User's Manual

Software Version: 1.2.0
Date: 2000/09/21

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

UCRL-MA-137155 DR

Contents

Chapter 1

Introduction

hypre is a software library for solving large, sparse linear systems of equations on massively parallel computers. The library was created with the primary goal of providing users with advanced parallel preconditioners. Issues of robustness, ease of use, flexibility, and interoperability also play an important role.

1.1 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. *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 and element-based algebraic 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 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 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* utilizes the GNU Autoconf package to allow 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`.
- **Interfaces in multiple languages provide greater flexibility for applications:** *hypre* contains interfaces for both Fortran and C users.

1.2 Assumptions and Limitations

- *hypre* **is designed for large, sparse, linear systems on parallel computers.** 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.
- **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.

Chapter 2

Getting Started

Before writing any code:

1. **Choose a conceptual interface (see Sections 2.2 and 2.3).** Generally, the choice is fairly obvious. A structured-grid interface is clearly inappropriate for an unstructured-grid application. It is desirable to use a more specific interface if appropriate, e.g., the linear-algebraic interface is usable from any type of grid but will involve much more user work and prevent access to some grid-type-specific preconditioners.

2. **Choose your desired solver strategy.** For the typical user, this will mean a single Krylov method and a single preconditioner.

3. **Look up matrix requirements for each solver and preconditioner.** Each specific solver and preconditioner has requirements from the input matrix. This information is provided in several places: Chapter 7, the *hypre* Reference Manual, and *hypre* header files).

4. **Choose a matrix 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*:

1. **Build any necessary auxiliary structures for your chosen conceptual interface.** This includes, e.g., the grid and stencil structures for the structured-grid interface.

2. **Build the matrix and vectors through your chosen conceptual interface.** Each conceptual interface provides a series of calls for entering information about your problem into *hypre*.

3. **Build solvers and preconditioners by giving them the input matrix.**

4. **Set solver parameters (this is optional).** Some parameters like convergence tolerance are the same across solvers, while others are solver specific.

5. **Call the solve function for the solver.**

6. **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.

2.1 A Simple Example

The following code serves as a simple example of the usage of *hypre*. In this example, the structured-grid interface (discussed in Chapter 3) is used to enter the problem into *hypre*, and the PFMG Multigrid solver is used to solve the system.

This example and all other examples in this manual are written in C, but *hypre* also supports Fortran. See Section 8.2 for details.

```
/*-----------------------------------------------------------
 * Set up the matrix
 *-----------------------------------------------------------*/

HYPRE_StructGridCreate(MPI_COMM_WORLD, dim, &grid);
HYPRE_StructGridSetExtents(grid, ilower, iupper);
...
HYPRE_StructGridAssemble(grid);

HYPRE_StructStencilCreate(dim, stencil_size, &stencil);
HYPRE_StructStencilSetElement(stencil, 0, offset0);
...

HYPRE_StructMatrixCreate(MPI_COMM_WORLD, grid, stencil, &A);
HYPRE_StructMatrixInitialize(A);
HYPRE_StructMatrixSetBoxValues(A, ilower, iupper, nelts, elts, Avalues);
...
HYPRE_StructMatrixAssemble(A);

/*-----------------------------------------------------------
 * Set up the right-hand side and initial guess
 *-----------------------------------------------------------*/

HYPRE_StructVectorCreate(MPI_COMM_WORLD, grid, &b);
HYPRE_StructVectorInitialize(b);
HYPRE_StructVectorSetBoxValues(b, ilower, iupper, bvalues);
...
HYPRE_StructVectorAssemble(b);
```

```
HYPRE_StructVectorCreate(MPI_COMM_WORLD, grid, &x);
HYPRE_StructVectorInitialize(x);
HYPRE_StructVectorSetBoxValues(x, ilower, iupper, xvalues);
...
HYPRE_StructVectorAssemble(x);

/*-----------------------------------------------------------
 * Set up solver
 *-----------------------------------------------------------*/

HYPRE_StructPFMGCreate(MPI_COMM_WORLD, &solver);
HYPRE_StructPFMGSetMaxIter(solver, 50);       /* optional */
HYPRE_StructPFMGSetTol(solver, 1.0e-06);      /* optional */
HYPRE_StructPFMGSetup(solver, A, b, x);

/*-----------------------------------------------------------
 * Solve the linear system
 *-----------------------------------------------------------*/

HYPRE_StructPFMGSolve(solver, A, b, x);

/*-----------------------------------------------------------
 * Get solution info
 *-----------------------------------------------------------*/

HYPRE_StructVectorGetBoxValues(x, ilower, iupper, xvalues);
...

/*-----------------------------------------------------------
 * Free up memory
 *-----------------------------------------------------------*/

HYPRE_StructPFMGDestroy(solver);
HYPRE_StructGridDestroy(grid);
HYPRE_StructStencilDestroy(stencil);
HYPRE_StructMatrixDestroy(A);
HYPRE_StructVectorDestroy(b);
HYPRE_StructVectorDestroy(x);
```

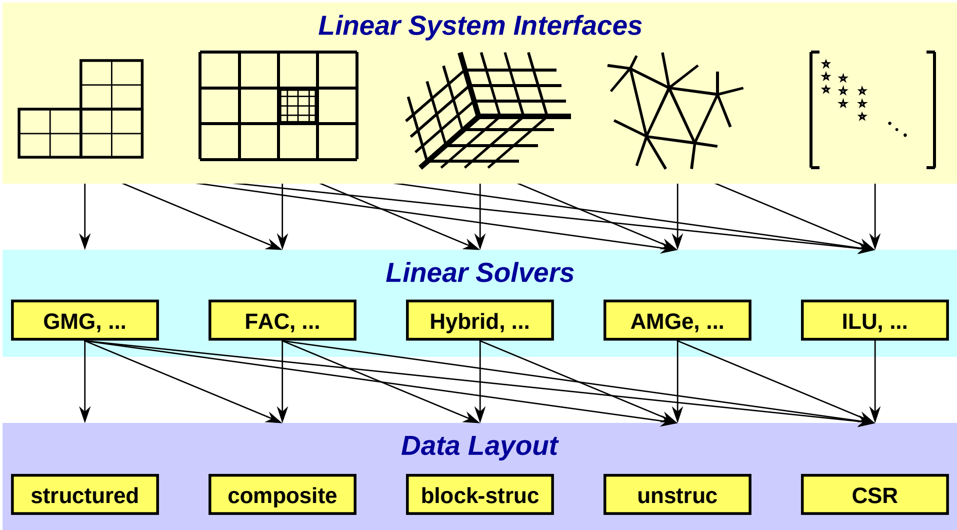

Figure 2.1: Graphic illustrating the notion of conceptual interfaces. All of these elements are not necessarily in *hypre*.

2.2 What are conceptual interfaces?

The top row of Figure 2.1 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. These conceptual interfaces are intended to represent the way that applications developers naturally think of their linear problem, and 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. For example, applications that use structured grids (such as in the left-most interface in the Figure 2.1) 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 second row of Figure 2.1 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.

The third row of Figure 2.1 is a list of data layouts or matrix/vector storage schemes. The relationship between linear solver and storage scheme is similar to that of interface and linear solver.

2.3 Which conceptual interface should I use?

hypre currently supports four conceptual interfaces:

- **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 3 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 4 for details. **NOTE:** This is a very new interface and should be used with caution as it matures.

- **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 (www.z.ca.sandia.gov/fei). See Chapter 5 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 6 for details.

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.

Chapter 3

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 5 and 6.

Figure 3.1 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). 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

$$\begin{cases} \nabla^2 u = f, & \text{in the domain,} \\ u = 0, & \text{on the boundary.} \end{cases} \tag{3.1}$$

Assume (3.1) is discretized using standard 5-pt finite-volumes on the uniform grid pictured in 3.1, and assume that the problem data is distributed across two processes as depicted.

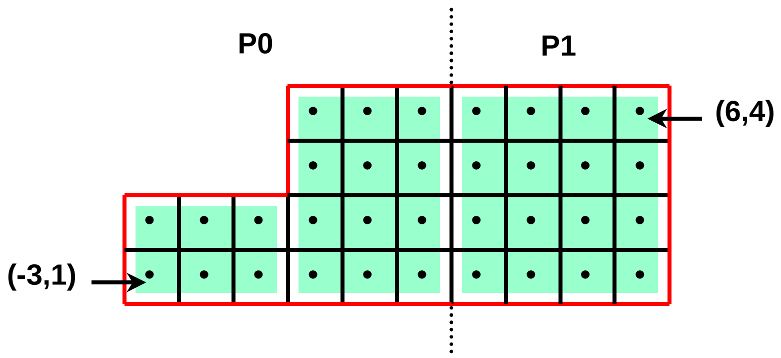

Figure 3.1: An example 2D structured grid, distributed accross two processors.

3.1 Setting Up the Grid

The grid is described via a global *index space*, i.e., via integer tuples (triples in 3D). 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. Each process describes that portion of the grid that it "owns", one *box* at a time. For example, in the figure, the global grid can be described in terms of three boxes, two owned by process 0, and one owned by process 1. A box is described in terms of a lower and upper index.

On process 0, the following code will set up the grid shown in the figure (the code for process 1 is similar).

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

HYPRE_StructGridCreate(MPI_COMM_WORLD, 2, &grid);

HYPRE_StructGridSetExtents(grid, ilower[0], iupper[0]);
HYPRE_StructGridSetExtents(grid, ilower[1], iupper[1]);

HYPRE_StructGridAssemble(grid);
```

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), and finalizes the grid assembly, making the grid "ready to use".

3.2 Setting Up the Stencil

The geometry of the discretization stencil is described by an array of integer tuples in 2D (triples in 3D), each representing a relative offset (in index space) from some gridpoint on the grid. For example, the geometry of the 5-pt stencil for the example problem being considered can be represented in the following way:

$$\begin{bmatrix} & (0,1) & \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) & \end{bmatrix} \equiv \begin{bmatrix} & S_4 & \\ S_1 & S_0 & S_2 \\ & S_3 & \end{bmatrix}. \tag{3.2}$$

In (3.2), the $(0,0)$ entry represents the "center" coefficient, and is the 0th entry in the array (S_0). The $(0,-1)$ entry represents the "south" coefficient, and is the 3rd entry in the array (S_3). And so on.

On process 0 or 1, the following code will set up the stencil in (3.2). The stencil must be the same on all processes.

```
HYPRE_StructStencil  stencil;
int                  offsets[5][2] = {{0,0}, {-1,0}, {1,0}, {0,-1}, {0,1}};
int                  s;

HYPRE_StructStencilCreate(2, 5, &stencil);

for (s = 0; s < 5; s++)
{
   HYPRE_StructStencilSetElement(stencil, s, offsets[s]);
}
```

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

3.3 Setting Up the Matrix

The matrix is set up in terms of the grid and stencil objects described in Sections 3.1 and 3.2. 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):

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}. \tag{3.3}$$

On process 0, the following code will set up matrix values associated with the center (S_0) and south (S_3) stencil entries in (3.2) / (3.3) (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 */
...

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 (i.e., must be called on all processes), 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 3.1, 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 */
...
```

3.4 Setting Up the Right-Hand-Side Vector

The right-hand-side vector is set up similarly to the matrix set up described in Section 3.3 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 will set 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);

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 (i.e., must be called on all processes), and finalizes the vector assembly, making the vector "ready to use".

3.5 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 (3.1) on the grid pictured in Figure 3.1. 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);
```

Note that symmetric storage may or may not actually be used, depending on the underlying storage scheme. Currently in *hypre*, symmetric storage is always used when indicated.

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 3.2 to an "appropriate" 3-pt stencil, then set matrix coefficients (as in Section 3.3) for these three stencil elements *only*. For example, we could use the following stencil

$$\left[\begin{array}{ccc} & (0,1) & \\ & (0,0) & (1,0) \\ & & \end{array}\right] \equiv \left[\begin{array}{ccc} & S_2 & \\ & S_0 & S_1 \\ & & \end{array}\right]. \tag{3.4}$$

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

Chapter 4

Semi-Structured-Grid System Interface (SStruct)

In order to get access to the most efficient and scalable solvers for semi-structured-grid applications (applications with grids that are mostly structured, but with some unstructured features), users should use the `SStruct` interface described in this chapter. This interface also provides access 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 5 and 6.

Figure 4.1 gives an example of the type of grid currently supported by the `SStruct` interface. The grid is composed of three "parts". There is a single cell-centered variable across the entire grid, except at the grid cell marked with a square, which has one additional cell-centered variable. 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,
3. set up the graph,
4. set up the matrix,
5. set up the right-hand-side vector.

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

$$\begin{cases} \nabla^2 u = f, & \text{in the domain,} \\ u = 0, & \text{on the boundary.} \end{cases} \tag{4.1}$$

Assume (3.1) is discretized using standard 9-pt finite-volumes on the grid pictured in 4.1, and assume that the problem data is distributed across three processes as depicted. In the figure, each grid part is distributed onto a different process, but this need not be the case; each grid part may also be distributed across several processes. Assume for simplicity and illustration that there is a

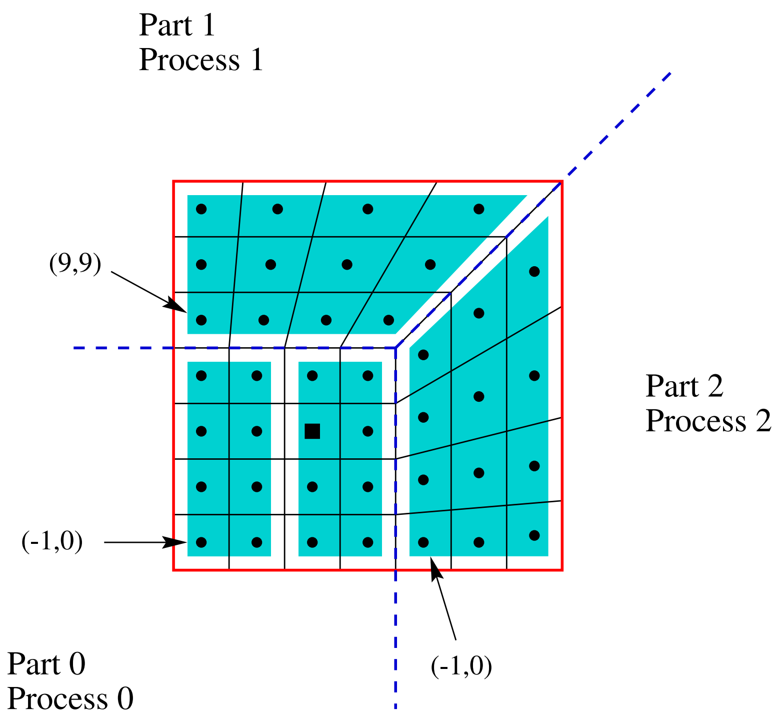

Figure 4.1: An example 2D block-structured grid, distributed accross three processes.

single coupling between the first and second variables in grid part 0 at index (1,2). In general, this variable may also be coupled to variables at neighboring indices.

4.1 Setting Up the Grid

On process 0, the following code will set up the grid shown in the figure (the code for processes 1 and 2 is similar).

```
HYPRE_SStructGrid     grid;
int                   ilower[2][2] = {{-1, 0}, {1, 0}};
int                   iupper[2][2] = {{ 0, 3}, {2, 3}};
HYPRE_SStructVariable vars[1]      = {HYPRE_SSTRUCT_VARIABLE_CELL};

int                   addindex[2]  = {1,2};
HYPRE_SStructVariable addvars[1]   = {HYPRE_SSTRUCT_VARIABLE_CELL};

HYPRE_SStructGridCreate(MPI_COMM_WORLD, 2, 3, &grid);

HYPRE_SStructGridSetExtents(grid, 0, ilower[0], iupper[0]);
```

```
HYPRE_SStructGridSetExtents(grid, 0, ilower[1], iupper[1]);

HYPRE_SStructGridSetVariables(grid, 0, 1, vars);
HYPRE_SStructGridAddVariables(grid, 0, addindex, 1, addvars);

HYPRE_SStructGridAssemble(grid);
```

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 `SetVariables()` routine sets the variables on a grid part, and the `AddVariables()` routine adds variables to a grid part index (in the above example, there are two cell-centered variables at index (1,2)). The `Assemble()` routine is a collective call (i.e., must be called on all processes), and finalizes the grid assembly, making the grid "ready to use".

4.2 Setting Up the Stencil

The geometry of the discretization stencil is described by an array of integer tuples in 2D (triples in 3D), each representing a relative offset (in index space) from some gridpoint on the grid. For example, the geometry of the 9-pt stencil for the example problem being considered can be represented in the following way:

$$\begin{bmatrix} (-1,1) & (0,1) & (1,1) \\ (-1,0) & (0,0) & (1,0) \\ (-1,-1) & (0,-1) & (1,-1) \end{bmatrix} \equiv \begin{bmatrix} S_7 & S_4 & S_8 \\ S_1 & S_0 & S_2 \\ S_5 & S_3 & S_6 \end{bmatrix}. \tag{4.2}$$

In (3.2), the $(0,0)$ entry represents the "center" coefficient, and is the 0th entry in the array (S_0). The $(0,-1)$ entry represents the "south" coefficient, and is the 3rd entry in the array (S_3). And so on.

On process 0, 1, or 2, the following code will set up the stencil in (3.2).

```
HYPRE_SStructStencil stencil;
int                  s;
int                  offsets[9][2] = {{0,0},
                                      {-1, 0}, { 1, 0}, { 0,-1}, { 0, 1}};
                                      {-1,-1}, { 1,-1}, {-1, 1}, { 1, 1}};

HYPRE_SStructStencilCreate(2, 9, &stencil);

for (s = 0; s < 9; s++)
   HYPRE_SStructStencilSetEntry(stencil, s, offsets[s], 0);
```

The `Create()` routine creates an empty 2D, 9-pt stencil object. The `SetEntry()` routine defines the geometry of the stencil, and assigns the array numbers for each of the stencil entries. None of the calls are collective calls.

4.3 Setting Up the Graph

The graph will represent the nonzero structure of the matrix in Section 4.4 below. It is defined in terms of the grid and stencil objects described in Sections 4.1 and 4.2 above.

On process 0, the following code will set up the graph for the example problem.

```
HYPRE_SStructGraph graph;
int                addindex[9][2]   = {{ 2,0},{2,1},{2,2},{2,3},
                                       {-1,3},{0,3},{1,3},{2,3},
                                       { 1,2}};
int                addentries[9][2] = {{-1,0},{-1,1},{-1,2},{-1,3},
                                       { 9,9},{10,9},{11,9},{12,9},
                                       { 1,2}};

HYPRE_SStructGraphCreate(MPI_COMM_WORLD, grid, &graph);

HYPRE_SStructGraphSetStencil(graph, 0, 0, stencil);

/* Add graph edge entries at grid part boundaries */
HYPRE_SStructGraphAddEntries(graph, 0, addindex[0], 0, 2, 2, &addentries[0], 0);
HYPRE_SStructGraphAddEntries(graph, 0, addindex[1], 0, 3, 2, &addentries[0], 0);
HYPRE_SStructGraphAddEntries(graph, 0, addindex[2], 0, 3, 2, &addentries[1], 0);
...

/* Add graph edge entries at index (1,2) */
HYPRE_SStructGraphAddEntries(graph, 0, addindex[8], 0, 1, 0, &addentries[8], 1);
HYPRE_SStructGraphAddEntries(graph, 0, addindex[8], 1, 1, 0, &addentries[8], 0);
```

4.4 Setting Up the Matrix

The matrix is set up in terms of the graph object described in Section 4.3 above. The coefficients associated with each graph 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):

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}. \tag{4.3}$$

On process 0, the following code will set up matrix values associated with the center (S_0) and south (S_3) stencil entries in (3.2) / (3.3). Matrix values associated with the non-stencil graph entries are also set up.

```
HYPRE_SStructMatrix  A;
double               values[32];
int                  sindices[2] = {0,3};
int                  gindices[4] = {9,10,11,0};
int                  i;

HYPRE_SStructMatrixCreate(MPI_COMM_WORLD, graph, &A);
HYPRE_SStructMatrixInitialize(A);

for (i = 0; i < 32; i += 2)
{
   values[i]   =  8.0;
   values[i+1] = -1.0;
}
HYPRE_SStructMatrixSetBoxValues(A, 0, ilower[0], iupper[0], 0,
                                2, sindices, values);

/* set values at non-stencil graph entries */
for (i = 0; i < 3; i++)
{
   values[i] = -1.0;
}
HYPRE_SStructMatrixSetValues(A, 0, addindex[0], 0, 2, gindices, values);
HYPRE_SStructMatrixSetValues(A, 0, addindex[1], 0, 3, gindices, values);
HYPRE_SStructMatrixSetValues(A, 0, addindex[2], 0, 3, gindices, values);
...
HYPRE_SStructMatrixSetValues(A, 0, addindex[7], 0, 1, gindices, values);
HYPRE_SStructMatrixSetValues(A, 0, addindex[7], 1, 1, &gindices[3], values);

/* zero out coefficients that reach outside of the domain or grid part */
...

/* set boundary conditions at domain boundaries, */
...

HYPRE_SStructMatrixAssemble(A);
```

4.5 Setting Up the Right-Hand-Side Vector

The right-hand-side vector is set up similarly to the matrix set up described in Section 4.4 above. The main difference is that there is no graph.

Chapter 5

Finite Element Interface (FEI)

The finite element interface (FEI) defines a linear solver interface for finite element applications. For information on how to use this interface, see [4]. This chapter describes the iterative methods and preconditioners in the *hypre* implementation of this interface.

Solving a linear system from a finite element problem consists of four steps in the FEI:

1. Initialize the structure of the finite-element data, including loading the element connectivity data

2. Load the element or super-element stiffness matrices and forcing terms

3. Set solver parameters and solve the linear system

4. Retrieve the solution to the linear system

Parameters to the *hypre* solvers are specified by calling

```
void FEI_parameters(int sysHandle, int numParams, char **paramStrings);
```

where `sysHandle` is an identifier for the linear system being solved, `numParams` is the number of parameter strings, and `paramStrings` is an array of null-terminated strings with the format: "*parameter_name value*". For example, setting the preconditioner can be accomplished by:

```
char **paramStrings[1];
paramStrings[0] = (char *) malloc(64*sizeof(char));
strcpy(paramStrings[0], "preconditioner parasails");
FEI_parameters(sysHandle, 1, paramStrings);
```

All possible parameters are listed in section 5.2 below.

A linear system is then solved by calling

```
void FEI_iterateToSolve(int sysHandle);
```

5.1 Iterative methods and preconditioners available

5.1.1 Iterative methods

1. conjugate gradient (CG)
2. generalized minimal residual (GMRES) methods

5.1.2 Preconditioners

1. diagonal
2. parallel incomplete LU with threshold (PILUT)
3. parallel algebraic multigrid (BoomerAMG)
4. parallel sparse approximate inverse (ParaSails)

5.2 Parameters

solver XXX - where XXX specifies one of cg, gmres, superlu, or superlux. The default is gmres.

preconditioner XXX - where XXX is one of diagonal, pilut, parasails, or boomeramg. The default is diagonal.

gmres-dim XXX - where XXX is an integer specifying the value of m in restarted GMRES(m). The default value is 50.

maxIterations XXX - where XXX is an integer specifying the maximum number of iterations permitted for CG or GMRES. The default value is 1000.

tolerance XXX - where XXX is a floating point number specifying the termination criterion for CG or GMRES. The default value is 1.0E-10.

pilut-row-size XXX - where XXX is an integer specifying the maximum number of nonzeros kept in the formation of imcomplete L and U). If this is not called, a value will be selected based on the sparsity of the matrix.

pilut-drop-tol XXX - where XXX is a floating point number specifying the threshold to drop small entries in L and U. The default value is 0.0.

superlu-ordering XXX - where XXX specifies on of natural or mmd (minimum degree ordering). This ordering is used to minimize the number of nonzeros generated in the LU decomposition. The default is natural ordering.

superlu-scale XXX - where XXX specifies on of y (perform row and column scalings before decomposition) or n. The default is no scaling.

amg-coarsen-type XXX - where XXX specifies one of falgout, ruge, or default coarsening for BOOMERAMG.

amg-num-sweeps XXX - where XXX is an integer specifying the number of pre- and post-smoothing at each level of BOOMERAMG. The default is one pre- and one post-smoothings.

amg-relax-type XXX - where XXX is one of jacobi (Damped Jacobi), gs-slow (sequential Gauss-Seidel), gs-fast (Gauss-Seidel on interior nodes), hybrid, or direct. The default is hybrid.

amg-relax-weight XXX - where XXX is a floating point number between 0 and 1 specifying the damping factor for BOOMERAMG's damped Jacobi smoother. The default value is 0.5.

amg-strong-threshold XXX - where XXX is a floating point number between 0 and 1 specifying the threshold used to determine strong coupling in BOOMERAMG's coasening. The default value is 0.25.

parasails-threshold XXX - where XXX is a floating point number between 0 and 1 specifying the threshold used to prune small entries in setting up the sparse approximate inverse. The default value is 0.0.

parasails-nlevels XXX - where XXX is an integer larger than 0 specifying the desired sparsity of the approximate inverse. The default value is 1.

Chapter 6

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*, but not to the specialized solvers that require more problem information.

6.1 IJ Matrix Interface

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

```
MPI_Comm        comm;
HYPRE_IJMatrix *ij_matrix;
int             global_m, global_n;
int             num_values;
int             row;
int            *cols;
double         *values;

HYPRE_IJMatrixCreate(comm, &ij_matrix, global_m, global_n);
HYPRE_IJMatrixSetLocalStorageType(ij_matrix, HYPRE_PARCSR);
HYPRE_IJMatrixInitialize(ij_matrix);

/* set matrix coefficients a row at a time */
HYPRE_IJMatrixSetValues(ij_matrix, num_values, row, cols, values);
...

HYPRE_IJMatrixAssemble(ij_matrix);
```

The Create() routine creates an empty global_m by global_n matrix object that lives on the comm communicator. The SetLocalStorageType routine sets the underlying matrix storage type to HYPRE_PARCSR (this is the only storage 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 Set routines mentioned later in this chapter and in the Reference Manual, should be called before this step. The SetValues() routine sets the matrix coefficients for a specified row and list of cols (num_values is the number of column coefficients being set). The Assemble() routine is a collective call (i.e., must be called on all processes), and finalizes the matrix assembly, making the matrix "ready to use".

6.2 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;
int             global_n;
int             num_values;
int            *global_indices;
int            *value_indices;
double         *values;

HYPRE_IJVectorCreate(comm, &ij_vector, global_n);
HYPRE_IJVectorSetLocalStorageType(ij_vector, HYPRE_PARCSR);
HYPRE_IJVectorInitialize(ij_vector);

/* set vector values */
HYPRE_IJVectorSetLocalComponents(ij_vector, num_values,
                                 global_indices, value_indices, values);
...

HYPRE_IJVectorAssemble(ij_vector);
```

The Create() routine creates an empty global_n-vector object that lives on the comm communicator. The SetLocalStorageType 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 optional Set routines mentioned later in this chapter and in the Reference Manual, should be called before this step. The SetLocalComponents() routine sets the vector coefficients for a specified list of global_indices (`num_values` is the number of values being set). The `Assemble()` routine is a collective call (i.e., must be called on all processes), and finalizes the vector assembly, making the vector "ready to use".

Chapter 7

Solvers and Preconditioners

There are several solvers available in *hypre* via different conceptual interfaces (see Table 7.1). 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.

	System Interfaces			
Solvers	Struct	SStruct	FEI	IJ
Jacobi	X			
SMG	X			
PFMG	X			
BoomerAMG		X	X	X
ParaSails		X	X	X
PILUT		X	X	X
PCG	X		X	X
GMRES		X	X	X

Table 7.1: Current solver availability via *hypre* conceptual interfaces.

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 coefficients of the matrix, e.g. the GetRow function in the `HYPRE_DistributedMatrix` class.

2. **Choose parameters that control construction of the preconditioner/solver.** Parameters are chosen through the `Set` calls provided by the solver. As is true throughout *hypre*, all parameters have reasonable defaults if not chosen. Note that in *hypre*, convergence criteria can be chosen after the preconditioner/solver has been setup.

3. **Pass the preconditioner to the solver.** For solvers that are not preconditioned, this step is omitted. The preconditioner is passed through the `SetPreconditioner` call. The object passed in as the preconditioner must be a fully constructed `Solver`.

4. **Set up the solver.** In the procedural C interface for *hypre*, this is the `Setup` call. In the ESI-compliant interface for *hypre*, this is the `GetConstructedObject` from the solver builder interface.

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. Generally speaking, Krylov solvers default to a tolerance-based convergence criteria, while solvers used traditionally as preconditioners default to convergence based on the number of iterations (usually a single iteration). Users are free to change these, though care must be taken. Some Krylov solvers, for example, are not guaranteed to converge unless the preconditioner is a constant operator, which is only satisfied for iterative methods if a fixed number of iterations (typically one) is taken. For these solvers, a preconditioner with a tolerance-based convergence criteria will not usually work.

2. **Apply the solver to the right hand side.** For the procedural C interface, this is the `Solve` function.

7.1 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,

$$\nabla \cdot (D\nabla u) + \sigma u = f \tag{7.1}$$

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 [6, 2, 5] 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.

7.2 PFMG

PFMG is a parallel semicoarsening multigrid solver similar to SMG. See [1, 5] 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.

7.3 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 [3].

7.3.1 Synopsis

```
#include "HYPRE_parcsr_ls.h"

int HYPRE_ParCSRParaSailsCreate(MPI_Comm comm, HYPRE_Solver *solver,
  int symmetry);
int HYPRE_ParCSRParaSailsSetParams(HYPRE_Solver solver,
  double thresh, int nlevel, double filter);
int HYPRE_ParCSRParaSailsSetup(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_ParCSRParaSailsSolve(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_ParCSRParaSailsStats(HYPRE_Solver solver);
int HYPRE_ParCSRParaSailsDestroy(HYPRE_Solver solver);
```

The accuracy and cost of ParaSails are parameterized by the real *thresh* and integer *nlevels* parameters, $0 \le thresh \le 1$, $0 \le nlevels$. 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_ParCSRParaSailsSetParams`.

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 [3].

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.

7.3.2 Interface functions

A ParaSails solver `solver` is returned with

```
int HYPRE_ParCSRParaSailsCreate(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 7.3.3.

Parameters for setting up the preconditioner are specified using

```
int HYPRE_ParCSRParaSailsSetParams(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:

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

When `thresh` < 0, then a threshold is selected such that $-$`thresh` represents the fraction of the 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 fraction of the 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.

7.3.3 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).

7.4 *PILUT*: Parallel Incomplete Factorization

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 **??** 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.

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.

7.5 BoomerAMG

BoomerAMG is a parallel implementation of algebraic multigrid. It can be used both as a solver or as a preconditioner. The user can choose between various different parallel coarsening techniques and relaxation schemes. The following coarsening techniques are available:

- the Cleary-Luby-Jones-Plassman (CLJP) coarsening,
- various variants of the classical Ruge-Stueben (RS) coarsening algorithm, and
- the Falgout coarsening which is a combination of CLJP and the classical RS coarsening algorithm.

The following relaxation techniques are available:

- weighted Jacobi relaxation,
- a hybrid Gauss-Seidel / Jacobi relaxation scheme, and
- sequential Gauss-Seidel relaxation.

7.5.1 Synopsis

The 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:

```
#include "HYPRE_parcsr_ls.h"

int HYPRE_BoomerAMGCreate(HYPRE_Solver *solver);

<set certain parameters if desired >

int HYPRE_BoomerAMGSetup(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_BoomerAMGSolve(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_BoomerAMGDestroy(HYPRE_Solver solver);
```

7.5.2 Interface functions

Parameters for setting up the code are specified using the following routines:

HYPRE_BoomerAMGSetMaxLevels

```
int HYPRE_BoomerAMGSetMaxLevels( HYPRE_Solver solver, int max_levels);
```

max_levels defines the maximal number of multigrid levels allowed. The default is 25.

HYPRE_BoomerAMGSetMaxIter

```
int HYPRE_BoomerAMGSetMaxIter( HYPRE_Solver solver, int max_iter);
```

max_iter defines the maximal number of iterations allowed. The default is 20.

HYPRE_BoomerAMGSetTol

```
int HYPRE_BoomerAMGSetTol( HYPRE_Solver solver, double tol);
```

tol defines the tolerance needed for the stopping criterion $\|b - Ax^{(n)}\|_2/\|b\|_2 \leq$ tol. The default for tol is 10^{-7}.

HYPRE_BoomerAMGSetStrongThreshold

```
int HYPRE_BoomerAMGSetStrongThreshold( HYPRE_Solver solver,
double strong_threshold);
```

A point i is strongly connected to j, if $-a_{i,j} \geq \theta \max_{j \neq i} a_{ij}$, where the strong threshold θ is a value between 0 and 1. Weak connections are usually ignored when determining the next lower level. Consequently, choosing a larger strong threshold leads in general to smaller coarse grids, but worse convergence rates. The default value for θ is 0.25, which appears to be a good choice for 2-dimensional problems. A better choice for 3-dimensional problems appears to be 0.5. However, the choice of the strength threshold is problem dependent and therefore there could be better choices than the two suggested ones.

HYPRE_BoomerAMGSetMaxRowSum

```
int HYPRE_BoomerAMGSetMaxRowSum( HYPRE_Solver solver, double max_row_sum);
```

This feature leads to a more efficient treatment of very diagonally dominant portions of the matrix. If the absolute row sum of row i weighted by the diagonal is greater than max_row_sum all dependencies of variable i are set to be weak. This feature can be switched off by setting max_row_sum to 1.0. The default is 0.9.

HYPRE_BoomerAMGSetCoarsenType

```
int HYPRE_BoomerAMGSetCoarsenType( HYPRE_Solver solver, int coarsen_type  );
```

coarsen_type defines the coarsening used. The following options are possible:

- 0 CLJP-coarsening
- 1 Ruge-Stueben coarsening without boundary treatment
- 3 Ruge-Stueben coarsening with a 3rd 'second' pass on the boundaries
- 6 Falgout coarsening (default)

HYPRE_BoomerAMGSetMeasureType

```
int HYPRE_BoomerAMGSetMeasureType( HYPRE_Solver solver, int measure_type  );
```

measure_type defines whether local (measure_type = 0, default) or global measures (measure_type = 1) are used within the coarsening algorithm. This feature is ignored for the CLJP and the Falgout coarsening.

HYPRE_BoomerAMGSetNumGridSweeps

```
int HYPRE_BoomerAMGSetNumGridSweeps( HYPRE_Solver solver, int* num_grid_sweeps );
```

num_grid_sweeps[k] defines the number of sweeps over the grid on the fine grid (k=0), the down cycle (k=1), the up cycle (k=2) and the coarse grid (k=3).

HYPRE_BoomerAMGSetGridRelaxType

```
int HYPRE_BoomerAMGSetGridRelaxType( HYPRE_Solver solver, int* grid_relax_type );
```

grid_relax_type[k] defines the relaxation used on the fine grid (k=0), the down cycle (k=1), the up cycle (k=2) and the coarse grid (k=3). The following options are possible for grid_relax_type[k]:

- 0 weighted Jacobi
- 3 Gauss-Seidel / Jacobi hybrid method (default)
- 9 Gaussian elimination (only for the coarsest level (k=3), not recommended if the system on the coarsest level is large)

HYPRE_BoomerAMGSetGridRelaxPoints

```
int HYPRE_BoomerAMGSetGridRelaxPoints( HYPRE_Solver solver,
int** grid_relax_points);
```

grid_relax_points[k][l] defines which points are to be relaxed during the (l+1)-th sweep on the fine grid (k=0), the down cycle (k=1), the up cycle (k=2) and the coarse grid (k=3), e.g. if grid_relax_points[1][0] is -1, all points marked -1 (which are in general fine points) are relaxed on the first sweep of the down cycle.

HYPRE_BoomerAMGSetRelaxWeight

```
int HYPRE_BoomerAMGSetRelaxWeight( HYPRE_Solver solver, double* relax_weight);
```

defines the relaxation weights used on each level, if weighted Jacobi is used as relaxation method. The default relaxation weight is 1.0 on each level.

HYPRE_BoomerAMGSetIOutDat

```
int HYPRE_BoomerAMGSetIOutDat( HYPRE_Solver solver, int ioutdat);
```

where ioutdat determines whether statistics information is generated and printed. The information is printed to standard output. The following options are possible:

0	no output (default)
1	matrix statistics (includes information on interpolation operators and matrices generated on each level)
2	cycle statistics (includes residuals generated during solve phase)
3	matrix and cycle statistics

Chapter 8

Additional Information

8.1 Building the Code

Usually, *hypre* can be built by simply typing `configure` followed by `make` in the top-level source directory.

8.1.1 Code configuration

To automatically generate machine specific makefiles, type `configure` in the top level directory. The configure script is a portable script generated by GNU Autoconf. It runs a series of tests to determine characteristics of the machine on which it is running, and it uses the results of the these tests to produce the machine specific makefiles, called 'Makefile', from template files called 'Makefile.in' in each directory. Once the makefiles are produced you can run make as you would with any other makefile.

The configure script primarily does the following things:

- selects a compiler
- provides either optimization or debugging options for the compiler
- finds the headers and libraries for MPI

The configure script makes these decisions based on a hierarchical check. First, it attempts to identify the machine on which it is running as a specific supported machine. Next it will try to identify the architecture as a supported architecture. If both of these fail, generic default decisions are made by the script. However, the script does have some command-line options that can give you control over the choices it will make. You can type `configure --help` to see the list of all of the command-line options to configure. This is the best resource for information on configure options. Below is some additional helpful information. Be aware that not all command line options have been tested on all machines and architectures, even supported machines and architectures.

–with- options Each `--with-` option that is listed in `configure --help` also was a `--without-` option (usually the default, but not always). Additionally, all `--with-` options have a

`--with-`*option*=*pathname*. This is not a supported feature for all `--with-` options however and may have no effect on configuration.

Compilers If you want to choose a compiler then is it recommended that you choose all (C, C++, Fortran) compilers.

Compiler Flags To choose optimization or debug, use `--enable-opt` (default) or `--enable-debug`. For other compiler flags use the `--with-CFLAGS` option.

BLAS The optimum BLAS for the platform should be obtained without specification of a configure option. In other words, by default, the systems native optimized BLAS library will the automatically chosen. To specify another BLAS library available on a platform, use `--with-BLAS=`*pathname* or `--with-BLAS=`*link list*. To configure and compile without BLAS use the `--without-BLAS` option.

Configure automatically generates a file `HYPRE_config.h` that includes all the header files found to be necessary by configure. This file may be used to see how a compiled version of the library was configured and may also be included by the user in his/her own code.

8.1.2 Linking into another program

A program linking with *hypre* must be compiled with `-I$HYPRE_DIR/include` and linked with `-L$HYPRE_DIR/lib -l`*hypre library name...* `-l`*hypre library name...*, where `$HYPRE_DIR` is the directory where *hypre* is installed. Additionally, any other libraries to which *hypre* is linked must also be linked to by the users application. For example, the BLAS library or PETSc library are often (but not always) linked in by *hypre* and would also need to be linked in by the users application.

It may be useful to reference the `Makefile` in the `test` subdirectory. This makefile is designed to build test applications that link with and call *hypre*. All include and linking flags that are used by *hypre* and needed by these test applications get exported to this file by the `configure` script.

8.2 Calling *hypre* from Fortran

Fortran 77 subroutine arguments always pass copies of the argument addresses upon execution of the subroutine call. This is referred to as call-by-address or call-by-reference. In the called subroutine, the memory space at the argument address can be altered, but the calling address cannot be altered.

C function parameters, whether pointers (addresses) or not, are directly copied upon entry to the function. This is referred to as call-by-value. Altering the copied C parameter in the called function has no effect on the parameter in the calling function. A pointer parameter in C can achieve the same effect as call-by-reference, and by this mechanism the languages can interoperate in a straightforward manner.

Portability across typical platforms is currently achieved with the specific Fortran-calling-C mapping:

calling Fortran argument type		called C function parameter type
(addr of) `integer*8`	⟶	`long int*`
(addr of) `integer`	⟶	`int*`
(addr of) `character`	⟶	`char*`
(addr of) `double precision`	⟶	`double*`

In particular, C-type `long int*` points to a space that can hold an address, a space which happens to be the size of that allocated by a Fortran `integer*8` declaration. In orther words, C can hold an address in a `long int` and assign that address to a Fortran `integer*8` memory space (using `long int*` call-by-value).

Addresses in `integer*8` variables might not have originated in a Fortran call, and if they address inhomogeneously typed collections of data (e.g. C structures), Fortran might not have enough information to dereference them. But Fortran can still pass around such addresses, and in particular hand them back and forth between various C functions. The *hypre* Fortran interface makes extensive use of this technique, as with the `addr` variable in the following general example:

Generalized example:

Fortran calling:

```
integer*8        addr
integer          intg, ierr
character        charact
double precision double_precis

call subroutine_name(addr, intg, charact, double_precis, ierr)
```

The interface is designed so that the label `subroutine_name` is exactly the label of the *hypre* C-function[1]. *hypre* interlanguage C-wrappers currently account for interlanguage linking issues involving appended underscores. The C-wrappers also accomodate Fortran subroutine name length limitations. Fortran variable `ierr` allows error handling through the interface.

Specific example:

Fortran calling:

```
integer*8        IJmatrix,
integer          num_nonzero_coefs, row_index
integer          col_indices(MAX_NUM_COLS), ierr
double precision coefs(MAX_NUM_COLS)
        .
```

[1]For C function names under 32 characters in length, the Fortran name is the same as the C name. For C function names over 31 characters, the Fortran name is condensed to less than 32 characters (see reference manual for Fortran name in such a case).

```
        .
        .

   call HYPRE_IJMatrixInsertRow(IJmatrix, num_nonzero_coefs, row_index,
                                col_indices, coefs, ierr)
```

C called:

```
int HYPRE_IJMatrixInsertRow( HYPRE_IJMatrix IJmatrix, int n,
                             int row, const int *cols,
                             const double *values)

{ int ierr = 0;
        .
        .
        .

  return(ierr); }
```

Since the C function name has less than 32 characters, Fortran uses the same name in the subroutine call.

The typical implementation above might not have the portability needed over a larger set of platforms. The previous mapping might not be universal. Furthermore, the implementation is hand-programmed, slow, and prone to human error. It will probably be replaced by a more automated implementation in accordance with the *Babel* project.

8.3 Bug Reporting

hypre has an automated bug reporting mechanism in place that may be used as a resource for submitting bugs, desired features, and documentation problems, as well as querying the status of previous reports. Access http://www-casc.llnl.gov/bugs for full bug tracking details or to submit or query a bug report. When using the CASC bug reporting site for the first time, click on "Open a new Bugzilla account" under the "User login account management" heading.

Bibliography

[1] 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.

[2] 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.

[3] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21:1804–1822, 2000.

[4] 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.

[5] 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.

[6] 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.