User's Manual

Software Version: 1.11.1b
Date: 2006/05/30 09:30:00

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

This work was produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL. The rights of the Federal Government are reserved under Contract 48 subject to the restrictions agreed upon by the DOE and University as allowed under DOE Acquisition Letter 97-1.

UCRL-MA-137155 DR

Contents

Chapter 1

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. 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 interfaces, 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`.

- **Interfaces in multiple languages provide greater flexibility for applications:** *hypre* is written in C and utilizes Babel to provide interfaces for users of Fortran 77, Fortran 90, C++, Python, and Java. For more information on Babel, see `http://www.llnl.gov/CASC/components/babel.html`.

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 or the solvers available in the library are available from the *hypre* web page at `http://www.llnl.gov/CASC/linear_solvers/`.

- **Mailing Lists:** There are three *hypre* mailing lists that can be subscribed to through the *hypre* web page at `http://www.llnl.gov/CASC/linear_solvers`:

1. *hypre-announce* (hypre-announce@lists.llnl.gov): The development team uses this list to announce new general releases of *hypre*. It cannot be posted to by users.
2. *hypre-beta-announce* (hypre-beta-announce@lists.llnl.gov): The development team uses this list to announce new beta releases of *hypre*. It cannot be posted to by users.
3. *hypre-users* (hypre-users@lists.llnl.gov): This list is for *hypre* users to communicate with the development team and each other. It can be posted to by any member of the list.

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. 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 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 1.1 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 the Figure 1.1) 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 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

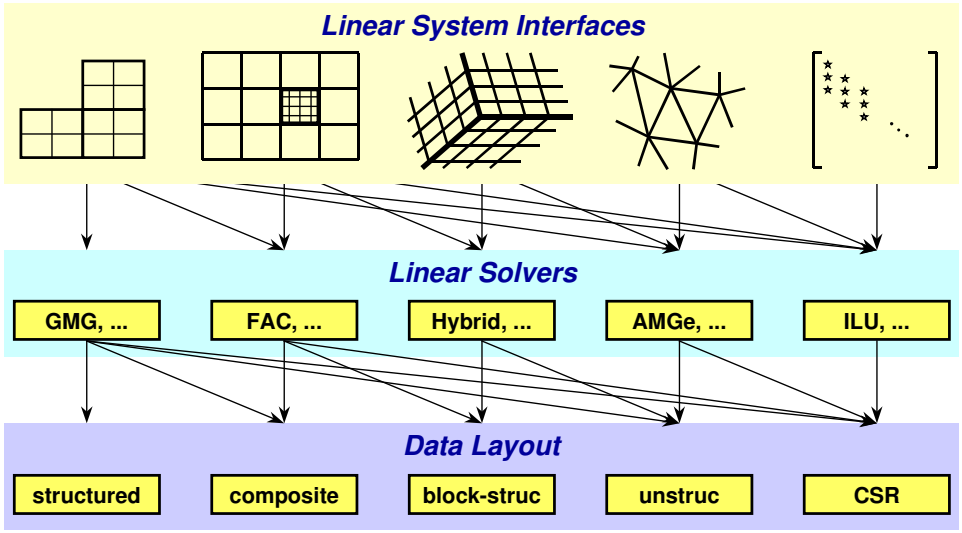


Figure 1.1: Graphic illustrating the notion of conceptual interfaces.

of nonzeros at each grid point. This interface supports only a single unknown per grid point. See Chapter 2 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 3 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 (`http://.z.ca.sandia.gov/fei`). See Chapter 4 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 5 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. For example, the second row of Figure 1.1 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 6 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 1.1 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:

1. Choose a conceptual interface.
2. Choose your desired solver strategy.
3. Look up matrix requirements for each solver and preconditioner.
4. 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*:

1. **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.
2. **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*.
3. **Build solvers and preconditioners and set solver parameters (optional).** Some parameters like convergence tolerance are the same across solvers, while others are solver specific.
4. **Call the solve function for the solver.**

5. **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 2

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 4 and 5.

Figure 2.1 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.

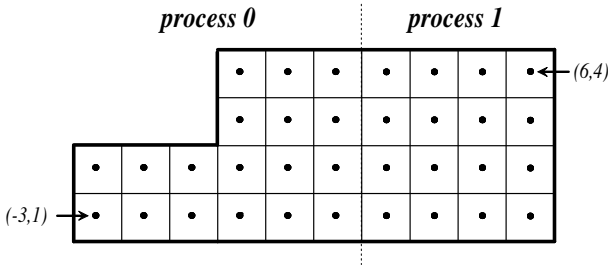


Figure 2.1: An example 2D structured grid, distributed accross two processors.

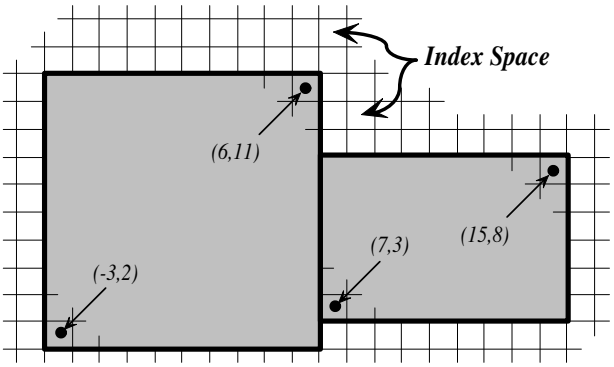


Figure 2.2: A box is a collection of abstract cell-centered indices, described by its minimum and maximum indices. Here, two boxes are illustrated.

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

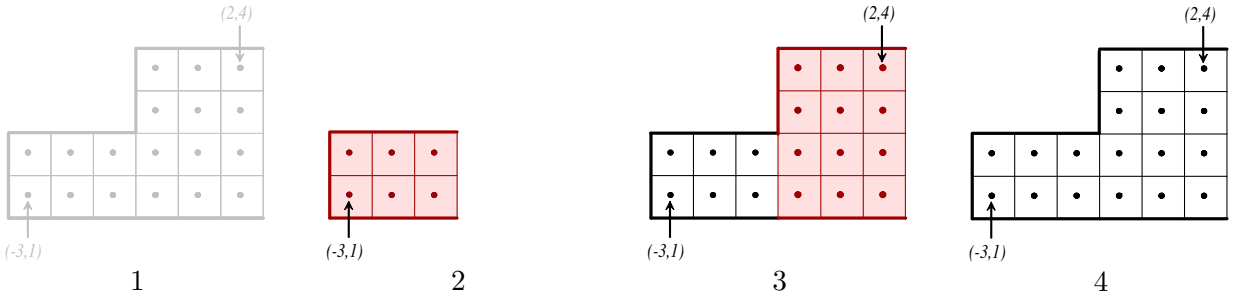
$$\begin{cases} \nabla^2 u = f, & \text{in the domain,} \\ u = 0, & \text{on the boundary.} \end{cases} \tag{2.1}$$

Assume (2.1) is discretized using standard 5-pt finite-volumes on the uniform grid pictured in 2.1, 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 2.2). 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 2.1 can be described in terms of three boxes, two owned by process 0, and one owned by process 1. Figure 2.3 shows the code for setting up the grid on process 0 (the code for process 1 is similar). The "icons" at the top of the figure 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".




```
    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);

    /* 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);
```

Figure 2.3: Code on process 0 for setting up the grid in Figure 2.1.

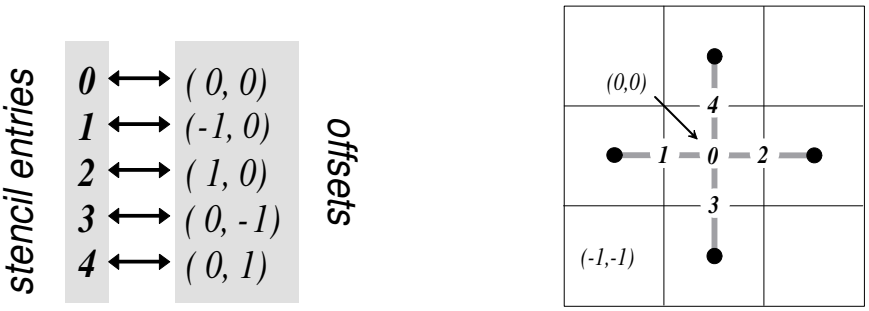


Figure 2.4: Representation of the 5-point discretization stencil for the example problem.

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 2.4. 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.

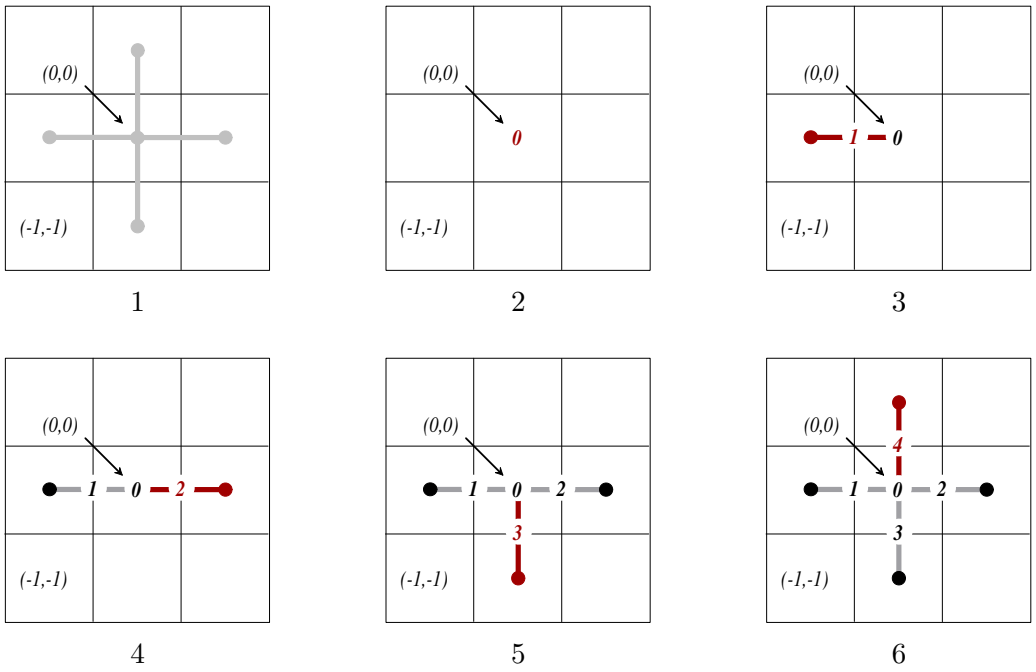
On process 0 or 1, the code in Figure 2.5 will set up the stencil in Figure 2.4. The stencil must be the same on all processes. The `Create()` routine creates an empty 2D, 5-pt stencil object. The `SetElement()` routine defines the geometry of the stencil and assigns the stencil numbers for each of the 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 2.1 and 2.2. 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):

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

On process 0, the code in Figure 2.6 will set up matrix values associated with the center (entry 0) and south (entry 3) stencil entries as given by 2.2 and Figure 2.6 (boundaries are ignored here temporarily). 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




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

Figure 2.5: Code for setting up the stencil in Figure 2.4.

```
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);
```

Figure 2.6: Code for setting up matrix values associated with stencil entries 0 and 3 as given by 2.2 and Figure 2.4.

```
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 */
...
```

Figure 2.7: Code for adjusting boundary conditions along the lower grid boundary in Figure 2.1.

"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 2.1, the "south" coefficient must be set to zero. To do this on process 0, the code in Figure 2.7 could be used:

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 2.3 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 code in Figure 2.8 will set up the right-hand-side vector values. 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()`

```
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);
```

Figure 2.8: Code for setting up right-hand-side vector values.

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 2.1. 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 2.2 to an "appropriate" 3-pt stencil, then set matrix

coefficients (as in Section 2.3) 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 3

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 3.2), composite grids in structured adaptive mesh refinement (AMR) applications (see Figure 3.7), 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. These latter solvers are usually provided via the `FEI` or `IJ` interfaces described in Chapters 4 and 5.

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 2.2) 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 3.1 for an illustration in 2D.

The `SStruct` interface uses a *graph* to allow nearly arbitrary relationships between part data. The graph is constructed from stencils plus some additional data-coupling information set by the `GraphAddEntries()` routine. Another method for relating part data is the `GridSetNeighborbox()` routine, which is particularly suited for block-structured grid problems.

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.

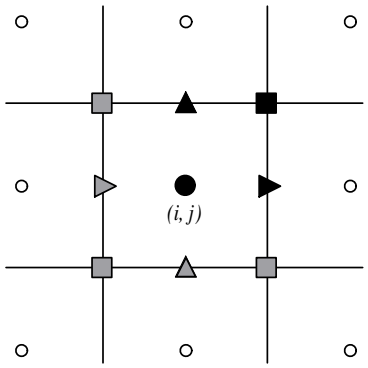


Figure 3.1: Grid variables in *hypre* are referenced by the abstract cell-centered index to the left and down in 2D (analogously in 3D). In the figure, index (i, j) is used to reference the variables in black. The variables in grey—although contained in the pictured cell—are not referenced by the (i, j) index.

3.1 Block-Structured Grids

In this section, we describe how to use the `SStruct` interface to define block-structured grid problems. We will do this primarily by example, paying particular attention to the construction of stencils and the use of the `GridSetNeighborbox()` 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 3.2, where D is a scalar diffusion coefficient, and $\sigma \geq 0$. The discretization [12] 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.

The grid in Figure 3.2 is defined in terms of five separate logically-rectangular parts as shown in Figure 3.3, 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 2.1), 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 2.2). 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.

As with the `Struct` interface, each process describes that portion of the grid that it "owns", one box at a time. Figure 3.4 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

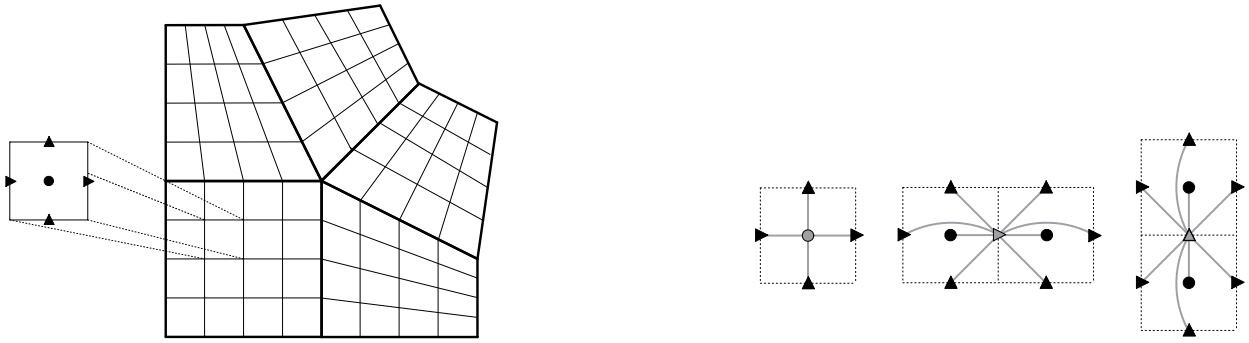


Figure 3.2: Example of a block-structured grid with five logically-rectangular blocks and three variables types: cell-centered, x-face, and y-face. Discretization stencils for the cell-centered (left), x-face (middle), and y-face (right) variables are also pictured.

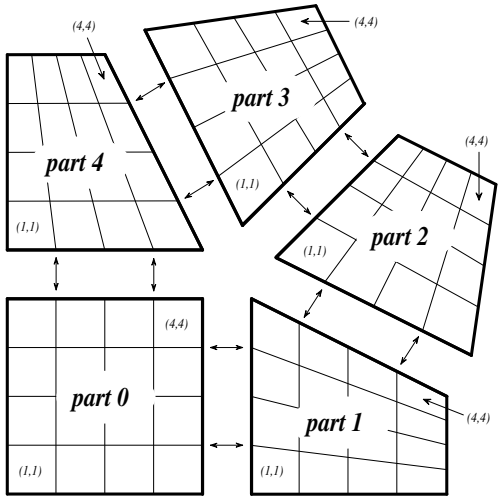
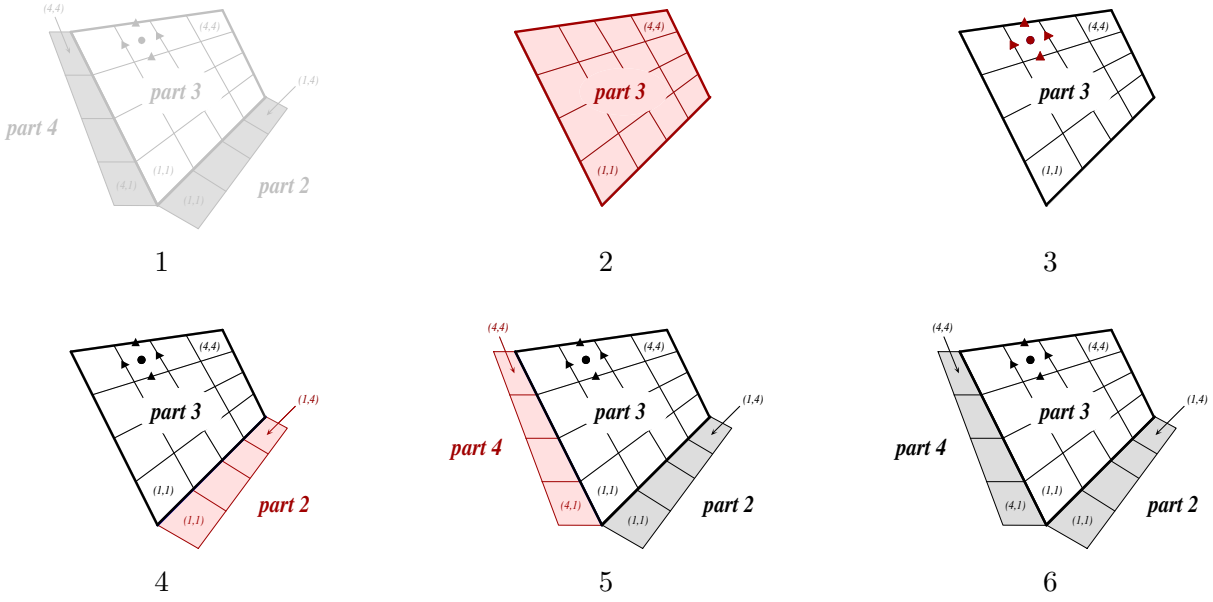


Figure 3.3: Test figure.




```
   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};

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_SStructGridSetNeighborBox(grid, part, nb2_exts[0], nb2_exts[1],
      nb2_n_part, nb2_n_exts[0], nb2_n_exts[1], nb2_map);
5: HYPRE_SStructGridSetNeighborBox(grid, part, nb4_exts[0], nb4_exts[1],
      nb4_n_part, nb4_n_exts[0], nb4_n_exts[1], nb4_map);

6: HYPRE_SStructGridAssemble(grid);
```

Figure 3.4: Code on process 3 for setting up the grid in Figure 3.2.

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 3.4 also correspond to boxes on parts 2 and 4. This is done through the two calls to the `SetNeighborbox()` routine. We will 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 2.2). 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 `SetNeighborbox()` call are simply the lower and upper indices on part 3 and the corresponding indices on part 2. The final argument to the routine indicates that the x-direction on part 3 (i.e., the i component of the tuple (i, j)) corresponds to the y-direction on part 2 and that the y-direction on part 3 corresponds to the 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 `SetNeighborbox()` 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 `SetNeighborbox()` 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 3.2, 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 3.5. 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 3.1 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 3.6 shows the code for setting up the graph .

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 2.3 using routines `MatrixSetValues()` and `MatrixSetBoxValues()` in the `SStruct` interface. As before,

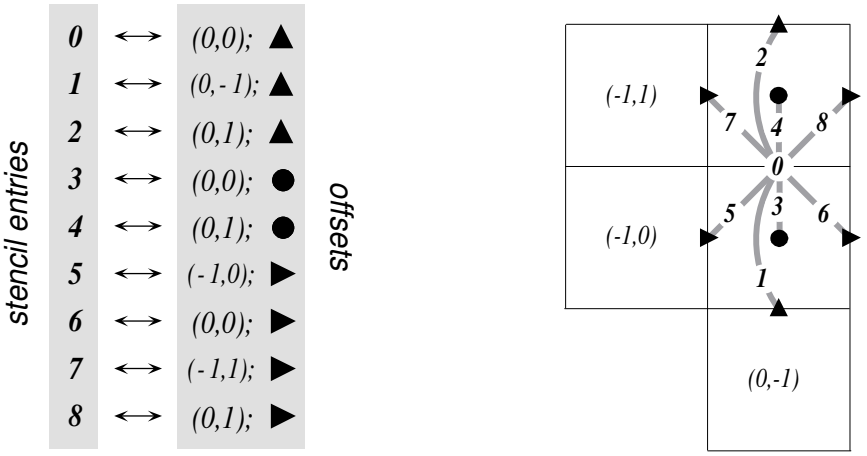


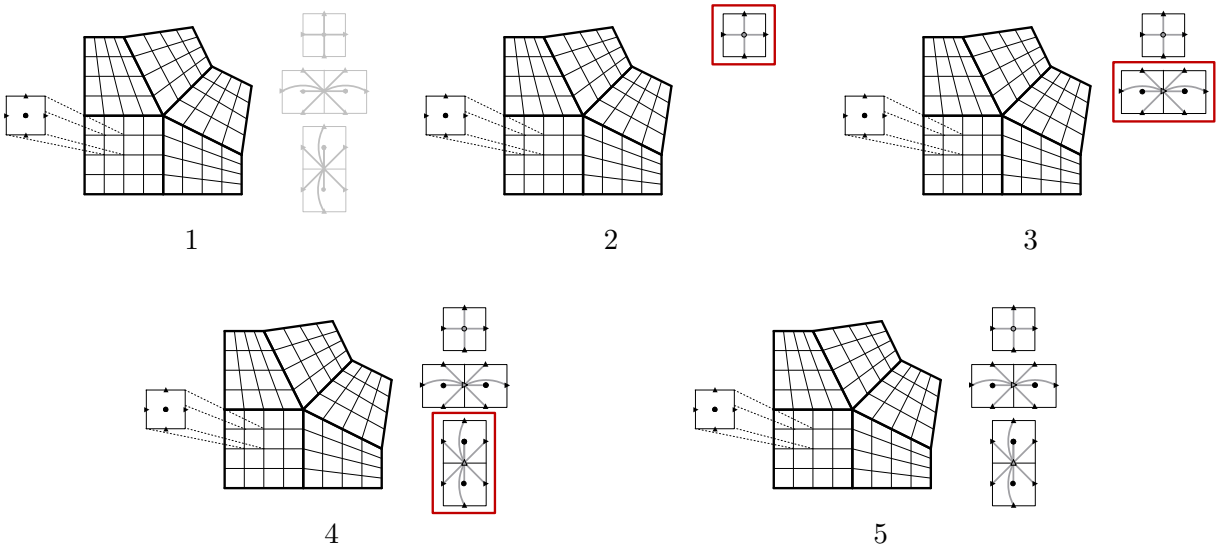
Figure 3.5: Assignment of labels and geometries to the y-face stencil in Figure 3.2. Stencil offsets are described relative to the $(0, 0)$ index for the "center" of the stencil.

there are also `AddTo` variants of these routines. Likewise, setting up the right-hand-side is similar to what is described in Section 2.4. 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 `GridSetNeighborBox()` 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 3.2 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 `GridSetNeighborBox()` for this example.

3.2 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 3.7. 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 3.7) 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 [11] 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).




```
    HYPRE_SStructGraph graph;
    HYPRE_SStructStencil c_stencil, x_stencil, y_stencil;
    int c_var = 0, x_var = 1, y_var = 2;
    int part;

    /* Create the graph object */
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);
3:     HYPRE_SStructGraphSetStencil(graph, part, x_var, x_stencil);
4:     HYPRE_SStructGraphSetStencil(graph, part, y_var, y_stencil);
    }

    /* No need to add non-stencil entries in this example */
5:  HYPRE_SStructGraphAssemble(graph);
```

Figure 3.6: Test figure.

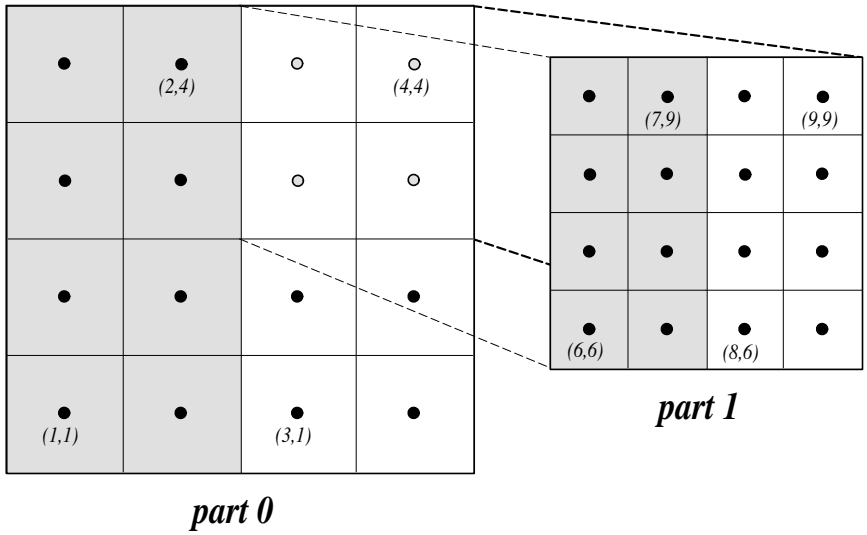


Figure 3.7: Structured AMR grid example. Shaded regions correspond to process 0, unshaded to process 1. The grey dots are dummy variables.

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 3.1 and Figure 3.4 (no `SetNeighborBox` 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 Figure 2.5 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 3.8. Let h be the coarse grid mesh size, and consider a local coordinate system with the origin at the center of cell $(6, 6)$. We approximate the flux as follows

$$\begin{aligned} \int_{-h/4}^{h/4} u_x(-h/4, s)ds &\approx \frac{h}{2} u_x(-h/4, 0) \approx \frac{h}{2} \frac{u(0,0) - u(-3h/4, 0)}{3h/4} \\ &\approx \frac{2}{3}(u_{6,6} - u_{2,3}). \end{aligned} \tag{3.2}$$

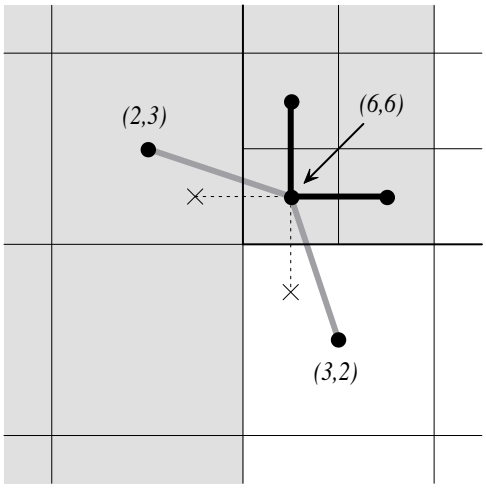


Figure 3.8: Coupling for equation at corner of refinement patch. Black lines (solid and broken) are stencil couplings. Gray line are non-stencil couplings.

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.

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.

Chapter 4

Finite Element Interface

4.1 Introduction

User applications access the *hypre* linear solvers via a pipeline of two interfaces - user to finite element interface (called FEI), and the finite element to linear solver interface (called LinearSystemCore). The purpose of FEI is to allow users to submit the global matrices in the form of element connectivities, element stiffness matrices, element loads, and boundary conditions. These element information are processed by an implementation of the FEI (see [5]) which loads the global matrix and right hand side vectors to the linear solver libraries via the LinearSystemCore interface. The LinearSystemCore interface also facilitates interfacing multiple linear system solver packages (such as PetSC or Aztec) with little change in the user code.

The specification of the FEI and its implementation was first developed at Sandia. A simplified implementation has been implemented at LLNL in *hypre*'s finite element module. In the next section, we describe the basic FEI functions and a sample program to demonstrate how to use them. A brief description of *hypre*'s internal data structure and solver capabilities is presented in Section 5.3. Associated with *hypre*'s finite element interface is an FE-based gray-box multilevel preconditioning module called MLI which provides fast multilevel preconditioners. A description of the MLI is given in Section 5.4. In Section 5.5, we describe the available options for using *hypre*'s rich solver capabilities. Users who prefer to create their own finite element packages but would like to use the *hypre* solvers can link their packages to *hypre* via the LinearSystemCore. A description of this interface is given in Section 5.6. Finally, some installation and usage issues are discussed in Section 5.7.

4.2 A Brief Description of The Finite Element Interface

Embedded in application finite element codes are data structures storing element connectivities, element stiffness matrices, element loads, boundary conditions, nodal coordinates, etc. An implicit finite element problem can be solved by assembling the global stiffness matrix and the corresponding right hand side vector, and then calling linear solver to calculate the solution. The first step in this process is thus to instantiate an FEI object by

```
feiPtr = new FEI_Implementation(mpiComm);
```

where mpiComm is an MPI communicator (e.g. MPI_COMM_WORLD). Next, various finite element information need to be sent into the FEI object.

The first entity to be submitted to the FEI is *field* information. A *field* has an identifier called *fieldID* and a rank or *fieldSize* (number of degree of freedom). For example, for a simple 3D incompressible Navier Stokes equation, the nodal variable is the velocity vector which has 3 degrees of freedom; and the element variable (constant over the element) is the pressure (scalar). 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;
fieldID = new int[nFields];
fieldID[0] = 7; /* velocity vector */
fieldID[1] = 8; /* pressure */
fieldSize = new int[nFields];
fieldSize[0] = 3; /* velocity vector */
fieldSize[1] = 1; /* pressure */
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;
nElems = 1000;
elemNNodes = 8; /* number of nodes per element */
nodeNFields = 1; /* nodal field - velocity */
nodeFieldIDs = new[nodeNFields];
nodeFieldIDs[0] = fieldID[0]; /* velocity */
elemNFields = 1; /* element field - pressure */
elemFieldIDs = new[elemNFields];
elemFieldIDs[0] = fieldID[1]; /* pressure */
feiPtr− >initElemBlock(elemBlkID, nElems, elemNNodes, nodeNFields, nodeFieldIDs,
                       elemNFields, elemFieldIDs, 0);
```

The last argument is to specify 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 the element blocks 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 whether the boundary conditions are Dirichlet, Neumann, or mixed, the three values should be passed into the FEI accordingly.

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 (iE = 0; iE ¡ nElems; iE++)
        feiPtr− >sumInElem(elemBlkID, elemID, elemConn[iE], elemStiff[iE],
                           elemLoads[iE], elemFormat);
```

Again, to complete the loading phase, a completion signal is sent to the FEI via

```
feiPtr− >loadComplete();
```

Now the global stiffness matrix and the corresponding right hand side have been assembled. Before the linear system is solved, a number of solver parameters have to be passed into the FEI. A detailed description of the solver parameters is given in Section 3. An example is given below

```
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 status is returned from the FEI indicating whether the solve is successful. Finally, the solution can be retrieved by

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

where solnOffsets[i] is the index pointing to the first location where the variables at node i is returned in solnValues.

4.3 *hypre* Solvers

On the other side of the LinearSystemCore interface is the linear solver package. Any linear solver package can be used as long as there is an implementation of the interface in the solver package itself. The LinearSystemCore implementation in *hypre* is encapsulated in the *hypre* finite element conceptual interface called HYPRE_LinSysCore. The incoming element stiffness matrices and other information are converted into *hypre*'s internal data structures. In this section we briefly describe some of these internal data structures and capabilities.

4.3.1 Parallel Matrix and Vector Construction

The matrix class in *hypre* accessible via the LinearSystemCore interface is the parallel compressed sparse row (ParCSR) matrix. The requirements about how the global matrix is partitioned among the processors are that each processor holds a contiguous block of rows and columns and the equation numbers in processors of lower rank are lower than those in processors of higher rank. The FEI is responsible for ensuring that these two requirements are followed. The matrix can be loaded in parallel - a row or a block of rows at a time. The solution and right hand side vectors are constructed accordingly. The matrix rows corresponding to the shared nodes can be assigned to either processor, and is determined by the FEI itself. Once the incoming matrix and vector data have been captured in the *hypre* ParCSR format, a whole of matrix and vector operators are available for use in the *hypre* solvers.

4.3.2 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. Krylov solvers (CG, GMRES, FGMRES, TFQMR, BiCGSTAB, symmetric QMR),
2. Boomeramg (a parallel algebraic multigrid solver),
3. SuperLU direct solver (sequential),
4. SuperLU direct solver with iterative refinement (sequential),

4.3.3 Parallel Preconditioners

The performance of the Krylov solvers can be improved by clever selection of preconditioners. *hypre* currently has the following preconditioners available via the LinearSystemCore interface:

1. no preconditioning,
2. diagonal,
3. another parallel incomplete LU (Euclid),
4. parallel sparse approximate inverse (ParaSails),
5. parallel algebraic multigrid (BoomerAMG),
6. another parallel algebraic multigrid (MLI based on the smoothed aggregation multigrid methods),
7. parallel domain decomposition with inexact local solves (DDIlut),
8. least-squares polynomial preconditioner,
9. 2×2 block preconditioner, and
10. 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 later sections.

4.3.4 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 different 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).

4.3.5 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.

4.4 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. 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. Details of how to set this preconditioners up can be found in the next section.

4.5 *hypre* LinearSystemCore Parameters

User applications interact directly with *hypre* via the parameters function. In this section we list the parameters recognized by HYPRE_LinSysCore.

4.5.1 Parameters for Solvers and Preconditioners

solver xxx where xxx specifies one of cg, gmres, fgmres, bicgs, bicgstab, tfqmr, symqmr, superlu, or superlux. The default is gmres. The solver type can be followed by override to specify its priority when multiple solvers are declared at random order.

preconditioner xxx where xxx is one of diagonal, pilut, euclid, parasails, boomeramg, ddilut, poly, blockP, Uzawa, or mli. The default is diagonal. Another option for xxx is reuse which allows the preconditioner to be reused (this is to be set after a preconditioner has been set up already). The preconditioner type can be followed by override to specify its priority when multiple preconditioners are declared at random order.

maxIterations xxx where xxx is an integer specifying the maximum number of iterations permitted for the iterative solvers. The default value is 1000.

tolerance xxx where xxx is a floating point number specifying the termination criterion for the iterative solvers. The default value is 1.0E-6.

gmresDim xxx where xxx is an integer specifying the value of m in restarted GMRES(m). The default value is 100.

stopCrit xxx where xxx is one of absolute or relative stopping criterion.

superluOrdering xxx - where xxx specifies one 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.

superluScale xxx where xxx specifies one of y (perform row and column scalings before decomposition) or n. The default is no scaling.

4.5.2 Parameters for ILUT, SPAI, and Polynomial Preconditioners

ddilutFillin xxx where xxx is a floating point number specifying the maximum number of nonzeros kept in the formation of local incomplete L and U (a value of 0.0 means same sparsity as A, and a value of 1.0 means two times the number of nonzeros as A.). If this is not called, a value will be selected based on the sparsity of the matrix.

ddilutDropTol 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.

euclidNlevels xxx where xxx is an non-negative integer specifying the desired sparsity of the incomplete factors. The default value is 0.

euclidThreshold xxx where xxx is a floating point number specifying the threshold used to sparsify the incomplete factors. The default value is 0.0.

parasailsThreshold 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.

parasailsNlevels xxx where xxx is an integer larger than 0 specifying the desired sparsity of the approximate inverse. The default value is 1.

parasailsFilter xxx where xxx is a floating point number between 0 and 1 specifying the threshold used to prune small entries in A. The default value is 0.0.

parasailsLoadbal xxx where xxx is a floating point number between 0 and 1 specifying how load balancing has to be done (Edmond, explain please). The default value is 0.0.

parasailsSymmetric sets Parasails to take A as symmetric.

parasailsUnSymmetric sets Parasails to take A as nonsymmetric (default).

parasailsReuse sets Parasails to reuse the sparsity pattern of A.

polyorder xxx where xxx is the order of the least-squares polynomial preconditioner.

4.5.3 Parameters for Multilevel Preconditioners

Parameters for BoomerAMG Preconditioner

amgCoarsenType xxx where xxx specifies one of falgout or ruge, or default (CLJP) coarsening for BOOMERAMG.

amgMeasureType xxx where xxx specifies one of local or or global. This parameter affects how coarsening is performed in parallel.

amgNumSweeps xxx where xxx is an integer specifying the number of pre- and post-smoothing at each level of BOOMERAMG. The default is two pre- and two post-smoothings.

amgRelaxType xxx where xxx is one of jacobi (Damped Jacobi), gs-slow (sequential Gauss-Seidel), gs-fast (Gauss-Seidel on interior nodes), or hybrid. The default is hybrid.

amgRelaxWeight xxx where xxx is a floating point number between 0 and 1 specifying the damping factor for BOOMERAMG's damped Jacobi and GS smoothers. The default value is 1.0.

amgRelaxOmega xxx where xxx is a floating point number between 0 and 1 specifying the damping factor for BOOMERAMG's hybrid smoother for multiple processors. The default value is 1.0.

amgStrongThreshold 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.

amgSystemSize xxx where xxx is the degree of freedom per node.

amgUseGSMG - tells BOOMERAMG to use a different coarsening called GSMG.

amgGSMGNumSamples where xxx is the number of samples to generate to determine how to coarsen for GSMG.

Parameters for MLI Preconditioner

outputLevel xxx where xxx is the output level for diagnostics.

method xxx where xxx is either AMGSA (default), AMGSAe, to indicate which MLI algorithm is to be used.

numLevels xxx where xxx is the maximum number of levels (default=30) used.

maxIterations xxx where xxx is the maximum number of iterations (default = 1 as preconditioner).

cycleType xxx where xxx is either 'V' or 'W' cycle (default = 'V').

strengthThreshold xxx strength threshold for coarsening (default = 0).

smoother xxx where xxx is either Jacobi, BJacobi, GS, SGS, HSGS (SSOR,default), BSGS, ParaSails, MLS, CGJacobi, CGBJacobi, or Chebyshev.

numSweeps xxx where xxx is the number of smoother sweeps (default = 2).

coarseSolver xxx where xxx is one of those in 'smoother' or SuperLU (default).

minCoarseSize xxx where xxx is the minimum coarse grid size to control the number of levels used (default = 3000).

Pweight xxx where xxx is the relaxation parameter for the prolongation smoother (default 0.0).

nodeDOF xxx where xxx is the degree of freedom for each node (default = 1).

nullSpaceDim xxx where xxx is the dimension of the null space for the coarse grid (default = 1).

useNodalCoord xxx where xxx is either 'on' or 'off' (default) to indicate whether the nodal coordinates are used to generate the initial null space.

saAMGCalibrationSize xxx where xxx is the additional null space vectors to be generated via calibration (default = 0).

numSmoothVecs xxx where xxx is the number of near null space vectors used to create the prolongation operator (default = 0).

smoothVecSteps xxx where xxx is the number of smoothing steps used to generate the smooth vectors (default = 0).

In addition, to use 'AMGSAe', the parameter 'haveSFEI' has to be sent into the FEI using the parameters function (this option is valid only for the Sandia FEI implementation).

4.5.4 Parameters for Block and Uzawa Preconditioners

Parameters for Block Preconditioners

The parameters for this preconditioner are preceded by the keyword sf blockP. The available parameters after this keywords are:

blockD turns on block diagonal preconditioning.

blockT turns on block tridiagonal preconditioning.

blockLU turns on block LU preconditioning.

outputLevel xxx where xxx is the output level for diagnostics.

block1FieldID xxx where xxx is field ID for the (1,1) block.

block2FieldID xxx where xxx is field ID for the (2,2) block (for ALE3D's implicit hydrodynamics with slide surfaces, the field ID for both blocks are -3.)

printInfo prints information about internal parameter settings.

lumpedMassScheme xxx where is either diag (take the diagonal of the (1,1) block) or ainv (take the approximate inverse of the (1,1) block.

invA11PSNlevels xxx where xxx is 0 or 1 to indicate the ParaSails nlevels to use to generate the lumped mass matrix (if ainv is selected.)

invA11PSThresh xxx where xxx is a floating point number between 0 and 1 to indicate the ParaSails threshold to use to generate the lumped mass matrix (if ainv is selected.)

A11Solver xxx where xxx is either cg or gmres as solver for the (1,1) block.

A11Tolerance xxx where xxx is convergence tolerance for the (1,1) block.

A11MaxIterations xxx where xxx is maximum number of iterations for the (1,1) block.

A11Precon xxx where xxx is either pilut, boomeramg, euclid, parasails, ddilut, or mli.

A11PreconPSNlevels xxx - ParaSails' nlevels.

A11PreconPSThresh xxx - ParaSails' threshold.

A11PreconPSFilter xxx - ParaSails' filter.

A11PreconAMGThresh xxx - Boomeramg's threshold.

A11PreconAMGRelaxType xxx - Boomeramg's smoother.

A11PreconAMGNumSweeps xxx - Boomeramg's numSweeps.

A11PreconAMGSystemSize xxx - Boomeramg's systemSize.

A11PreconEuclidNLevels xxx - Euclid's nlevels.

A11PreconEuclidThresh xxx - Euclid's threshold.

A11PreconPilutFillin xxx - Pilut's fillin.

A11PreconPilutDropTol xxx - Pilut's drop tolerance.

A11PreconDDIlutFillin xxx - DDILUT's fillin.

A11PreconDDIlutDropTol xxx - DDILUT's drop tolerance.

A11PreconMLIRelaxType xxx - MLI's smoother.

A11PreconMLIThresh xxx - MLI's threshold.

A11PreconMLIPweight xxx - MLI's Pweight.

A11PreconMLINumSweeps xxx - MLI's numSweeps.

A11PreconMLINodeDOF xxx - MLI's nodeDOF.

A11PreconMLINullDim xxx - MLI's null space dimension.

A22Solver xxx where xxx is either cg or gmres as solver for the (2,2) block.

A22Tolerance xxx where xxx is convergence tolerance for the (2,2) block.

A22MaxIterations xxx where xxx is maximum number of iterations for the (2,2) block.

A22Precon xxx where xxx is either pilut, boomeramg, euclid, parasails, ddilut, or mli.

A22PreconPSNlevels xxx - ParaSails nlevels.

A22PreconPSThresh xxx - ParaSails' threshold.

A22PreconPSFilter xxx - ParaSails' filter.

A22PreconAMGThresh xxx - Boomeramg's threshold.

A22PreconAMGRelaxType xxx - Boomeramg's smoother.

A22PreconAMGNumSweeps xxx - Boomeramg's numSweeps.

A22PreconAMGSystemSize xxx - Boomeramg's systemSize.

A22PreconEuclidNLevels xxx - Euclid's nlevels.

A22PreconEuclidThresh xxx - Euclid's threshold.

A22PreconPilutFillin xxx - Pilut's fillin.

A22PreconPilutDropTol xxx - Pilut's drop tolerance.

A22PreconDDIlutFillin xxx - DDILUT's fillin.

A22PreconDDIlutDropTol xxx - DDILUT's drop tolerance.

A22PreconMLIRelaxType xxx - MLI's smoother.

A22PreconMLIThresh xxx - MLI's threshold.

A22PreconMLIPweight xxx - MLI's Pweight.

A22PreconMLINumSweeps xxx - MLI's numSweeps.

A22PreconMLINodeDOF xxx - MLI's nodeDOF.

A22PreconMLINullDim xxx - MLI's null space dimension.

Parameters for Uzawa Preconditioner

The Uzawa preconditioner has a similar parameter set as block preconditioner, as described in the following (except that DDIlut is not available here).

outputLevel xxx - where xxx is the output level for diagnostics.

A11Solver xxx where xxx is either cg or gmres as solver for the (1,1) block.

A11Tolerance xxx where xxx is convergence tolerance for the (1,1) block.

A11MaxIterations xxx where xxx is maximum number of iterations for the (1,1) block.

A11Precon xxx where xxx is either pilut, boomeramg, euclid, parasails, ddilut, or mli.

A11PreconPSNlevels xxx - ParaSails' nlevels.

A11PreconPSThresh xxx - ParaSails' threshold.

A11PreconPSFilter xxx - ParaSails' filter.

A11PreconAMGThresh xxx - Boomeramg's threshold.

A11PreconAMGRelaxType xxx - Boomeramg's smoother.

A11PreconAMGNumSweeps xxx - Boomeramg's numSweeps.

A11PreconAMGSystemSize xxx - Boomeramg's systemSize.

A11PreconEuclidNLevels xxx - Euclid's nlevels.

A11PreconEuclidThresh xxx - Euclid's threshold.

A11PreconPilutFillin xxx - Pilut's fillin.

A11PreconPilutDropTol xxx - Pilut's drop tolerance.

A11PreconMLIRelaxType xxx - MLI's smoother.

A11PreconMLIThresh xxx - MLI's threshold.

A11PreconMLIPweight xxx - MLI's Pweight.

A11PreconMLINumSweeps xxx - MLI's numSweeps.

A11PreconMLINodeDOF xxx - MLI's nodeDOF.

A11PreconMLINullDim xxx - MLI's null space dimension.

S22SolverDampingFactor xxx where xxx is the damping (scaling) factor for the Schur complement approximation of the (2,2) block.

S22Solver xxx where xxx is either cg or gmres as solver for the (2,2) block.

S22Tolerance xxx where xxx is convergence tolerance for the (2,2) block.

S22MaxIterations xxx where xxx is maximum number of iterations for the (2,2) block.

S22Precon xxx where xxx is either pilut, boomeramg, euclid, parasails, ddilut, or mli.

S22PreconPSNlevels xxx - ParaSails' nlevels.

S22PreconPSThresh xxx - ParaSails' threshold.

S22PreconPSFilter xxx - ParaSails' filter.

S22PreconAMGThresh xxx - Boomeramg's threshold.

S22PreconAMGRelaxType xxx - Boomeramg's smoother.

S22PreconAMGNumSweeps xxx - Boomeramg's numSweeps.

S22PreconAMGSystemSize xxx - Boomeramg's systemSize.

S22PreconEuclidNLevels xxx - Euclid's nlevels.

S22PreconEuclidThresh xxx - Euclid's threshold.

S22PreconPilutFillin xxx - Pilut's fillin.

S22PreconPilutDropTol xxx - Pilut's drop tolerance.

S22PreconMLIRelaxType xxx - MLI's smoother.

S22PreconMLIThresh xxx - MLI's threshold.

S22PreconMLIPweight xxx - MLI's Pweight.

S22PreconMLINumSweeps xxx - MLI's numSweeps.

S22PreconMLINodeDOF xxx - MLI's nodeDOF.

S22PreconMLINullDim xxx - MLI's null space dimension.

4.5.5 Parameters for Matrix Reduction

schurReduction turns on the Schur reduction mode.

slideReduction turns on the slide reduction mode.

slideReduction2 turns on the slide reduction mode version 2 (see section 2).

slideReduction3 turns on the slide reduction mode version 3 (see section 2).

4.5.6 Parameters for Diagnostics and Performance Tuning

outputLevel xxx where xxx is an integer specifying the output level. An output level of 1 prints only the solver information such as number of iterations and timings. An output level of 2 prints debug information such as the functions visited and preconditioner information. An output level of 3 or higher prints more debug information such as the matrix and right hand side loaded via the LinearSystemCore functions to the standard output.

setDebug xxx where xxx is one of slideReduction1, slideReduction2, slideReduction3 (level 1,2,3 diagnostics in the slide surface reduction code), printMat (print the original matrix into a file), printReducedMat (print the reduced matrix into a file), printSol (print the solution into a file), ddilut (output diagnostic information for DDIlut preconditioner setup), and amgDebug (output diagnostic information for AMG).

optimizeMemory cleans up the matrix sparsity pattern after the matrix has been loaded. (It has been kept to allow matrix reuse.)

imposeNoBC turns off the boundary condition to allow diagnosing the matrix (for example, checking the null space.)

4.5.7 Miscellaneous Parameters

AConjugateProjection xxx where xxx specifies the number of previous solution vectors to keep for the A-conjugate projection. The default is 0 (the projection is off).

minResProjection xxx where xxx specifies the number of previous solution vectors to keep for projection. The default is 0 (the projection is off).

haveFEData indicates that additional finite element information are available to assist in building more efficient solvers.

haveSFEI indicates that the simplified finite element information are available to assist in building more efficient solvers.

4.6 The LinearSystemCore Interface

As described before, users who prefer to create their own finite element interface package can also take advantage of the rich solver capabilities in *hypre*. In this section we show how to access HYPRE_LinSysCore's internal solver directly. Users who choose this path need first to construct an array (say, eqnOffsets describing the matrix row partitioning across all processors (so eqnOffsets[p] and eqnOffset[p+1] have the starting and ending row indices for processor p). Furthermore, suppose the local submatrix has been constructed as a compressed sparse row (CSR) matrix in the ia, ja, val arrays. The following program segment describes the function calls to set up the internal matrix and solve the linear system.

```
Program Segment
startRow = eqnOffsets[mypid];
endRow = eqnOffsets[mypid+1] - 1;
nrows = endRow - startRow + 1
for ( i = startRow; i <= endRow; i++ ) {
      ncnt = ia[i+1] - ia[i];
      rowLengths[i-startRow] = ncnt;
      colIndices[i-startRow] = new int[ncnt];
      k = 0;
      for (j = ia[i]; j ¡ ia[i+1]; j++) colIndices[i-startRow][k++] = ja[j];
}
HYPRE_LinSysCore_create(&lsc, MPI_COMM_WORLD);
HYPRE_setGlobalOffsets(lsc, nrows, NULL, eqnOffsets, NULL);
HYPRE_setMatrixStructure(lsc, colIndices, rowLengths, NULL, NULL, NULL);
for ( i = startRow; i ¡= endRow; i++ ) {
      ncnt = ia[i+1] - ia[i];
      HYPRE_sumIntoSystemMatrix(lsc, i, ncnt, &val[ia[i]], &ja[ia[i]]);
      HYPRE_sumIntoRHSVector(1, &rhs[i], &i);
}
HYPRE_matrixLoadComplete();
strcpy(paramString, "solver gmres");
HYPRE_parameters(1, &paramString);
strcpy(paramString, "preconditioner boomeramg");
HYPRE_parameters(1, &paramString);
HYPRE_launchSolver(&status, &iterations);
```

A list of available functions is given in the following.

HYPRE_LinSysCore_create(LinSysCore **lsc, MPI_Comm comm)

HYPRE_LinSysCore_destroy(LinSysCore **lsc)

HYPRE_parameters(LinSysCore *lsc, int nParams, char **params)

HYPRE_setGlobalOffsets(LinSysCore* lsc, int leng, int* nodeOffsets,
int* eqnOffsets, int* blkEqnOffsets)

HYPRE_setMatrixStructure(LinSysCore *lsc, int** ptColIndices,
int* ptRowLengths, int** blkColIndices, int* blkRowLengths, int* ptRowsPerBlkRow)

HYPRE_resetMatrixAndVector(LinSysCore *lsc, double val)

HYPRE_resetMatrix(LinSysCore *lsc, double val)

HYPRE_resetRHSVector(LinSysCore *lsc, double val)

HYPRE_sumIntoSystemMatrix(LinSysCore *lsc, int numPtRows, const int* ptRows,
int numPtCols, const int* ptCols, int numBlkRows, const int* blkRows,
int numBlkCols, const int* blkCols, const double* const* values)

HYPRE_sumIntoRHSVector(LinSysCore *lsc, int num, const double* values, const int* indices)

HYPRE_matrixLoadComplete(LinSysCore *lsc)

HYPRE_enforceEssentialBC(LinSysCore *lsc, int* globalEqn, double* alpha, double* gamma, int leng)

HYPRE_enforceRemoteEssBCs(LinSysCore *lsc,int numEqns,int* globalEqns, int** colIndices,
int* colIndLen, double** coefs)

HYPRE_enforceOtherBC(LinSysCore *lsc, int* globalEqn, double* alpha, double *beta
double* gamma, int leng)

HYPRE_putInitialGuess(LinSysCore *lsc, const int* eqnNumbers, const double* values, int leng)

HYPRE_getSolution(LinSysCore *lsc, double *answers, int leng)

HYPRE_getSolnEntry(LinSysCore *lsc, int eqnNumber, double *answer)

HYPRE_formResidual(LinSysCore *lsc, double *values, int leng)

HYPRE_launchSolver(LinSysCore *lsc, int *solveStatus, int *iter)

4.7 HYPRE LinearSystemCore Installation

The ultimate objective is for application users to have immediate access to the latest FEI/*hypre* library files on different computing platforms via public lib directories. While this feature is forthcoming, careful version control is needed for users to keep track of capabilities and bug fixes for different installations. Users who would like to set up the FEI/*hypre* on their own should do the following :

1. obtain the *hypre* and the Sandia FEI source codes (alternatively, use the FEI implementation in *hypre*),
2. compile Sandia's FEI (fei-2.5.0) to create the libfei_base.a file.
3. compile *hypre*
   (a) download *hypre* from the web, ungzip and untar it
   (b) go into the linear_solvers directory
   (c) do a 'configure' with the –with-fei-inc-dir option set to the FEI include directory plus other compile options
   (d) compile with make install to create the libHYPRE_LSI.a file in the linear_solvers/hypre/lib directory.
4. call the FEI functions in your application code (example given previously)
   (a) include cfeihypre.h in your file
   (b) include FEI_Implementation.h in your file
   (c) make sure your application has an include and an lib path to the include and lib directories created above.

4.7.1 Linking with the library files

To link the FEI and *hypre* into the executable, the following has to be attached to the linking command :

```
-L${LIBPATHS} -lfei_base -lHYPRE_LSI
```

along with all the other libraries (Note : the order in which the libraries are listed may be important), where LIBPATHS are where the *hypre* and FEI libraray files can be found.

Since some of these library files make calls to LAPACK and BLAS functions, the corresponding libraries need to be linked along with (placed after) these library files. For example, on the DEC cluster, it suffices to link with the dxml library, (So -ldxml is placed after the above link sequence, with -lm placed after -ldxml.) while the *essl* library can be used on the blue machine. If SuperLU is also needed, -lHYPRE_superlu should be placed immediately after HYPRE_LSI.

4.7.2 Some more caveats for application developers

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.

Chapter 5

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:

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{P-1} \end{bmatrix} \tag{5.1}$$

In the above example, the matrix is distributed accross 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 */
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`. After the coefficients are set, they can be added to with an `AddTo()` routine. Each process should set only those matrix values that it "owns" in the data distribution.

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`. Each process should set only those vector values that it "owns" in the data distribution.

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.

Chapter 6

Solvers and Preconditioners

There are several solvers available in *hypre* via different conceptual interfaces (see Table 6.1). 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			
SysPFMG	X			
Split		X		
BoomerAMG		X	X	X
MLI		X	X	X
ParaSails		X	X	X
Euclid		X	X	X
PILUT		X	X	X
PCG	X	X	X	X
GMRES	X	X	X	X
BiCGSTAB	X	X	X	X
Hybrid	X	X	X	X

Table 6.1: 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 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.

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,

$$\nabla \cdot (D\nabla u) + \sigma u = f \tag{6.1}$$

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 [14, 2, 6] 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 [1, 6] 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 BoomerAMG

BoomerAMG is a parallel implementation of the algebraic multigrid method [13]. 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. See [7, 17] for a detailed description of the coarsening algorithms, interpolation and relaxation schemes as well as numerical results. The following 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,
- PMIS and HMIS coarsening algorithms which lead to coarsenings with lower complexities [4] and
- aggressive coarsening, which can be applied to any of the coarsening techniques mentioned above and thus achieving much lower complexities and lower memory use [15].

The following interpolation techniques are available:

- the “classical” interpolation as defined in [13],
- direct interpolation [15],

- multipass interpolation [15].
- the "classical" interpolation modified for hyperbolic PDEs.

The following relaxation techniques are available:

- weighted Jacobi relaxation,
- a hybrid Gauss-Seidel / Jacobi relaxation scheme,
- a symmetric hybrid Gauss-Seidel / Jacobi relaxation scheme, and
- hybrid block smoothers [16].

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.

6.3.1 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);
```

For best performance, it might be necessary to set certain parameters. 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. 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. In some cases (such as a 3D 7-point Laplace problem) good performance could be obtained by setting the strength threshold to 0 and additionally setting the interpolation truncation factor to 0.3 via HYPRE_BoomerAMGSetTruncFactor.

For a complete listing of all parameters see the reference manual.

6.4 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].

6.4.1 Synopsis

```
#include "HYPRE_parcsr_ls.h"

int HYPRE_ParaSailsCreate(MPI_Comm comm, HYPRE_Solver *solver,
  int symmetry);
int HYPRE_ParaSailsSetParams(HYPRE_Solver solver,
  double thresh, int nlevel, double filter);
int HYPRE_ParaSailsSetup(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_ParaSailsSolve(HYPRE_Solver solver, HYPRE_ParCSRMatrix A,
  HYPRE_ParVector b, HYPRE_ParVector x);
int HYPRE_ParaSailsStats(HYPRE_Solver solver);
int HYPRE_ParaSailsDestroy(HYPRE_Solver solver);
```

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

6.4.2 Interface functions

A ParaSails solver `solver` is returned with

```
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 6.4.3.

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:

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.

6.4.3 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).

6.5 Euclid

The Euclid library is a scalable implementation of the Parallel ILU algorithm that was presented at SC99 [8], and published in expanded form in the SIAM Journal on Scientific Computing [9]. 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.

6.5.1 Synopsis

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 two calls 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");

//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
//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);
```

6.5.2 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
```

6.5.3 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.

The following options are partially implemented, but not yet fully functional (i.e, don't use them until further notice).

-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.

-maxNzPerRow $\langle int \rangle$ This sets the maximum number of nonzeros that is permitted in any row of F, in addition to the number that would result from an ILU(0) factorization. A negative value indicates infinity (no limit). This setting is effective for both ILU(k) and ILUT factorization methods. Default: infinity, for ILU(k); 5, for ILUT.

6.6 *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 [10] 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.

Chapter 7

Building the HYPRE Library

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 Configure and Make

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).

Now that all of the files are available, 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
```

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.

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 are used, `src/hypre/lib` and `src/hypre/include`.

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.

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.

7.2.2 Configure Execution and Sample Output

```
Usage: ./configure [OPTION] ... [VAR=VALUE]...
```

Examples of using `configure` with options and variable settings. The user can mix and match the options and variable settings as needed to satisfy their requirements.

```
./configure --with-openmp --enable-debug

./configure  CC=mpiicc --with-babel

./configure  --with-blas-lib="essl" --with-blas-lib-dirs="/usr/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.

7.2.3 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
                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
```

7.2.4 Make Execution and Sample Output

`Usage: make [TARGET]`

Examples of using `make` with different targets. Note that only ONE target can be specified at a time. The output from `make` is several pages long.

```
make

make install

make test
```

7.3 Testing the Library

The `test` subdirectory contains several codes that can be used to test the newly created *hypre* library. To create the executable versions, move into the `test` subdirectory, enter `make test` then execute the codes as described by their help packages.

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 -l`*hypre library name*... `-l`*hypre library name*..., where `$PREFIX` is the directory where *hypre* is installed, specified by the configure option `--prefix=PREFIX`. Additionally, any other libraries to which *hypre* is linked must also be linked to by the users application.

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`

Table 7.1: Conversion from C parameters to Fortran arguments

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

7.5 Calling HYPRE from Fortran

A Fortran interface is provided in *hypre* to enable such applications to call its C routines. 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 Table 7.1.

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 (although we are not aware of any at this time). 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,
```

```
                              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)
```

7.6 Bug Reporting

An automated bug reporting mechanism has been set up for *hypre* to be used 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 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. 51

[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. 51

[3] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21:1804–1822, 2000. 53

[4] H. De Sterck, U. M. Yang, and J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, to appear, 2004. Also available as LLNL technical report UCRL-JRNL-206780. 51

[5] 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. 27

[6] 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. 51

[7] 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. 51

[8] 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. 55

[9] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, 2001. 55

[10] 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. 58

[11] S. F. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*, volume 6 of *Frontiers in Applied Mathematics*. SIAM Books, Philadelphia, 1989. 22

[12] 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. 18

[13] 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. 51

[14] 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. 51

[15] K. Stüben. Algebraic multigrid (AMG): an introduction with applications. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*. Academic Press, 2001. 51, 52

[16] U. M. Yang. On the use of relaxation parameters in hybrid smoothers. *Numerical Linear Algebra with Applications*, 11:155–172, 2004. 52

[17] 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, 2005. Also available as LLNL technical report UCRL-BOOK-208032. 51