the sequential `SuperLU` direct solver (developed at U.C. Berkeley) suitable to small problems (may be up to the size of 10000). In the following we list some of these internal solvers.

1. Additional Krylov solvers (FGMRES, TFQMR, symmetric QMR),
2. SuperLU direct solver (sequential),
3. SuperLU direct solver with iterative refinement (sequential),

Parallel Preconditioners

The performance of the Krylov solvers can be improved by clever selection of preconditioners. Besides those mentioned previously in this chapter, the following preconditioners are available via the `LinearSystemCore` interface:

1. the modified version of MLI, which requires the finite element substructure matrices to construct the prolongation operators,
2. parallel domain decomposition with inexact local solves (`DDIlut`),
3. least-squares polynomial preconditioner,
4. 2×2 block preconditioner, and
5. 2×2 Uzawa preconditioner.

Some of these preconditioners can be tuned by a number of internal parameters modifiable by users. A description of these parameters is given in the reference manual.

Matrix Reduction

For some structural mechanics problems with multi-point constraints the discretization matrix is indefinite (eigenvalues lie in both sides of the imaginary axis). Indefinite matrices are much more difficult to solve than definite matrices. Methods have been developed to reduce these indefinite matrices to definite matrices. Two matrix reduction algorithms have been implemented in hypre, as presented in the following subsections.

Schur Complement Reduction

The incoming linear system of equations is assumed to be in the form:

$$\begin{bmatrix} D & B \\ B^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

where D is a diagonal matrix. After Schur complement reduction is applied, the resulting linear system becomes

$$-B^T D^{-1} B x_2 = b_2 - B^T D^{-1} b_1.$$

Slide Surface Reduction

With the presence of slide surfaces, the matrix is in the same form as in the case of Schur complement reduction. Here A represents the relationship between the master, slave, and other degrees of freedom. The matrix block $[B^T 0]$ corresponds to the constraint equations. The goal of reduction is to eliminate the constraints. As proposed by Manteuffel, the trick is to re-order the system into a 3×3 block matrix.

$$\begin{bmatrix} A_{11} & A_{12} & N \\ A_{21} & A_{22} & D \\ N_T & D & 0 \end{bmatrix} = \begin{bmatrix} A_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22}. \end{bmatrix}$$

The reduced system has the form :

$$(A_{11} - \hat{A}_{21} \hat{A}_{22}^{-1} \hat{A}_{12}) x_1 = b_1 - \hat{A}_{21} \hat{A}_{22}^{-1} b_2,$$

which is symmetric positive definite (SPD) if the original matrix is PD. In addition, $\hat{A}_{22}^{-1}$ is easy to compute.

There are three slide surface reduction algorithms in hypre. The first follows the matrix formulation in this section. The second is similar except that it replaces the eliminated slave equations with identity rows so that the degree of freedom at each node is preserved. This is essential for certain block algorithms such as the smoothed aggregation multilevel preconditioners. The third is similar to the second except that it is more general and can be applied to problems with intersecting slide surfaces (sequential only for intersecting slide surfaces).

Other Features

To improve the efficiency of the hypre solvers, a few other features have been incorporated. We list a few of these features below :

1. Preconditioner reuse - For multiple linear solves with matrices that are slightly perturbed from each other, oftentimes the use of the same preconditioners can save preconditioner setup times but suffer little convergence rate degradation.
2. Projection methods - For multiple solves that use the same matrix, previous solution vectors can sometimes be used to give a better initial guess for subsequent solves. Two projection schemes have been implemented in hypre - A-conjugate projection (for SPD matrices) and minimal residual projection (for both SPD and non-SPD matrices).
3. The sparsity pattern of the matrix is in general not destroyed after it has been loaded to an hypre matrix. But if the matrix is not to be reused, an option is provided to clean up this pattern matrix to conserve memory usage.

CHAPTER

SEVEN

GENERAL INFORMATION

7.1 Getting the Source Code

The hypre distribution tar file is available from the Software link of the hypre web page, http://www.llnl.gov/CASC/hypre/. The hypre Software distribution page allows access to the tar files of the latest and previous general and beta distributions as well as documentation.

7.2 Building the Library

In this and the following several sections, we discuss the steps to install and use hypre on a Unix-like operating system, such as Linux, AIX, and Mac OS X. Alternatively, the CMake build system [CMakeWeb] can be used, and is the best approach for building hypre on Windows systems in particular (see the `INSTALL` file for details).

After unpacking the hypre tar file, the source code will be in the `src` sub-directory of a directory named hypre-VERSION, where VERSION is the current version number (e.g., hypre-1.8.4, with a "b" appended for a beta release).

Move to the `src` sub-directory to build hypre for the host platform. The simplest method is to configure, compile and install the libraries in `./hypre/lib` and `./hypre/include` directories, which is accomplished by:

```
./configure
make
```

NOTE: when executing on an IBM platform `configure` must be executed under the nopoe script (`./nopoe ./configure <option> ...<option>`) to force a single task to be run on the log-in node.

There are many options to `configure` and `make` to customize such things as installation directories, compilers used, compile and load flags, etc.

Executing `configure` results in the creation of platform specific files that are used when building the library. The information may include such things as the system type being used for building and executing, compilers being used, libraries being searched, option flags being set, etc. When all of the searching is done two files are left in the `src` directory; `config.status` contains information to recreate the current configuration and `config.log` contains compiler messages which may help in debugging `configure` errors.

Upon successful completion of `configure` the file `config/Makefile.config` is created from its template `config/Makefile.config.in` and hypre is ready to be built.

Executing `make`, with or without targets being specified, in the `src` directory initiates compiling of all of the source code and building of the hypre library. If any errors occur while compiling, the user can edit the file `config/Makefile.config` directly then run `make` again; without having to re-run configure.

When building hypre without the install target, the libraries and include files will be copied into the default directories, `src/hypre/lib` and `src/hypre/include`, respectively.

When building hypre using the install target, the libraries and include files will be copied into the directories that the user specified in the options to `configure`, e.g. `--prefix=/usr/apps`. If none were specified the default directories, `src/hypre/lib` and `src/hypre/include`, are used.

7.2.1 Configure Options

There are many options to `configure` to allow the user to override and refine the defaults for any system. The best way to find out what options are available is to display the help package, by executing `./configure --help`, which also includes the usage information. The user can mix and match the configure options and variable settings to meet their needs.

Some of the commonly used options include:

```
--enable-debug                Sets compiler flags to generate information
                              needed for debugging.
--enable-shared               Build shared libraries.
                              NOTE: in order to use the resulting shared
                                    libraries the user MUST have the path to
                                    the libraries defined in the environment
                                    variable LD_LIBRARY_PATH.
--with-print-errors           Print HYPRE errors
--with-openmp                 Use OpenMP. This may affect which compiler is
                              chosen.
--enable-bigint               Use long long int for HYPRE_Int (default is NO).
--enable-mixedint             Use long long int for HYPRE_BigInt and int for
                              HYPRE_Int.
                              NOTE: This option disables Euclid, ParaSails,
                                    pilut and CGC coarsening.
```

The user can mix and match the configure options and variable settings to meet their needs. It should be noted that hypre can be configured with external BLAS and LAPACK libraries, which can be combined with any other option. This is done as follows (currently, both libraries must be configured as external together):

```
./configure  --with-blas-lib="blas-lib-name" \
             --with-blas-lib-dirs="path-to-blas-lib" \
             --with-lapack-lib="lapack-lib-name" \
             --with-lapack-lib-dirs="path-to-lapack-lib"
```

The output from `configure` is several pages long. It reports the system type being used for building and executing, compilers being used, libraries being searched, option flags being set, etc.

7.2.2 Make Targets

The make step in building hypre is where the compiling, loading and creation of libraries occurs. Make has several options that are called targets. These include:

```
help         prints the details of each target

all          default target in all directories
             compile the entire library
             does NOT rebuild documentation

clean        deletes all files from the current directory that are
```

(continues on next page)

(continued from previous page)

```
                created by building the library

distclean    deletes all files from the current directory that are created
                by configuring or building the library

install      compile the source code, build the library and copy executables,
                 libraries, etc to the appropriate directories for user access

uninstall    deletes all files that the install target created

tags         runs etags to create a tags table
             file is named TAGS and is saved in the top-level directory

test         depends on the all target to be completed
             removes existing temporary installation directories
             creates temporary installation directories
             copies all libHYPRE* and *.h files to the temporary locations
             builds the test drivers; linking to the temporary locations to
                simulate how application codes will link to HYPRE
```

7.2.3 GPU build

Hypre can support NVIDIA GPUs with CUDA and OpenMP ($\geq$ 4.5). The related `configure` options are

```
--with-cuda             Use CUDA. Require cuda-8.0 or higher (default is
                        NO).

--with-device-openmp    Use OpenMP 4.5 Device Directives. This may affect
                        which compiler is chosen.
```

The related environment variables

```
HYPRE_CUDA_SM          (default 70)

CUDA_HOME              the CUDA home directory
```

need to be set properly, which can be also set by

```
--with-gpu-arch=ARG    (e.g., --with-gpu-arch='60 70')

--with-cuda-home=DIR
```

When configured with `--with-cuda` or `--with-device-openmp`, the memory allocated on the GPUs, by default, is the GPU device memory, which is not accessible from the CPUs. Hypre's structured solvers can work fine with device memory, whereas only selected unstructured solvers can run with device memory. See Chapter *GPU-supported Options* for details. In general, BoomerAMG and the SStruct require unified (CUDA managed) memory, for which the following option should be added

```
--enable-unified-memory Use unified memory for allocating the memory
                        (default is NO).
```

Hypre's Struct solvers can also choose RAJA and Kokkos as the backend. The `configure` options are

```
--with-raja             Use RAJA. Require RAJA package to be compiled
                        properly (default is NO).

--with-kokkos           Use Kokkos. Require kokkos package to be compiled
                        properly(default is NO).
```

To run on the GPUs with RAJA and Kokkos, the options `--with-cuda` and `--with-device-openmp` are also needed, and the RAJA and Kokkos libraries should be built with CUDA or OpenMP 4.5 correspondingly.

The other NVIDIA GPU related options include:

- `--enable-gpu-profiling` Use NVTX on CUDA, rocTX on HIP (default is NO)
- `--enable-cusparse` Use cuSPARSE for GPU sparse kernels (default is YES)
- `--enable-cublas` Use cuBLAS for GPU dense kernels (default is NO)
- `--enable-curand` Use random numbers generators on GPUs (default is YES)

Allocations and deallocations of GPU memory are expensive. Memory pooling is a common approach to reduce such overhead and improve performance. hypre provides caching allocators for GPU device memory and unified memory, enabled by

```
--enable-device-memory-pool  Enable the caching GPU memory allocator in hypre
                             (default is NO)
```

hypre also supports Umpire [Umpire]. To enable Umpire pool, include the following options:

```
--with-umpire               Use Umpire Allocator for device and unified memory
                            (default is NO)
--with-umpire-include=/path-of-umpire-install/include
--with-umpire-lib-dirs=/path-of-umpire-install/lib
--with-umpire-libs=umpire
```

For running on AMD GPUs, configure with

```
--with-hip              Use HIP for AMD GPUs. (default is NO)
--with-gpu-arch=ARG     Use appropriate AMD GPU architecture
```

Currently, only BoomerAMG is supported with HIP. The other AMD GPU related options include:

- `--enable-gpu-profiling` Use NVTX on CUDA, rocTX on HIP (default is NO)
- `--enable-rocsparse` Use rocSPARSE (default is YES)
- `--enable-rocblas` Use rocBLAS (default is NO)
- `--enable-rocrand` Use rocRAND (default is YES)

7.3 Testing the Library

The `examples` subdirectory contains several codes that can be used to test the newly created hypre library. To create the executable versions, move into the `examples` subdirectory, enter `make` then execute the codes as described in the initial comments section of each source code.

7.4 Linking to the Library

An application code linking with hypre must be compiled with `-I$PREFIX/include` and linked with `-L$PREFIX/lib -lHYPRE`, where `$PREFIX` is the directory where hypre is installed, default is `hypre`, or as defined by the configure option `--prefix=PREFIX`. As noted above, if hypre was built as a shared library the user MUST have its location defined in the environment variable `LD_LIBRARY_PATH`.

As an example of linking with hypre, a user may refer to the `Makefile` in the `examples` sub-directory. It is designed to build codes similar to user applications that link with and call hypre. All include and linking flags are defined in the `Makefile.config` file by `configure`.

7.5 Error Flags

Every hypre function returns an integer, which is used to indicate errors during execution. Note that the error flag returned by a given function reflects the errors from {em all} previous calls to hypre functions. In particular, a value of zero means that all hypre functions up to (and including) the current one have completed successfully. This new error flag system is being implemented throughout the library, but currently there are still functions that do not support it. The error flag value is a combination of one or a few of the following error codes:

1. `HYPRE_ERROR_GENERIC` – describes a generic error
2. `HYPRE_ERROR_MEMORY` – hypre was unable to allocate memory
3. `HYPRE_ERROR_ARG` – error in one of the arguments of a hypre function
4. `HYPRE_ERROR_CONV` – a hypre solver did not converge as expected

One can use the `HYPRE_CheckError` function to determine exactly which errors have occurred:

```
/* call some HYPRE functions */
int  hypre_ierr;
hypre_ierr = HYPRE_Function();

/* check if the previously called hypre functions returned error(s) */
if (hypre_ierr)
   /* check if the error with code HYPRE_ERROR_CODE has occurred */
   if (HYPRE_CheckError(hypre_ierr,HYPRE_ERROR_CODE))
```

The corresponding FORTRAN code is

```
! header file with hypre error codes
include 'HYPRE_error_f.h'

! call some HYPRE functions
integer  hypre_ierr
call HYPRE_Function(hypre_ierr)
```

(continues on next page)

(continued from previous page)

```
! check if the previously called hypre functions returned error(s)
if (hypre_ierr .ne. 0) then
   ! check if the error with code HYPRE_ERROR_CODE has occurred
   call HYPRE_CheckError(hypre_ierr, HYPRE_ERROR_CODE, check)
   if (check .ne. 0) then
```

The global error flag can also be obtained directly, between calls to other hypre functions, by calling `HYPRE_GetError()`. If an argument error (`HYPRE_ERROR_ARG`) has occurred, the argument index (counting from 1) can be obtained from `HYPRE_GetErrorArg()`. To get a character string with a description of all errors in a given error flag, use

```
HYPRE_DescribeError(int hypre_ierr, char *descr);
```

The global error flag can be cleared manually by calling `HYPRE_ClearAllErrors()`, which will essentially ignore all previous hypre errors. To only clear a specific error code, the user can call `HYPRE_ClearError(HYPRE_ERROR_CODE)`. Finally, if hypre was configured with `--with-print-errors`, additional error information will be printed to the standard error during execution.

7.6 Bug Reporting and General Support

Simply create an issue at `https://github.com/hypre-space/hypre/issues` to report bugs, request features, or ask general usage questions.

Users should include as much relevant information as possible in their issue report, including at a minimum, the hypre version number being used. For compile and runtime problems, please also include the machine type, operating system, MPI implementation, compiler, and any error messages produced.

7.7 Using HYPRE in External FEI Implementations

To set up hypre for use in external, e.g. Sandia's, FEI implementations one needs to follow the following steps:

1. obtain the hypre and Sandia's FEI source codes,
2. compile Sandia's FEI (fei-2.5.0) to create the `fei_base` library.
3. compile hypre
   - unpack the archive and go into the `src` directory
   - do a `configure` with the `--with-fei-inc-dir` option set to the FEI include directory plus other compile options
   - compile with `make install` to create the `HYPRE_LSI` library in `hypre/lib`.
4. call the FEI functions in your application code (as shown in Chapters *Finite Element Interface* and *Solvers and Preconditioners*)
   - include `cfei-hypre.h` in your file
   - include `FEI_Implementation.h` in your file
5. Modify your `Makefile`
   - include hypre's `include` and `lib` directories in the search paths.

- Link with `-lfei_base -lHYPRE_LSI`. Note that the order in which the libraries are listed may be important.

Building an application executable often requires linking with many different software packages, and many software packages use some LAPACK and/or BLAS functions. In order to alleviate the problem of multiply defined functions at link time, it is recommended that all software libraries are stripped of all LAPACK and BLAS function definitions. These LAPACK and BLAS functions should then be resolved at link time by linking with the system LAPACK and BLAS libraries (e.g. dxml on DEC cluster). Both hypre and SuperLU were built with this in mind. However, some other software library files needed may have the BLAS functions defined in them. To avoid the problem of multiply defined functions, it is recommended that the offending library files be stripped of the BLAS functions.

7.8 Calling HYPRE from Other Languages

The hypre library currently supports two languages: C (native) and Fortran (in version 2.10.1 and earlier, additional language interfaces were also provided through a tool called Babel). The Fortran interface is manually supported to mirror the "native" C interface used throughout most of this manual. We describe this interface next.

Typically, the Fortran subroutine name is the same as the C name, unless it is longer than 31 characters. In these situations, the name is condensed to 31 characters, usually by simple truncation. For now, users should look at the Fortran test drivers (`*.f` codes) in the `test` directory for the correct condensed names. In the future, this aspect of the interface conversion will be made consistent and straightforward.

The Fortran subroutine argument list is always the same as the corresponding C routine, except that the error return code `ierr` is always last. Conversion from C parameter types to Fortran argument type is summarized in following table:

C parameter	Fortran argument
`int i`	`integer i`
`double d`	`double precision d`
`int *array`	`integer array(*)`
`double *array`	`double precision array(*)`
`char *string`	`character string(*)`
`HYPRE_Type object`	`integer*8 object`
`HYPRE_Type *object`	`integer*8 object`

Array arguments in hypre are always of type `(int *)` or `(double *)`, and the corresponding Fortran types are simply `integer` or `double precision` arrays. Note that the Fortran arrays may be indexed in any manner. For example, an integer array of length `N` may be declared in fortran as either of the following:

```
integer  array(N)
integer  array(0:N-1)
```

hypre objects can usually be declared as in the table because `integer*8` usually corresponds to the length of a pointer. However, there may be some machines where this is not the case. On such machines, the Fortran type for a hypre object should be an `integer` of the appropriate length.

This simple example illustrates the above information:

C prototype:

```
int HYPRE_IJMatrixSetValues(HYPRE_IJMatrix  matrix,
                            int  nrows, int  *ncols,
```

(continues on next page)

(continued from previous page)

```
                              const int *rows, const int  *cols,
                              const double  *values);
```

The corresponding Fortran code for calling this routine is as follows:

```
integer*8         matrix
integer           nrows, ncols(MAX_NCOLS)
integer           rows(MAX_ROWS), cols(MAX_COLS)
double precision  values(MAX_COLS)
integer           ierr

call HYPRE_IJMatrixSetValues(matrix, nrows, ncols, rows, cols, values, ierr)
```

CHAPTER

EIGHT

API

8.1 Struct System Interface

group **StructSystemInterface**

This interface represents a structured-grid conceptual view of a linear system.

@memo A structured-grid conceptual interface

Struct Grids

typedef struct hypre_StructGrid_struct ***HYPRE_StructGrid**

A grid object is constructed out of several "boxes", defined on a global abstract index space.

HYPRE_Int **HYPRE_StructGridCreate**(MPI_Comm comm, HYPRE_Int ndim, *HYPRE_StructGrid* *grid)

Create an *ndim*-dimensional grid object.

HYPRE_Int **HYPRE_StructGridDestroy**(*HYPRE_StructGrid* grid)

Destroy a grid object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_StructGridSetExtents**(*HYPRE_StructGrid* grid, HYPRE_Int *ilower, HYPRE_Int *iupper)

Set the extents for a box on the grid.

HYPRE_Int **HYPRE_StructGridAssemble**(*HYPRE_StructGrid* grid)

Finalize the construction of the grid before using.

HYPRE_Int **HYPRE_StructGridSetPeriodic**(*HYPRE_StructGrid* grid, HYPRE_Int *periodic)

Set the periodicity for the grid.

The argument *periodic* is an *ndim*-dimensional integer array that contains the periodicity for each dimension. A zero value for a dimension means non-periodic, while a nonzero value means periodic and contains the actual period. For example, periodicity in the first and third dimensions for a 10x11x12 grid is indicated by the array [10,0,12].

NOTE: Some of the solvers in hypre have power-of-two restrictions on the size of the periodic dimensions.

HYPRE_Int **HYPRE_StructGridSetNumGhost**(*HYPRE_StructGrid* grid, HYPRE_Int *num_ghost)

Set the ghost layer in the grid object

Struct Stencils

typedef struct hypre_StructStencil_struct ***HYPRE_StructStencil**

The stencil object.

HYPRE_Int **HYPRE_StructStencilCreate**(HYPRE_Int ndim, HYPRE_Int size, *HYPRE_StructStencil* *stencil)

Create a stencil object for the specified number of spatial dimensions and stencil entries.

HYPRE_Int **HYPRE_StructStencilDestroy**(*HYPRE_StructStencil* stencil)

Destroy a stencil object.

HYPRE_Int **HYPRE_StructStencilSetElement**(*HYPRE_StructStencil* stencil, HYPRE_Int entry, HYPRE_Int *offset)

Set a stencil entry.

NOTE: The name of this routine will eventually be changed to *HYPRE_StructStencilSetEntry*.

Struct Matrices

typedef struct hypre_StructMatrix_struct ***HYPRE_StructMatrix**

The matrix object.

HYPRE_Int **HYPRE_StructMatrixCreate**(MPI_Comm comm, *HYPRE_StructGrid* grid, *HYPRE_StructStencil* stencil, *HYPRE_StructMatrix* *matrix)

Create a matrix object.

HYPRE_Int **HYPRE_StructMatrixDestroy**(*HYPRE_StructMatrix* matrix)

Destroy a matrix object.

HYPRE_Int **HYPRE_StructMatrixInitialize**(*HYPRE_StructMatrix* matrix)

Prepare a matrix object for setting coefficient values.

HYPRE_Int **HYPRE_StructMatrixSetValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *index, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Set matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_StructMatrixSetBoxValues* to set coefficients a box at a time.

HYPRE_Int **HYPRE_StructMatrixAddToValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *index, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Add to matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_StructMatrixAddToBoxValues* to set coefficients a box at a time.

HYPRE_Int **HYPRE_StructMatrixSetConstantValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Set matrix coefficients which are constant over the grid. The *values* array is of length *nentries*.

HYPRE_Int **HYPRE_StructMatrixAddToConstantValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Add to matrix coefficients which are constant over the grid. The *values* array is of length *nentries*.

HYPRE_Int **HYPRE_StructMatrixSetBoxValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Set matrix coefficients a box at a time. The data in *values* is ordered as follows:

```
m = 0;
for (k = ilower[2]; k <= iupper[2]; k++)
   for (j = ilower[1]; j <= iupper[1]; j++)
      for (i = ilower[0]; i <= iupper[0]; i++)
         for (entry = 0; entry < nentries; entry++)
         {
            values[m] = ...;
            m++;
         }
```

HYPRE_Int **HYPRE_StructMatrixAddToBoxValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Add to matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructMatrixSetBoxValues*.

HYPRE_Int **HYPRE_StructMatrixSetBoxValues2**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Set matrix coefficients a box at a time. The *values* array is logically box shaped with value-box extents *vilower* and *viupper* that must contain the set-box extents *ilower* and *iupper* . The data in the *values* array is ordered as in *HYPRE_StructMatrixSetBoxValues*, but based on the value-box extents.

HYPRE_Int **HYPRE_StructMatrixAddToBoxValues2**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Add to matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructMatrixSetBoxValues2*.

HYPRE_Int **HYPRE_StructMatrixAssemble**(*HYPRE_StructMatrix* matrix)

Finalize the construction of the matrix before using.

HYPRE_Int **HYPRE_StructMatrixGetValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *index, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Get matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_StructMatrixGetBoxValues* to get coefficients a box at a time.

HYPRE_Int **HYPRE_StructMatrixGetBoxValues**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Get matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructMatrixSetBoxValues*.

HYPRE_Int **HYPRE_StructMatrixGetBoxValues2**(*HYPRE_StructMatrix* matrix, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Get matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructMatrixSetBoxValues2*.

HYPRE_Int **HYPRE_StructMatrixSetSymmetric**(*HYPRE_StructMatrix* matrix, HYPRE_Int symmetric)

Define symmetry properties of the matrix. By default, matrices are assumed to be nonsymmetric. Significant storage savings can be made if the matrix is symmetric.

HYPRE_Int **HYPRE_StructMatrixSetConstantEntries**(*HYPRE_StructMatrix* matrix, HYPRE_Int nentries, HYPRE_Int *entries)

Specify which stencil entries are constant over the grid. Declaring entries to be "constant over the grid" yields significant memory savings because the value for each declared entry will only be stored once. However, not all solvers are able to utilize this feature.

Presently supported:

- no entries constant (this function need not be called)
- all entries constant
- all but the diagonal entry constant

HYPRE_Int **HYPRE_StructMatrixSetNumGhost**(*HYPRE_StructMatrix* matrix, HYPRE_Int *num_ghost)

Set the ghost layer in the matrix

HYPRE_Int **HYPRE_StructMatrixPrint**(const char *filename, *HYPRE_StructMatrix* matrix, HYPRE_Int all)

Print the matrix to file. This is mainly for debugging purposes.

HYPRE_Int **HYPRE_StructMatrixMatvec**(HYPRE_Complex alpha, *HYPRE_StructMatrix* A, HYPRE_StructVector x, HYPRE_Complex beta, HYPRE_StructVector y)

Matvec operator. This operation is $y = \alpha Ax + \beta y$. Note that you can do a simple matrix-vector multiply by setting $\alpha = 1$ and $\beta = 0$.

Struct Vectors

HYPRE_Int **HYPRE_StructVectorCreate**(MPI_Comm comm, *HYPRE_StructGrid* grid, HYPRE_StructVector *vector)

The vector object. Create a vector object.

HYPRE_Int **HYPRE_StructVectorDestroy**(HYPRE_StructVector vector)

Destroy a vector object.

HYPRE_Int **HYPRE_StructVectorInitialize**(HYPRE_StructVector vector)

Prepare a vector object for setting coefficient values.

HYPRE_Int **HYPRE_StructVectorSetValues**(HYPRE_StructVector vector, HYPRE_Int *index, HYPRE_Complex value)

Set vector coefficients index by index.

NOTE: For better efficiency, use *HYPRE_StructVectorSetBoxValues* to set coefficients a box at a time.

HYPRE_Int **HYPRE_StructVectorAddToValues**(HYPRE_StructVector vector, HYPRE_Int *index, HYPRE_Complex value)

Add to vector coefficients index by index.

NOTE: For better efficiency, use *HYPRE_StructVectorAddToBoxValues* to set coefficients a box at a time.

HYPRE_Int **HYPRE_StructVectorSetBoxValues**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Complex *values)

Set vector coefficients a box at a time. The data in *values* is ordered as follows:

```
m = 0;
for (k = ilower[2]; k <= iupper[2]; k++)
   for (j = ilower[1]; j <= iupper[1]; j++)
      for (i = ilower[0]; i <= iupper[0]; i++)
      {
         values[m] = ...;
         m++;
      }
```

HYPRE_Int **HYPRE_StructVectorAddToBoxValues**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Complex *values)

Add to vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructVectorSetBoxValues*.

HYPRE_Int **HYPRE_StructVectorSetBoxValues2**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Set vector coefficients a box at a time. The *values* array is logically box shaped with value-box extents *vilower* and *viupper* that must contain the set-box extents *ilower* and *iupper* . The data in the *values* array is ordered as in *HYPRE_StructVectorSetBoxValues*, but based on the value-box extents.

HYPRE_Int **HYPRE_StructVectorAddToBoxValues2**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Add to vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructVectorSetBoxValues2*.

HYPRE_Int **HYPRE_StructVectorAssemble**(HYPRE_StructVector vector)

Finalize the construction of the vector before using.

HYPRE_Int **HYPRE_StructVectorGetValues**(HYPRE_StructVector vector, HYPRE_Int *index, HYPRE_Complex *value)

Get vector coefficients index by index.

NOTE: For better efficiency, use *HYPRE_StructVectorGetBoxValues* to get coefficients a box at a time.

HYPRE_Int **HYPRE_StructVectorGetBoxValues**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Complex *values)

Get vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructVectorSetBoxValues*.

HYPRE_Int **HYPRE_StructVectorGetBoxValues2**(HYPRE_StructVector vector, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Get vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_StructVectorSetBoxValues2*.

HYPRE_Int **HYPRE_StructVectorPrint**(const char *filename, HYPRE_StructVector vector, HYPRE_Int all)

Print the vector to file. This is mainly for debugging purposes.

8.2 SStruct System Interface

group **SStructSystemInterface**

This interface represents a semi-structured-grid conceptual view of a linear system.

@memo A semi-structured-grid conceptual interface

SStruct Grids

typedef struct hypre_SStructGrid_struct ***HYPRE_SStructGrid**

A grid object is constructed out of several structured "parts" and an optional unstructured "part". Each structured part has its own abstract index space.

typedef HYPRE_Int **HYPRE_SStructVariable**

An enumerated type that supports cell centered, node centered, face centered, and edge centered variables. Face centered variables are split into x-face, y-face, and z-face variables, and edge centered variables are split into x-edge, y-edge, and z-edge variables. The edge centered variable types are only used in 3D. In 2D, edge centered variables are handled by the face centered types.

Variables are referenced relative to an abstract (cell centered) index in the following way:

- cell centered variables are aligned with the index;
- node centered variables are aligned with the cell corner at relative index (1/2, 1/2, 1/2);
- x-face, y-face, and z-face centered variables are aligned with the faces at relative indexes (1/2, 0, 0), (0, 1/2, 0), and (0, 0, 1/2), respectively;
- x-edge, y-edge, and z-edge centered variables are aligned with the edges at relative indexes (0, 1/2, 1/2), (1/2, 0, 1/2), and (1/2, 1/2, 0), respectively.

The supported identifiers are:

- HYPRE_SSTRUCT_VARIABLE_CELL
- HYPRE_SSTRUCT_VARIABLE_NODE
- HYPRE_SSTRUCT_VARIABLE_XFACE
- HYPRE_SSTRUCT_VARIABLE_YFACE
- HYPRE_SSTRUCT_VARIABLE_ZFACE
- HYPRE_SSTRUCT_VARIABLE_XEDGE
- HYPRE_SSTRUCT_VARIABLE_YEDGE
- HYPRE_SSTRUCT_VARIABLE_ZEDGE

NOTE: Although variables are referenced relative to a unique abstract cell-centered index, some variables are associated with multiple grid cells. For example, node centered variables in 3D are associated with 8 cells (away from boundaries). Although grid cells are distributed uniquely to different processes, variables may be owned by multiple processes because they may be associated with multiple cells.

HYPRE_Int **HYPRE_SStructGridCreate**(MPI_Comm comm, HYPRE_Int ndim, HYPRE_Int nparts, *HYPRE_SStructGrid* *grid)

Create an *ndim*-dimensional grid object with *nparts* structured parts.

HYPRE_Int **HYPRE_SStructGridDestroy**(*HYPRE_SStructGrid* grid)

Destroy a grid object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructGridSetExtents**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper)

Set the extents for a box on a structured part of the grid.

HYPRE_Int **HYPRE_SStructGridSetVariables**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int nvars, *HYPRE_SStructVariable* *vartypes)

Describe the variables that live on a structured part of the grid.

HYPRE_Int **HYPRE_SStructGridAddVariables**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int nvars, *HYPRE_SStructVariable* *vartypes)

Describe additional variables that live at a particular index. These variables are appended to the array of variables set in *HYPRE_SStructGridSetVariables*, and are referenced as such.

NOTE: This routine is not yet supported.

HYPRE_Int **HYPRE_SStructGridSetFEMOrdering**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *ordering)

Set the ordering of variables in a finite element problem. This overrides the default ordering described below.

Array *ordering* is composed of blocks of size (1 + *ndim*). Each block indicates a specific variable in the element and the ordering of the blocks defines the ordering of the variables. A block contains a variable number followed by an offset direction relative to the element's center. For example, a block containing (2, 1, -1, 0) means variable 2 on the edge located in the (1, -1, 0) direction from the center of the element. Note that here variable 2 must be of type ZEDGE for this to make sense. The *ordering* array must account for all variables in the element. This routine can only be called after *HYPRE_SStructGridSetVariables*.

The default ordering for element variables (var, i, j, k) varies fastest in index i, followed by j, then k, then var. For example, if var 0, var 1, and var 2 are declared to be XFACE, YFACE, and NODE variables, respectively, then the default ordering (in 2D) would first list the two XFACE variables, then the two YFACE variables, then the four NODE variables as follows:

(0,-1,0), (0,1,0), (1,0,-1), (1,0,1), (2,-1,-1), (2,1,-1), (2,-1,1), (2,1,1)

HYPRE_Int **HYPRE_SStructGridSetNeighborPart**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int nbor_part, HYPRE_Int *nbor_ilower, HYPRE_Int *nbor_iupper, HYPRE_Int *index_map, HYPRE_Int *index_dir)

Describe how regions just outside of a part relate to other parts. This is done a box at a time.

Parts *part* and *nbor_part* must be different, except in the case where only cell-centered data is used.

Indexes should increase from *ilower* to *iupper*. It is not necessary that indexes increase from *nbor_ilower* to *nbor_iupper*.

The *index_map* describes the mapping of indexes 0, 1, and 2 on part *part* to the corresponding indexes on part *nbor_part*. For example, triple (1, 2, 0) means that indexes 0, 1, and 2 on part *part* map to indexes 1, 2, and 0 on part *nbor_part*, respectively.

The *index_dir* describes the direction of the mapping in *index_map*. For example, triple (1, 1, -1) means that for indexes 0 and 1, increasing values map to increasing values on *nbor_part*, while for index 2, decreasing values map to increasing values.

NOTE: All parts related to each other via this routine must have an identical list of variables and variable types. For example, if part 0 has only two variables on it, a cell centered variable and a node centered variable, and we declare part 1 to be a neighbor of part 0, then part 1 must also have only two variables on it, and they must be of type cell and node. In addition, variables associated with FACEs or EDGEs must be grouped together and listed in X, Y, Z order. This is to enable the code to correctly associate variables on one part with variables on its neighbor part when a coordinate transformation is specified. For example, an XFACE variable on one part may correspond to a YFACE variable on a neighbor part under a particular tranformation, and the code determines this association by assuming that the variable lists are as noted here.

HYPRE_Int **HYPRE_SStructGridSetSharedPart**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int *offset, HYPRE_Int shared_part, HYPRE_Int *shared_ilower, HYPRE_Int *shared_iupper, HYPRE_Int *shared_offset, HYPRE_Int *index_map, HYPRE_Int *index_dir)

Describe how regions inside a part are shared with regions in other parts.

Parts *part* and *shared_part* must be different.

Indexes should increase from *ilower* to *iupper*. It is not necessary that indexes increase from *shared_ilower* to *shared_iupper*. This is to maintain consistency with the `SetNeighborPart` function, which is also able to describe shared regions but in a more limited fashion.

The *offset* is a triple (in 3D) used to indicate the dimensionality of the shared set of data and its position with respect to the box extents *ilower* and *iupper* on part *part*. The dimensionality is given by the number of 0's in the triple, and the position is given by plus or minus 1's. For example: (0, 0, 0) indicates sharing of all data in the given box; (1, 0, 0) indicates sharing of data on the faces in the (1, 0, 0) direction; (1, 0, -1) indicates sharing of data on the edges in the (1, 0, -1) direction; and (1, -1, 1) indicates sharing of data on the nodes in the (1, -1, 1) direction. To ensure the dimensionality, it is required that for every nonzero entry, the corresponding extents of the box are the same. For example, if *offset* is (0, 1, 0), then (2, 1, 3) and (10, 1, 15) are valid box extents, whereas (2, 1, 3) and (10, 7, 15) are invalid (because 1 and 7 are not the same).

The *shared_offset* is used in the same way as *offset*, but with respect to the box extents *shared_ilower* and *shared_iupper* on part *shared_part*.

The *index_map* describes the mapping of indexes 0, 1, and 2 on part *part* to the corresponding indexes on part *shared_part*. For example, triple (1, 2, 0) means that indexes 0, 1, and 2 on part *part* map to indexes 1, 2, and 0 on part *shared_part*, respectively.

The *index_dir* describes the direction of the mapping in *index_map*. For example, triple (1, 1, -1) means that for indexes 0 and 1, increasing values map to increasing values on *shared_part*, while for index 2, decreasing values map to increasing values.

NOTE: All parts related to each other via this routine must have an identical list of variables and variable types. For example, if part 0 has only two variables on it, a cell centered variable and a node centered variable, and we declare part 1 to have shared regions with part 0, then part 1 must also have only two variables on it, and they must be of type cell and node. In addition, variables associated with FACEs or EDGEs must be grouped together and listed in X, Y, Z order. This is to enable the code to correctly associate variables on one part with variables on a shared part when a coordinate transformation is specified. For example, an XFACE variable on one part may correspond to a YFACE variable on a shared part under a

particular tranformation, and the code determines this association by assuming that the variable lists are as noted here.

HYPRE_Int **HYPRE_SStructGridAddUnstructuredPart**(*HYPRE_SStructGrid* grid, HYPRE_Int ilower, HYPRE_Int iupper)

Add an unstructured part to the grid. The variables in the unstructured part of the grid are referenced by a global rank between 0 and the total number of unstructured variables minus one. Each process owns some unique consecutive range of variables, defined by *ilower* and *iupper*.

NOTE: This is just a placeholder. This part of the interface is not finished.

HYPRE_Int **HYPRE_SStructGridAssemble**(*HYPRE_SStructGrid* grid)

Finalize the construction of the grid before using.

HYPRE_Int **HYPRE_SStructGridSetPeriodic**(*HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int *periodic)

Set the periodicity on a particular part.

The argument *periodic* is an *ndim-dimensional* integer array that contains the periodicity for each dimension. A zero value for a dimension means non-periodic, while a nonzero value means periodic and contains the actual period. For example, periodicity in the first and third dimensions for a 10x11x12 part is indicated by the array [10,0,12].

NOTE: Some of the solvers in hypre have power-of-two restrictions on the size of the periodic dimensions.

HYPRE_Int **HYPRE_SStructGridSetNumGhost**(*HYPRE_SStructGrid* grid, HYPRE_Int *num_ghost)

Setting ghost in the sgrids.

HYPRE_SSTRUCT_VARIABLE_UNDEFINED

HYPRE_SSTRUCT_VARIABLE_CELL

HYPRE_SSTRUCT_VARIABLE_NODE

HYPRE_SSTRUCT_VARIABLE_XFACE

HYPRE_SSTRUCT_VARIABLE_YFACE

HYPRE_SSTRUCT_VARIABLE_ZFACE

HYPRE_SSTRUCT_VARIABLE_XEDGE

HYPRE_SSTRUCT_VARIABLE_YEDGE

HYPRE_SSTRUCT_VARIABLE_ZEDGE

SStruct Stencils

typedef struct hypre_SStructStencil_struct ***HYPRE_SStructStencil**

The stencil object.

HYPRE_Int **HYPRE_SStructStencilCreate**(HYPRE_Int ndim, HYPRE_Int size, *HYPRE_SStructStencil* *stencil)

Create a stencil object for the specified number of spatial dimensions and stencil entries.

HYPRE_Int **HYPRE_SStructStencilDestroy**(*HYPRE_SStructStencil* stencil)

Destroy a stencil object.

HYPRE_Int **HYPRE_SStructStencilSetEntry**(*HYPRE_SStructStencil* stencil, HYPRE_Int entry, HYPRE_Int *offset, HYPRE_Int var)

Set a stencil entry.

SStruct Graphs

typedef struct hypre_SStructGraph_struct ***HYPRE_SStructGraph**

The graph object is used to describe the nonzero structure of a matrix.

HYPRE_Int **HYPRE_SStructGraphCreate**(MPI_Comm comm, *HYPRE_SStructGrid* grid, *HYPRE_SStructGraph* *graph)

Create a graph object.

HYPRE_Int **HYPRE_SStructGraphDestroy**(*HYPRE_SStructGraph* graph)

Destroy a graph object.

HYPRE_Int **HYPRE_SStructGraphSetDomainGrid**(*HYPRE_SStructGraph* graph, *HYPRE_SStructGrid* domain_grid)

Set the domain grid.

HYPRE_Int **HYPRE_SStructGraphSetStencil**(*HYPRE_SStructGraph* graph, HYPRE_Int part, HYPRE_Int var, *HYPRE_SStructStencil* stencil)

Set the stencil for a variable on a structured part of the grid.

HYPRE_Int **HYPRE_SStructGraphSetFEM**(*HYPRE_SStructGraph* graph, HYPRE_Int part)

Indicate that an FEM approach will be used to set matrix values on this part.

HYPRE_Int **HYPRE_SStructGraphSetFEMSparsity**(*HYPRE_SStructGraph* graph, HYPRE_Int part, HYPRE_Int nsparse, HYPRE_Int *sparsity)

Set the finite element stiffness matrix sparsity. This overrides the default full sparsity pattern described below.

Array *sparsity* contains *nsparse* row/column tuples (I,J) that indicate the nonzeroes of the local stiffness matrix. The layout of the values passed into the routine *HYPRE_SStructMatrixAddFEMValues* is determined here.

The default sparsity is full (each variable is coupled to all others), and the values passed into the routine *HYPRE_SStructMatrixAddFEMValues* are assumed to be by rows (that is, column indices vary fastest).

HYPRE_Int **HYPRE_SStructGraphAddEntries**(*HYPRE_SStructGraph* graph, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Int to_part, HYPRE_Int *to_index, HYPRE_Int to_var)

Add a non-stencil graph entry at a particular index. This graph entry is appended to the existing graph entries, and is referenced as such.

NOTE: Users are required to set graph entries on all processes that own the associated variables. This means that some data will be multiply defined.

HYPRE_Int **HYPRE_SStructGraphAssemble**(*HYPRE_SStructGraph* graph)

Finalize the construction of the graph before using.

HYPRE_Int **HYPRE_SStructGraphSetObjectType**(*HYPRE_SStructGraph* graph, HYPRE_Int type)

Set the storage type of the associated matrix object. It is used before AddEntries and Assemble to compute the right ranks in the graph.

NOTE: This routine is only necessary for implementation reasons, and will eventually be removed.

See also:

HYPRE_SStructMatrixSetObjectType

SStruct Matrices

typedef struct hypre_SStructMatrix_struct ***HYPRE_SStructMatrix**

The matrix object.

HYPRE_Int **HYPRE_SStructMatrixCreate**(MPI_Comm comm, *HYPRE_SStructGraph* graph, *HYPRE_SStructMatrix* *matrix)

Create a matrix object.

HYPRE_Int **HYPRE_SStructMatrixDestroy**(*HYPRE_SStructMatrix* matrix)

Destroy a matrix object.

HYPRE_Int **HYPRE_SStructMatrixInitialize**(*HYPRE_SStructMatrix* matrix)

Prepare a matrix object for setting coefficient values.

HYPRE_Int **HYPRE_SStructMatrixSetValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Set matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_SStructMatrixSetBoxValues* to set coefficients a box at a time.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

NOTE: The entries in this routine must all be of the same type: either stencil or non-stencil, but not both. Also, if they are stencil entries, they must all represent couplings to the same variable type (there are no such restrictions for non-stencil entries).

HYPRE_Int **HYPRE_SStructMatrixAddToValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Add to matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_SStructMatrixAddToBoxValues* to set coefficients a box at a time.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

NOTE: The entries in this routine must all be of the same type: either stencil or non-stencil, but not both. Also, if they are stencil entries, they must all represent couplings to the same variable type.

HYPRE_Int **HYPRE_SStructMatrixAddFEMValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *index, HYPRE_Complex *values)

Add finite element stiffness matrix coefficients index by index. The layout of the data in *values* is determined by the routines *HYPRE_SStructGridSetFEMOrdering* and *HYPRE_SStructGraphSetFEMSparsity*.

HYPRE_Int **HYPRE_SStructMatrixGetValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Get matrix coefficients index by index. The *values* array is of length *nentries*.

NOTE: For better efficiency, use *HYPRE_SStructMatrixGetBoxValues* to get coefficients a box at a time.

NOTE: Users may get values on any process that owns the associated variables.

NOTE: The entries in this routine must all be of the same type: either stencil or non-stencil, but not both. Also, if they are stencil entries, they must all represent couplings to the same variable type (there are no such restrictions for non-stencil entries).

HYPRE_Int **HYPRE_SStructMatrixGetFEMValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *index, HYPRE_Complex *values)

Get finite element stiffness matrix coefficients index by index. The layout of the data in *values* is determined by the routines *HYPRE_SStructGridSetFEMOrdering* and *HYPRE_SStructGraphSetFEMSparsity*.

HYPRE_Int **HYPRE_SStructMatrixSetBoxValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Set matrix coefficients a box at a time. The data in *values* is ordered as follows:

```
m = 0;
for (k = ilower[2]; k <= iupper[2]; k++)
   for (j = ilower[1]; j <= iupper[1]; j++)
      for (i = ilower[0]; i <= iupper[0]; i++)
         for (entry = 0; entry < nentries; entry++)
         {
            values[m] = ...;
            m++;
         }
```

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

NOTE: The entries in this routine must all be of the same type: either stencil or non-stencil, but not both. Also, if they are stencil entries, they must all represent couplings to the same variable type (there are no such restrictions for non-stencil entries).

HYPRE_Int **HYPRE_SStructMatrixAddToBoxValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Add to matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructMatrixSetBoxValues*.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

NOTE: The entries in this routine must all be of stencil type. Also, they must all represent couplings to the same variable type.

HYPRE_Int **HYPRE_SStructMatrixSetBoxValues2**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Set matrix coefficients a box at a time. The *values* array is logically box shaped with value-box extents *vilower* and *viupper* that must contain the set-box extents *ilower* and *iupper* . The data in the *values* array is ordered as in *HYPRE_SStructMatrixSetBoxValues*, but based on the value-box extents.

HYPRE_Int **HYPRE_SStructMatrixAddToBoxValues2**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Add to matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructMatrixSetBoxValues2*.

HYPRE_Int **HYPRE_SStructMatrixAssemble**(*HYPRE_SStructMatrix* matrix)

Finalize the construction of the matrix before using.

HYPRE_Int **HYPRE_SStructMatrixGetBoxValues**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Complex *values)

Get matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructMatrixSetBoxValues*.

NOTE: Users may get values on any process that owns the associated variables.

NOTE: The entries in this routine must all be of stencil type. Also, they must all represent couplings to the same variable type.

HYPRE_Int **HYPRE_SStructMatrixGetBoxValues2**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int nentries, HYPRE_Int *entries, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Get matrix coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructMatrixSetBoxValues2*.

HYPRE_Int **HYPRE_SStructMatrixSetSymmetric**(*HYPRE_SStructMatrix* matrix, HYPRE_Int part, HYPRE_Int var, HYPRE_Int to_var, HYPRE_Int symmetric)

Define symmetry properties for the stencil entries in the matrix. The boolean argument *symmetric* is applied to stencil entries on part *part* that couple variable *var* to variable *to_var*. A value of -1 may be used for *part*, *var*, or *to_var* to specify "all". For example, if *part* and *to_var* are set to -1, then the boolean is applied to stencil entries on all parts that couple variable *var* to all other variables.

By default, matrices are assumed to be nonsymmetric. Significant storage savings can be made if the matrix is symmetric.

HYPRE_Int **HYPRE_SStructMatrixSetNSSymmetric**(*HYPRE_SStructMatrix* matrix, HYPRE_Int symmetric)

Define symmetry properties for all non-stencil matrix entries.

HYPRE_Int **HYPRE_SStructMatrixSetObjectType**(*HYPRE_SStructMatrix* matrix, HYPRE_Int type)

Set the storage type of the matrix object to be constructed. Currently, *type* can be either `HYPRE_SSTRUCT` (the default), `HYPRE_STRUCT`, or `HYPRE_PARCSR`.

See also:

HYPRE_SStructMatrixGetObject

HYPRE_Int **HYPRE_SStructMatrixGetObject**(*HYPRE_SStructMatrix* matrix, void **object)

Get a reference to the constructed matrix object.

See also:

HYPRE_SStructMatrixSetObjectType

HYPRE_Int **HYPRE_SStructMatrixPrint**(const char *filename, *HYPRE_SStructMatrix* matrix, HYPRE_Int all)

Print the matrix to file. This is mainly for debugging purposes.

SStruct Vectors

typedef struct hypre_SStructVector_struct ***HYPRE_SStructVector**

The vector object.

HYPRE_Int **HYPRE_SStructVectorCreate**(MPI_Comm comm, *HYPRE_SStructGrid* grid, *HYPRE_SStructVector* *vector)

Create a vector object.

HYPRE_Int **HYPRE_SStructVectorDestroy**(*HYPRE_SStructVector* vector)

Destroy a vector object.

HYPRE_Int **HYPRE_SStructVectorInitialize**(*HYPRE_SStructVector* vector)

Prepare a vector object for setting coefficient values.

HYPRE_Int **HYPRE_SStructVectorSetValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Complex *value)

Set vector coefficients index by index.

NOTE: For better efficiency, use *HYPRE_SStructVectorSetBoxValues* to set coefficients a box at a time.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

HYPRE_Int **HYPRE_SStructVectorAddToValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Complex *value)

Add to vector coefficients index by index.

NOTE: For better efficiency, use *HYPRE_SStructVectorAddToBoxValues* to set coefficients a box at a time.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

HYPRE_Int **HYPRE_SStructVectorAddFEMValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *index, HYPRE_Complex *values)

Add finite element vector coefficients index by index. The layout of the data in *values* is determined by the routine *HYPRE_SStructGridSetFEMOrdering*.

HYPRE_Int **HYPRE_SStructVectorGetValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *index, HYPRE_Int var, HYPRE_Complex *value)

Get vector coefficients index by index. Users must first call the routine *HYPRE_SStructVectorGather* to ensure that data owned by multiple processes is correct.

NOTE: For better efficiency, use *HYPRE_SStructVectorGetBoxValues* to get coefficients a box at a time.

NOTE: Users may only get values on processes that own the associated variables.

HYPRE_Int **HYPRE_SStructVectorGetFEMValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *index, HYPRE_Complex *values)

Get finite element vector coefficients index by index. The layout of the data in *values* is determined by the routine *HYPRE_SStructGridSetFEMOrdering*. Users must first call the routine *HYPRE_SStructVectorGather* to ensure that data owned by multiple processes is correct.

HYPRE_Int **HYPRE_SStructVectorSetBoxValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Complex *values)

Set vector coefficients a box at a time. The data in *values* is ordered as follows:

```
m = 0;
for (k = ilower[2]; k <= iupper[2]; k++)
   for (j = ilower[1]; j <= iupper[1]; j++)
      for (i = ilower[0]; i <= iupper[0]; i++)
      {
         values[m] = ...;
         m++;
      }
```

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

HYPRE_Int **HYPRE_SStructVectorAddToBoxValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Complex *values)

Add to vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructVectorSetBoxValues*.

NOTE: Users are required to set values on all processes that own the associated variables. This means that some data will be multiply defined.

HYPRE_Int **HYPRE_SStructVectorSetBoxValues2**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Set vector coefficients a box at a time. The *values* array is logically box shaped with value-box extents *vilower* and *viupper* that must contain the set-box extents *ilower* and *iupper* . The data in the *values* array is ordered as in *HYPRE_SStructVectorSetBoxValues*, but based on the value-box extents.

HYPRE_Int **HYPRE_SStructVectorAddToBoxValues2**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Add to vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructVectorSetBoxValues2*.

HYPRE_Int **HYPRE_SStructVectorAssemble**(*HYPRE_SStructVector* vector)

Finalize the construction of the vector before using.

HYPRE_Int **HYPRE_SStructVectorGetBoxValues**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Complex *values)

Get vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructVectorSetBoxValues*. Users must first call the routine *HYPRE_SStructVectorGather* to ensure that data owned by multiple processes is correct.

NOTE: Users may only get values on processes that own the associated variables.

HYPRE_Int **HYPRE_SStructVectorGetBoxValues2**(*HYPRE_SStructVector* vector, HYPRE_Int part, HYPRE_Int *ilower, HYPRE_Int *iupper, HYPRE_Int var, HYPRE_Int *vilower, HYPRE_Int *viupper, HYPRE_Complex *values)

Get vector coefficients a box at a time. The data in *values* is ordered as in *HYPRE_SStructVectorSetBoxValues2*.

HYPRE_Int **HYPRE_SStructVectorGather**(*HYPRE_SStructVector* vector)

Gather vector data so that efficient `GetValues` can be done. This routine must be called prior to calling `GetValues` to ensure that correct and consistent values are returned, especially for non cell-centered data that is shared between more than one processor.

HYPRE_Int **HYPRE_SStructVectorSetObjectType**(*HYPRE_SStructVector* vector, HYPRE_Int type)

Set the storage type of the vector object to be constructed. Currently, *type* can be either `HYPRE_SSTRUCT` (the default), `HYPRE_STRUCT`, or `HYPRE_PARCSR`.

See also:

HYPRE_SStructVectorGetObject

HYPRE_Int **HYPRE_SStructVectorGetObject**(*HYPRE_SStructVector* vector, void **object)

Get a reference to the constructed vector object.

See also:

HYPRE_SStructVectorSetObjectType

HYPRE_Int **HYPRE_SStructVectorPrint**(const char *filename, *HYPRE_SStructVector* vector, HYPRE_Int all)

Print the vector to file. This is mainly for debugging purposes.

8.3 IJ System Interface

group **IJSystemInterface**

This interface represents a linear-algebraic conceptual view of a linear system. The 'I' and 'J' in the name are meant to be mnemonic for the traditional matrix notation A(I,J).

@memo A linear-algebraic conceptual interface

IJ Matrices

typedef struct hypre_IJMatrix_struct ***HYPRE_IJMatrix**

The matrix object.

HYPRE_Int **HYPRE_IJMatrixCreate**(MPI_Comm comm, HYPRE_BigInt ilower, HYPRE_BigInt iupper, HYPRE_BigInt jlower, HYPRE_BigInt jupper, *HYPRE_IJMatrix* *matrix)

Create a matrix object. Each process owns some unique consecutive range of rows, indicated by the global row indices *ilower* and *iupper*. The row data is required to be such that the value of *ilower* on any process p be exactly one more than the value of *iupper* on process $p-1$. Note that the first row of the global matrix may start with any integer value. In particular, one may use zero- or one-based indexing.

For square matrices, *jlower* and *jupper* typically should match *ilower* and *iupper*, respectively. For rectangular matrices, *jlower* and *jupper* should define a partitioning of the columns. This partitioning must be used for any vector v that will be used in matrix-vector products with the rectangular matrix. The matrix data structure may use *jlower* and *jupper* to store the diagonal blocks (rectangular in general) of the matrix separately from the rest of the matrix.

Collective.

HYPRE_Int **HYPRE_IJMatrixDestroy**(*HYPRE_IJMatrix* matrix)

Destroy a matrix object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_IJMatrixInitialize**(*HYPRE_IJMatrix* matrix)

Prepare a matrix object for setting coefficient values. This routine will also re-initialize an already assembled matrix, allowing users to modify coefficient values.

HYPRE_Int **HYPRE_IJMatrixInitialize_v2**(*HYPRE_IJMatrix* matrix, HYPRE_MemoryLocation memory_location)

Prepare a matrix object for setting coefficient values. This routine will also re-initialize an already assembled matrix, allowing users to modify coefficient values. This routine also specifies the memory location, i.e. host or device.

HYPRE_Int **HYPRE_IJMatrixSetValues**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_Int *ncols, const HYPRE_BigInt *rows, const HYPRE_BigInt *cols, const HYPRE_Complex *values)

Sets values for *nrows* rows or partial rows of the matrix. The arrays *ncols* and *rows* are of dimension *nrows* and contain the number of columns in each row and the row indices, respectively. The array *cols* contains the column indices for each of the *rows*, and is ordered by rows. The data in the *values* array corresponds directly to the column entries in *cols*. Erases any previous values at the specified locations and replaces them with new ones, or, if there was no value there before, inserts a new one if set locally. Note that it is not

possible to set values on other processors. If one tries to set a value from proc i on proc j, proc i will erase all previous occurrences of this value in its stack (including values generated with AddToValues), and treat it like a zero value. The actual value needs to be set on proc j.

Note that a threaded version (threaded over the number of rows) will be called if HYPRE_IJMatrixSetOMPFlag is set to a value != 0. This requires that rows[i] != rows[j] for i!= j and is only efficient if a large number of rows is set in one call to HYPRE_IJMatrixSetValues.

Not collective.

HYPRE_Int **HYPRE_IJMatrixSetConstantValues**(*HYPRE_IJMatrix* matrix, HYPRE_Complex value)

Sets all matrix coefficients of an already assembled matrix to *value*

HYPRE_Int **HYPRE_IJMatrixAddToValues**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_Int *ncols, const HYPRE_BigInt *rows, const HYPRE_BigInt *cols, const HYPRE_Complex *values)

Adds to values for *nrows* rows or partial rows of the matrix. Usage details are analogous to *HYPRE_IJMatrixSetValues*. Adds to any previous values at the specified locations, or, if there was no value there before, inserts a new one. AddToValues can be used to add to values on other processors.

Note that a threaded version (threaded over the number of rows) will be called if HYPRE_IJMatrixSetOMPFlag is set to a value != 0. This requires that rows[i] != rows[j] for i!= j and is only efficient if a large number of rows is added in one call to HYPRE_IJMatrixAddToValues.

Not collective.

HYPRE_Int **HYPRE_IJMatrixSetValues2**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_Int *ncols, const HYPRE_BigInt *rows, const HYPRE_Int *row_indexes, const HYPRE_BigInt *cols, const HYPRE_Complex *values)

Sets values for *nrows* rows or partial rows of the matrix.

Same as IJMatrixSetValues, but with an additional *row_indexes* array that provides indexes into the *cols* and *values* arrays. Because of this, there can be gaps between the row data in these latter two arrays.

HYPRE_Int **HYPRE_IJMatrixAddToValues2**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_Int *ncols, const HYPRE_BigInt *rows, const HYPRE_Int *row_indexes, const HYPRE_BigInt *cols, const HYPRE_Complex *values)

Adds to values for *nrows* rows or partial rows of the matrix.

Same as IJMatrixAddToValues, but with an additional *row_indexes* array that provides indexes into the *cols* and *values* arrays. Because of this, there can be gaps between the row data in these latter two arrays.

HYPRE_Int **HYPRE_IJMatrixAssemble**(*HYPRE_IJMatrix* matrix)

Finalize the construction of the matrix before using.

HYPRE_Int **HYPRE_IJMatrixGetRowCounts**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_BigInt *rows, HYPRE_Int *ncols)

Gets number of nonzeros elements for *nrows* rows specified in *rows* and returns them in *ncols*, which needs to be allocated by the user.

HYPRE_Int **HYPRE_IJMatrixGetValues**(*HYPRE_IJMatrix* matrix, HYPRE_Int nrows, HYPRE_Int *ncols, HYPRE_BigInt *rows, HYPRE_BigInt *cols, HYPRE_Complex *values)

Gets values for *nrows* rows or partial rows of the matrix. Usage details are mostly analogous to *HYPRE_IJMatrixSetValues*. Note that if nrows is negative, the routine will return the column_indices and matrix coefficients of the (-nrows) rows contained in rows.

HYPRE_Int **HYPRE_IJMatrixSetObjectType**(*HYPRE_IJMatrix* matrix, HYPRE_Int type)

Set the storage type of the matrix object to be constructed. Currently, *type* can only be HYPRE_PARCSR.

Not collective, but must be the same on all processes.

See also:

HYPRE_IJMatrixGetObject

HYPRE_Int **HYPRE_IJMatrixGetObjectType**(*HYPRE_IJMatrix* matrix, HYPRE_Int *type)

Get the storage type of the constructed matrix object.

HYPRE_Int **HYPRE_IJMatrixGetLocalRange**(*HYPRE_IJMatrix* matrix, HYPRE_BigInt *ilower, HYPRE_BigInt *iupper, HYPRE_BigInt *jlower, HYPRE_BigInt *jupper)

Gets range of rows owned by this processor and range of column partitioning for this processor.

HYPRE_Int **HYPRE_IJMatrixGetObject**(*HYPRE_IJMatrix* matrix, void **object)

Get a reference to the constructed matrix object.

See also:

HYPRE_IJMatrixSetObjectType

HYPRE_Int **HYPRE_IJMatrixSetRowSizes**(*HYPRE_IJMatrix* matrix, const HYPRE_Int *sizes)

(Optional) Set the max number of nonzeros to expect in each row. The array *sizes* contains estimated sizes for each row on this process. This call can significantly improve the efficiency of matrix construction, and should always be utilized if possible.

Not collective.

HYPRE_Int **HYPRE_IJMatrixSetDiagOffdSizes**(*HYPRE_IJMatrix* matrix, const HYPRE_Int *diag_sizes, const HYPRE_Int *offdiag_sizes)

(Optional) Sets the exact number of nonzeros in each row of the diagonal and off-diagonal blocks. The diagonal block is the submatrix whose column numbers correspond to rows owned by this process, and the off-diagonal block is everything else. The arrays *diag_sizes* and *offdiag_sizes* contain estimated sizes for each row of the diagonal and off-diagonal blocks, respectively. This routine can significantly improve the efficiency of matrix construction, and should always be utilized if possible.

Not collective.

HYPRE_Int **HYPRE_IJMatrixSetMaxOffProcElmts**(*HYPRE_IJMatrix* matrix, HYPRE_Int max_off_proc_elmts)

(Optional) Sets the maximum number of elements that are expected to be set (or added) on other processors from this processor This routine can significantly improve the efficiency of matrix construction, and should always be utilized if possible.

Not collective.

HYPRE_Int **HYPRE_IJMatrixSetPrintLevel**(*HYPRE_IJMatrix* matrix, HYPRE_Int print_level)

(Optional) Sets the print level, if the user wants to print error messages. The default is 0, i.e. no error messages are printed.

HYPRE_Int **HYPRE_IJMatrixSetOMPFlag**(*HYPRE_IJMatrix* matrix, HYPRE_Int omp_flag)

(Optional) if set, will use a threaded version of HYPRE_IJMatrixSetValues and HYPRE_IJMatrixAddToValues. This is only useful if a large number of rows is set or added to at once.

NOTE that the values in the rows array of HYPRE_IJMatrixSetValues or HYPRE_IJMatrixAddToValues must be different from each other !!!

This option is VERY inefficient if only a small number of rows is set or added at once and/or if reallocation of storage is required and/or if values are added to off processor values.

HYPRE_Int **HYPRE_IJMatrixRead**(const char *filename, MPI_Comm comm, HYPRE_Int type, *HYPRE_IJMatrix* *matrix)

Read the matrix from file. This is mainly for debugging purposes.

HYPRE_Int **HYPRE_IJMatrixPrint**(*HYPRE_IJMatrix* matrix, const char *filename)

Print the matrix to file. This is mainly for debugging purposes.

IJ Vectors

typedef struct hypre_IJVector_struct ***HYPRE_IJVector**

The vector object.

HYPRE_Int **HYPRE_IJVectorCreate**(MPI_Comm comm, HYPRE_BigInt jlower, HYPRE_BigInt jupper, *HYPRE_IJVector* *vector)

Create a vector object. Each process owns some unique consecutive range of vector unknowns, indicated by the global indices *jlower* and *jupper*. The data is required to be such that the value of *jlower* on any process p be exactly one more than the value of *jupper* on process $p-1$. Note that the first index of the global vector may start with any integer value. In particular, one may use zero- or one-based indexing.

Collective.

HYPRE_Int **HYPRE_IJVectorDestroy**(*HYPRE_IJVector* vector)

Destroy a vector object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_IJVectorInitialize**(*HYPRE_IJVector* vector)

Prepare a vector object for setting coefficient values. This routine will also re-initialize an already assembled vector, allowing users to modify coefficient values.

HYPRE_Int **HYPRE_IJVectorInitialize_v2**(*HYPRE_IJVector* vector, HYPRE_MemoryLocation memory_location)

Prepare a vector object for setting coefficient values. This routine will also re-initialize an already assembled vector, allowing users to modify coefficient values. This routine also specifies the memory location, i.e. host or device.

HYPRE_Int **HYPRE_IJVectorSetMaxOffProcElmts**(*HYPRE_IJVector* vector, HYPRE_Int max_off_proc_elmts)

(Optional) Sets the maximum number of elements that are expected to be set (or added) on other processors from this processor This routine can significantly improve the efficiency of matrix construction, and should always be utilized if possible.

Not collective.

HYPRE_Int **HYPRE_IJVectorSetValues**(*HYPRE_IJVector* vector, HYPRE_Int nvalues, const HYPRE_BigInt *indices, const HYPRE_Complex *values)

Sets values in vector. The arrays *values* and *indices* are of dimension *nvalues* and contain the vector values to be set and the corresponding global vector indices, respectively. Erases any previous values at the specified

locations and replaces them with new ones. Note that it is not possible to set values on other processors. If one tries to set a value from proc i on proc j, proc i will erase all previous occurrences of this value in its stack (including values generated with AddToValues), and treat it like a zero value. The actual value needs to be set on proc j.

Not collective.

HYPRE_Int **HYPRE_IJVectorAddToValues**(*HYPRE_IJVector* vector, HYPRE_Int nvalues, const HYPRE_BigInt *indices, const HYPRE_Complex *values)

Adds to values in vector. Usage details are analogous to *HYPRE_IJVectorSetValues*. Adds to any previous values at the specified locations, or, if there was no value there before, inserts a new one. AddToValues can be used to add to values on other processors.

Not collective.

HYPRE_Int **HYPRE_IJVectorAssemble**(*HYPRE_IJVector* vector)

Finalize the construction of the vector before using.

HYPRE_Int **HYPRE_IJVectorGetValues**(*HYPRE_IJVector* vector, HYPRE_Int nvalues, const HYPRE_BigInt *indices, HYPRE_Complex *values)

Gets values in vector. Usage details are analogous to *HYPRE_IJVectorSetValues*.

Not collective.

HYPRE_Int **HYPRE_IJVectorSetObjectType**(*HYPRE_IJVector* vector, HYPRE_Int type)

Set the storage type of the vector object to be constructed. Currently, *type* can only be `HYPRE_PARCSR`.

Not collective, but must be the same on all processes.

See also:

HYPRE_IJVectorGetObject

HYPRE_Int **HYPRE_IJVectorGetObjectType**(*HYPRE_IJVector* vector, HYPRE_Int *type)

Get the storage type of the constructed vector object.

HYPRE_Int **HYPRE_IJVectorGetLocalRange**(*HYPRE_IJVector* vector, HYPRE_BigInt *jlower, HYPRE_BigInt *jupper)

Returns range of the part of the vector owned by this processor.

HYPRE_Int **HYPRE_IJVectorGetObject**(*HYPRE_IJVector* vector, void **object)

Get a reference to the constructed vector object.

See also:

HYPRE_IJVectorSetObjectType

HYPRE_Int **HYPRE_IJVectorSetPrintLevel**(*HYPRE_IJVector* vector, HYPRE_Int print_level)

(Optional) Sets the print level, if the user wants to print error messages. The default is 0, i.e. no error messages are printed.

HYPRE_Int **HYPRE_IJVectorRead**(const char *filename, MPI_Comm comm, HYPRE_Int type, *HYPRE_IJVector* *vector)

Read the vector from file. This is mainly for debugging purposes.

HYPRE_Int **HYPRE_IJVectorPrint**(*HYPRE_IJVector* vector, const char *filename)

Print the vector to file. This is mainly for debugging purposes.

8.4 Struct Solvers

group **StructSolvers**

These solvers use matrix/vector storage schemes that are tailored to structured grid problems.

@memo Linear solvers for structured grids

Struct Solvers

typedef struct hypre_StructSolver_struct ***HYPRE_StructSolver**

The solver object.

typedef HYPRE_Int (***HYPRE_PtrToStructSolverFcn**)(*HYPRE_StructSolver*, *HYPRE_StructMatrix*, HYPRE_StructVector, HYPRE_StructVector)

typedef struct hypre_Solver_struct ***HYPRE_Solver**

typedef HYPRE_Int (***HYPRE_PtrToModifyPCFcn**)(*HYPRE_Solver*, HYPRE_Int, HYPRE_Real)

HYPRE_MODIFYPC

HYPRE_SOLVER_STRUCT

Struct Jacobi Solver

HYPRE_Int **HYPRE_StructJacobiCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructJacobiDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_StructJacobiSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_StructJacobiSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Solve the system.

HYPRE_Int **HYPRE_StructJacobiSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_StructJacobiGetTol**(*HYPRE_StructSolver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_StructJacobiSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_StructJacobiGetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_StructJacobiSetZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_StructJacobiGetZeroGuess**(*HYPRE_StructSolver* solver, HYPRE_Int *zeroguess)

HYPRE_Int **HYPRE_StructJacobiSetNonZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_StructJacobiGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_StructJacobiGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

Struct PFMG Solver

PFMG is a semicoarsening multigrid solver that uses pointwise relaxation. For periodic problems, users should try to set the grid size in periodic dimensions to be as close to a power-of-two as possible. That is, if the grid size in a periodic dimension is given by $N = 2^m * M$ where M is not a power-of-two, then M should be as small as possible. Large values of M will generally result in slower convergence rates.

HYPRE_Int **HYPRE_StructPFMGCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructPFMGDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructPFMGSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_StructPFMGSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Solve the system.

HYPRE_Int **HYPRE_StructPFMGSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_StructPFMGGetTol**(*HYPRE_StructSolver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_StructPFMGSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_StructPFMGGetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_StructPFMGSetMaxLevels**(*HYPRE_StructSolver* solver, HYPRE_Int max_levels)

(Optional) Set maximum number of multigrid grid levels.

HYPRE_Int **HYPRE_StructPFMGGetMaxLevels**(*HYPRE_StructSolver* solver, HYPRE_Int *max_levels)

HYPRE_Int **HYPRE_StructPFMGSetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_StructPFMGGetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int *rel_change)

HYPRE_Int **HYPRE_StructPFMGSetZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_StructPFMGGetZeroGuess**(*HYPRE_StructSolver* solver, HYPRE_Int *zeroguess)

HYPRE_Int **HYPRE_StructPFMGSetNonZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_StructPFMGSetRelaxType**(*HYPRE_StructSolver* solver, HYPRE_Int relax_type)

(Optional) Set relaxation type.

Current relaxation methods set by *relax_type* are:

- 0 : Jacobi
- 1 : Weighted Jacobi (default)
- 2 : Red/Black Gauss-Seidel (symmetric: RB pre-relaxation, BR post-relaxation)
- 3 : Red/Black Gauss-Seidel (nonsymmetric: RB pre- and post-relaxation)

HYPRE_Int **HYPRE_StructPFMGGetRelaxType**(*HYPRE_StructSolver* solver, HYPRE_Int *relax_type)

HYPRE_Int **HYPRE_StructPFMGSetJacobiWeight**(*HYPRE_StructSolver* solver, HYPRE_Real weight)

HYPRE_Int **HYPRE_StructPFMGGetJacobiWeight**(*HYPRE_StructSolver* solver, HYPRE_Real *weight)

HYPRE_Int **HYPRE_StructPFMGSetRAPType**(*HYPRE_StructSolver* solver, HYPRE_Int rap_type)

(Optional) Set type of coarse-grid operator to use.

Current operators set by *rap_type* are:

- 0 : Galerkin (default)
- 1 : non-Galerkin 5-pt or 7-pt stencils

Both operators are constructed algebraically. The non-Galerkin option maintains a 5-pt stencil in 2D and a 7-pt stencil in 3D on all grid levels. The stencil coefficients are computed by averaging techniques.

HYPRE_Int **HYPRE_StructPFMGGetRAPType**(*HYPRE_StructSolver* solver, HYPRE_Int *rap_type)

HYPRE_Int **HYPRE_StructPFMGSetNumPreRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_pre_relax)

(Optional) Set number of relaxation sweeps before coarse-grid correction.

HYPRE_Int **HYPRE_StructPFMGGetNumPreRelax**(*HYPRE_StructSolver* solver, HYPRE_Int *num_pre_relax)

HYPRE_Int **HYPRE_StructPFMGSetNumPostRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_post_relax)

(Optional) Set number of relaxation sweeps after coarse-grid correction.

HYPRE_Int **HYPRE_StructPFMGGetNumPostRelax**(*HYPRE_StructSolver* solver, HYPRE_Int *num_post_relax)

HYPRE_Int **HYPRE_StructPFMGSetSkipRelax**(*HYPRE_StructSolver* solver, HYPRE_Int skip_relax)

(Optional) Skip relaxation on certain grids for isotropic problems. This can greatly improve efficiency by eliminating unnecessary relaxations when the underlying problem is isotropic.

HYPRE_Int **HYPRE_StructPFMGGetSkipRelax**(*HYPRE_StructSolver* solver, HYPRE_Int *skip_relax)

HYPRE_Int **HYPRE_StructPFMGSetDxyz**(*HYPRE_StructSolver* solver, HYPRE_Real *dxyz)

HYPRE_Int **HYPRE_StructPFMGSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_StructPFMGGetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int *logging)

HYPRE_Int **HYPRE_StructPFMGSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int print_level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_StructPFMGGetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int *print_level)

HYPRE_Int **HYPRE_StructPFMGGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_StructPFMGGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

Struct SMG Solver

SMG is a semicoarsening multigrid solver that uses plane smoothing (in 3D). The plane smoother calls a 2D SMG algorithm with line smoothing, and the line smoother is cyclic reduction (1D SMG). For periodic problems, the grid size in periodic dimensions currently must be a power-of-two.

HYPRE_Int **HYPRE_StructSMGCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructSMGDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructSMGSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_StructSMGSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Solve the system.

HYPRE_Int **HYPRE_StructSMGSetMemoryUse**(*HYPRE_StructSolver* solver, HYPRE_Int memory_use)

HYPRE_Int **HYPRE_StructSMGGetMemoryUse**(*HYPRE_StructSolver* solver, HYPRE_Int *memory_use)

HYPRE_Int **HYPRE_StructSMGSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_StructSMGGetTol**(*HYPRE_StructSolver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_StructSMGSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_StructSMGGetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_StructSMGSetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_StructSMGGetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int *rel_change)

HYPRE_Int **HYPRE_StructSMGSetZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_StructSMGGetZeroGuess**(*HYPRE_StructSolver* solver, HYPRE_Int *zeroguess)

HYPRE_Int **HYPRE_StructSMGSetNonZeroGuess**(*HYPRE_StructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_StructSMGSetNumPreRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_pre_relax)

(Optional) Set number of relaxation sweeps before coarse-grid correction.

HYPRE_Int **HYPRE_StructSMGGetNumPreRelax**(*HYPRE_StructSolver* solver, HYPRE_Int *num_pre_relax)

HYPRE_Int **HYPRE_StructSMGSetNumPostRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_post_relax)

(Optional) Set number of relaxation sweeps after coarse-grid correction.

HYPRE_Int **HYPRE_StructSMGGetNumPostRelax**(*HYPRE_StructSolver* solver, HYPRE_Int *num_post_relax)

HYPRE_Int **HYPRE_StructSMGSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_StructSMGGetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int *logging)

HYPRE_Int **HYPRE_StructSMGSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int print_level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_StructSMGGetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int *print_level)

HYPRE_Int **HYPRE_StructSMGGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_StructSMGGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

Struct CycRed Solver

CycRed is a cyclic reduction solver that simultaneously solves a collection of 1D tridiagonal systems embedded in a d-dimensional grid.

HYPRE_Int **HYPRE_StructCycRedCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructCycRedDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructCycRedSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_StructCycRedSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Solve the system.

HYPRE_Int **HYPRE_StructCycRedSetTDim**(*HYPRE_StructSolver* solver, HYPRE_Int tdim)

(Optional) Set the dimension number for the embedded 1D tridiagonal systems. The default is *tdim* = 0.

HYPRE_Int **HYPRE_StructCycRedSetBase**(*HYPRE_StructSolver* solver, HYPRE_Int ndim, HYPRE_Int *base_index, HYPRE_Int *base_stride)

(Optional) Set the base index and stride for the embedded 1D systems. The stride must be equal one in the dimension corresponding to the 1D systems (see *HYPRE_StructCycRedSetTDim*).

Struct PCG Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_StructPCGCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructPCGDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructPCGSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructPCGSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructPCGSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructPCGSetAbsoluteTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructPCGSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructPCGSetTwoNorm**(*HYPRE_StructSolver* solver, HYPRE_Int two_norm)

HYPRE_Int **HYPRE_StructPCGSetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int rel_change)

HYPRE_Int **HYPRE_StructPCGSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

HYPRE_Int **HYPRE_StructPCGSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructPCGSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_StructPCGGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructPCGGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_StructPCGGetResidual**(*HYPRE_StructSolver* solver, void **residual)

HYPRE_Int **HYPRE_StructDiagScaleSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector y, HYPRE_StructVector x)

Setup routine for diagonal preconditioning.

HYPRE_Int **HYPRE_StructDiagScale**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* HA, HYPRE_StructVector Hy, HYPRE_StructVector Hx)

Solve routine for diagonal preconditioning.

Struct GMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_StructGMRESCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructGMRESDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructGMRESSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructGMRESSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructGMRESSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructGMRESSetAbsoluteTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructGMRESSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructGMRESSetKDim**(*HYPRE_StructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_StructGMRESSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

HYPRE_Int **HYPRE_StructGMRESSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructGMRESSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_StructGMRESGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructGMRESGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_StructGMRESGetResidual**(*HYPRE_StructSolver* solver, void **residual)

Struct FlexGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_StructFlexGMRESCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructFlexGMRESDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructFlexGMRESSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructFlexGMRESSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructFlexGMRESSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructFlexGMRESSetAbsoluteTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructFlexGMRESSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructFlexGMRESSetKDim**(*HYPRE_StructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_StructFlexGMRESSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

HYPRE_Int **HYPRE_StructFlexGMRESSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructFlexGMRESSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_StructFlexGMRESGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructFlexGMRESGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_StructFlexGMRESGetResidual**(*HYPRE_StructSolver* solver, void **residual)

HYPRE_Int **HYPRE_StructFlexGMRESSetModifyPC**(*HYPRE_StructSolver* solver, *HYPRE_PtrToModifyPCFcn* modify_pc)

Struct LGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_StructLGMRESCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructLGMRESDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructLGMRESSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructLGMRESSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructLGMRESSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructLGMRESSetAbsoluteTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructLGMRESSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructLGMRESSetKDim**(*HYPRE_StructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_StructLGMRESSetAugDim**(*HYPRE_StructSolver* solver, HYPRE_Int aug_dim)

HYPRE_Int **HYPRE_StructLGMRESSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

HYPRE_Int **HYPRE_StructLGMRESSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructLGMRESSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_StructLGMRESGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructLGMRESGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_StructLGMRESGetResidual**(*HYPRE_StructSolver* solver, void **residual)

Struct BiCGSTAB Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_StructBiCGSTABCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructBiCGSTABDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructBiCGSTABSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructBiCGSTABSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructBiCGSTABSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructBiCGSTABSetAbsoluteTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructBiCGSTABSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructBiCGSTABSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

HYPRE_Int **HYPRE_StructBiCGSTABSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructBiCGSTABSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_StructBiCGSTABGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructBiCGSTABGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_StructBiCGSTABGetResidual**(*HYPRE_StructSolver* solver, void **residual)

Struct Hybrid Solver

HYPRE_Int **HYPRE_StructHybridCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_StructHybridDestroy**(*HYPRE_StructSolver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_StructHybridSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_StructHybridSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

Solve the system.

HYPRE_Int **HYPRE_StructHybridSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_StructHybridSetConvergenceTol**(*HYPRE_StructSolver* solver, HYPRE_Real cf_tol)

(Optional) Set an accepted convergence tolerance for diagonal scaling (DS). The solver will switch preconditioners if the convergence of DS is slower than *cf_tol*.

HYPRE_Int **HYPRE_StructHybridSetDSCGMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int ds_max_its)

(Optional) Set maximum number of iterations for diagonal scaling (DS). The solver will switch preconditioners if DS reaches *ds_max_its*.

HYPRE_Int **HYPRE_StructHybridSetPCGMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int pre_max_its)

(Optional) Set maximum number of iterations for general preconditioner (PRE). The solver will stop if PRE reaches *pre_max_its*.

HYPRE_Int **HYPRE_StructHybridSetTwoNorm**(*HYPRE_StructSolver* solver, HYPRE_Int two_norm)

(Optional) Use the two-norm in stopping criteria.

HYPRE_Int **HYPRE_StructHybridSetStopCrit**(*HYPRE_StructSolver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_StructHybridSetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_StructHybridSetSolverType**(*HYPRE_StructSolver* solver, HYPRE_Int solver_type)

(Optional) Set the type of Krylov solver to use.

Current krylov methods set by *solver_type* are:

- 0 : PCG (default)

- 1 : GMRES
- 2 : BiCGSTAB

HYPRE_Int **HYPRE_StructHybridSetRecomputeResidual**(*HYPRE_StructSolver* solver, HYPRE_Int recompute_residual)

(Optional) Set recompute residual (don't rely on 3-term recurrence).

HYPRE_Int **HYPRE_StructHybridGetRecomputeResidual**(*HYPRE_StructSolver* solver, HYPRE_Int *recompute_residual)

(Optional) Get recompute residual option.

HYPRE_Int **HYPRE_StructHybridSetRecomputeResidualP**(*HYPRE_StructSolver* solver, HYPRE_Int recompute_residual_p)

(Optional) Set recompute residual period (don't rely on 3-term recurrence).

Recomputes residual after every specified number of iterations.

HYPRE_Int **HYPRE_StructHybridGetRecomputeResidualP**(*HYPRE_StructSolver* solver, HYPRE_Int *recompute_residual_p)

(Optional) Get recompute residual period option.

HYPRE_Int **HYPRE_StructHybridSetKDim**(*HYPRE_StructSolver* solver, HYPRE_Int k_dim)

(Optional) Set the maximum size of the Krylov space when using GMRES.

HYPRE_Int **HYPRE_StructHybridSetPrecond**(*HYPRE_StructSolver* solver, *HYPRE_PtrToStructSolverFcn* precond, *HYPRE_PtrToStructSolverFcn* precond_setup, *HYPRE_StructSolver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_StructHybridSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_StructHybridSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int print_level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_StructHybridGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_its)

Return the number of iterations taken.

HYPRE_Int **HYPRE_StructHybridGetDSCGNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *ds_num_its)

Return the number of diagonal scaling iterations taken.

HYPRE_Int **HYPRE_StructHybridGetPCGNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *pre_num_its)

Return the number of general preconditioning iterations taken.

HYPRE_Int **HYPRE_StructHybridGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_StructHybridSetPCGAbsoluteTolFactor**(*HYPRE_StructSolver* solver, HYPRE_Real pcg_atolf)

Struct LOBPCG Eigensolver

These routines should be used in conjunction with the generic interface in *Eigensolvers*.

HYPRE_Int **HYPRE_StructSetupInterpreter**(mv_InterfaceInterpreter *i)

Load interface interpreter. Vector part loaded with hypre_StructKrylov functions and multivector part loaded with mv_TempMultiVector functions.

HYPRE_Int **HYPRE_StructSetupMatvec**(HYPRE_MatvecFunctions *mv)

Load Matvec interpreter with hypre_StructKrylov functions.

Functions

HYPRE_Int **HYPRE_StructSparseMSGCreate**(MPI_Comm comm, *HYPRE_StructSolver* *solver)

HYPRE_Int **HYPRE_StructSparseMSGDestroy**(*HYPRE_StructSolver* solver)

HYPRE_Int **HYPRE_StructSparseMSGSetup**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructSparseMSGSolve**(*HYPRE_StructSolver* solver, *HYPRE_StructMatrix* A, HYPRE_StructVector b, HYPRE_StructVector x)

HYPRE_Int **HYPRE_StructSparseMSGSetTol**(*HYPRE_StructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_StructSparseMSGSetMaxIter**(*HYPRE_StructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_StructSparseMSGSetJump**(*HYPRE_StructSolver* solver, HYPRE_Int jump)

HYPRE_Int **HYPRE_StructSparseMSGSetRelChange**(*HYPRE_StructSolver* solver, HYPRE_Int rel_change)

HYPRE_Int **HYPRE_StructSparseMSGSetZeroGuess**(*HYPRE_StructSolver* solver)

HYPRE_Int **HYPRE_StructSparseMSGSetNonZeroGuess**(*HYPRE_StructSolver* solver)

HYPRE_Int **HYPRE_StructSparseMSGSetRelaxType**(*HYPRE_StructSolver* solver, HYPRE_Int relax_type)

HYPRE_Int **HYPRE_StructSparseMSGSetJacobiWeight**(*HYPRE_StructSolver* solver, HYPRE_Real weight)

HYPRE_Int **HYPRE_StructSparseMSGSetNumPreRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_pre_relax)

HYPRE_Int **HYPRE_StructSparseMSGSetNumPostRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_post_relax)

HYPRE_Int **HYPRE_StructSparseMSGSetNumFineRelax**(*HYPRE_StructSolver* solver, HYPRE_Int num_fine_relax)

HYPRE_Int **HYPRE_StructSparseMSGSetLogging**(*HYPRE_StructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_StructSparseMSGSetPrintLevel**(*HYPRE_StructSolver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_StructSparseMSGGetNumIterations**(*HYPRE_StructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_StructSparseMSGGetFinalRelativeResidualNorm**(*HYPRE_StructSolver* solver, HYPRE_Real *norm)

8.5 SStruct Solvers

group **SStructSolvers**

These solvers use matrix/vector storage schemes that are taylored to semi-structured grid problems.

@memo Linear solvers for semi-structured grids

SStruct Solvers

typedef struct hypre_SStructSolver_struct ***HYPRE_SStructSolver**

The solver object.

typedef HYPRE_Int (***HYPRE_PtrToSStructSolverFcn**)(*HYPRE_SStructSolver*, *HYPRE_SStructMatrix*, *HYPRE_SStructVector*, *HYPRE_SStructVector*)

typedef struct hypre_Solver_struct ***HYPRE_Solver**

typedef HYPRE_Int (***HYPRE_PtrToModifyPCFcn**)(*HYPRE_Solver*, HYPRE_Int, HYPRE_Real)

HYPRE_MODIFYPC

HYPRE_SOLVER_STRUCT

SStruct SysPFMG Solver

SysPFMG is a semicoarsening multigrid solver similar to PFMG, but for systems of PDEs. For periodic problems, users should try to set the grid size in periodic dimensions to be as close to a power-of-two as possible (for more details, see Struct PFMG Solver).

HYPRE_Int **HYPRE_SStructSysPFMGCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructSysPFMGDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructSysPFMGSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_SStructSysPFMGSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Solve the system.

HYPRE_Int **HYPRE_SStructSysPFMGSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_SStructSysPFMGSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_SStructSysPFMGSetRelChange**(*HYPRE_SStructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_SStructSysPFMGSetZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_SStructSysPFMGSetNonZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_SStructSysPFMGSetRelaxType**(*HYPRE_SStructSolver* solver, HYPRE_Int relax_type)

(Optional) Set relaxation type.

Current relaxation methods set by *relax_type* are:

- 0 : Jacobi
- 1 : Weighted Jacobi (default)
- 2 : Red/Black Gauss-Seidel (symmetric: RB pre-relaxation, BR post-relaxation)

HYPRE_Int **HYPRE_SStructSysPFMGSetJacobiWeight**(*HYPRE_SStructSolver* solver, HYPRE_Real weight)

(Optional) Set Jacobi Weight.

HYPRE_Int **HYPRE_SStructSysPFMGSetNumPreRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_pre_relax)

(Optional) Set number of relaxation sweeps before coarse-grid correction.

HYPRE_Int **HYPRE_SStructSysPFMGSetNumPostRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_post_relax)

(Optional) Set number of relaxation sweeps after coarse-grid correction.

HYPRE_Int **HYPRE_SStructSysPFMGSetSkipRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int skip_relax)

(Optional) Skip relaxation on certain grids for isotropic problems. This can greatly improve efficiency by eliminating unnecessary relaxations when the underlying problem is isotropic.

HYPRE_Int **HYPRE_SStructSysPFMGSetDxyz**(*HYPRE_SStructSolver* solver, HYPRE_Real *dxyz)

HYPRE_Int **HYPRE_SStructSysPFMGSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_SStructSysPFMGSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int print_level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_SStructSysPFMGGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_SStructSysPFMGGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

SStruct Split Solver

HYPRE_Int **HYPRE_SStructSplitCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructSplitDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructSplitSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_SStructSplitSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Solve the system.

HYPRE_Int **HYPRE_SStructSplitSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_SStructSplitSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_SStructSplitSetZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_SStructSplitSetNonZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_SStructSplitSetStructSolver**(*HYPRE_SStructSolver* solver, HYPRE_Int ssolver)

(Optional) Set up the type of diagonal struct solver. Either *ssolver* is set to *HYPRE_SMG* or *HYPRE_PFMG*.

HYPRE_Int **HYPRE_SStructSplitGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_SStructSplitGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_PFMG

HYPRE_SMG

HYPRE_Jacobi

SStruct FAC Solver

HYPRE_Int **HYPRE_SStructFACCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructFACDestroy2**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructFACAMR_RAP**(*HYPRE_SStructMatrix* A, HYPRE_Int (*rfactors)[HYPRE_MAXDIM], *HYPRE_SStructMatrix* *fac_A)

Re-distribute the composite matrix so that the amr hierachy is approximately nested. Coarse underlying operators are also formed.

HYPRE_Int **HYPRE_SStructFACSetup2**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Set up the FAC solver structure .

HYPRE_Int **HYPRE_SStructFACSolve3**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Solve the system.

HYPRE_Int **HYPRE_SStructFACSetPLevels**(*HYPRE_SStructSolver* solver, HYPRE_Int nparts, HYPRE_Int *plevels)

Set up amr structure

HYPRE_Int **HYPRE_SStructFACSetPRefinements**(*HYPRE_SStructSolver* solver, HYPRE_Int nparts, HYPRE_Int (*rfactors)[HYPRE_MAXDIM])

Set up amr refinement factors

HYPRE_Int **HYPRE_SStructFACZeroCFSten**(*HYPRE_SStructMatrix* A, *HYPRE_SStructGrid* grid, HYPRE_Int part, HYPRE_Int rfactors[HYPRE_MAXDIM])

(Optional, but user must make sure that they do this function otherwise.) Zero off the coarse level stencils reaching into a fine level grid.

HYPRE_Int **HYPRE_SStructFACZeroFCSten**(*HYPRE_SStructMatrix* A, *HYPRE_SStructGrid* grid, HYPRE_Int part)

(Optional, but user must make sure that they do this function otherwise.) Zero off the fine level stencils reaching into a coarse level grid.

HYPRE_Int **HYPRE_SStructFACZeroAMRMatrixData**(*HYPRE_SStructMatrix* A, HYPRE_Int part_crse, HYPRE_Int rfactors[HYPRE_MAXDIM])

(Optional, but user must make sure that they do this function otherwise.) Places the identity in the coarse grid matrix underlying the fine patches. Required between each pair of amr levels.

HYPRE_Int **HYPRE_SStructFACZeroAMRVectorData**(*HYPRE_SStructVector* b, HYPRE_Int *plevels, HYPRE_Int (*rfactors)[HYPRE_MAXDIM])

(Optional, but user must make sure that they do this function otherwise.) Places zeros in the coarse grid vector underlying the fine patches. Required between each pair of amr levels.

HYPRE_Int **HYPRE_SStructFACSetMaxLevels**(*HYPRE_SStructSolver* solver, HYPRE_Int max_levels)

(Optional) Set maximum number of FAC levels.

HYPRE_Int **HYPRE_SStructFACSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_SStructFACSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_SStructFACSetRelChange**(*HYPRE_SStructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_SStructFACSetZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a zero initial guess. This allows the solver to cut corners in the case where a zero initial guess is needed (e.g., for preconditioning) to reduce compuational cost.

HYPRE_Int **HYPRE_SStructFACSetNonZeroGuess**(*HYPRE_SStructSolver* solver)

(Optional) Use a nonzero initial guess. This is the default behavior, but this routine allows the user to switch back after using *SetZeroGuess*.

HYPRE_Int **HYPRE_SStructFACSetRelaxType**(*HYPRE_SStructSolver* solver, HYPRE_Int relax_type)

(Optional) Set relaxation type. See *HYPRE_SStructSysPFMGSetRelaxType* for appropriate values of *relax_type*.

HYPRE_Int **HYPRE_SStructFACSetJacobiWeight**(*HYPRE_SStructSolver* solver, HYPRE_Real weight)

(Optional) Set Jacobi weight if weighted Jacobi is used.

HYPRE_Int **HYPRE_SStructFACSetNumPreRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_pre_relax)

(Optional) Set number of relaxation sweeps before coarse-grid correction.

HYPRE_Int **HYPRE_SStructFACSetNumPostRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_post_relax)

(Optional) Set number of relaxation sweeps after coarse-grid correction.

HYPRE_Int **HYPRE_SStructFACSetCoarseSolverType**(*HYPRE_SStructSolver* solver, HYPRE_Int csolver_type)

(Optional) Set coarsest solver type.

Current solver types set by *csolver_type* are:

- 1 : SysPFMG-PCG (default)
- 2 : SysPFMG

HYPRE_Int **HYPRE_SStructFACSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_SStructFACGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_SStructFACGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

SStruct Maxwell Solver

HYPRE_Int **HYPRE_SStructMaxwellCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructMaxwellDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructMaxwellSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_SStructMaxwellSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Solve the system. Full coupling of the augmented system used throughout the multigrid hierarchy.

HYPRE_Int **HYPRE_SStructMaxwellSolve2**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

Solve the system. Full coupling of the augmented system used only on the finest level, i.e., the node and edge multigrid cycles are coupled only on the finest level.

HYPRE_Int **HYPRE_SStructMaxwellSetGrad**(*HYPRE_SStructSolver* solver, HYPRE_ParCSRMatrix T)

Sets the gradient operator in the Maxwell solver.

HYPRE_Int **HYPRE_SStructMaxwellSetRfactors**(*HYPRE_SStructSolver* solver, HYPRE_Int rfactors[HYPRE_MAXDIM])

Sets the coarsening factor.

HYPRE_Int **HYPRE_SStructMaxwellPhysBdy**(*HYPRE_SStructGrid* *grid_l, HYPRE_Int num_levels, HYPRE_Int rfactors[HYPRE_MAXDIM], HYPRE_Int ***BdryRanks_ptr, HYPRE_Int **BdryRanksCnt_ptr)

Finds the physical boundary row ranks on all levels.

HYPRE_Int **HYPRE_SStructMaxwellEliminateRowsCols**(HYPRE_ParCSRMatrix parA, HYPRE_Int nrows, HYPRE_Int *rows)

Eliminates the rows and cols corresponding to the physical boundary in a parcsr matrix.

HYPRE_Int **HYPRE_SStructMaxwellZeroVector**(HYPRE_ParVector b, HYPRE_Int *rows, HYPRE_Int nrows)

Zeros the rows corresponding to the physical boundary in a par vector.

HYPRE_Int **HYPRE_SStructMaxwellSetSetConstantCoef**(*HYPRE_SStructSolver* solver, HYPRE_Int flag)

(Optional) Set the constant coefficient flag- Nedelec interpolation used.

HYPRE_Int **HYPRE_SStructMaxwellGrad**(*HYPRE_SStructGrid* grid, HYPRE_ParCSRMatrix *T)

(Optional) Creates a gradient matrix from the grid. This presupposes a particular orientation of the edge elements.

HYPRE_Int **HYPRE_SStructMaxwellSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_SStructMaxwellSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_SStructMaxwellSetRelChange**(*HYPRE_SStructSolver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_SStructMaxwellSetNumPreRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_pre_relax)

(Optional) Set number of relaxation sweeps before coarse-grid correction.

HYPRE_Int **HYPRE_SStructMaxwellSetNumPostRelax**(*HYPRE_SStructSolver* solver, HYPRE_Int num_post_relax)

(Optional) Set number of relaxation sweeps after coarse-grid correction.

HYPRE_Int **HYPRE_SStructMaxwellSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_SStructMaxwellGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_SStructMaxwellGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

SStruct PCG Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_SStructPCGCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructPCGDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructPCGSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructPCGSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructPCGSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructPCGSetAbsoluteTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructPCGSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_SStructPCGSetTwoNorm**(*HYPRE_SStructSolver* solver, HYPRE_Int two_norm)

HYPRE_Int **HYPRE_SStructPCGSetRelChange**(*HYPRE_SStructSolver* solver, HYPRE_Int rel_change)

HYPRE_Int **HYPRE_SStructPCGSetPrecond**(*HYPRE_SStructSolver* solver, *HYPRE_PtrToSStructSolverFcn* precond, *HYPRE_PtrToSStructSolverFcn* precond_setup, void *precond_solver)

HYPRE_Int **HYPRE_SStructPCGSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_SStructPCGSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_SStructPCGGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_SStructPCGGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_SStructPCGGetResidual**(*HYPRE_SStructSolver* solver, void **residual)

HYPRE_Int **HYPRE_SStructDiagScaleSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* y, *HYPRE_SStructVector* x)

Setup routine for diagonal preconditioning.

HYPRE_Int **HYPRE_SStructDiagScale**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* y, *HYPRE_SStructVector* x)

Solve routine for diagonal preconditioning.

SStruct GMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_SStructGMRESCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructGMRESDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructGMRESSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructGMRESSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructGMRESSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructGMRESSetAbsoluteTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructGMRESSetMinIter**(*HYPRE_SStructSolver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_SStructGMRESSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_SStructGMRESSetKDim**(*HYPRE_SStructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_SStructGMRESSetStopCrit**(*HYPRE_SStructSolver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_SStructGMRESSetPrecond**(*HYPRE_SStructSolver* solver,
HYPRE_PtrToSStructSolverFcn precond,
HYPRE_PtrToSStructSolverFcn precond_setup, void
*precond_solver)

HYPRE_Int **HYPRE_SStructGMRESSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_SStructGMRESSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_SStructGMRESGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int
*num_iterations)

HYPRE_Int **HYPRE_SStructGMRESGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver,
HYPRE_Real *norm)

HYPRE_Int **HYPRE_SStructGMRESGetResidual**(*HYPRE_SStructSolver* solver, void **residual)

SStruct FlexGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_SStructFlexGMRESCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructFlexGMRESDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructFlexGMRESSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A,
HYPRE_SStructVector b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructFlexGMRESSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A,
HYPRE_SStructVector b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructFlexGMRESSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructFlexGMRESSetAbsoluteTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructFlexGMRESSetMinIter**(*HYPRE_SStructSolver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_SStructFlexGMRESSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_SStructFlexGMRESSetKDim**(*HYPRE_SStructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_SStructFlexGMRESSetPrecond**(*HYPRE_SStructSolver* solver,
HYPRE_PtrToSStructSolverFcn precond,
HYPRE_PtrToSStructSolverFcn precond_setup, void
*precond_solver)

HYPRE_Int **HYPRE_SStructFlexGMRESSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_SStructFlexGMRESSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_SStructFlexGMRESGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_SStructFlexGMRESGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_SStructFlexGMRESGetResidual**(*HYPRE_SStructSolver* solver, void **residual)

HYPRE_Int **HYPRE_SStructFlexGMRESSetModifyPC**(*HYPRE_SStructSolver* solver, *HYPRE_PtrToModifyPCFcn* modify_pc)

SStruct LGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_SStructLGMRESCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructLGMRESDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructLGMRESSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructLGMRESSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructLGMRESSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructLGMRESSetAbsoluteTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructLGMRESSetMinIter**(*HYPRE_SStructSolver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_SStructLGMRESSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_SStructLGMRESSetKDim**(*HYPRE_SStructSolver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_SStructLGMRESSetAugDim**(*HYPRE_SStructSolver* solver, HYPRE_Int aug_dim)

HYPRE_Int **HYPRE_SStructLGMRESSetPrecond**(*HYPRE_SStructSolver* solver, *HYPRE_PtrToSStructSolverFcn* precond, *HYPRE_PtrToSStructSolverFcn* precond_setup, void *precond_solver)

HYPRE_Int **HYPRE_SStructLGMRESSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_SStructLGMRESSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_SStructLGMRESGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_SStructLGMRESGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_SStructLGMRESGetResidual**(*HYPRE_SStructSolver* solver, void **residual)

SStruct BiCGSTAB Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_SStructBiCGSTABCreate**(MPI_Comm comm, *HYPRE_SStructSolver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_SStructBiCGSTABDestroy**(*HYPRE_SStructSolver* solver)

Destroy a solver object. An object should be explicitly destroyed using this destructor when the user's code no longer needs direct access to it. Once destroyed, the object must not be referenced again. Note that the object may not be deallocated at the completion of this call, since there may be internal package references to the object. The object will then be destroyed when all internal reference counts go to zero.

HYPRE_Int **HYPRE_SStructBiCGSTABSetup**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructBiCGSTABSolve**(*HYPRE_SStructSolver* solver, *HYPRE_SStructMatrix* A, *HYPRE_SStructVector* b, *HYPRE_SStructVector* x)

HYPRE_Int **HYPRE_SStructBiCGSTABSetTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructBiCGSTABSetAbsoluteTol**(*HYPRE_SStructSolver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_SStructBiCGSTABSetMinIter**(*HYPRE_SStructSolver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_SStructBiCGSTABSetMaxIter**(*HYPRE_SStructSolver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_SStructBiCGSTABSetStopCrit**(*HYPRE_SStructSolver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_SStructBiCGSTABSetPrecond**(*HYPRE_SStructSolver* solver, *HYPRE_PtrToSStructSolverFcn* precond, *HYPRE_PtrToSStructSolverFcn* precond_setup, void *precond_solver)

HYPRE_Int **HYPRE_SStructBiCGSTABSetLogging**(*HYPRE_SStructSolver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_SStructBiCGSTABSetPrintLevel**(*HYPRE_SStructSolver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_SStructBiCGSTABGetNumIterations**(*HYPRE_SStructSolver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_SStructBiCGSTABGetFinalRelativeResidualNorm**(*HYPRE_SStructSolver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_SStructBiCGSTABGetResidual**(*HYPRE_SStructSolver* solver, void **residual)

SStruct LOBPCG Eigensolver

These routines should be used in conjunction with the generic interface in *Eigensolvers*.

HYPRE_Int **HYPRE_SStructSetupInterpreter**(mv_InterfaceInterpreter *i)

Load interface interpreter. Vector part loaded with hypre_SStructKrylov functions and multivector part loaded with mv_TempMultiVector functions.

HYPRE_Int **HYPRE_SStructSetupMatvec**(HYPRE_MatvecFunctions *mv)

Load Matvec interpreter with hypre_SStructKrylov functions.

8.6 ParCSR Solvers

group **ParCSRSolvers**

These solvers use matrix/vector storage schemes that are taylored for general sparse matrix systems.

@memo Linear solvers for sparse matrix systems

ParCSR Solvers

typedef struct hypre_Solver_struct ***HYPRE_Solver**

typedef HYPRE_Int (***HYPRE_PtrToParSolverFcn**)(*HYPRE_Solver*, HYPRE_ParCSRMatrix, HYPRE_ParVector, HYPRE_ParVector)

typedef HYPRE_Int (***HYPRE_PtrToModifyPCFcn**)(*HYPRE_Solver*, HYPRE_Int, HYPRE_Real)

HYPRE_SOLVER_STRUCT

The solver object.

HYPRE_MODIFYPC

ParCSR BoomerAMG Solver and Preconditioner

Parallel unstructured algebraic multigrid solver and preconditioner

HYPRE_Int **HYPRE_BoomerAMGCreate**(*HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_BoomerAMGDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_BoomerAMGSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the BoomerAMG solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.

- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_BoomerAMGSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply AMG as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_BoomerAMGSolveT**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the transpose system $A^T x = b$ or apply AMG as a preconditioner to the transpose system . Note that this function should only be used when preconditioning CGNR with BoomerAMG. It can only be used with Jacobi smoothing (relax_type 0 or 7) and without CF smoothing, i.e relax_order needs to be set to 0. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_BoomerAMGSetOldDefault**(*HYPRE_Solver* solver)

Recovers old default for coarsening and interpolation, i.e Falgout coarsening and untruncated modified classical interpolation. This option might be preferred for 2 dimensional problems.

HYPRE_Int **HYPRE_BoomerAMGGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

Returns the residual.

HYPRE_Int **HYPRE_BoomerAMGGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Returns the number of iterations taken.

HYPRE_Int **HYPRE_BoomerAMGGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *rel_resid_norm)

Returns the norm of the final relative residual.

HYPRE_Int **HYPRE_BoomerAMGSetNumFunctions**(*HYPRE_Solver* solver, HYPRE_Int num_functions)

(Optional) Sets the size of the system of PDEs, if using the systems version. The default is 1, i.e. a scalar system.

HYPRE_Int **HYPRE_BoomerAMGSetDofFunc**(*HYPRE_Solver* solver, HYPRE_Int *dof_func)

(Optional) Sets the mapping that assigns the function to each variable, if using the systems version. If no assignment is made and the number of functions is k > 1, the mapping generated is (0,1,...,k-1,0,1,...,k-1,...).

HYPRE_Int **HYPRE_BoomerAMGSetConvergeType**(*HYPRE_Solver* solver, HYPRE_Int type)

(Optional) Set the type convergence checking 0: (default) norm(r)/norm(b), or norm(r) when b == 0 1: nomr(r) / norm(r_0)

HYPRE_Int **HYPRE_BoomerAMGSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance, if BoomerAMG is used as a solver. If it is used as a preconditioner, it should be set to 0. The default is 1.e-6.

HYPRE_Int **HYPRE_BoomerAMGSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Sets maximum number of iterations, if BoomerAMG is used as a solver. If it is used as a preconditioner, it should be set to 1. The default is 20.

HYPRE_Int **HYPRE_BoomerAMGSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

(Optional)

HYPRE_Int **HYPRE_BoomerAMGSetMaxCoarseSize**(*HYPRE_Solver* solver, HYPRE_Int max_coarse_size)

(Optional) Sets maximum size of coarsest grid. The default is 9.

HYPRE_Int **HYPRE_BoomerAMGSetMinCoarseSize**(*HYPRE_Solver* solver, HYPRE_Int min_coarse_size)

(Optional) Sets minimum size of coarsest grid. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetMaxLevels**(*HYPRE_Solver* solver, HYPRE_Int max_levels)

(Optional) Sets maximum number of multigrid levels. The default is 25.

HYPRE_Int **HYPRE_BoomerAMGSetCoarsenCutFactor**(*HYPRE_Solver* solver, HYPRE_Int coarsen_cut_factor)

(Optional) Sets cut factor for choosing isolated points during coarsening according to the rows' density. The default is 0. If nnzrow > coarsen_cut_factor*avg_nnzrow, where avg_nnzrow is the average number of nonzeros per row of the global matrix, holds for a given row, it is set as fine, and interpolation weights are not computed.

HYPRE_Int **HYPRE_BoomerAMGSetStrongThreshold**(*HYPRE_Solver* solver, HYPRE_Real strong_threshold)

(Optional) Sets AMG strength threshold. The default is 0.25. For 2D Laplace operators, 0.25 is a good value, for 3D Laplace operators, 0.5 or 0.6 is a better value. For elasticity problems, a large strength threshold, such as 0.9, is often better.

HYPRE_Int **HYPRE_BoomerAMGSetStrongThresholdR**(*HYPRE_Solver* solver, HYPRE_Real strong_threshold)

(Optional) The strong threshold for R is strong connections used in building an approximate ideal restriction. Default value is 0.25.

HYPRE_Int **HYPRE_BoomerAMGSetFilterThresholdR**(*HYPRE_Solver* solver, HYPRE_Real filter_threshold)

(Optional) The filter threshold for R is used to eliminate small entries of the approximate ideal restriction after building it. Default value is 0.0, which disables filtering.

HYPRE_Int **HYPRE_BoomerAMGSetSCommPkgSwitch**(*HYPRE_Solver* solver, HYPRE_Real S_commpkg_switch)

(Optional) Deprecated. This routine now has no effect.

HYPRE_Int **HYPRE_BoomerAMGSetMaxRowSum**(*HYPRE_Solver* solver, HYPRE_Real max_row_sum)

(Optional) Sets a parameter to modify the definition of strength for diagonal dominant portions of the matrix. The default is 0.9. If *max_row_sum* is 1, no checking for diagonally dominant rows is performed.

HYPRE_Int **HYPRE_BoomerAMGSetCoarsenType**(*HYPRE_Solver* solver, HYPRE_Int coarsen_type)

(Optional) Defines which parallel coarsening algorithm is used. There are the following options for *coarsen_type:*

- 0 : CLJP-coarsening (a parallel coarsening algorithm using independent sets.
- 1 : classical Ruge-Stueben coarsening on each processor, no boundary treatment (not recommended!)
- 3 : classical Ruge-Stueben coarsening on each processor, followed by a third pass, which adds coarse points on the boundaries
- 6 : Falgout coarsening (uses 1 first, followed by CLJP using the interior coarse points generated by 1 as its first independent set)
- 7 : CLJP-coarsening (using a fixed random vector, for debugging purposes only)
- 8 : PMIS-coarsening (a parallel coarsening algorithm using independent sets, generating lower complexities than CLJP, might also lead to slower convergence)
- 9 : PMIS-coarsening (using a fixed random vector, for debugging purposes only)
- 10 : HMIS-coarsening (uses one pass Ruge-Stueben on each processor independently, followed by PMIS using the interior C-points generated as its first independent set)
- 11 : one-pass Ruge-Stueben coarsening on each processor, no boundary treatment (not recommended!)
- 21 : CGC coarsening by M. Griebel, B. Metsch and A. Schweitzer
- 22 : CGC-E coarsening by M. Griebel, B. Metsch and A.Schweitzer

The default is 10.

HYPRE_Int **HYPRE_BoomerAMGSetNonGalerkinTol**(*HYPRE_Solver* solver, HYPRE_Real nongalerkin_tol)

(Optional) Defines the non-Galerkin drop-tolerance for sparsifying coarse grid operators and thus reducing communication. Value specified here is set on all levels. This routine should be used before HYPRE_BoomerAMGSetLevelNonGalerkinTol, which then can be used to change individual levels if desired

HYPRE_Int **HYPRE_BoomerAMGSetLevelNonGalerkinTol**(*HYPRE_Solver* solver, HYPRE_Real nongalerkin_tol, HYPRE_Int level)

(Optional) Defines the level specific non-Galerkin drop-tolerances for sparsifying coarse grid operators and thus reducing communication. A drop-tolerance of 0.0 means to skip doing non-Galerkin on that level. The maximum drop tolerance for a level is 1.0, although much smaller values such as 0.03 or 0.01 are recommended.

Note that if the user wants to set a specific tolerance on all levels, HYPRE_BooemrAMGSetNonGalerkinTol should be used. Individual levels can then be changed using this routine.

In general, it is safer to drop more aggressively on coarser levels. For instance, one could use 0.0 on the finest level, 0.01 on the second level and then using 0.05 on all remaining levels. The best way to achieve this is to set 0.05 on all levels with HYPRE_BoomerAMGSetNonGalerkinTol and then change the tolerance on level 0 to 0.0 and the tolerance on level 1 to 0.01 with HYPRE_BoomerAMGSetLevelNonGalerkinTol. Like many AMG parameters, these drop tolerances can be tuned. It is also common to delay the start of the non-Galerkin process further to a later level than level 1.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **nongalerkin_tol** – [IN] level specific drop tolerance
- **level** – [IN] level on which drop tolerance is used

HYPRE_Int **HYPRE_BoomerAMGSetNonGalerkTol**(*HYPRE_Solver* solver, HYPRE_Int nongalerk_num_tol, HYPRE_Real *nongalerk_tol)

(Optional) Defines the non-Galerkin drop-tolerance (old version)

HYPRE_Int **HYPRE_BoomerAMGSetMeasureType**(*HYPRE_Solver* solver, HYPRE_Int measure_type)

(Optional) Defines whether local or global measures are used.

HYPRE_Int **HYPRE_BoomerAMGSetAggNumLevels**(*HYPRE_Solver* solver, HYPRE_Int agg_num_levels)

(Optional) Defines the number of levels of aggressive coarsening. The default is 0, i.e. no aggressive coarsening.

HYPRE_Int **HYPRE_BoomerAMGSetNumPaths**(*HYPRE_Solver* solver, HYPRE_Int num_paths)

(Optional) Defines the degree of aggressive coarsening. The default is 1. Larger numbers lead to less aggressive coarsening.

HYPRE_Int **HYPRE_BoomerAMGSetCGCIts**(*HYPRE_Solver* solver, HYPRE_Int its)

(optional) Defines the number of pathes for CGC-coarsening.

HYPRE_Int **HYPRE_BoomerAMGSetNodal**(*HYPRE_Solver* solver, HYPRE_Int nodal)

(Optional) Sets whether to use the nodal systems coarsening. Should be used for linear systems generated from systems of PDEs. The default is 0 (unknown-based coarsening, only coarsens within same function). For the remaining options a nodal matrix is generated by applying a norm to the nodal blocks and applying the coarsening algorithm to this matrix.

- 1 : Frobenius norm
- 2 : sum of absolute values of elements in each block
- 3 : largest element in each block (not absolute value)
- 4 : row-sum norm
- 6 : sum of all values in each block

HYPRE_Int **HYPRE_BoomerAMGSetNodalDiag**(*HYPRE_Solver* solver, HYPRE_Int nodal_diag)

(Optional) Sets whether to give special treatment to diagonal elements in the nodal systems version. The default is 0. If set to 1, the diagonal entry is set to the negative sum of all off diagonal entries. If set to 2, the signs of all diagonal entries are inverted.

HYPRE_Int **HYPRE_BoomerAMGSetKeepSameSign**(*HYPRE_Solver* solver, HYPRE_Int keep_same_sign)

HYPRE_Int **HYPRE_BoomerAMGSetInterpType**(*HYPRE_Solver* solver, HYPRE_Int interp_type)

(Optional) Defines which parallel interpolation operator is used. There are the following options for *interp_type:*

- 0 : classical modified interpolation
- 1 : LS interpolation (for use with GSMG)
- 2 : classical modified interpolation for hyperbolic PDEs
- 3 : direct interpolation (with separation of weights) (also for GPU use)
- 4 : multipass interpolation
- 5 : multipass interpolation (with separation of weights)
- 6 : extended+i interpolation (also for GPU use)
- 7 : extended+i (if no common C neighbor) interpolation

- 8 : standard interpolation
- 9 : standard interpolation (with separation of weights)
- 10 : classical block interpolation (for use with nodal systems version only)
- 11 : classical block interpolation (for use with nodal systems version only) with diagonalized diagonal blocks
- 12 : FF interpolation
- 13 : FF1 interpolation
- 14 : extended interpolation (also for GPU use)
- 15 : interpolation with adaptive weights (GPU use only)
- 16 : extended interpolation in matrix-matrix form
- 17 : extended+i interpolation in matrix-matrix form
- 18 : extended+e interpolation in matrix-matrix form

The default is ext+i interpolation (interp_type 6) trunctated to at most 4 elements per row. (see HYPRE_BoomerAMGSetPMaxElmts).

HYPRE_Int **HYPRE_BoomerAMGSetTruncFactor**(*HYPRE_Solver* solver, HYPRE_Real trunc_factor)

(Optional) Defines a truncation factor for the interpolation. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetPMaxElmts**(*HYPRE_Solver* solver, HYPRE_Int P_max_elmts)

(Optional) Defines the maximal number of elements per row for the interpolation. The default is 4. To turn off truncation, it needs to be set to 0.

HYPRE_Int **HYPRE_BoomerAMGSetSepWeight**(*HYPRE_Solver* solver, HYPRE_Int sep_weight)

(Optional) Defines whether separation of weights is used when defining strength for standard interpolation or multipass interpolation. Default: 0, i.e. no separation of weights used.

HYPRE_Int **HYPRE_BoomerAMGSetAggInterpType**(*HYPRE_Solver* solver, HYPRE_Int agg_interp_type)

(Optional) Defines the interpolation used on levels of aggressive coarsening The default is 4, i.e. multipass interpolation. The following options exist:

- 1 : 2-stage extended+i interpolation
- 2 : 2-stage standard interpolation
- 3 : 2-stage extended interpolation
- 4 : multipass interpolation
- 5 : 2-stage extended interpolation in matrix-matrix form
- 6 : 2-stage extended+i interpolation in matrix-matrix form
- 7 : 2-stage extended+e interpolation in matrix-matrix form

HYPRE_Int **HYPRE_BoomerAMGSetAggTruncFactor**(*HYPRE_Solver* solver, HYPRE_Real agg_trunc_factor)

(Optional) Defines the truncation factor for the interpolation used for aggressive coarsening. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetAggP12TruncFactor**(*HYPRE_Solver* solver, HYPRE_Real agg_P12_trunc_factor)

(Optional) Defines the truncation factor for the matrices P1 and P2 which are used to build 2-stage interpolation. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetAggPMaxElmts**(*HYPRE_Solver* solver, HYPRE_Int agg_P_max_elmts)

(Optional) Defines the maximal number of elements per row for the interpolation used for aggressive coarsening. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetAggP12MaxElmts**(*HYPRE_Solver* solver, HYPRE_Int agg_P12_max_elmts)

(Optional) Defines the maximal number of elements per row for the matrices P1 and P2 which are used to build 2-stage interpolation. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetInterpVectors**(*HYPRE_Solver* solver, HYPRE_Int num_vectors, HYPRE_ParVector *interp_vectors)

(Optional) Allows the user to incorporate additional vectors into the interpolation for systems AMG, e.g. rigid body modes for linear elasticity problems. This can only be used in context with nodal coarsening and still requires the user to choose an interpolation.

HYPRE_Int **HYPRE_BoomerAMGSetInterpVecVariant**(*HYPRE_Solver* solver, HYPRE_Int var)

(Optional) Defines the interpolation variant used for HYPRE_BoomerAMGSetInterpVectors:

- 1 : GM approach 1
- 2 : GM approach 2 (to be preferred over 1)
- 3 : LN approach

HYPRE_Int **HYPRE_BoomerAMGSetInterpVecQMax**(*HYPRE_Solver* solver, HYPRE_Int q_max)

(Optional) Defines the maximal elements per row for Q, the additional columns added to the original interpolation matrix P, to reduce complexity. The default is no truncation.

HYPRE_Int **HYPRE_BoomerAMGSetInterpVecAbsQTrunc**(*HYPRE_Solver* solver, HYPRE_Real q_trunc)

(Optional) Defines a truncation factor for Q, the additional columns added to the original interpolation matrix P, to reduce complexity. The default is no truncation.

HYPRE_Int **HYPRE_BoomerAMGSetGSMG**(*HYPRE_Solver* solver, HYPRE_Int gsmg)

(Optional) Specifies the use of GSMG - geometrically smooth coarsening and interpolation. Currently any nonzero value for gsmg will lead to the use of GSMG. The default is 0, i.e. (GSMG is not used)

HYPRE_Int **HYPRE_BoomerAMGSetNumSamples**(*HYPRE_Solver* solver, HYPRE_Int num_samples)

(Optional) Defines the number of sample vectors used in GSMG or LS interpolation.

HYPRE_Int **HYPRE_BoomerAMGSetCycleType**(*HYPRE_Solver* solver, HYPRE_Int cycle_type)

(Optional) Defines the type of cycle. For a V-cycle, set *cycle_type* to 1, for a W-cycle set *cycle_type* to 2. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetFCycle**(*HYPRE_Solver* solver, HYPRE_Int fcycle)

(Optional) Specifies the use of Full multigrid cycle. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetAdditive**(*HYPRE_Solver* solver, HYPRE_Int addlvl)

(Optional) Defines use of an additive V(1,1)-cycle using the classical additive method starting at level 'addlvl'. The multiplicative approach is used on levels 0, ...'addlvl+1'. 'addlvl' needs to be > -1 for this to have an effect. Can only be used with weighted Jacobi and l1-Jacobi(default).

Can only be used when AMG is used as a preconditioner !!!

HYPRE_Int **HYPRE_BoomerAMGSetMultAdditive**(*HYPRE_Solver* solver, HYPRE_Int addlvl)

(Optional) Defines use of an additive V(1,1)-cycle using the mult-additive method starting at level 'addlvl'. The multiplicative approach is used on levels 0, ... 'addlvl+1'. 'addlvl' needs to be > -1 for this to have an effect. Can only be used with weighted Jacobi and l1-Jacobi(default).

Can only be used when AMG is used as a preconditioner !!!

HYPRE_Int **HYPRE_BoomerAMGSetSimple**(*HYPRE_Solver* solver, HYPRE_Int addlvl)

(Optional) Defines use of an additive V(1,1)-cycle using the simplified mult-additive method starting at level 'addlvl'. The multiplicative approach is used on levels 0, ... 'addlvl+1'. 'addlvl' needs to be > -1 for this to have an effect. Can only be used with weighted Jacobi and l1-Jacobi(default).

Can only be used when AMG is used as a preconditioner !!!

HYPRE_Int **HYPRE_BoomerAMGSetAddLastLvl**(*HYPRE_Solver* solver, HYPRE_Int add_last_lvl)

(Optional) Defines last level where additive, mult-additive or simple cycle is used. The multiplicative approach is used on levels > add_last_lvl.

Can only be used when AMG is used as a preconditioner !!!

HYPRE_Int **HYPRE_BoomerAMGSetMultAddTruncFactor**(*HYPRE_Solver* solver, HYPRE_Real add_trunc_factor)

(Optional) Defines the truncation factor for the smoothed interpolation used for mult-additive or simple method. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetMultAddPMaxElmts**(*HYPRE_Solver* solver, HYPRE_Int add_P_max_elmts)

(Optional) Defines the maximal number of elements per row for the smoothed interpolation used for mult-additive or simple method. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetAddRelaxType**(*HYPRE_Solver* solver, HYPRE_Int add_rlx_type)

(Optional) Defines the relaxation type used in the (mult)additive cycle portion (also affects simple method.) The default is 18 (L1-Jacobi). Currently the only other option allowed is 0 (Jacobi) which should be used in combination with HYPRE_BoomerAMGSetAddRelaxWt.

HYPRE_Int **HYPRE_BoomerAMGSetAddRelaxWt**(*HYPRE_Solver* solver, HYPRE_Real add_rlx_wt)

(Optional) Defines the relaxation weight used for Jacobi within the (mult)additive or simple cycle portion. The default is 1. The weight only affects the Jacobi method, and has no effect on L1-Jacobi

HYPRE_Int **HYPRE_BoomerAMGSetSeqThreshold**(*HYPRE_Solver* solver, HYPRE_Int seq_threshold)

(Optional) Sets maximal size for agglomeration or redundant coarse grid solve. When the system is smaller than this threshold, sequential AMG is used on process 0 or on all remaining active processes (if redundant = 1).

HYPRE_Int **HYPRE_BoomerAMGSetRedundant**(*HYPRE_Solver* solver, HYPRE_Int redundant)

(Optional) operates switch for redundancy. Needs to be used with HYPRE_BoomerAMGSetSeqThreshold. Default is 0, i.e. no redundancy.

HYPRE_Int **HYPRE_BoomerAMGSetNumGridSweeps**(*HYPRE_Solver* solver, HYPRE_Int *num_grid_sweeps)

(Optional) Defines the number of sweeps for the fine and coarse grid, the up and down cycle.

Note: This routine will be phased out!!!! Use HYPRE_BoomerAMGSetNumSweeps or HYPRE_BoomerAMGSetCycleNumSweeps instead.

HYPRE_Int **HYPRE_BoomerAMGSetNumSweeps**(*HYPRE_Solver* solver, HYPRE_Int num_sweeps)

(Optional) Sets the number of sweeps. On the finest level, the up and the down cycle the number of sweeps are set to *num_sweeps* and on the coarsest level to 1. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetCycleNumSweeps**(*HYPRE_Solver* solver, HYPRE_Int num_sweeps, HYPRE_Int k)

(Optional) Sets the number of sweeps at a specified cycle. There are the following options for *k:*

- 1 : the down cycle
- 2 : the up cycle
- 3 : the coarsest level

HYPRE_Int **HYPRE_BoomerAMGSetGridRelaxType**(*HYPRE_Solver* solver, HYPRE_Int *grid_relax_type)

(Optional) Defines which smoother is used on the fine and coarse grid, the up and down cycle.

Note: This routine will be phased out!!!! Use HYPRE_BoomerAMGSetRelaxType or HYPRE_BoomerAMGSetCycleRelaxType instead.

HYPRE_Int **HYPRE_BoomerAMGSetRelaxType**(*HYPRE_Solver* solver, HYPRE_Int relax_type)

(Optional) Defines the smoother to be used. It uses the given smoother on the fine grid, the up and the down cycle and sets the solver on the coarsest level to Gaussian elimination (9). The default is ℓ_1-Gauss-Seidel, forward solve (13) on the down cycle and backward solve (14) on the up cycle.

There are the following options for *relax_type:*

- 0 : Jacobi
- 1 : Gauss-Seidel, sequential (very slow!)
- 2 : Gauss-Seidel, interior points in parallel, boundary sequential (slow!)
- 3 : hybrid Gauss-Seidel or SOR, forward solve
- 4 : hybrid Gauss-Seidel or SOR, backward solve
- 5 : hybrid chaotic Gauss-Seidel (works only with OpenMP)
- 6 : hybrid symmetric Gauss-Seidel or SSOR
- 8 : ℓ_1-scaled hybrid symmetric Gauss-Seidel
- 9 : Gaussian elimination (only on coarsest level)
- 13 : ℓ_1 Gauss-Seidel, forward solve
- 14 : ℓ_1 Gauss-Seidel, backward solve
- 15 : CG (warning - not a fixed smoother - may require FGMRES)
- 16 : Chebyshev
- 17 : FCF-Jacobi
- 18 : ℓ_1-scaled jacobi

HYPRE_Int **HYPRE_BoomerAMGSetCycleRelaxType**(*HYPRE_Solver* solver, HYPRE_Int relax_type, HYPRE_Int k)

(Optional) Defines the smoother at a given cycle. For options of *relax_type* see description of HYPRE_BoomerAMGSetRelaxType). Options for *k* are

- 1 : the down cycle
- 2 : the up cycle

- 3 : the coarsest level

HYPRE_Int **HYPRE_BoomerAMGSetRelaxOrder**(*HYPRE_Solver* solver, HYPRE_Int relax_order)

(Optional) Defines in which order the points are relaxed. There are the following options for *relax_order:*

- 0 : the points are relaxed in natural or lexicographic order on each processor
- 1 : CF-relaxation is used, i.e on the fine grid and the down cycle the coarse points are relaxed first, followed by the fine points; on the up cycle the F-points are relaxed first, followed by the C-points. On the coarsest level, if an iterative scheme is used, the points are relaxed in lexicographic order.

The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetGridRelaxPoints**(*HYPRE_Solver* solver, HYPRE_Int **grid_relax_points)

(Optional) Defines in which order the points are relaxed.

Note: This routine will be phased out!!!! Use HYPRE_BoomerAMGSetRelaxOrder instead.

HYPRE_Int **HYPRE_BoomerAMGSetRelaxWeight**(*HYPRE_Solver* solver, HYPRE_Real *relax_weight)

(Optional) Defines the relaxation weight for smoothed Jacobi and hybrid SOR.

Note: This routine will be phased out!!!! Use HYPRE_BoomerAMGSetRelaxWt or HYPRE_BoomerAMGSetLevelRelaxWt instead.

HYPRE_Int **HYPRE_BoomerAMGSetRelaxWt**(*HYPRE_Solver* solver, HYPRE_Real relax_weight)

(Optional) Defines the relaxation weight for smoothed Jacobi and hybrid SOR on all levels.

Values for *relax_weight* are

- > 0 : this assigns the given relaxation weight on all levels
- = 0 : the weight is determined on each level with the estimate $\frac{3}{4\|D^{-1/2}AD^{-1/2}\|}$, where D is the diagonal of A (this should only be used with Jacobi)
- = -k : the relaxation weight is determined with at most k CG steps on each level (this should only be used for symmetric positive definite problems)

The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetLevelRelaxWt**(*HYPRE_Solver* solver, HYPRE_Real relax_weight, HYPRE_Int level)

(Optional) Defines the relaxation weight for smoothed Jacobi and hybrid SOR on the user defined level. Note that the finest level is denoted 0, the next coarser level 1, etc. For nonpositive *relax_weight*, the parameter is determined on the given level as described for HYPRE_BoomerAMGSetRelaxWt. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetOmega**(*HYPRE_Solver* solver, HYPRE_Real *omega)

(Optional) Defines the outer relaxation weight for hybrid SOR. Note: This routine will be phased out!!!! Use HYPRE_BoomerAMGSetOuterWt or HYPRE_BoomerAMGSetLevelOuterWt instead.

HYPRE_Int **HYPRE_BoomerAMGSetOuterWt**(*HYPRE_Solver* solver, HYPRE_Real omega)

(Optional) Defines the outer relaxation weight for hybrid SOR and SSOR on all levels.

Values for *omega* are

- > 0 : this assigns the same outer relaxation weight omega on each level
- = -k : an outer relaxation weight is determined with at most k CG steps on each level (this only makes sense for symmetric positive definite problems and smoothers such as SSOR)

The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetLevelOuterWt**(*HYPRE_Solver* solver, HYPRE_Real omega, HYPRE_Int level)

(Optional) Defines the outer relaxation weight for hybrid SOR or SSOR on the user defined level. Note that the finest level is denoted 0, the next coarser level 1, etc. For nonpositive omega, the parameter is determined on the given level as described for HYPRE_BoomerAMGSetOuterWt. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetChebyOrder**(*HYPRE_Solver* solver, HYPRE_Int order)

(Optional) Defines the Order for Chebyshev smoother. The default is 2 (valid options are 1-4).

HYPRE_Int **HYPRE_BoomerAMGSetChebyFraction**(*HYPRE_Solver* solver, HYPRE_Real ratio)

(Optional) Fraction of the spectrum to use for the Chebyshev smoother. The default is .3 (i.e., damp on upper 30% of the spectrum).

HYPRE_Int **HYPRE_BoomerAMGSetChebyScale**(*HYPRE_Solver* solver, HYPRE_Int scale)

(Optional) Defines whether matrix should be scaled. The default is 1 (i.e., scaled).

HYPRE_Int **HYPRE_BoomerAMGSetChebyVariant**(*HYPRE_Solver* solver, HYPRE_Int variant)

(Optional) Defines which polynomial variant should be used. The default is 0 (i.e., scaled).

HYPRE_Int **HYPRE_BoomerAMGSetChebyEigEst**(*HYPRE_Solver* solver, HYPRE_Int eig_est)

(Optional) Defines how to estimate eigenvalues. The default is 10 (i.e., 10 CG iterations are used to find extreme eigenvalues.) If eig_est=0, the largest eigenvalue is estimated using Gershgorin, the smallest is set to 0. If eig_est is a positive number n, n iterations of CG are used to determine the smallest and largest eigenvalue.

HYPRE_Int **HYPRE_BoomerAMGSetSmoothType**(*HYPRE_Solver* solver, HYPRE_Int smooth_type)

(Optional) Enables the use of more complex smoothers. The following options exist for *smooth_type:*

- 6 : Schwarz (routines needed to set: HYPRE_BoomerAMGSetDomainType, HYPRE_BoomerAMGSetOverlap, HYPRE_BoomerAMGSetVariant, HYPRE_BoomerAMGSetSchwarzRlxWeight)
- 7 : Pilut (routines needed to set: HYPRE_BoomerAMGSetDropTol, HYPRE_BoomerAMGSetMaxNzPerRow)
- 8 : ParaSails (routines needed to set: HYPRE_BoomerAMGSetSym, HYPRE_BoomerAMGSetLevel, HYPRE_BoomerAMGSetFilter, HYPRE_BoomerAMGSetThreshold)
- 9 : Euclid (routines needed to set: HYPRE_BoomerAMGSetEuclidFile)
- 5 : ParILUK (routines needed to set: HYPRE_ILUSetLevelOfFill, HYPRE_ILUSetType)

The default is 6. Also, if no smoother parameters are set via the routines mentioned in the table above, default values are used.

HYPRE_Int **HYPRE_BoomerAMGSetSmoothNumLevels**(*HYPRE_Solver* solver, HYPRE_Int smooth_num_levels)

(Optional) Sets the number of levels for more complex smoothers. The smoothers, as defined by HYPRE_BoomerAMGSetSmoothType, will be used on level 0 (the finest level) through level *smooth_num_levels-1*. The default is 0, i.e. no complex smoothers are used.

HYPRE_Int **HYPRE_BoomerAMGSetSmoothNumSweeps**(*HYPRE_Solver* solver, HYPRE_Int smooth_num_sweeps)

(Optional) Sets the number of sweeps for more complex smoothers. The default is 1.

HYPRE_Int **HYPRE_BoomerAMGSetVariant**(*HYPRE_Solver* solver, HYPRE_Int variant)

(Optional) Defines which variant of the Schwarz method is used. The following options exist for *variant:*

- 0 : hybrid multiplicative Schwarz method (no overlap across processor boundaries)
- 1 : hybrid additive Schwarz method (no overlap across processor boundaries)
- 2 : additive Schwarz method
- 3 : hybrid multiplicative Schwarz method (with overlap across processor boundaries)

The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetOverlap**(*HYPRE_Solver* solver, HYPRE_Int overlap)

(Optional) Defines the overlap for the Schwarz method. The following options exist for overlap:

- 0 : no overlap
- 1 : minimal overlap (default)
- 2 : overlap generated by including all neighbors of domain boundaries

HYPRE_Int **HYPRE_BoomerAMGSetDomainType**(*HYPRE_Solver* solver, HYPRE_Int domain_type)

(Optional) Defines the type of domain used for the Schwarz method. The following options exist for *domain_type:*

- 0 : each point is a domain
- 1 : each node is a domain (only of interest in "systems" AMG)
- 2 : each domain is generated by agglomeration (default)

HYPRE_Int **HYPRE_BoomerAMGSetSchwarzRlxWeight**(*HYPRE_Solver* solver, HYPRE_Real schwarz_rlx_weight)

(Optional) Defines a smoothing parameter for the additive Schwarz method.

HYPRE_Int **HYPRE_BoomerAMGSetSchwarzUseNonSymm**(*HYPRE_Solver* solver, HYPRE_Int use_nonsymm)

(Optional) Indicates that the aggregates may not be SPD for the Schwarz method. The following options exist for *use_nonsymm:*

- 0 : assume SPD (default)
- 1 : assume non-symmetric

HYPRE_Int **HYPRE_BoomerAMGSetSym**(*HYPRE_Solver* solver, HYPRE_Int sym)

(Optional) Defines symmetry for ParaSAILS. For further explanation see description of ParaSAILS.

HYPRE_Int **HYPRE_BoomerAMGSetLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Defines number of levels for ParaSAILS. For further explanation see description of ParaSAILS.

HYPRE_Int **HYPRE_BoomerAMGSetThreshold**(*HYPRE_Solver* solver, HYPRE_Real threshold)

(Optional) Defines threshold for ParaSAILS. For further explanation see description of ParaSAILS.

HYPRE_Int **HYPRE_BoomerAMGSetFilter**(*HYPRE_Solver* solver, HYPRE_Real filter)

(Optional) Defines filter for ParaSAILS. For further explanation see description of ParaSAILS.

HYPRE_Int **HYPRE_BoomerAMGSetDropTol**(*HYPRE_Solver* solver, HYPRE_Real drop_tol)

(Optional) Defines drop tolerance for PILUT. For further explanation see description of PILUT.

HYPRE_Int **HYPRE_BoomerAMGSetMaxNzPerRow**(*HYPRE_Solver* solver, HYPRE_Int max_nz_per_row)

(Optional) Defines maximal number of nonzeros for PILUT. For further explanation see description of PILUT.

HYPRE_Int **HYPRE_BoomerAMGSetEuclidFile**(*HYPRE_Solver* solver, char *euclidfile)

(Optional) Defines name of an input file for Euclid parameters. For further explanation see description of Euclid.

HYPRE_Int **HYPRE_BoomerAMGSetEuLevel**(*HYPRE_Solver* solver, HYPRE_Int eu_level)

(Optional) Defines number of levels for ILU(k) in Euclid. For further explanation see description of Euclid.

HYPRE_Int **HYPRE_BoomerAMGSetEuSparseA**(*HYPRE_Solver* solver, HYPRE_Real eu_sparse_A)

(Optional) Defines filter for ILU(k) for Euclid. For further explanation see description of Euclid.

HYPRE_Int **HYPRE_BoomerAMGSetEuBJ**(*HYPRE_Solver* solver, HYPRE_Int eu_bj)

(Optional) Defines use of block jacobi ILUT for Euclid. For further explanation see description of Euclid.

HYPRE_Int **HYPRE_BoomerAMGSetILUType**(*HYPRE_Solver* solver, HYPRE_Int ilu_type)

Defines type of ILU smoother to use For further explanation see description of ILU.

HYPRE_Int **HYPRE_BoomerAMGSetILULevel**(*HYPRE_Solver* solver, HYPRE_Int ilu_lfil)

Defines level k for ILU(k) smoother For further explanation see description of ILU.

HYPRE_Int **HYPRE_BoomerAMGSetILUMaxRowNnz**(*HYPRE_Solver* solver, HYPRE_Int ilu_max_row_nnz)

Defines max row nonzeros for ILUT smoother For further explanation see description of ILU.

HYPRE_Int **HYPRE_BoomerAMGSetILUMaxIter**(*HYPRE_Solver* solver, HYPRE_Int ilu_max_iter)

Defines number of iterations for ILU smoother on each level For further explanation see description of ILU.

HYPRE_Int **HYPRE_BoomerAMGSetILUDroptol**(*HYPRE_Solver* solver, HYPRE_Real ilu_droptol)

Defines drop tolorance for iLUT smoother For further explanation see description of ILU.

HYPRE_Int **HYPRE_BoomerAMGSetRestriction**(*HYPRE_Solver* solver, HYPRE_Int restr_par)

(Optional) Defines which parallel restriction operator is used. There are the following options for restr_type:

- 0 : P^T - Transpose of the interpolation operator
- 1 : AIR-1 - Approximate Ideal Restriction (distance 1)
- 2 : AIR-2 - Approximate Ideal Restriction (distance 2)

The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetIsTriangular**(*HYPRE_Solver* solver, HYPRE_Int is_triangular)

(Optional) Assumes the matrix is triangular in some ordering to speed up the setup time of approximate ideal restriction.

The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetGMRESSwitchR**(*HYPRE_Solver* solver, HYPRE_Int gmres_switch)

(Optional) Set local problem size at which GMRES is used over a direct solve in approximating ideal restriction. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetADropTol**(*HYPRE_Solver* solver, HYPRE_Real A_drop_tol)

(Optional) Defines the drop tolerance for the A-matrices from the 2nd level of AMG. The default is 0.

HYPRE_Int **HYPRE_BoomerAMGSetADropType**(*HYPRE_Solver* solver, HYPRE_Int A_drop_type)

(Optional) Drop the entries that are not on the diagonal and smaller than its row norm: type 1: 1-norm, 2: 2-norm, -1: infinity norm

HYPRE_Int **HYPRE_BoomerAMGSetPrintFileName**(*HYPRE_Solver* solver, const char *print_file_name)

(Optional) Name of file to which BoomerAMG will print; cf HYPRE_BoomerAMGSetPrintLevel. (Presently this is ignored).

HYPRE_Int **HYPRE_BoomerAMGSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

(Optional) Requests automatic printing of setup and solve information.

- 0 : no printout (default)
- 1 : print setup information
- 2 : print solve information
- 3 : print both setup and solve information

Note, that if one desires to print information and uses BoomerAMG as a preconditioner, suggested *print_level* is 1 to avoid excessive output, and use *print_level* of solver for solve phase information.

HYPRE_Int **HYPRE_BoomerAMGSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Requests additional computations for diagnostic and similar data to be logged by the user. Default to 0 for do nothing. The latest residual will be available if logging > 1.

HYPRE_Int **HYPRE_BoomerAMGSetDebugFlag**(*HYPRE_Solver* solver, HYPRE_Int debug_flag)

(Optional)

HYPRE_Int **HYPRE_BoomerAMGInitGridRelaxation**(HYPRE_Int **num_grid_sweeps_ptr, HYPRE_Int **grid_relax_type_ptr, HYPRE_Int ***grid_relax_points_ptr, HYPRE_Int coarsen_type, HYPRE_Real **relax_weights_ptr, HYPRE_Int max_levels)

(Optional) This routine will be eliminated in the future.

HYPRE_Int **HYPRE_BoomerAMGSetRAP2**(*HYPRE_Solver* solver, HYPRE_Int rap2)

(Optional) If rap2 not equal 0, the triple matrix product RAP is replaced by two matrix products. (Required for triple matrix product generation on GPUs)

HYPRE_Int **HYPRE_BoomerAMGSetModuleRAP2**(*HYPRE_Solver* solver, HYPRE_Int mod_rap2)

(Optional) If mod_rap2 not equal 0, the triple matrix product RAP is replaced by two matrix products with modularized kernels (Required for triple matrix product generation on GPUs)

HYPRE_Int **HYPRE_BoomerAMGSetKeepTranspose**(*HYPRE_Solver* solver, HYPRE_Int keepTranspose)

(Optional) If set to 1, the local interpolation transposes will be saved to use more efficient matvecs instead of matvecTs (Recommended for efficient use on GPUs)

HYPRE_Int **HYPRE_BoomerAMGSetPlotGrids**(*HYPRE_Solver* solver, HYPRE_Int plotgrids)

HYPRE_BoomerAMGSetPlotGrids

HYPRE_Int **HYPRE_BoomerAMGSetPlotFileName**(*HYPRE_Solver* solver, const char *plotfilename)

HYPRE_BoomerAMGSetPlotFilename

HYPRE_Int **HYPRE_BoomerAMGSetCoordDim**(*HYPRE_Solver* solver, HYPRE_Int coorddim)

HYPRE_BoomerAMGSetCoordDim

HYPRE_Int **HYPRE_BoomerAMGSetCoordinates**(*HYPRE_Solver* solver, float *coordinates)

HYPRE_BoomerAMGSetCoordinates

HYPRE_Int **HYPRE_BoomerAMGGetGridHierarchy**(*HYPRE_Solver* solver, HYPRE_Int *cgrid)

(Optional) Get the coarse grid hierarchy. Assumes input/ output array is preallocated to the size of the local matrix. On return, *cgrid*[i] returns the last grid level containing node *i*.

Parameters

- **solver** – [IN] solver or preconditioner
- **cgrid** – [IN/ OUT] preallocated array. On return, contains grid hierarchy info.

HYPRE_Int **HYPRE_BoomerAMGSetCPoints**(*HYPRE_Solver* solver, HYPRE_Int cpt_coarse_level, HYPRE_Int num_cpt_coarse, HYPRE_BigInt *cpt_coarse_index)

(Optional) Fix C points to be kept till a specified coarse level.

Parameters

- **solver** – [IN] solver or preconditioner
- **cpt_coarse_level** – [IN] coarse level up to which to keep C points
- **num_cpt_coarse** – [IN] number of C points to be kept
- **cpt_coarse_index** – [IN] indexes of C points to be kept

HYPRE_Int **HYPRE_BoomerAMGSetCpointsToKeep**(*HYPRE_Solver* solver, HYPRE_Int cpt_coarse_level, HYPRE_Int num_cpt_coarse, HYPRE_BigInt *cpt_coarse_index)

(Optional) Deprecated function. Use HYPRE_BoomerAMGSetCPoints instead.

HYPRE_Int **HYPRE_BoomerAMGSetFPoints**(*HYPRE_Solver* solver, HYPRE_Int num_fpt, HYPRE_BigInt *fpt_index)

(Optional) Set fine points in the first level.

Parameters

- **solver** – [IN] solver or preconditioner
- **num_fpt** – [IN] number of fine points
- **fpt_index** – [IN] global indices of fine points

HYPRE_Int **HYPRE_BoomerAMGSetIsolatedFPoints**(*HYPRE_Solver* solver, HYPRE_Int num_isolated_fpt, HYPRE_BigInt *isolated_fpt_index)

(Optional) Set isolated fine points in the first level. Interpolation weights are not computed for these points.

Parameters

- **solver** – [IN] solver or preconditioner
- **num_isolated_fpt** – [IN] number of isolated fine points
- **isolated_fpt_index** – [IN] global indices of isolated fine points

HYPRE_Int **HYPRE_BoomerAMGSetSabs**(*HYPRE_Solver* solver, HYPRE_Int Sabs)

(Optional) if Sabs equals 1, the strength of connection test is based on the absolute value of the matrix coefficients

ParCSR BoomerAMGDD Solver and Preconditioner

Communication reducing solver and preconditioner built on top of algebraic multigrid

HYPRE_Int **HYPRE_BoomerAMGDDCreate**(*HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_BoomerAMGDDDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_BoomerAMGDDSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the BoomerAMGDD solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_BoomerAMGDDSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply AMG-DD as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_BoomerAMGDDSetFACNumRelax**(*HYPRE_Solver* solver, HYPRE_Int amgdd_fac_num_relax)

(Optional) Set the number of pre- and post-relaxations per level for AMG-DD inner FAC cycles. Default is 1.

HYPRE_Int **HYPRE_BoomerAMGDDSetFACNumCycles**(*HYPRE_Solver* solver, HYPRE_Int amgdd_fac_num_cycles)

(Optional) Set the number of inner FAC cycles per AMG-DD iteration. Default is 2.

HYPRE_Int **HYPRE_BoomerAMGDDSetFACCycleType**(*HYPRE_Solver* solver, HYPRE_Int amgdd_fac_cycle_type)

(Optional) Set the cycle type for the AMG-DD inner FAC cycles. 1 (default) = V-cycle, 2 = W-cycle, 3 = F-cycle

HYPRE_Int **HYPRE_BoomerAMGDDSetFACRelaxType**(*HYPRE_Solver* solver, HYPRE_Int amgdd_fac_relax_type)

(Optional) Set the relaxation type for the AMG-DD inner FAC cycles. 0 = Jacobi, 1 = Gauss-Seidel, 2 = ordered Gauss-Seidel, 3 (default) = C/F L1-scaled Jacobi

HYPRE_Int **HYPRE_BoomerAMGDDSetFACRelaxWeight**(*HYPRE_Solver* solver, HYPRE_Real amgdd_fac_relax_weight)

(Optional) Set the relaxation weight for the AMG-DD inner FAC cycles. Default is 1.0.

HYPRE_Int **HYPRE_BoomerAMGDDSetStartLevel**(*HYPRE_Solver* solver, HYPRE_Int start_level)

(Optional) Set the AMG-DD start level. Default is 0.

HYPRE_Int **HYPRE_BoomerAMGDDSetPadding**(*HYPRE_Solver* solver, HYPRE_Int padding)

(Optional) Set the AMG-DD padding. Default is 1.

HYPRE_Int **HYPRE_BoomerAMGDDSetNumGhostLayers**(*HYPRE_Solver* solver, HYPRE_Int num_ghost_layers)

(Optional) Set the AMG-DD number of ghost layers. Default is 1.

HYPRE_Int **HYPRE_BoomerAMGDDSetUserFACRelaxation**(*HYPRE_Solver* solver, HYPRE_Int (*userFACRelaxation)(void *amgdd_vdata, HYPRE_Int level, HYPRE_Int cycle_param))

(Optional) Pass a custom user-defined function as a relaxation method for the AMG-DD FAC cycles. Function should have the following form, where amgdd_solver is of type hypre_ParAMGDDData* and level is the level on which to relax: HYPRE_Int userFACRelaxation(HYPRE_Solver amgdd_solver, HYPRE_Int level)

HYPRE_Int **HYPRE_BoomerAMGDDGetAMG**(*HYPRE_Solver* solver, *HYPRE_Solver* *amg_solver)

(Optional) Get the underlying AMG hierarchy as a HYPRE_Solver object.

HYPRE_Int **HYPRE_BoomerAMGDDGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *rel_resid_norm)

Returns the norm of the final relative residual.

HYPRE_Int **HYPRE_BoomerAMGDDGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Returns the number of iterations taken.

ParCSR ParaSails Preconditioner

Parallel sparse approximate inverse preconditioner for the ParCSR matrix format.

HYPRE_Int **HYPRE_ParaSailsCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a ParaSails preconditioner.

HYPRE_Int **HYPRE_ParaSailsDestroy**(*HYPRE_Solver* solver)

Destroy a ParaSails preconditioner.

HYPRE_Int **HYPRE_ParaSailsSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the ParaSails preconditioner. This function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] Preconditioner object to set up.
- **A** – [IN] ParCSR matrix used to construct the preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_ParaSailsSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Apply the ParaSails preconditioner. This function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] Preconditioner object to apply.
- **A** – Ignored by this function.
- **b** – [IN] Vector to precondition.
- **x** – [OUT] Preconditioned vector.

HYPRE_Int **HYPRE_ParaSailsSetParams**(*HYPRE_Solver* solver, HYPRE_Real thresh, HYPRE_Int nlevels)

Set the threshold and levels parameter for the ParaSails preconditioner. The accuracy and cost of ParaSails are parameterized by these two parameters. Lower values of the threshold parameter and higher values of levels parameter lead to more accurate, but more expensive preconditioners.

Parameters

- **solver** – [IN] Preconditioner object for which to set parameters.
- **thresh** – [IN] Value of threshold parameter, $0 \leq \text{thresh} \leq 1$. The default value is 0.1.
- **nlevels** – [IN] Value of levels parameter, $0 \leq \text{nlevels}$. The default value is 1.

HYPRE_Int **HYPRE_ParaSailsSetFilter**(*HYPRE_Solver* solver, HYPRE_Real filter)

Set the filter parameter for the ParaSails preconditioner.

Parameters

- **solver** – [IN] Preconditioner object for which to set filter parameter.
- **filter** – [IN] Value of filter parameter. The filter parameter is used to drop small nonzeros in the preconditioner, to reduce the cost of applying the preconditioner. Values from 0.05 to 0.1 are recommended. The default value is 0.1.

HYPRE_Int **HYPRE_ParaSailsSetSym**(*HYPRE_Solver* solver, HYPRE_Int sym)

Set the symmetry parameter for the ParaSails preconditioner.

Values for *sym*

- 0 : nonsymmetric and/or indefinite problem, and nonsymmetric preconditioner
- 1 : SPD problem, and SPD (factored) preconditioner
- 2 : nonsymmetric, definite problem, and SPD (factored) preconditioner

Parameters

- **solver** – [IN] Preconditioner object for which to set symmetry parameter.
- **sym** – [IN] Symmetry parameter.

HYPRE_Int **HYPRE_ParaSailsSetLoadbal**(*HYPRE_Solver* solver, HYPRE_Real loadbal)

Set the load balance parameter for the ParaSails preconditioner.

Parameters

- **solver** – [IN] Preconditioner object for which to set the load balance parameter.

- **loadbal** – [IN] Value of the load balance parameter, $0 \leq$ loadbal ≤ 1. A zero value indicates that no load balance is attempted; a value of unity indicates that perfect load balance will be attempted. The recommended value is 0.9 to balance the overhead of data exchanges for load balancing. No load balancing is needed if the preconditioner is very sparse and fast to construct. The default value when this parameter is not set is 0.

HYPRE_Int **HYPRE_ParaSailsSetReuse**(*HYPRE_Solver* solver, HYPRE_Int reuse)

Set the pattern reuse parameter for the ParaSails preconditioner.

Parameters

- **solver** – [IN] Preconditioner object for which to set the pattern reuse parameter.
- **reuse** – [IN] Value of the pattern reuse parameter. A nonzero value indicates that the pattern of the preconditioner should be reused for subsequent constructions of the preconditioner. A zero value indicates that the preconditioner should be constructed from scratch. The default value when this parameter is not set is 0.

HYPRE_Int **HYPRE_ParaSailsSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

Set the logging parameter for the ParaSails preconditioner.

Parameters

- **solver** – [IN] Preconditioner object for which to set the logging parameter.
- **logging** – [IN] Value of the logging parameter. A nonzero value sends statistics of the setup procedure to stdout. The default value when this parameter is not set is 0.

HYPRE_Int **HYPRE_ParaSailsBuildIJMatrix**(*HYPRE_Solver* solver, *HYPRE_IJMatrix* *pij_A)

Build IJ Matrix of the sparse approximate inverse (factor). This function explicitly creates the IJ Matrix corresponding to the sparse approximate inverse or the inverse factor. Example: HYPRE_IJMatrix ij_A; HYPRE_ParaSailsBuildIJMatrix(solver, &ij_A);

Parameters

- **solver** – [IN] Preconditioner object.
- **pij_A** – [OUT] Pointer to the IJ Matrix.

HYPRE_Int **HYPRE_ParCSRParaSailsCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

HYPRE_Int **HYPRE_ParCSRParaSailsDestroy**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_ParCSRParaSailsSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRParaSailsSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRParaSailsSetParams**(*HYPRE_Solver* solver, HYPRE_Real thresh, HYPRE_Int nlevels)

HYPRE_Int **HYPRE_ParCSRParaSailsSetFilter**(*HYPRE_Solver* solver, HYPRE_Real filter)

HYPRE_Int **HYPRE_ParCSRParaSailsSetSym**(*HYPRE_Solver* solver, HYPRE_Int sym)

HYPRE_Int **HYPRE_ParCSRParaSailsSetLoadbal**(*HYPRE_Solver* solver, HYPRE_Real loadbal)

HYPRE_Int **HYPRE_ParCSRParaSailsSetReuse**(*HYPRE_Solver* solver, HYPRE_Int reuse)

HYPRE_Int **HYPRE_ParCSRParaSailsSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

ParCSR Euclid Preconditioner

MPI Parallel ILU preconditioner

Options summary:

Option	Default	Synopsis
-level	1	ILU(k) factorization level
-bj	0 (false)	Use Block Jacobi ILU instead of PILU
-eu_stats	0 (false)	Print internal timing and statistics
-eu_mem	0 (false)	Print internal memory usage

HYPRE_Int **HYPRE_EuclidCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a Euclid object.

HYPRE_Int **HYPRE_EuclidDestroy**(*HYPRE_Solver* solver)

Destroy a Euclid object.

HYPRE_Int **HYPRE_EuclidSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the Euclid preconditioner. This function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] Preconditioner object to set up.
- **A** – [IN] ParCSR matrix used to construct the preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_EuclidSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Apply the Euclid preconditioner. This function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] Preconditioner object to apply.
- **A** – Ignored by this function.
- **b** – [IN] Vector to precondition.
- **x** – [OUT] Preconditioned vector.

HYPRE_Int **HYPRE_EuclidSetParams**(*HYPRE_Solver* solver, HYPRE_Int argc, char *argv[])

Insert (name, value) pairs in Euclid's options database by passing Euclid the command line (or an array of strings). All Euclid options (e.g, level, drop-tolerance) are stored in this database. If a (name, value) pair already exists, this call updates the value. See also: HYPRE_EuclidSetParamsFromFile.

Parameters

- **argc** – [IN] Length of argv array
- **argv** – [IN] Array of strings

HYPRE_Int **HYPRE_EuclidSetParamsFromFile**(*HYPRE_Solver* solver, char *filename)

Insert (name, value) pairs in Euclid's options database. Each line of the file should either begin with a "\#", indicating a comment line, or contain a (name value) pair, e.g:

```
>cat optionsFile
\#sample runtime parameter file
-blockJacobi 3
-matFile     /home/hysom/myfile.euclid
-doSomething true
-xx_coeff -1.0
```

See also: HYPRE_EuclidSetParams.

Parameters
filename[IN] – Pathname/filename to read

HYPRE_Int **HYPRE_EuclidSetLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

Set level k for ILU(k) factorization, default: 1

HYPRE_Int **HYPRE_EuclidSetBJ**(*HYPRE_Solver* solver, HYPRE_Int bj)

Use block Jacobi ILU preconditioning instead of PILU

HYPRE_Int **HYPRE_EuclidSetStats**(*HYPRE_Solver* solver, HYPRE_Int eu_stats)

If *eu_stats* not equal 0, a summary of runtime settings and timing information is printed to stdout.

HYPRE_Int **HYPRE_EuclidSetMem**(*HYPRE_Solver* solver, HYPRE_Int eu_mem)

If *eu_mem* not equal 0, a summary of Euclid's memory usage is printed to stdout.

HYPRE_Int **HYPRE_EuclidSetSparseA**(*HYPRE_Solver* solver, HYPRE_Real sparse_A)

Defines a drop tolerance for ILU(k). Default: 0 Use with HYPRE_EuclidSetRowScale. Note that this can destroy symmetry in a matrix.

HYPRE_Int **HYPRE_EuclidSetRowScale**(*HYPRE_Solver* solver, HYPRE_Int row_scale)

If *row_scale* not equal 0, values are scaled prior to factorization so that largest value in any row is +1 or -1. Note that this can destroy symmetry in a matrix.

HYPRE_Int **HYPRE_EuclidSetILUT**(*HYPRE_Solver* solver, HYPRE_Real drop_tol)

uses ILUT and defines a drop tolerance relative to the largest absolute value of any entry in the row being factored.

ParCSR Pilut Preconditioner

HYPRE_Int **HYPRE_ParCSRPilutCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a preconditioner object.

HYPRE_Int **HYPRE_ParCSRPilutDestroy**(*HYPRE_Solver* solver)

Destroy a preconditioner object.

HYPRE_Int **HYPRE_ParCSRPilutSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRPilutSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Precondition the system.

HYPRE_Int **HYPRE_ParCSRPilutSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_ParCSRPilutSetDropTolerance**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional)

HYPRE_Int **HYPRE_ParCSRPilutSetFactorRowSize**(*HYPRE_Solver* solver, HYPRE_Int size)

(Optional)

HYPRE_Int **HYPRE_ParCSRPilutSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

ParCSR AMS Solver and Preconditioner

Parallel auxiliary space Maxwell solver and preconditioner

HYPRE_Int **HYPRE_AMSCreate**(*HYPRE_Solver* *solver)

Create an AMS solver object.

HYPRE_Int **HYPRE_AMSDestroy**(*HYPRE_Solver* solver)

Destroy an AMS solver object.

HYPRE_Int **HYPRE_AMSSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the AMS solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_AMSSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply AMS as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_AMSSetDimension**(*HYPRE_Solver* solver, HYPRE_Int dim)

(Optional) Sets the problem dimension (2 or 3). The default is 3.

HYPRE_Int **HYPRE_AMSSetDiscreteGradient**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix G)

Sets the discrete gradient matrix *G*. This function should be called before *HYPRE_AMSSetup()*!

HYPRE_Int **HYPRE_AMSSetCoordinateVectors**(*HYPRE_Solver* solver, HYPRE_ParVector x, HYPRE_ParVector y, HYPRE_ParVector z)

Sets the *x*, *y* and *z* coordinates of the vertices in the mesh.

Either *HYPRE_AMSSetCoordinateVectors()* or *HYPRE_AMSSetEdgeConstantVectors()* should be called before *HYPRE_AMSSetup()*!

HYPRE_Int **HYPRE_AMSSetEdgeConstantVectors**(*HYPRE_Solver* solver, HYPRE_ParVector Gx, HYPRE_ParVector Gy, HYPRE_ParVector Gz)

Sets the vectors *Gx*, *Gy* and *Gz* which give the representations of the constant vector fields (1,0,0), (0,1,0) and (0,0,1) in the edge element basis.

Either *HYPRE_AMSSetCoordinateVectors()* or *HYPRE_AMSSetEdgeConstantVectors()* should be called before *HYPRE_AMSSetup()*!

HYPRE_Int **HYPRE_AMSSetInterpolations**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix Pi, HYPRE_ParCSRMatrix Pix, HYPRE_ParCSRMatrix Piy, HYPRE_ParCSRMatrix Piz)

(Optional) Set the (components of) the Nedelec interpolation matrix $\Pi = [\Pi^x, \Pi^y, \Pi^z]$.

This function is generally intended to be used only for high-order Nedelec discretizations (in the lowest order case, Π is constructed internally in AMS from the discreet gradient matrix and the coordinates of the vertices), though it can also be used in the lowest-order case or for other types of discretizations (e.g. ones based on the second family of Nedelec elements).

By definition, Π is the matrix representation of the linear operator that interpolates (high-order) vector nodal finite elements into the (high-order) Nedelec space. The component matrices are defined as $\Pi^x \varphi = \Pi(\varphi, 0, 0)$ and similarly for Π^y and Π^z. Note that all these operators depend on the choice of the basis and degrees of freedom in the high-order spaces.

The column numbering of Pi should be node-based, i.e. the x/ y/ z components of the first node (vertex or high-order dof) should be listed first, followed by the x/ y/ z components of the second node and so on (see the documentation of HYPRE_BoomerAMGSetDofFunc).

If used, this function should be called before *HYPRE_AMSSetup()* and there is no need to provide the vertex coordinates. Furthermore, only one of the sets $\{\Pi\}$ and $\{\Pi^x, \Pi^y, \Pi^z\}$ needs to be specified (though it is OK to provide both). If Pix is NULL, then scalar Π-based AMS cycles, i.e. those with *cycle_type* > 10, will be unavailable. Similarly, AMS cycles based on monolithic Π (*cycle_type* < 10) require that Pi is not NULL.

HYPRE_Int **HYPRE_AMSSetAlphaPoissonMatrix**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A_alpha)

(Optional) Sets the matrix A_α corresponding to the Poisson problem with coefficient α (the curl-curl term coefficient in the Maxwell problem).

If this function is called, the coarse space solver on the range of Π^T is a block-diagonal version of A_Π. If this function is not called, the coarse space solver on the range of Π^T is constructed as $\Pi^T A \Pi$ in *HYPRE_AMSSetup()*. See the user's manual for more details.

HYPRE_Int **HYPRE_AMSSetBetaPoissonMatrix**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A_beta)

(Optional) Sets the matrix A_β corresponding to the Poisson problem with coefficient β (the mass term coefficient in the Maxwell problem).

If not given, the Poisson matrix will be computed in *HYPRE_AMSSetup()*. If the given matrix is NULL, we assume that β is identically 0 and use two-level (instead of three-level) methods. See the user's manual for more details.

HYPRE_Int **HYPRE_AMSSetInteriorNodes**(*HYPRE_Solver* solver, HYPRE_ParVector interior_nodes)

(Optional) Set the list of nodes which are interior to a zero-conductivity region. This way, a more robust solver is constructed, that can be iterated to lower tolerance levels. A node is interior if its entry in the array is 1.0. This function should be called before *HYPRE_AMSSetup()*!

HYPRE_Int **HYPRE_AMSSetProjectionFrequency**(*HYPRE_Solver* solver, HYPRE_Int projection_frequency)

(Optional) Set the frequency at which a projection onto the compatible subspace for problems with zero-conductivity regions is performed. The default value is 5.

HYPRE_Int **HYPRE_AMSSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int maxit)

(Optional) Sets maximum number of iterations, if AMS is used as a solver. To use AMS as a preconditioner, set the maximum number of iterations to 1. The default is 20.

HYPRE_Int **HYPRE_AMSSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance, if AMS is used as a solver. When using AMS as a preconditioner, set the tolerance to 0.0. The default is 10^{-6}.

HYPRE_Int **HYPRE_AMSSetCycleType**(*HYPRE_Solver* solver, HYPRE_Int cycle_type)

(Optional) Choose which three-level solver to use. Possible values are:

- 1 : 3-level multiplicative solver (01210)
- 2 : 3-level additive solver (0+1+2)
- 3 : 3-level multiplicative solver (02120)
- 4 : 3-level additive solver (010+2)
- 5 : 3-level multiplicative solver (0102010)
- 6 : 3-level additive solver (1+020)
- 7 : 3-level multiplicative solver (0201020)
- 8 : 3-level additive solver (0(1+2)0)
- 11 : 5-level multiplicative solver (013454310)
- 12 : 5-level additive solver (0+1+3+4+5)
- 13 : 5-level multiplicative solver (034515430)
- 14 : 5-level additive solver (01(3+4+5)10)

The default is 1. See the user's manual for more details.

HYPRE_Int **HYPRE_AMSSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

(Optional) Control how much information is printed during the solution iterations. The default is 1 (print residual norm at each step).

HYPRE_Int **HYPRE_AMSSetSmoothingOptions**(*HYPRE_Solver* solver, HYPRE_Int relax_type, HYPRE_Int relax_times, HYPRE_Real relax_weight, HYPRE_Real omega)

(Optional) Sets relaxation parameters for A. The defaults are 2, 1, 1.0, 1.0.

The available options for *relax_type* are:

- 1 : ℓ_1-scaled Jacobi

- 2 : ℓ_1-scaled block symmetric Gauss-Seidel/SSOR
- 3 : Kaczmarz
- 4 : truncated version of ℓ_1-scaled block symmetric Gauss-Seidel/SSOR
- 16 : Chebyshev

HYPRE_Int **HYPRE_AMSSetAlphaAMGOptions**(*HYPRE_Solver* solver, HYPRE_Int alpha_coarsen_type, HYPRE_Int alpha_agg_levels, HYPRE_Int alpha_relax_type, HYPRE_Real alpha_strength_threshold, HYPRE_Int alpha_interp_type, HYPRE_Int alpha_Pmax)

(Optional) Sets AMG parameters for B_Π. The defaults are 10, 1, 3, 0.25, 0, 0. See the user's manual for more details.

HYPRE_Int **HYPRE_AMSSetAlphaAMGCoarseRelaxType**(*HYPRE_Solver* solver, HYPRE_Int alpha_coarse_relax_type)

(Optional) Sets the coarsest level relaxation in the AMG solver for B_Π. The default is 8 (l1-GS). Use 9, 19, 29 or 99 for a direct solver.

HYPRE_Int **HYPRE_AMSSetBetaAMGOptions**(*HYPRE_Solver* solver, HYPRE_Int beta_coarsen_type, HYPRE_Int beta_agg_levels, HYPRE_Int beta_relax_type, HYPRE_Real beta_strength_threshold, HYPRE_Int beta_interp_type, HYPRE_Int beta_Pmax)

(Optional) Sets AMG parameters for B_G. The defaults are 10, 1, 3, 0.25, 0, 0. See the user's manual for more details.

HYPRE_Int **HYPRE_AMSSetBetaAMGCoarseRelaxType**(*HYPRE_Solver* solver, HYPRE_Int beta_coarse_relax_type)

(Optional) Sets the coarsest level relaxation in the AMG solver for B_G. The default is 8 (l1-GS). Use 9, 19, 29 or 99 for a direct solver.

HYPRE_Int **HYPRE_AMSGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Returns the number of iterations taken.

HYPRE_Int **HYPRE_AMSGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *rel_resid_norm)

Returns the norm of the final relative residual.

HYPRE_Int **HYPRE_AMSProjectOutGradients**(*HYPRE_Solver* solver, HYPRE_ParVector x)

For problems with zero-conductivity regions, project the vector onto the compatible subspace: $x = (I - G_0(G_0^t G_0)^{-1} G_0^T)x$, where G_0 is the discrete gradient restricted to the interior nodes of the regions with zero conductivity. This ensures that x is orthogonal to the gradients in the range of G_0.

This function is typically called after the solution iteration is complete, in order to facilitate the visualization of the computed field. Without it the values in the zero-conductivity regions contain kernel components.

HYPRE_Int **HYPRE_AMSConstructDiscreteGradient**(HYPRE_ParCSRMatrix A, HYPRE_ParVector x_coord, HYPRE_BigInt *edge_vertex, HYPRE_Int edge_orientation, HYPRE_ParCSRMatrix *G)

Construct and return the lowest-order discrete gradient matrix G using some edge and vertex information. We assume that *edge_vertex* lists the edge vertices consecutively, and that the orientation of all edges is consistent.

If *edge_orientation* = 1, the edges are already oriented.

If *edge_orientation* = 2, the orientation of edge i depends only on the sign of *edge_vertex*[2*i+1] - *edge_vertex*[2*i].

ParCSR ADS Solver and Preconditioner

Parallel auxiliary space divergence solver and preconditioner

HYPRE_Int **HYPRE_ADSCreate**(*HYPRE_Solver* *solver)

Create an ADS solver object.

HYPRE_Int **HYPRE_ADSDestroy**(*HYPRE_Solver* solver)

Destroy an ADS solver object.

HYPRE_Int **HYPRE_ADSSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Set up the ADS solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_ADSSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply ADS as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_ADSSetDiscreteCurl**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix C)

Sets the discrete curl matrix *C*. This function should be called before *HYPRE_ADSSetup()*!

HYPRE_Int **HYPRE_ADSSetDiscreteGradient**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix G)

Sets the discrete gradient matrix *G*. This function should be called before *HYPRE_ADSSetup()*!

HYPRE_Int **HYPRE_ADSSetCoordinateVectors**(*HYPRE_Solver* solver, HYPRE_ParVector x, HYPRE_ParVector y, HYPRE_ParVector z)

Sets the x, y and z coordinates of the vertices in the mesh. This function should be called before *HYPRE_ADSSetup()*!

HYPRE_Int **HYPRE_ADSSetInterpolations**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix RT_Pi, HYPRE_ParCSRMatrix RT_Pix, HYPRE_ParCSRMatrix RT_Piy, HYPRE_ParCSRMatrix RT_Piz, HYPRE_ParCSRMatrix ND_Pi, HYPRE_ParCSRMatrix ND_Pix, HYPRE_ParCSRMatrix ND_Piy, HYPRE_ParCSRMatrix ND_Piz)

(Optional) Set the (components of) the Raviart-Thomas (Π_{RT}) and the Nedelec (Π_{ND}) interpolation matrices.

This function is generally intended to be used only for high-order $H(div)$ discretizations (in the lowest order case, these matrices are constructed internally in ADS from the discreet gradient and curl matrices and the coordinates of the vertices), though it can also be used in the lowest-order case or for other types of discretizations.

By definition, *RT_Pi* and *ND_Pi* are the matrix representations of the linear operators Π_{RT} and Π_{ND} that interpolate (high-order) vector nodal finite elements into the (high-order) Raviart-Thomas and Nedelec spaces. The component matrices are defined in both cases as $\Pi^x\varphi = \Pi(\varphi, 0, 0)$ and similarly for Π^y and Π^z. Note that all these operators depend on the choice of the basis and degrees of freedom in the high-order spaces.

The column numbering of *RT_Pi* and *ND_Pi* should be node-based, i.e. the x/ y/ z components of the first node (vertex or high-order dof) should be listed first, followed by the x/ y/ z components of the second node and so on (see the documentation of HYPRE_BoomerAMGSetDofFunc).

If used, this function should be called before hypre_ADSSetup() and there is no need to provide the vertex coordinates. Furthermore, only one of the sets $\{\Pi_{RT}\}$ and $\{\Pi^x_{RT}, \Pi^y_{RT}, \Pi^z_{RT}\}$ needs to be specified (though it is OK to provide both). If *RT_Pix* is NULL, then scalar Π-based ADS cycles, i.e. those with *cycle_type* > 10, will be unavailable. Similarly, ADS cycles based on monolithic Π (*cycle_type* < 10) require that *RT_Pi* is not NULL. The same restrictions hold for the sets $\{\Pi_{ND}\}$ and $\{\Pi^x_{ND}, \Pi^y_{ND}, \Pi^z_{ND}\}$ — only one of them needs to be specified, and the availability of each enables different AMS cycle type options.

HYPRE_Int **HYPRE_ADSSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int maxit)

(Optional) Sets maximum number of iterations, if ADS is used as a solver. To use ADS as a preconditioner, set the maximum number of iterations to 1. The default is 20.

HYPRE_Int **HYPRE_ADSSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance, if ADS is used as a solver. When using ADS as a preconditioner, set the tolerance to 0.0. The default is 10^{-6}.

HYPRE_Int **HYPRE_ADSSetCycleType**(*HYPRE_Solver* solver, HYPRE_Int cycle_type)

(Optional) Choose which auxiliary-space solver to use. Possible values are:

- 1 : 3-level multiplicative solver (01210)
- 2 : 3-level additive solver (0+1+2)
- 3 : 3-level multiplicative solver (02120)
- 4 : 3-level additive solver (010+2)
- 5 : 3-level multiplicative solver (0102010)
- 6 : 3-level additive solver (1+020)
- 7 : 3-level multiplicative solver (0201020)
- 8 : 3-level additive solver (0(1+2)0)
- 11 : 5-level multiplicative solver (013454310)
- 12 : 5-level additive solver (0+1+3+4+5)
- 13 : 5-level multiplicative solver (034515430)
- 14 : 5-level additive solver (01(3+4+5)10)

The default is 1. See the user's manual for more details.

HYPRE_Int **HYPRE_ADSSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

(Optional) Control how much information is printed during the solution iterations. The default is 1 (print residual norm at each step).

HYPRE_Int **HYPRE_ADSSetSmoothingOptions**(*HYPRE_Solver* solver, HYPRE_Int relax_type, HYPRE_Int relax_times, HYPRE_Real relax_weight, HYPRE_Real omega)

(Optional) Sets relaxation parameters for A. The defaults are 2, 1, 1.0, 1.0.

The available options for *relax_type* are:

- 1 : ℓ_1-scaled Jacobi
- 2 : ℓ_1-scaled block symmetric Gauss-Seidel/SSOR
- 3 : Kaczmarz
- 4 : truncated version of ℓ_1-scaled block symmetric Gauss-Seidel/SSOR
- 16 : Chebyshev

HYPRE_Int **HYPRE_ADSSetChebySmoothingOptions**(*HYPRE_Solver* solver, HYPRE_Int cheby_order, HYPRE_Int cheby_fraction)

(Optional) Sets parameters for Chebyshev relaxation. The defaults are 2, 0.3.

HYPRE_Int **HYPRE_ADSSetAMSOptions**(*HYPRE_Solver* solver, HYPRE_Int cycle_type, HYPRE_Int coarsen_type, HYPRE_Int agg_levels, HYPRE_Int relax_type, HYPRE_Real strength_threshold, HYPRE_Int interp_type, HYPRE_Int Pmax)

(Optional) Sets AMS parameters for B_C. The defaults are 11, 10, 1, 3, 0.25, 0, 0. Note that *cycle_type* should be greater than 10, unless the high-order interface of HYPRE_ADSSetInterpolations is being used! See the user's manual for more details.

HYPRE_Int **HYPRE_ADSSetAMGOptions**(*HYPRE_Solver* solver, HYPRE_Int coarsen_type, HYPRE_Int agg_levels, HYPRE_Int relax_type, HYPRE_Real strength_threshold, HYPRE_Int interp_type, HYPRE_Int Pmax)

(Optional) Sets AMG parameters for B_Π. The defaults are 10, 1, 3, 0.25, 0, 0. See the user's manual for more details.

HYPRE_Int **HYPRE_ADSGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Returns the number of iterations taken.

HYPRE_Int **HYPRE_ADSGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *rel_resid_norm)

Returns the norm of the final relative residual.

ParCSR PCG Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_ParCSRPCGCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_ParCSRPCGDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRPCGSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRPCGSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRPCGSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRPCGSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRPCGSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRPCGSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_ParCSRPCGSetTwoNorm**(*HYPRE_Solver* solver, HYPRE_Int two_norm)

HYPRE_Int **HYPRE_ParCSRPCGSetRelChange**(*HYPRE_Solver* solver, HYPRE_Int rel_change)

HYPRE_Int **HYPRE_ParCSRPCGSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRPCGGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRPCGSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRPCGSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRPCGGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRPCGGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRPCGGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

Returns the residual.

HYPRE_Int **HYPRE_ParCSRDiagScaleSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector y, HYPRE_ParVector x)

Setup routine for diagonal preconditioning.

HYPRE_Int **HYPRE_ParCSRDiagScale**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix HA, HYPRE_ParVector Hy, HYPRE_ParVector Hx)

Solve routine for diagonal preconditioning.

HYPRE_Int **HYPRE_ParCSROnProcTriSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix HA, HYPRE_ParVector Hy, HYPRE_ParVector Hx)

HYPRE_Int **HYPRE_ParCSROnProcTriSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix HA, HYPRE_ParVector Hy, HYPRE_ParVector Hx)

ParCSR GMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_ParCSRGMRESCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_ParCSRGMRESDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRGMRESSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRGMRESSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_ParCSRGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

HYPRE_Int **HYPRE_ParCSRGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRGMRESSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_ParCSRGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRGMRESGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

Returns the residual.

HYPRE_Int **HYPRE_ParCSRCOGMRESCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_ParCSRCOGMRESDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRCOGMRESSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRCOGMRESSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetUnroll**(*HYPRE_Solver* solver, HYPRE_Int unroll)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetCGS**(*HYPRE_Solver* solver, HYPRE_Int cgs)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRCOGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRCOGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRCOGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRCOGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRCOGMRESGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

Returns the residual.

ParCSR FlexGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_ParCSRFlexGMRESCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_ParCSRFlexGMRESDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRFlexGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRFlexGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRFlexGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRFlexGMRESGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

HYPRE_Int **HYPRE_ParCSRFlexGMRESSetModifyPC**(*HYPRE_Solver* solver, *HYPRE_PtrToModifyPCFcn* modify_pc)

ParCSR LGMRES Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_ParCSRLGMRESCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object.

HYPRE_Int **HYPRE_ParCSRLGMRESDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRLGMRESSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRLGMRESSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRLGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

HYPRE_Int **HYPRE_ParCSRLGMRESSetAugDim**(*HYPRE_Solver* solver, HYPRE_Int aug_dim)

HYPRE_Int **HYPRE_ParCSRLGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRLGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

HYPRE_Int **HYPRE_ParCSRLGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRLGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRLGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRLGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRLGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRLGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRLGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRLGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRLGMRESGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

ParCSR BiCGSTAB Solver

These routines should be used in conjunction with the generic interface in *Krylov Solvers*.

HYPRE_Int **HYPRE_ParCSRBiCGSTABCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

Create a solver object

HYPRE_Int **HYPRE_ParCSRBiCGSTABDestroy**(*HYPRE_Solver* solver)

Destroy a solver object.

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRBiCGSTABGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRBiCGSTABSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_ParCSRBiCGSTABGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRBiCGSTABGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

HYPRE_Int **HYPRE_ParCSRBiCGSTABGetResidual**(*HYPRE_Solver* solver, HYPRE_ParVector *residual)

ParCSR Hybrid Solver

HYPRE_Int **HYPRE_ParCSRHybridCreate**(*HYPRE_Solver* *solver)

Create solver object

HYPRE_Int **HYPRE_ParCSRHybridDestroy**(*HYPRE_Solver* solver)

Destroy solver object

HYPRE_Int **HYPRE_ParCSRHybridSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Setup the hybrid solver

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – Ignored by this function.
- **x** – Ignored by this function.

HYPRE_Int **HYPRE_ParCSRHybridSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve linear system

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_ParCSRHybridSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

Set the convergence tolerance for the Krylov solver. The default is 1.e-6.

HYPRE_Int **HYPRE_ParCSRHybridSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

Set the absolute convergence tolerance for the Krylov solver. The default is 0.

HYPRE_Int **HYPRE_ParCSRHybridSetConvergenceTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

Set the desired convergence factor

HYPRE_Int **HYPRE_ParCSRHybridSetDSCGMaxIter**(*HYPRE_Solver* solver, HYPRE_Int dscg_max_its)

Set the maximal number of iterations for the diagonally preconditioned solver

HYPRE_Int **HYPRE_ParCSRHybridSetPCGMaxIter**(*HYPRE_Solver* solver, HYPRE_Int pcg_max_its)

Set the maximal number of iterations for the AMG preconditioned solver

HYPRE_Int **HYPRE_ParCSRHybridSetSetupType**(*HYPRE_Solver* solver, HYPRE_Int setup_type)

HYPRE_Int **HYPRE_ParCSRHybridSetSolverType**(*HYPRE_Solver* solver, HYPRE_Int solver_type)

Set the desired solver type. There are the following options:

- 1 : PCG (default)
- 2 : GMRES
- 3 : BiCGSTAB

HYPRE_Int **HYPRE_ParCSRHybridSetRecomputeResidual**(*HYPRE_Solver* solver, HYPRE_Int recompute_residual)

(Optional) Set recompute residual (don't rely on 3-term recurrence).

HYPRE_Int **HYPRE_ParCSRHybridGetRecomputeResidual**(*HYPRE_Solver* solver, HYPRE_Int *recompute_residual)

(Optional) Get recompute residual option.

HYPRE_Int **HYPRE_ParCSRHybridSetRecomputeResidualP**(*HYPRE_Solver* solver, HYPRE_Int recompute_residual_p)

(Optional) Set recompute residual period (don't rely on 3-term recurrence).

Recomputes residual after every specified number of iterations.

HYPRE_Int **HYPRE_ParCSRHybridGetRecomputeResidualP**(*HYPRE_Solver* solver, HYPRE_Int *recompute_residual_p)

(Optional) Get recompute residual period option.

HYPRE_Int **HYPRE_ParCSRHybridSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

Set the Krylov dimension for restarted GMRES. The default is 5.

HYPRE_Int **HYPRE_ParCSRHybridSetTwoNorm**(*HYPRE_Solver* solver, HYPRE_Int two_norm)

Set the type of norm for PCG.

HYPRE_Int **HYPRE_ParCSRHybridSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

RE-VISIT

HYPRE_Int **HYPRE_ParCSRHybridSetRelChange**(*HYPRE_Solver* solver, HYPRE_Int rel_change)

HYPRE_Int **HYPRE_ParCSRHybridSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

Set preconditioner if wanting to use one that is not set up by the hybrid solver.

HYPRE_Int **HYPRE_ParCSRHybridSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

Set logging parameter (default: 0, no logging).

HYPRE_Int **HYPRE_ParCSRHybridSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

Set print level (default: 0, no printing) 2 will print residual norms per iteration 10 will print AMG setup information if AMG is used 12 both Setup information and iterations.

HYPRE_Int **HYPRE_ParCSRHybridSetStrongThreshold**(*HYPRE_Solver* solver, HYPRE_Real strong_threshold)

(Optional) Sets AMG strength threshold. The default is 0.25. For elasticity problems, a larger strength threshold, such as 0.7 or 0.8, is often better.

HYPRE_Int **HYPRE_ParCSRHybridSetMaxRowSum**(*HYPRE_Solver* solver, HYPRE_Real max_row_sum)

(Optional) Sets a parameter to modify the definition of strength for diagonal dominant portions of the matrix. The default is 0.9. If *max_row_sum* is 1, no checking for diagonally dominant rows is performed.

HYPRE_Int **HYPRE_ParCSRHybridSetTruncFactor**(*HYPRE_Solver* solver, HYPRE_Real trunc_factor)

(Optional) Defines a truncation factor for the interpolation. The default is 0.

HYPRE_Int **HYPRE_ParCSRHybridSetPMaxElmts**(*HYPRE_Solver* solver, HYPRE_Int P_max_elmts)

(Optional) Defines the maximal number of elements per row for the interpolation. The default is 0.

HYPRE_Int **HYPRE_ParCSRHybridSetMaxLevels**(*HYPRE_Solver* solver, HYPRE_Int max_levels)

(Optional) Defines the maximal number of levels used for AMG. The default is 25.

HYPRE_Int **HYPRE_ParCSRHybridSetMeasureType**(*HYPRE_Solver* solver, HYPRE_Int measure_type)

(Optional) Defines whether local or global measures are used.

HYPRE_Int **HYPRE_ParCSRHybridSetCoarsenType**(*HYPRE_Solver* solver, HYPRE_Int coarsen_type)

(Optional) Defines which parallel coarsening algorithm is used. There are the following options for *coarsen_type:*

- 0 : CLJP-coarsening (a parallel coarsening algorithm using independent sets).
- 1 : classical Ruge-Stueben coarsening on each processor, no boundary treatment
- 3 : classical Ruge-Stueben coarsening on each processor, followed by a third pass, which adds coarse points on the boundaries
- 6 : Falgout coarsening (uses 1 first, followed by CLJP using the interior coarse points generated by 1 as its first independent set)
- 7 : CLJP-coarsening (using a fixed random vector, for debugging purposes only)
- 8 : PMIS-coarsening (a parallel coarsening algorithm using independent sets with lower complexities than CLJP, might also lead to slower convergence)
- 9 : PMIS-coarsening (using a fixed random vector, for debugging purposes only)
- 10 : HMIS-coarsening (uses one pass Ruge-Stueben on each processor independently, followed by PMIS using the interior C-points as its first independent set)
- 11 : one-pass Ruge-Stueben coarsening on each processor, no boundary treatment

The default is 10.

HYPRE_Int **HYPRE_ParCSRHybridSetInterpType**(*HYPRE_Solver* solver, HYPRE_Int interp_type)

(Optional) Specifies which interpolation operator is used The default is ext+i interpolation truncated to at most 4 elements per row.

HYPRE_Int **HYPRE_ParCSRHybridSetCycleType**(*HYPRE_Solver* solver, HYPRE_Int cycle_type)

(Optional) Defines the type of cycle. For a V-cycle, set *cycle_type* to 1, for a W-cycle set *cycle_type* to 2. The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetGridRelaxType**(*HYPRE_Solver* solver, HYPRE_Int *grid_relax_type)

HYPRE_Int **HYPRE_ParCSRHybridSetGridRelaxPoints**(*HYPRE_Solver* solver, HYPRE_Int **grid_relax_points)

HYPRE_Int **HYPRE_ParCSRHybridSetNumSweeps**(*HYPRE_Solver* solver, HYPRE_Int num_sweeps)

(Optional) Sets the number of sweeps. On the finest level, the up and the down cycle the number of sweeps are set to *num_sweeps* and on the coarsest level to 1. The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetCycleNumSweeps**(*HYPRE_Solver* solver, HYPRE_Int num_sweeps, HYPRE_Int k)

(Optional) Sets the number of sweeps at a specified cycle. There are the following options for *k:*

- 1 : the down cycle
- 2 : the up cycle

- 3 : the coarsest level

HYPRE_Int **HYPRE_ParCSRHybridSetRelaxType**(*HYPRE_Solver* solver, HYPRE_Int relax_type)

(Optional) Defines the smoother to be used. It uses the given smoother on the fine grid, the up and the down cycle and sets the solver on the coarsest level to Gaussian elimination (9). The default is l1-Gauss-Seidel, forward solve on the down cycle (13) and backward solve on the up cycle (14).

There are the following options for *relax_type:*

- 0 : Jacobi
- 1 : Gauss-Seidel, sequential (very slow!)
- 2 : Gauss-Seidel, interior points in parallel, boundary sequential (slow!)
- 3 : hybrid Gauss-Seidel or SOR, forward solve
- 4 : hybrid Gauss-Seidel or SOR, backward solve
- 6 : hybrid symmetric Gauss-Seidel or SSOR
- 8 : hybrid symmetric l1-Gauss-Seidel or SSOR
- 13 : l1-Gauss-Seidel, forward solve
- 14 : l1-Gauss-Seidel, backward solve
- 18 : l1-Jacobi
- 9 : Gaussian elimination (only on coarsest level)

HYPRE_Int **HYPRE_ParCSRHybridSetCycleRelaxType**(*HYPRE_Solver* solver, HYPRE_Int relax_type, HYPRE_Int k)

(Optional) Defines the smoother at a given cycle. For options of *relax_type* see description of HYPRE_BoomerAMGSetRelaxType). Options for k are

- 1 : the down cycle
- 2 : the up cycle
- 3 : the coarsest level

HYPRE_Int **HYPRE_ParCSRHybridSetRelaxOrder**(*HYPRE_Solver* solver, HYPRE_Int relax_order)

(Optional) Defines in which order the points are relaxed. There are the following options for *relax_order:*

- 0 : the points are relaxed in natural or lexicographic order on each processor
- 1 : CF-relaxation is used, i.e on the fine grid and the down cycle the coarse points are relaxed first, followed by the fine points; on the up cycle the F-points are relaxed first, followed by the C-points. On the coarsest level, if an iterative scheme is used, the points are relaxed in lexicographic order.

The default is 0 (CF-relaxation).

HYPRE_Int **HYPRE_ParCSRHybridSetRelaxWt**(*HYPRE_Solver* solver, HYPRE_Real relax_wt)

(Optional) Defines the relaxation weight for smoothed Jacobi and hybrid SOR on all levels.

Values for *relax_wt* are

- > 0 : this assigns the given relaxation weight on all levels

- = 0 : the weight is determined on each level with the estimate $\frac{3}{4\|D^{-1/2}AD^{-1/2}\|}$, where D is the diagonal of A (this should only be used with Jacobi)
- = -k : the relaxation weight is determined with at most k CG steps on each level (this should only be used for symmetric positive definite problems)

The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetLevelRelaxWt**(*HYPRE_Solver* solver, HYPRE_Real relax_wt, HYPRE_Int level)

(Optional) Defines the relaxation weight for smoothed Jacobi and hybrid SOR on the user defined level. Note that the finest level is denoted 0, the next coarser level 1, etc. For nonpositive *relax_weight*, the parameter is determined on the given level as described for HYPRE_BoomerAMGSetRelaxWt. The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetOuterWt**(*HYPRE_Solver* solver, HYPRE_Real outer_wt)

(Optional) Defines the outer relaxation weight for hybrid SOR and SSOR on all levels.

Values for *outer_wt* are

- > 0 : this assigns the same outer relaxation weight omega on each level
- = -k : an outer relaxation weight is determined with at most k CG steps on each level (this only makes sense for symmetric positive definite problems and smoothers such as SSOR)

The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetLevelOuterWt**(*HYPRE_Solver* solver, HYPRE_Real outer_wt, HYPRE_Int level)

(Optional) Defines the outer relaxation weight for hybrid SOR or SSOR on the user defined level. Note that the finest level is denoted 0, the next coarser level 1, etc. For nonpositive omega, the parameter is determined on the given level as described for HYPRE_BoomerAMGSetOuterWt. The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetMaxCoarseSize**(*HYPRE_Solver* solver, HYPRE_Int max_coarse_size)

(Optional) Defines the maximal coarse grid size. The default is 9.

HYPRE_Int **HYPRE_ParCSRHybridSetMinCoarseSize**(*HYPRE_Solver* solver, HYPRE_Int min_coarse_size)

(Optional) Defines the minimal coarse grid size. The default is 0.

HYPRE_Int **HYPRE_ParCSRHybridSetSeqThreshold**(*HYPRE_Solver* solver, HYPRE_Int seq_threshold)

(Optional) enables redundant coarse grid size. If the system size becomes smaller than seq_threshold, sequential AMG is used on all remaining processors. The default is 0.

HYPRE_Int **HYPRE_ParCSRHybridSetRelaxWeight**(*HYPRE_Solver* solver, HYPRE_Real *relax_weight)

HYPRE_Int **HYPRE_ParCSRHybridSetOmega**(*HYPRE_Solver* solver, HYPRE_Real *omega)

HYPRE_Int **HYPRE_ParCSRHybridSetAggNumLevels**(*HYPRE_Solver* solver, HYPRE_Int agg_num_levels)

(Optional) Defines the number of levels of aggressive coarsening, starting with the finest level. The default is 0, i.e. no aggressive coarsening.

HYPRE_Int **HYPRE_ParCSRHybridSetAggInterpType**(*HYPRE_Solver* solver, HYPRE_Int agg_interp_type)

HYPRE_Int **HYPRE_ParCSRHybridSetNumPaths**(*HYPRE_Solver* solver, HYPRE_Int num_paths)

(Optional) Defines the degree of aggressive coarsening. The default is 1, which leads to the most aggressive coarsening. Setting *num_paths* to 2 will increase complexity somewhat, but can lead to better convergence.

HYPRE_Int **HYPRE_ParCSRHybridSetNumFunctions**(*HYPRE_Solver* solver, HYPRE_Int num_functions)

(Optional) Sets the size of the system of PDEs, if using the systems version. The default is 1.

HYPRE_Int **HYPRE_ParCSRHybridSetDofFunc**(*HYPRE_Solver* solver, HYPRE_Int *dof_func)

(Optional) Sets the mapping that assigns the function to each variable, if using the systems version. If no assignment is made and the number of functions is k > 1, the mapping generated is (0,1,...,k-1,0,1,...,k-1,...).

HYPRE_Int **HYPRE_ParCSRHybridSetNodal**(*HYPRE_Solver* solver, HYPRE_Int nodal)

(Optional) Sets whether to use the nodal systems version. The default is 0 (the unknown based approach).

HYPRE_Int **HYPRE_ParCSRHybridSetKeepTranspose**(*HYPRE_Solver* solver, HYPRE_Int keepT)

(Optional) Sets whether to store local transposed interpolation The default is 0 (don't store).

HYPRE_Int **HYPRE_ParCSRHybridSetNonGalerkinTol**(*HYPRE_Solver* solver, HYPRE_Int num_levels, HYPRE_Real *nongalerkin_tol)

(Optional) Sets whether to use non-Galerkin option The default is no non-Galerkin option num_levels sets the number of levels where to use it nongalerkin_tol contains the tolerances for <num_levels> levels

HYPRE_Int **HYPRE_ParCSRHybridGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_its)

Retrieves the total number of iterations.

HYPRE_Int **HYPRE_ParCSRHybridGetDSCGNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *dscg_num_its)

Retrieves the number of iterations used by the diagonally scaled solver.

HYPRE_Int **HYPRE_ParCSRHybridGetPCGNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *pcg_num_its)

Retrieves the number of iterations used by the AMG preconditioned solver.

HYPRE_Int **HYPRE_ParCSRHybridGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Retrieves the final relative residual norm.

HYPRE_Int **HYPRE_ParCSRHybridSetNumGridSweeps**(*HYPRE_Solver* solver, HYPRE_Int *num_grid_sweeps)

HYPRE_Int **HYPRE_ParCSRHybridGetSetupSolveTime**(*HYPRE_Solver* solver, HYPRE_Real *time)

ParCSR MGR Solver

Parallel multigrid reduction solver and preconditioner. This solver or preconditioner is designed with systems of PDEs in mind. However, it can also be used for scalar linear systems, particularly for problems where the user can exploit information from the physics of the problem. In this way, the MGR solver could potentially be used as a foundation for a physics-based preconditioner.

HYPRE_Int **HYPRE_MGRCreate**(*HYPRE_Solver* *solver)

Create a solver object

HYPRE_Int **HYPRE_MGRDestroy**(*HYPRE_Solver* solver)

Destroy a solver object

HYPRE_Int **HYPRE_MGRSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Setup the MGR solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – right-hand-side of the linear system to be solved (Ignored by this function).
- **x** – approximate solution of the linear system to be solved (Ignored by this function).

HYPRE_Int **HYPRE_MGRSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply MGR as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_MGRSetCpointsByContiguousBlock**(*HYPRE_Solver* solver, HYPRE_Int block_size, HYPRE_Int max_num_levels, HYPRE_BigInt *idx_array, HYPRE_Int *num_block_coarse_points, HYPRE_Int **block_coarse_indexes)

Set the block data assuming that the physical variables are ordered contiguously, i.e. p_1, p_2, …, p_n, s_1, s_2, …, s_n, …

Parameters

- **solver** – [IN] solver or preconditioner object
- **block_size** – [IN] system block size
- **max_num_levels** – [IN] maximum number of reduction levels
- **num_block_coarse_points** – [IN] number of coarse points per block per level
- **block_coarse_indexes** – [IN] index for each block coarse point per level

HYPRE_Int **HYPRE_MGRSetCpointsByBlock**(*HYPRE_Solver* solver, HYPRE_Int block_size, HYPRE_Int max_num_levels, HYPRE_Int *num_block_coarse_points, HYPRE_Int **block_coarse_indexes)

Set the block data (by grid points) and prescribe the coarse indexes per block for each reduction level.

Parameters

- **solver** – [IN] solver or preconditioner object
- **block_size** – [IN] system block size
- **max_num_levels** – [IN] maximum number of reduction levels
- **num_block_coarse_points** – [IN] number of coarse points per block per level
- **block_coarse_indexes** – [IN] index for each block coarse point per level

HYPRE_Int **HYPRE_MGRSetCpointsByPointMarkerArray**(*HYPRE_Solver* solver, HYPRE_Int block_size, HYPRE_Int max_num_levels, HYPRE_Int *num_block_coarse_points, HYPRE_Int **lvl_block_coarse_indexes, HYPRE_Int *point_marker_array)

Set the coarse indices for the levels using an array of tags for all the local degrees of freedom. TODO: Rename the function to make it more descriptive.

Parameters

- **solver** – [IN] solver or preconditioner object
- **block_size** – [IN] system block size
- **max_num_levels** – [IN] maximum number of reduction levels
- **num_block_coarse_points** – [IN] number of coarse points per block per level
- **lvl_block_coarse_indexes** – [IN] indices for the coarse points per level
- **point_marker_array** – [IN] array of tags for the local degrees of freedom

HYPRE_Int **HYPRE_MGRSetNonCpointsToFpoints**(*HYPRE_Solver* solver, HYPRE_Int nonCptToFptFlag)

(Optional) Set non C-points to F-points. This routine determines how the coarse points are selected for the next level reduction. Options for *nonCptToFptFlag* are:

- 0 : Allow points not prescribed as C points to be potentially set as C points using classical AMG coarsening strategies (currently uses CLJP-coarsening).
- 1 : Fix points not prescribed as C points to be F points for the next reduction

HYPRE_Int **HYPRE_MGRSetMaxCoarseLevels**(*HYPRE_Solver* solver, HYPRE_Int maxlev)

(Optional) Set maximum number of coarsening (or reduction) levels. The default is 10.

HYPRE_Int **HYPRE_MGRSetBlockSize**(*HYPRE_Solver* solver, HYPRE_Int bsize)

(Optional) Set the system block size. This should match the block size set in the MGRSetCpointsByBlock function. The default is 1.

HYPRE_Int **HYPRE_MGRSetReservedCoarseNodes**(*HYPRE_Solver* solver, HYPRE_Int reserved_coarse_size, HYPRE_BigInt *reserved_coarse_nodes)

(Optional) Defines indexes of coarse nodes to be kept to the coarsest level. These indexes are passed down through the MGR hierarchy to the coarsest grid of the coarse grid (BoomerAMG) solver.

Parameters

- **solver** – [IN] solver or preconditioner object
- **reserved_coarse_size** – [IN] number of reserved coarse points
- **reserved_coarse_nodes** – [IN] (global) indexes of reserved coarse points

HYPRE_Int **HYPRE_MGRSetReservedCpointsLevelToKeep**(*HYPRE_Solver* solver, HYPRE_Int level)

HYPRE_Int **HYPRE_MGRSetRelaxType**(*HYPRE_Solver* solver, HYPRE_Int relax_type)

(Optional) Set the relaxation type for F-relaxation. Currently supports the following flavors of relaxation types as described in the *BoomerAMGSetRelaxType: relax_type* 0 - 8, 13, 14, 18, 19, 98.

HYPRE_Int **HYPRE_MGRSetFRelaxMethod**(*HYPRE_Solver* solver, HYPRE_Int relax_method)

(Optional) Set the strategy for F-relaxation. Options for *relax_method* are:

- 0 : Single-level relaxation sweeps for F-relaxation as prescribed by *MGRSetRelaxType*
- 1 : Multi-level relaxation strategy for F-relaxation (V(1,0) cycle currently supported).

HYPRE_Int **HYPRE_MGRSetLevelFRelaxMethod**(*HYPRE_Solver* solver, HYPRE_Int *relax_method)

HYPRE_Int **HYPRE_MGRSetCoarseGridMethod**(*HYPRE_Solver* solver, HYPRE_Int *cg_method)

(Optional) Set the strategy for coarse grid computation. Options for *cg_method* are:

- 0 : Galerkin coarse grid computation using RAP.
- 1 : Non-Galerkin coarse grid computation with dropping strategy.

HYPRE_Int **HYPRE_MGRSetLevelFRelaxNumFunctions**(*HYPRE_Solver* solver, HYPRE_Int *num_functions)

(Optional) Set the number of functions for F-relaxation V-cycle. For problems like elasticity, one may want to perform coarsening and interpolation for block matrices. The number of functions corresponds to the number of scalar PDEs in the system.

HYPRE_Int **HYPRE_MGRSetRestrictType**(*HYPRE_Solver* solver, HYPRE_Int restrict_type)

(Optional) Set the strategy for computing the MGR restriction operator.

Options for *restrict_type* are:

- 0 : injection $[0I]$
- 1 : unscaled (not recommended)
- 2 : diagonal scaling (Jacobi)
- 3 : approximate inverse
- 4 : pAIR distance 1
- 5 : pAIR distance 2
- else : use classical modified interpolation

The default is injection.

HYPRE_Int **HYPRE_MGRSetLevelRestrictType**(*HYPRE_Solver* solver, HYPRE_Int *restrict_type)

HYPRE_Int **HYPRE_MGRSetNumRestrictSweeps**(*HYPRE_Solver* solver, HYPRE_Int nsweeps)

(Optional) Set number of restriction sweeps. This option is for *restrict_type* > 2.

HYPRE_Int **HYPRE_MGRSetInterpType**(*HYPRE_Solver* solver, HYPRE_Int interp_type)

(Optional) Set the strategy for computing the MGR interpolation operator. Options for *interp_type* are:

- 0 : injection $[0I]^T$
- 1 : unscaled (not recommended)
- 2 : diagonal scaling (Jacobi)
- 3 : classical modified interpolation

- 4 : approximate inverse
- else : classical modified interpolation

The default is diagonal scaling.

HYPRE_Int **HYPRE_MGRSetLevelInterpType**(*HYPRE_Solver* solver, HYPRE_Int *interp_type)

HYPRE_Int **HYPRE_MGRSetNumRelaxSweeps**(*HYPRE_Solver* solver, HYPRE_Int nsweeps)

(Optional) Set number of relaxation sweeps. This option is for the "single level" F-relaxation (*relax_method* = 0).

HYPRE_Int **HYPRE_MGRSetNumInterpSweeps**(*HYPRE_Solver* solver, HYPRE_Int nsweeps)

(Optional) Set number of interpolation sweeps. This option is for *interp_type* > 2.

HYPRE_Int **HYPRE_MGRSetFSolver**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* fine_grid_solver_solve, *HYPRE_PtrToParSolverFcn* fine_grid_solver_setup, *HYPRE_Solver* fsolver)

HYPRE_Int **HYPRE_MGRBuildAff**(HYPRE_ParCSRMatrix A, HYPRE_Int *CF_marker, HYPRE_Int debug_flag, HYPRE_ParCSRMatrix *A_ff)

HYPRE_Int **HYPRE_MGRSetCoarseSolver**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* coarse_grid_solver_solve, *HYPRE_PtrToParSolverFcn* coarse_grid_solver_setup, *HYPRE_Solver* coarse_grid_solver)

(Optional) Set the coarse grid solver. Currently uses BoomerAMG. The default, if not set, is BoomerAMG with default options.

Parameters

- **solver** – [IN] solver or preconditioner object
- **coarse_grid_solver_solve** – [IN] solve routine for BoomerAMG
- **coarse_grid_solver_setup** – [IN] setup routine for BoomerAMG
- **coarse_grid_solver** – [IN] BoomerAMG solver

HYPRE_Int **HYPRE_MGRSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

(Optional) Set the print level to print setup and solve information.

- 0 : no printout (default)
- 1 : print setup information
- 2 : print solve information
- 3 : print both setup and solve information

HYPRE_Int **HYPRE_MGRSetFrelaxPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_MGRSetCoarseGridPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

HYPRE_Int **HYPRE_MGRSetTruncateCoarseGridThreshold**(*HYPRE_Solver* solver, HYPRE_Real threshold)

(Optional) Set the threshold to compress the coarse grid at each level Use threshold = 0.0 if no truncation is applied. Otherwise, set the threshold value for dropping entries for the coarse grid. The default is 0.0.

HYPRE_Int **HYPRE_MGRSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Requests logging of solver diagnostics. Requests additional computations for diagnostic and similar data to be logged by the user. Default to 0 for do nothing. The latest residual will be available if logging > 1.

HYPRE_Int **HYPRE_MGRSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations if used as a solver. Set this to 1 if MGR is used as a preconditioner. The default is 20.

HYPRE_Int **HYPRE_MGRSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance for the MGR solver. Use tol = 0.0 if MGR is used as a preconditioner. The default is 1.e-6.

HYPRE_Int **HYPRE_MGRSetMaxGlobalsmoothIters**(*HYPRE_Solver* solver, HYPRE_Int smooth_iter)

(Optional) Determines how many sweeps of global smoothing to do. Default is 0 (no global smoothing).

HYPRE_Int **HYPRE_MGRSetGlobalsmoothType**(*HYPRE_Solver* solver, HYPRE_Int smooth_type)

(Optional) Determines type of global smoother. Options for *smooth_type* are:

- 0 : block Jacobi (default)
- 1 : Jacobi
- 2 : Gauss-Seidel, sequential (very slow!)
- 3 : Gauss-Seidel, interior points in parallel, boundary sequential (slow!)
- 4 : hybrid Gauss-Seidel or SOR, forward solve
- 5 : hybrid Gauss-Seidel or SOR, backward solve
- 6 : hybrid chaotic Gauss-Seidel (works only with OpenMP)
- 7 : hybrid symmetric Gauss-Seidel or SSOR
- 8 : Euclid (ILU)

HYPRE_Int **HYPRE_MGRGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

(Optional) Return the number of MGR iterations.

HYPRE_Int **HYPRE_MGRGetCoarseGridConvergenceFactor**(*HYPRE_Solver* solver, HYPRE_Real *conv_factor)

HYPRE_Int **HYPRE_MGRSetPMaxElmts**(*HYPRE_Solver* solver, HYPRE_Int P_max_elmts)

(Optional) Set the number of maximum points for interpolation operator.

HYPRE_Int **HYPRE_MGRGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *res_norm)

(Optional) Return the norm of the final relative residual.

ParCSR ILU Solver

(Parallel) ILU smoother

HYPRE_Int **HYPRE_ILUCreate**(*HYPRE_Solver* *solver)

Create a solver object

HYPRE_Int **HYPRE_ILUDestroy**(*HYPRE_Solver* solver)

Destroy a solver object

HYPRE_Int **HYPRE_ILUSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Setup the ILU solver or preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] object to be set up.
- **A** – [IN] ParCSR matrix used to construct the solver/preconditioner.
- **b** – right-hand-side of the linear system to be solved (Ignored by this function).
- **x** – approximate solution of the linear system to be solved (Ignored by this function).

HYPRE_Int **HYPRE_ILUSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

Solve the system or apply ILU as a preconditioner. If used as a preconditioner, this function should be passed to the iterative solver *SetPrecond* function.

Parameters

- **solver** – [IN] solver or preconditioner object to be applied.
- **A** – [IN] ParCSR matrix, matrix of the linear system to be solved
- **b** – [IN] right hand side of the linear system to be solved
- **x** – [OUT] approximated solution of the linear system to be solved

HYPRE_Int **HYPRE_ILUSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations if used as a solver. Set this to 1 if ILU is used as a preconditioner. The default is 20.

HYPRE_Int **HYPRE_ILUSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance for the ILU smoother. Use tol = 0.0 if ILU is used as a preconditioner. The default is 1.e-7.

HYPRE_Int **HYPRE_ILUSetLevelOfFill**(*HYPRE_Solver* solver, HYPRE_Int lfil)

(Optional) Set the level of fill k, for level-based ILU(k) The default is 0 (for ILU(0)).

HYPRE_Int **HYPRE_ILUSetMaxNnzPerRow**(*HYPRE_Solver* solver, HYPRE_Int nzmax)

(Optional) Set the max non-zeros per row in L and U factors (for ILUT) The default is 1000.

HYPRE_Int **HYPRE_ILUSetDropThreshold**(*HYPRE_Solver* solver, HYPRE_Real threshold)

(Optional) Set the threshold for dropping in L and U factors (for ILUT). Any fill-in less than this threshold is dropped in the factorization. The default is 1.0e-2.

HYPRE_Int **HYPRE_ILUSetDropThresholdArray**(*HYPRE_Solver* solver, HYPRE_Real *threshold)

(Optional) Set the array of thresholds for dropping in ILUT. B, E, and F correspond to upper left, lower left and upper right of 2 x 2 block decomposition respectively. Any fill-in less than threshold is dropped in the factorization.

- threshold[0] : threshold for matrix B.
- threshold[1] : threshold for matrix E and F.
- threshold[2] : threshold for matrix S (Schur Complement). The default is 1.0e-2.

HYPRE_Int **HYPRE_ILUSetNSHDropThreshold**(*HYPRE_Solver* solver, HYPRE_Real threshold)

(Optional) Set the threshold for dropping in Newton–Schulz–Hotelling iteration (NHS-ILU). Any entries less than this threshold are dropped when forming the approximate inverse matrix. The default is 1.0e-2.

HYPRE_Int **HYPRE_ILUSetNSHDropThresholdArray**(*HYPRE_Solver* solver, HYPRE_Real *threshold)

(Optional) Set the array of thresholds for dropping in Newton–Schulz–Hotelling iteration (for NHS-ILU). Any fill-in less than thresholds is dropped when forming the approximate inverse matrix.

- threshold[0] : threshold for Minimal Residual iteration (initial guess for NSH).
- threshold[1] : threshold for Newton–Schulz–Hotelling iteration.

The default is 1.0e-2.

HYPRE_Int **HYPRE_ILUSetSchurMaxIter**(*HYPRE_Solver* solver, HYPRE_Int ss_max_iter)

(Optional) Set maximum number of iterations for Schur System Solve. For GMRES-ILU, this is the maximum number of iterations for GMRES. The Krylov dimension for GMRES is set equal to this value to avoid restart. For NSH-ILU, this is the maximum number of iterations for NSH solve. The default is 5.

HYPRE_Int **HYPRE_ILUSetType**(*HYPRE_Solver* solver, HYPRE_Int ilu_type)

Set the type of ILU factorization.

Options for *ilu_type* are:

- 0 : BJ with ILU(k) (default, with k = 0)
- 1 : BJ with ILUT
- 10 : GMRES with ILU(k)
- 11 : GMRES with ILUT
- 20 : NSH with ILU(k)
- 21 : NSH with ILUT
- 30 : RAS with ILU(k)
- 31 : RAS with ILUT
- 40 : (nonsymmetric permutation) DDPQ-GMRES with ILU(k)
- 41 : (nonsymmetric permutation) DDPQ-GMRES with ILUT
- 50 : GMRES with RAP-ILU(0) using MILU(0) for P

HYPRE_Int **HYPRE_ILUSetLocalReordering**(*HYPRE_Solver* solver, HYPRE_Int reordering_type)

Set the type of reordering for the local matrix.

Options for *reordering_type* are:

- 0 : No reordering
- 1 : RCM (default)

HYPRE_Int **HYPRE_ILUSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int print_level)

(Optional) Set the print level to print setup and solve information.

- 0 : no printout (default)
- 1 : print setup information
- 2 : print solve information
- 3 : print both setup and solve information

HYPRE_Int **HYPRE_ILUSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Requests logging of solver diagnostics. Requests additional computations for diagnostic and similar data to be logged by the user. Default is 0, do nothing. The latest residual will be available if logging > 1.

HYPRE_Int **HYPRE_ILUGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

(Optional) Return the number of ILU iterations.

HYPRE_Int **HYPRE_ILUGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *res_norm)

(Optional) Return the norm of the final relative residual.

HYPRE_ParCSRMatrix **GenerateLaplacian**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Real *value)

HYPRE_ParCSRMatrix **GenerateLaplacian27pt**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Real *value)

HYPRE_ParCSRMatrix **GenerateLaplacian9pt**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int p, HYPRE_Int q, HYPRE_Real *value)

HYPRE_ParCSRMatrix **GenerateDifConv**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Real *value)

HYPRE_ParCSRMatrix **GenerateRotate7pt**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int p, HYPRE_Int q, HYPRE_Real alpha, HYPRE_Real eps)

HYPRE_ParCSRMatrix **GenerateVarDifConv**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Real eps, HYPRE_ParVector *rhs_ptr)

HYPRE_ParCSRMatrix **GenerateRSVarDifConv**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Real eps, HYPRE_ParVector *rhs_ptr, HYPRE_Int type)

float ***GenerateCoordinates**(MPI_Comm comm, HYPRE_BigInt nx, HYPRE_BigInt ny, HYPRE_BigInt nz, HYPRE_Int P, HYPRE_Int Q, HYPRE_Int R, HYPRE_Int p, HYPRE_Int q, HYPRE_Int r, HYPRE_Int coorddim)

HYPRE_Int **HYPRE_BoomerAMGSetPostInterpType**(*HYPRE_Solver* solver, HYPRE_Int post_interp_type)

HYPRE_Int **HYPRE_BoomerAMGSetJacobiTruncThreshold**(*HYPRE_Solver* solver, HYPRE_Real jacobi_trunc_threshold)

HYPRE_Int **HYPRE_BoomerAMGSetNumCRRelaxSteps**(*HYPRE_Solver* solver, HYPRE_Int num_CR_relax_steps)

HYPRE_Int **HYPRE_BoomerAMGSetCRRate**(*HYPRE_Solver* solver, HYPRE_Real CR_rate)

HYPRE_Int **HYPRE_BoomerAMGSetCRStrongTh**(*HYPRE_Solver* solver, HYPRE_Real CR_strong_th)

HYPRE_Int **HYPRE_BoomerAMGSetCRUseCG**(*HYPRE_Solver* solver, HYPRE_Int CR_use_CG)

HYPRE_Int **HYPRE_BoomerAMGSetISType**(*HYPRE_Solver* solver, HYPRE_Int IS_type)

ParCSR LOBPCG Eigensolver

These routines should be used in conjunction with the generic interface in *Eigensolvers*.

HYPRE_Int **HYPRE_ParCSRSetupInterpreter**(mv_InterfaceInterpreter *i)

Load interface interpreter. Vector part loaded with hypre_ParKrylov functions and multivector part loaded with mv_TempMultiVector functions.

HYPRE_Int **HYPRE_ParCSRSetupMatvec**(HYPRE_MatvecFunctions *mv)

Load Matvec interpreter with hypre_ParKrylov functions.

HYPRE_Int **HYPRE_ParCSRMultiVectorPrint**(void *x_, const char *fileName)

void ***HYPRE_ParCSRMultiVectorRead**(MPI_Comm comm, void *ii_, const char *fileName)

Functions

HYPRE_Int **HYPRE_SchwarzCreate**(*HYPRE_Solver* *solver)

HYPRE_Int **HYPRE_SchwarzDestroy**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_SchwarzSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_SchwarzSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_SchwarzSetVariant**(*HYPRE_Solver* solver, HYPRE_Int variant)

HYPRE_Int **HYPRE_SchwarzSetOverlap**(*HYPRE_Solver* solver, HYPRE_Int overlap)

HYPRE_Int **HYPRE_SchwarzSetDomainType**(*HYPRE_Solver* solver, HYPRE_Int domain_type)

HYPRE_Int **HYPRE_SchwarzSetRelaxWeight**(*HYPRE_Solver* solver, HYPRE_Real relax_weight)

HYPRE_Int **HYPRE_SchwarzSetDomainStructure**(*HYPRE_Solver* solver, HYPRE_CSRMatrix domain_structure)

HYPRE_Int **HYPRE_SchwarzSetNumFunctions**(*HYPRE_Solver* solver, HYPRE_Int num_functions)

HYPRE_Int **HYPRE_SchwarzSetDofFunc**(*HYPRE_Solver* solver, HYPRE_Int *dof_func)

HYPRE_Int **HYPRE_SchwarzSetNonSymm**(*HYPRE_Solver* solver, HYPRE_Int use_nonsymm)

HYPRE_Int **HYPRE_ParCSRCGNRCreate**(MPI_Comm comm, *HYPRE_Solver* *solver)

HYPRE_Int **HYPRE_ParCSRCGNRDestroy**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_ParCSRCGNRSetup**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRCGNRSolve**(*HYPRE_Solver* solver, HYPRE_ParCSRMatrix A, HYPRE_ParVector b, HYPRE_ParVector x)

HYPRE_Int **HYPRE_ParCSRCGNRSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

HYPRE_Int **HYPRE_ParCSRCGNRSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_ParCSRCGNRSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

HYPRE_Int **HYPRE_ParCSRCGNRSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_ParCSRCGNRSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToParSolverFcn* precond, *HYPRE_PtrToParSolverFcn* precondT, *HYPRE_PtrToParSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

HYPRE_Int **HYPRE_ParCSRCGNRGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data)

HYPRE_Int **HYPRE_ParCSRCGNRSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

HYPRE_Int **HYPRE_ParCSRCGNRGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

HYPRE_Int **HYPRE_ParCSRCGNRGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

8.7 Krylov Solvers

group **KrylovSolvers**

These solvers support many of the matrix/vector storage schemes in hypre. They should be used in conjunction with the storage-specific interfaces, particularly the specific Create() and Destroy() functions.

@memo A basic interface for Krylov solvers

Krylov Solvers

typedef struct hypre_Solver_struct ***HYPRE_Solver**

The solver object.

typedef struct hypre_Matrix_struct ***HYPRE_Matrix**

The matrix object.

typedef struct hypre_Vector_struct ***HYPRE_Vector**

The vector object.

typedef HYPRE_Int (***HYPRE_PtrToSolverFcn**)(*HYPRE_Solver*, *HYPRE_Matrix*, *HYPRE_Vector*, *HYPRE_Vector*)

typedef HYPRE_Int (***HYPRE_PtrToModifyPCFcn**)(*HYPRE_Solver*, HYPRE_Int, HYPRE_Real)

HYPRE_SOLVER_STRUCT

HYPRE_MATRIX_STRUCT

HYPRE_VECTOR_STRUCT

HYPRE_MODIFYPC

PCG Solver

HYPRE_Int **HYPRE_PCGSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_PCGSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_PCGSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the relative convergence tolerance.

HYPRE_Int **HYPRE_PCGSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The default convergence test is $< C*r, r > \leq$ max(relative _tolerance $^2 * < C*b, b >$, absolute _tolerance ^{2}).)

HYPRE_Int **HYPRE_PCGSetResidualTol**(*HYPRE_Solver* solver, HYPRE_Real rtol)

(Optional) Set a residual-based convergence tolerance which checks if $\|r_{old} - r_{new}\| < rtol\|b\|$. This is useful when trying to converge to very low relative and/or absolute tolerances, in order to bail-out before roundoff errors affect the approximation.

HYPRE_Int **HYPRE_PCGSetAbsoluteTolFactor**(*HYPRE_Solver* solver, HYPRE_Real abstolf)

HYPRE_Int **HYPRE_PCGSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_PCGSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_PCGSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_PCGSetTwoNorm**(*HYPRE_Solver* solver, HYPRE_Int two_norm)

(Optional) Use the two-norm in stopping criteria.

HYPRE_Int **HYPRE_PCGSetRelChange**(*HYPRE_Solver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_PCGSetRecomputeResidual**(*HYPRE_Solver* solver, HYPRE_Int recompute_residual)

(Optional) Recompute the residual at the end to double-check convergence.

HYPRE_Int **HYPRE_PCGSetRecomputeResidualP**(*HYPRE_Solver* solver, HYPRE_Int recompute_residual_p)

(Optional) Periodically recompute the residual while iterating.

HYPRE_Int **HYPRE_PCGSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_PCGSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_PCGSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_PCGGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_PCGGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_PCGGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_PCGGetTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_PCGGetResidualTol**(*HYPRE_Solver* solver, HYPRE_Real *rtol)

HYPRE_Int **HYPRE_PCGGetAbsoluteTolFactor**(*HYPRE_Solver* solver, HYPRE_Real *abstolf)

HYPRE_Int **HYPRE_PCGGetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real *cf_tol)

HYPRE_Int **HYPRE_PCGGetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int *stop_crit)

HYPRE_Int **HYPRE_PCGGetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_PCGGetTwoNorm**(*HYPRE_Solver* solver, HYPRE_Int *two_norm)

HYPRE_Int **HYPRE_PCGGetRelChange**(*HYPRE_Solver* solver, HYPRE_Int *rel_change)

HYPRE_Int **HYPRE_GMRESGetSkipRealResidualCheck**(*HYPRE_Solver* solver, HYPRE_Int *skip_real_r_check)

HYPRE_Int **HYPRE_PCGGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_PCGGetLogging**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_PCGGetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_PCGGetConverged**(*HYPRE_Solver* solver, HYPRE_Int *converged)

GMRES Solver

HYPRE_Int **HYPRE_GMRESSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_GMRESSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_GMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the relative convergence tolerance.

HYPRE_Int **HYPRE_GMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The convergence test is $\|r\| \leq$ max(relative _tolerance $*\|b\|$, absolute _tolerance).)

HYPRE_Int **HYPRE_GMRESSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_GMRESSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_GMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_GMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_GMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

(Optional) Set the maximum size of the Krylov space.

HYPRE_Int **HYPRE_GMRESSetRelChange**(*HYPRE_Solver* solver, HYPRE_Int rel_change)

(Optional) Additionally require that the relative difference in successive iterates be small.

HYPRE_Int **HYPRE_GMRESSetSkipRealResidualCheck**(*HYPRE_Solver* solver, HYPRE_Int skip_real_r_check)

(Optional) By default, hypre checks for convergence by evaluating the actual residual before returnig from GMRES (with restart if the true residual does not indicate convergence). This option allows users to skip the evaluation and the check of the actual residual for badly conditioned problems where restart is not expected to be beneficial.

HYPRE_Int **HYPRE_GMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_GMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_GMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_GMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_GMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_GMRESGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_GMRESGetTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_GMRESGetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_GMRESGetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real *cf_tol)

HYPRE_Int **HYPRE_GMRESGetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int *stop_crit)

HYPRE_Int **HYPRE_GMRESGetMinIter**(*HYPRE_Solver* solver, HYPRE_Int *min_iter)

HYPRE_Int **HYPRE_GMRESGetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_GMRESGetKDim**(*HYPRE_Solver* solver, HYPRE_Int *k_dim)

HYPRE_Int **HYPRE_GMRESGetRelChange**(*HYPRE_Solver* solver, HYPRE_Int *rel_change)

HYPRE_Int **HYPRE_GMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_GMRESGetLogging**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_GMRESGetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_GMRESGetConverged**(*HYPRE_Solver* solver, HYPRE_Int *converged)

FlexGMRES Solver

HYPRE_Int **HYPRE_FlexGMRESSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_FlexGMRESSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_FlexGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_FlexGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The convergence test is $\|r\| \leq$ max(relative _tolerance $*\|b\|$, absolute _tolerance).)

HYPRE_Int **HYPRE_FlexGMRESSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_FlexGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_FlexGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_FlexGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

(Optional) Set the maximum size of the Krylov space.

HYPRE_Int **HYPRE_FlexGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_FlexGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_FlexGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_FlexGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_FlexGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_FlexGMRESGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_FlexGMRESGetTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_FlexGMRESGetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real *cf_tol)

HYPRE_Int **HYPRE_FlexGMRESGetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int *stop_crit)

HYPRE_Int **HYPRE_FlexGMRESGetMinIter**(*HYPRE_Solver* solver, HYPRE_Int *min_iter)

HYPRE_Int **HYPRE_FlexGMRESGetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_FlexGMRESGetKDim**(*HYPRE_Solver* solver, HYPRE_Int *k_dim)

HYPRE_Int **HYPRE_FlexGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_FlexGMRESGetLogging**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_FlexGMRESGetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_FlexGMRESGetConverged**(*HYPRE_Solver* solver, HYPRE_Int *converged)

HYPRE_Int **HYPRE_FlexGMRESSetModifyPC**(*HYPRE_Solver* solver, *HYPRE_PtrToModifyPCFcn* modify_pc)

(Optional) Set a user-defined function to modify solve-time preconditioner attributes.

LGMRES Solver

HYPRE_Int **HYPRE_LGMRESSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_LGMRESSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system. Details on LGMRES may be found in A. H. Baker, E.R. Jessup, and T.A. Manteuffel, "A technique for accelerating the

convergence of restarted GMRES." SIAM Journal on Matrix Analysis and Applications, 26 (2005), pp. 962-984. LGMRES(m,k) in the paper corresponds to LGMRES(Kdim+AugDim, AugDim).

HYPRE_Int **HYPRE_LGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_LGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The convergence test is $\|r\| \leq$ max(relative _tolerance $*\|b\|$, absolute _tolerance).)

HYPRE_Int **HYPRE_LGMRESSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_LGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_LGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_LGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

(Optional) Set the maximum size of the approximation space (includes the augmentation vectors).

HYPRE_Int **HYPRE_LGMRESSetAugDim**(*HYPRE_Solver* solver, HYPRE_Int aug_dim)

(Optional) Set the number of augmentation vectors (default: 2).

HYPRE_Int **HYPRE_LGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_LGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_LGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_LGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_LGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_LGMRESGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_LGMRESGetTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_LGMRESGetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real *cf_tol)

HYPRE_Int **HYPRE_LGMRESGetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int *stop_crit)

HYPRE_Int **HYPRE_LGMRESGetMinIter**(*HYPRE_Solver* solver, HYPRE_Int *min_iter)

HYPRE_Int **HYPRE_LGMRESGetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_LGMRESGetKDim**(*HYPRE_Solver* solver, HYPRE_Int *k_dim)

HYPRE_Int **HYPRE_LGMRESGetAugDim**(*HYPRE_Solver* solver, HYPRE_Int *k_dim)

HYPRE_Int **HYPRE_LGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_LGMRESGetLogging**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_LGMRESGetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_LGMRESGetConverged**(*HYPRE_Solver* solver, HYPRE_Int *converged)

COGMRES Solver

HYPRE_Int **HYPRE_COGMRESSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_COGMRESSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_COGMRESSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_COGMRESSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The convergence test is $\|r\| \leq$ max(relative _tolerance $*\|b\|$, absolute _tolerance).)

HYPRE_Int **HYPRE_COGMRESSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_COGMRESSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_COGMRESSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_COGMRESSetKDim**(*HYPRE_Solver* solver, HYPRE_Int k_dim)

(Optional) Set the maximum size of the Krylov space.

HYPRE_Int **HYPRE_COGMRESSetUnroll**(*HYPRE_Solver* solver, HYPRE_Int unroll)

(Optional) Set number of unrolling in mass funcyions in COGMRES Can be 4 or 8. Default: no unrolling.

HYPRE_Int **HYPRE_COGMRESSetCGS**(*HYPRE_Solver* solver, HYPRE_Int cgs)

(Optional) Set the number of orthogonalizations in COGMRES (at most 2).

HYPRE_Int **HYPRE_COGMRESSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_COGMRESSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_COGMRESSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_COGMRESGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_COGMRESGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_COGMRESGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_COGMRESGetTol**(*HYPRE_Solver* solver, HYPRE_Real *tol)

HYPRE_Int **HYPRE_COGMRESGetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real *cf_tol)

HYPRE_Int **HYPRE_COGMRESGetMinIter**(*HYPRE_Solver* solver, HYPRE_Int *min_iter)

HYPRE_Int **HYPRE_COGMRESGetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int *max_iter)

HYPRE_Int **HYPRE_COGMRESGetKDim**(*HYPRE_Solver* solver, HYPRE_Int *k_dim)

HYPRE_Int **HYPRE_COGMRESGetUnroll**(*HYPRE_Solver* solver, HYPRE_Int *unroll)

HYPRE_Int **HYPRE_COGMRESGetCGS**(*HYPRE_Solver* solver, HYPRE_Int *cgs)

HYPRE_Int **HYPRE_COGMRESGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_COGMRESGetLogging**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_COGMRESGetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int *level)

HYPRE_Int **HYPRE_COGMRESGetConverged**(*HYPRE_Solver* solver, HYPRE_Int *converged)

HYPRE_Int **HYPRE_COGMRESSetModifyPC**(*HYPRE_Solver* solver, *HYPRE_PtrToModifyPCFcn* modify_pc)

(Optional) Set a user-defined function to modify solve-time preconditioner attributes.

BiCGSTAB Solver

HYPRE_Int **HYPRE_BiCGSTABDestroy**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_BiCGSTABSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_BiCGSTABSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_BiCGSTABSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_BiCGSTABSetAbsoluteTol**(*HYPRE_Solver* solver, HYPRE_Real a_tol)

(Optional) Set the absolute convergence tolerance (default is 0). If one desires the convergence test to check the absolute convergence tolerance *only*, then set the relative convergence tolerance to 0.0. (The convergence test is $\|r\| \leq$ max(relative _tolerance $*\|b\|$, absolute _tolerance).)

HYPRE_Int **HYPRE_BiCGSTABSetConvergenceFactorTol**(*HYPRE_Solver* solver, HYPRE_Real cf_tol)

HYPRE_Int **HYPRE_BiCGSTABSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_BiCGSTABSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_BiCGSTABSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_BiCGSTABSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use.

HYPRE_Int **HYPRE_BiCGSTABSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_BiCGSTABSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

HYPRE_Int **HYPRE_BiCGSTABGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_BiCGSTABGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_BiCGSTABGetResidual**(*HYPRE_Solver* solver, void *residual)

Return the residual.

HYPRE_Int **HYPRE_BiCGSTABGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

CGNR Solver

HYPRE_Int **HYPRE_CGNRDestroy**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_CGNRSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Prepare to solve the system. The coefficient data in *b* and *x* is ignored here, but information about the layout of the data may be used.

HYPRE_Int **HYPRE_CGNRSolve**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Solve the system.

HYPRE_Int **HYPRE_CGNRSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the convergence tolerance.

HYPRE_Int **HYPRE_CGNRSetStopCrit**(*HYPRE_Solver* solver, HYPRE_Int stop_crit)

HYPRE_Int **HYPRE_CGNRSetMinIter**(*HYPRE_Solver* solver, HYPRE_Int min_iter)

HYPRE_Int **HYPRE_CGNRSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_CGNRSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precondT, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use. Note that the only preconditioner available in hypre for use with CGNR is currently BoomerAMG. It requires to use Jacobi as a smoother without CF smoothing, i.e. relax_type needs to be set to 0 or 7 and relax_order needs to be set to 0 by the user, since these are not default values. It can be used with a relaxation weight for Jacobi, which can significantly improve convergence.

HYPRE_Int **HYPRE_CGNRSetLogging**(*HYPRE_Solver* solver, HYPRE_Int logging)

(Optional) Set the amount of logging to do.

HYPRE_Int **HYPRE_CGNRGetNumIterations**(*HYPRE_Solver* solver, HYPRE_Int *num_iterations)

Return the number of iterations taken.

HYPRE_Int **HYPRE_CGNRGetFinalRelativeResidualNorm**(*HYPRE_Solver* solver, HYPRE_Real *norm)

Return the norm of the final relative residual.

HYPRE_Int **HYPRE_CGNRGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

8.8 Eigensolvers

group **Eigensolvers**

These eigensolvers support many of the matrix/vector storage schemes in hypre. They should be used in conjunction with the storage-specific interfaces.

@memo A basic interface for eigensolvers

LOBPCG Eigensolver

HYPRE_Int **HYPRE_LOBPCGCreate**(mv_InterfaceInterpreter *interpreter, HYPRE_MatvecFunctions *mvfunctions, *HYPRE_Solver* *solver)

LOBPCG constructor.

HYPRE_Int **HYPRE_LOBPCGDestroy**(*HYPRE_Solver* solver)

LOBPCG destructor.

HYPRE_Int **HYPRE_LOBPCGSetPrecond**(*HYPRE_Solver* solver, *HYPRE_PtrToSolverFcn* precond, *HYPRE_PtrToSolverFcn* precond_setup, *HYPRE_Solver* precond_solver)

(Optional) Set the preconditioner to use. If not called, preconditioning is not used.

HYPRE_Int **HYPRE_LOBPCGGetPrecond**(*HYPRE_Solver* solver, *HYPRE_Solver* *precond_data_ptr)

HYPRE_Int **HYPRE_LOBPCGSetup**(*HYPRE_Solver* solver, *HYPRE_Matrix* A, *HYPRE_Vector* b, *HYPRE_Vector* x)

Set up *A* and the preconditioner (if there is one).

HYPRE_Int **HYPRE_LOBPCGSetupB**(*HYPRE_Solver* solver, *HYPRE_Matrix* B, *HYPRE_Vector* x)

(Optional) Set up *B*. If not called, B = I.

HYPRE_Int **HYPRE_LOBPCGSetupT**(*HYPRE_Solver* solver, *HYPRE_Matrix* T, *HYPRE_Vector* x)

(Optional) Set the preconditioning to be applied to Tx = b, not Ax = b.

HYPRE_Int **HYPRE_LOBPCGSolve**(*HYPRE_Solver* solver, mv_MultiVectorPtr y, mv_MultiVectorPtr x, HYPRE_Real *lambda)

Solve A x = lambda B x, y'x = 0.

HYPRE_Int **HYPRE_LOBPCGSetTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the absolute convergence tolerance.

HYPRE_Int **HYPRE_LOBPCGSetRTol**(*HYPRE_Solver* solver, HYPRE_Real tol)

(Optional) Set the relative convergence tolerance.

HYPRE_Int **HYPRE_LOBPCGSetMaxIter**(*HYPRE_Solver* solver, HYPRE_Int max_iter)

(Optional) Set maximum number of iterations.

HYPRE_Int **HYPRE_LOBPCGSetPrecondUsageMode**(*HYPRE_Solver* solver, HYPRE_Int mode)

Define which initial guess for inner PCG iterations to use: *mode* = 0: use zero initial guess, otherwise use RHS.

HYPRE_Int **HYPRE_LOBPCGSetPrintLevel**(*HYPRE_Solver* solver, HYPRE_Int level)

(Optional) Set the amount of printing to do to the screen.

utilities_FortranMatrix ***HYPRE_LOBPCGResidualNorms**(*HYPRE_Solver* solver)

utilities_FortranMatrix ***HYPRE_LOBPCGResidualNormsHistory**(*HYPRE_Solver* solver)

utilities_FortranMatrix ***HYPRE_LOBPCGEigenvaluesHistory**(*HYPRE_Solver* solver)

HYPRE_Int **HYPRE_LOBPCGIterations**(*HYPRE_Solver* solver)

void **hypre_LOBPCGMultiOperatorB**(void *data, void *x, void *y)

void **lobpcg_MultiVectorByMultiVector**(mv_MultiVectorPtr x, mv_MultiVectorPtr y, utilities_FortranMatrix *xy)

BIBLIOGRAPHY

[AsFa1996] S. F. Ashby and R. D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, September 1996. Also available as LLNL Technical Report UCRL-JC-122359.

[BFKY2011] A. Baker, R. Falgout, T. Kolev, and U. M. Yang. Multigrid smoothers for ultra-parallel computing. *SIAM J. on Sci. Comp.*, 33:2864–2887, 2011. Also available as LLNL technical report LLLNL-JRNL-473191.

[BaFY2006] A.H. Baker, R.D. Falgout, and U.M. Yang. An assumed partition algorithm for determining processor inter-communication. *Parallel Computing*, 32:394–414, 2006.

[BaKY2010] A. Baker, T. Kolev, and U. M. Yang. Improving algebraic multigrid interpolation operators for linear elasticity problems. *Numer. Linear Algebra Appl.*, 17:495–517, 2010. Also available as LLNL technical report LLLNL-JRNL-412928.

[BKRHSMTY2021] Luc Berger-Vergiat, Brian Kelley, Sivasankaran Rajamanickam, Jonathan Hu, Katarzyna Swirydowicz, Paul Mullowney, Stephen Thomas, Ichitaro Yamazaki. Two-Stage Gauss–Seidel Preconditioners and Smoothers for Krylov Solvers on a GPU cluster. https://arxiv.org/abs/2104.01196.

[BLOPEWeb] BLOPEX, parallel preconditioned eigenvalue solvers. http://code.google.com/p/blopex/.

[BrFJ2000] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000. Special issue on the Fifth Copper Mountain Conference on Iterative Methods. Also available as LLNL technical report UCRL-JC-130720.

[Chow2000] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21:1804–1822, 2000.

[ClEA1999] R. L. Clay et al. An annotated reference guide to the Finite Element Interface (FEI) specification, Version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, Livermore, CA, 1999.

[CMakeWeb] CMake, a cross-platform open-source build system. http://www.cmake.org/.

[DFNY2008] H. De Sterck, R. Falgout, J. Nolting, and U. M. Yang. Distance-two interpolation for parallel algebraic multigrid. *Numer. Linear Algebra Appl.*, 15:115–139, 2008. Also available as LLNL technical report UCRL-JRNL-230844.

[DeYH2004] H. De Sterck, U. M. Yang, and J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27:1019–1039, 2006. Also available as LLNL technical report UCRL-JRNL-206780.

[FaJo2000] R. D. Falgout and J. E. Jones. Multigrid on massively parallel architectures. In E. Dick, K. Riemslagh, and J. Vierendeels, editors, *Multigrid Methods VI*, volume 14 of *Lecture Notes in Computational Science and Engineering*, pages 101–107, Berlin, 2000. Springer. Proc. of the Sixth European Multigrid Conference held in Gent, Belgium, September 27-30, 1999. Also available as LLNL technical report UCRL-JC-133948.

[FaJY2004] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 267–294. Springer–Verlag, 2006. Also available as LLNL technical report UCRL-JRNL-205459.

[FaJY2005] R. D. Falgout, J. E. Jones, and U. M. Yang. Conceptual interfaces in hypre. *Future Generation Computer Systems*, 22:239–251, 2006. Special issue on PDE software. Also available as LLNL technical report UCRL-JC-148957.

[FaSc2014] Robert D. Falgout and Jacob B. Schroder. Non-galerkin coarse grids for algebraic multigrid. *SIAM J. Sci. Comput.*, 36(3):309–334, 2014.

[GrKo2015] A. Grayver and Tz. Kolev. Large-scale 3D geoelectromagnetic modeling using parallel adaptive high-order finite element method. *Geophysics*, 80(6):E277–E291, 2015. Also available as LLNL technical report LLNL-JRNL-665742.

[GrMS2006a] M. Griebel, B. Metsch, and M. A. Schweitzer. Coarse grid classification: A parallel coarsening scheme for algebraic multigrid methods. *Numerical Linear Algebra with Applications*, 13(2–3):193–214, 2006. Also available as SFB 611 preprint No. 225, Universität Bonn, 2005.

[GrMS2006b] M. Griebel, B. Metsch, and M. A. Schweitzer. Coarse grid classification - Part II: Automatic coarse grid agglomeration for parallel AMG. Preprint No. 271, Sonderforschungsbereich 611, Universität Bonn, 2006.

[HeYa2002] V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(5):155–177, 2002. Also available as LLNL technical report UCRL-JC-141495.

[HiXu2006] R. Hiptmair and J. Xu. Nodal auxiliary space preconditioning in $H(curl)$ and $H(div)$ spaces. *Numer. Math.*, 2006.

[HyPo1999] D. Hysom and A. Pothen. Efficient parallel computation of ILU(k) preconditioners. In *Proceedings of Supercomputing '99*. ACM, November 1999. Published on CDROM, ISBN #1-58113-091-0, ACM Order #415990, IEEE Computer Society Press Order # RS00197.

[HyPo2001] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, 2001.

[KaKu1998] G. Karypis and V. Kumar. Parallel threshold-based ILU factorization. Technical Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, 1998.

[Knya2001] A. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[KLAO2007] A. Knyazev, I. Lashuk, M. Argentati, and E. Ovchinnikov. Block locally optimal preconditioned eigenvalue xolvers (blopex) in hypre and petsc. *SIAM J. Sci. Comput.*, 25(5):2224–2239, 2007.

[KoVa2009] Tz. Kolev and P. Vassilevski. Parallel auxiliary space AMG for $H(curl)$ problems. *J. Comput. Math.*, 27:604–623, 2009. Special issue on Adaptive and Multilevel Methods in Electromagnetics. UCRL-JRNL-237306.

[JoLe2006] J. Jones and B. Lee. A multigrid method for variable coefficient maxwell's equations. *SIAM J. Sci. Comput.*, 27:1689–1708, 2006.

[McCo1989] S. F. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*, volume 6 of *Frontiers in Applied Mathematics*. SIAM Books, Philadelphia, 1989.

[MoRS1998] J.E. Morel, Randy M. Roberts, and Mikhail J. Shashkov. A local support-operators diffusion discretization scheme for quadrilateral r-z meshes. *J. Comp. Physics*, 144:17–51, 1998.

[RuSt1987] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[Scha1998] S. Schaffer. A semi-coarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM J. Sci. Comput.*, 20(1):228–242, 1998.

[Stue1999] K. Stüben. Algebraic multigrid (AMG): an introduction with applications. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*. Academic Press, 2001.

[Umpire] Umpire: Managing Heterogeneous Memory Resources. https://github.com/LLNL/Umpire.

[VaMB1996] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, 1996.

[VaBM2001] P. Vaněk, M. Brezina, and J. Mandel. Convergence of algebraic multigrid based on smoothed aggregation. *Numerische Mathematik*, 88:559–579, 2001.

[VaYa2014] P. Vassilevski and U. M. Yang. Reducing communication in algebraic multigrid using additive variants. *Numer. Linear Algebra Appl.*, 21:275–296, 2014. Also available as LLNL technical report LLLNL-JRNL-637872.

[Yang2004] U. M. Yang. On the use of relaxation parameters in hybrid smoothers. *Numerical Linear Algebra with Applications*, 11:155–172, 2004.

[Yang2005] U. M. Yang. Parallel algebraic multigrid methods - high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 209–236. Springer-Verlag, 2006. Also available as LLNL technical report UCRL-BOOK-208032.

[Yang2010] U. M. Yang. On long range interpolation operators for aggressive coarsening. *Numer. Linear Algebra Appl.*, 17:453–472, 2010. Also available as LLNL technical report LLLNL-JRNL-417371.

INDEX

G

H

L