CONCEPTUAL INTERFACES IN *hypre*

ROBERT D. FALGOUT , JIM E. JONES , AND ULRIKE MEIER YANG*

Abstract. The *hypre* software library is being developed with the aim of providing scalable solvers for the solution of large, sparse linear systems on massively parallel computers. To this end, the notion of conceptual interfaces was introduced. These interfaces give applications users a more natural means for describing their linear systems, and provide access to methods such as geometric multigrid which require additional information beyond just the matrix. This paper discusses the design of the conceptual interfaces in *hypre* and illustrates their use with various examples. A brief overview of the solvers and preconditioners available through these interfaces is also given.

1. Introduction. The increasing demands of computationally challenging applications and the advance of larger more powerful computers with more complicated architectures have necessitated the development of new solvers and preconditioners. Since the implementation of these methods is quite complex, the use of high performance libraries with the newest efficient solvers and preconditioners becomes more important for promulgating their use into applications with relative ease.

hypre [10, 13] has been designed with the primary goal of providing users with advanced scalable parallel preconditioners. Issues of robustness, ease of use, flexibility and interoperability have also been important. It can be used both as a solver package and as a framework for algorithm development. Its object model is more general and flexible than the current generation of solver libraries [6]. *hypre* also provides several of the most commonly used solvers, such as conjugate gradient for symmetric systems or GMRES for nonsymmetric systems to be used in conjunction with the preconditioners.

Design innovations have been made to enable access to the library in the way that applications users naturally think about their problems. For example, application developers that use structured grids, typically think of their problems in terms of stencils and grids. *hypre*'s users do not have to learn complicated sparse matrix structures; instead *hypre* does the work of building these data structures through various *conceptual interfaces*. The conceptual interfaces currently implemented include stencil-based structured / semi-structured interfaces, a finite-element based unstructured interface, and a traditional linear-algebra based interface.

The primary focus of this paper is on the design and use of the conceptual interfaces in *hypre*. Implementation and performance of these interfaces will be discussed in a future paper. However, the reader should note that parallel implementations of these interfaces exist and are already being used in many application codes. In Section 2, we introduce and motivate the idea of conceptual interfaces. In Sections 3, 4, 5, and 6, we describe each of the four interfaces available in *hypre* with various examples illustrating their use. Particular attention is given to the new semi-structured grid interface, where both a block-structured grid example and a structured adaptive mesh refinement (AMR) example are given. Section 7 is a brief overview of the solvers and preconditioners currently available in *hypre*. Section 8 gives additional information on how to obtain and build the library. The paper concludes with some comments on future plans to enhance the library.

2. Conceptual Interfaces. Each application to be implemented lends itself to natural ways of thinking of the problem. If the application uses structured grids, a

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551.

natural way of formulating it would be in terms of grids and stencils, whereas for an application that uses unstructured grids and finite elements it is more natural to access the preconditioners and solvers via elements and element stiffness matrices. Consequently the provision of various interfaces facilitates the use of the library.

Conceptual interfaces also decrease the coding burden for users. The most common interface used in libraries today is a linear-algebraic one. This interface requires that the user compute the mapping of their discretization to row-column entries in a matrix. This code can be quite complex; for example, consider the problem of ordering the equations and unknowns on the composite grids used in structured AMR codes. The use of a conceptual interface merely requires the user to input the information that defines the problem to be solved, leaving the forming of the actual linear system as a library implementation detail hidden from the user.

Another reason for conceptual interfaces—maybe the most compelling one—is that they provide access to a large array of powerful scalable linear solvers that need the extra information beyond just the matrix. For example, geometric multigrid (GMG) cannot be used through a linear-algebraic interface, since it is formulated in terms of grids. Similarly, in many cases, these interfaces allow the use of other data storage schemes with less memory overhead and provide for more efficient computational kernels.

Fig. 2.1 illustrates the idea of conceptual interfaces. The level of generality increases from left to right. On the left are specific interfaces with algorithms and data structures that take advantage of more specific information. On the right are more general interfaces, algorithms and data structures. Note that the more specific interfaces also give users access to general solvers like algebraic multigrid (AMG) or incomplete LU factorization (ILU). The top row shows various concepts: structured grids, composite grids, unstructured grids or just plain matrices. In the second row, various solvers and preconditioners are listed. Each of these requires different information from the user, which is provided through the conceptual interfaces. For example, GMG needs a structured grid and can only be used with the leftmost interface. AMGe [2], an algebraic multigrid method, needs finite element information, whereas general solvers can be used with any interface. The bottom row contains a list of data layouts or matrix / vector storage schemes that can be used for the implementation of the various algorithms. The relationship between linear solver and storage scheme is similar to that of interface and linear solver.

In *hypre*, four conceptual interfaces are currently supported: a structured-grid interface, a semi-structured-grid interface, a finite-element interface, and a linear-algebraic interface. For the purposes of this paper, we will refer to these interfaces by the names `Struct`, `semiStruct`, `FEI`, and `IJ`, respectively; the actual names of the interfaces in *hypre* are slightly different. Similarly, when interface routines are discussed below, we will not strictly adhere to the prototypes in *hypre*, as these prototypes may change slightly in the future in response to user needs. Instead, we will focus on the basic design components and basic use of the interfaces, and refer the reader to the *hypre* documentation [13] for current details.

Note that *hypre* does not partition the problem, but builds the internal parallel data structures (often quite complicated) according to the partitioning of the application that the user provides.

3. The Structured-Grid Interface (`Struct`). The `Struct` interface is appropriate for scalar applications on structured grids with a fixed stencil pattern of nonzeros at each grid point. It provides access to *hypre*'s most efficient scalable

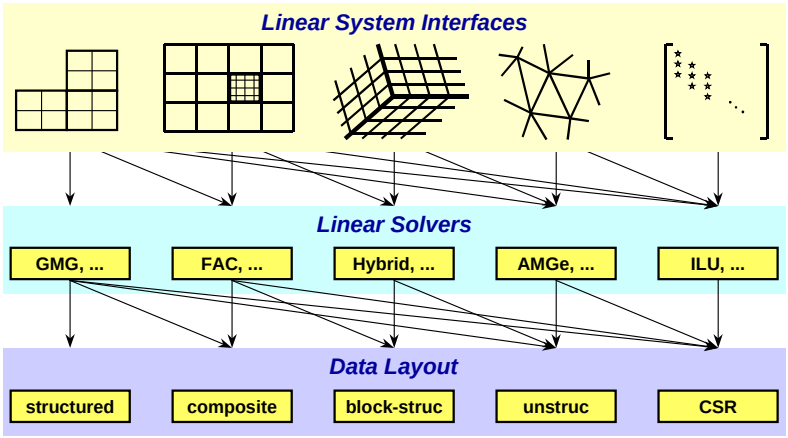

FIG. 2.1. *Graphic illustrating the notion of conceptual interfaces.*

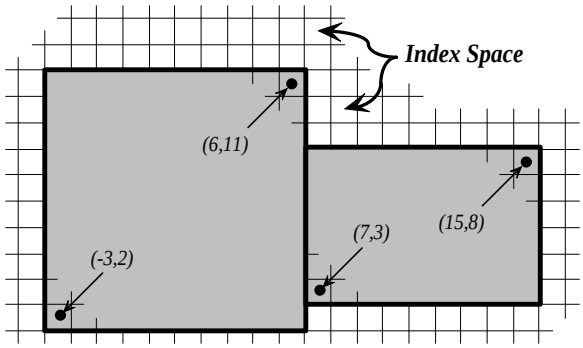

FIG. 3.1. *A box is a collection of abstract cell-centered indices, described by its minimum and maximum indices. Here, two boxes are illustrated.*

solvers for scalar structured-grid applications, the geometric multigrid methods SMG and PFMG. The user defines the *grid* and the *stencil*; the matrix and right-hand-side vector are then defined in terms of the grid and the stencil.

The `Struct` grid is described via a global d-dimensional *index space*, i.e. via integer singles in 1D, tuples in 2D, or triples in 3D (the integers may have any value, positive or negative). The global indices are used 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 (see Figure 3.1). The scalar grid data is always associated with cell centers, unlike the more general `semiStruct` interface which allows data to be associated with box indices in several different ways. Each process describes the portion of the grid that it "owns", one box at a time. Note that it is assumed that the data has already been distributed, and that it is handed to the library in this distributed form.

The stencil is described by an array of integer indices, each representing a relative offset (in index space) from some gridpoint on the grid. For example, the geometry

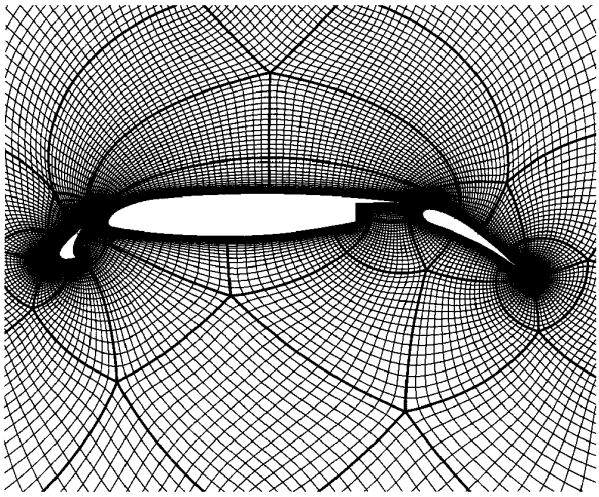

FIG. 4.1. *An example block-structured grid, distributed across many processes.*

of the standard 5-pt stencil can be represented in the following way:

$$\begin{bmatrix} & (0,1) & \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) & \end{bmatrix}. \tag{3.1}$$

After the grid and stencil are defined, the matrix coefficients are set using the `MatrixSetBoxValues()` routine with the following arguments: a box specifying where on the grid the stencil coefficients are to be set; a list of stencil entries indicating which coefficients are to be set (e.g., the "center", "south", and "north" entries of the 5-point stencil above); and the actual coefficient values.

4. The Semi-Structured-Grid Interface (`semiStruct`). The `semiStruct` interface is appropriate for applications with grids that are mostly—but not entirely—structured, e.g. block-structured grids (see Fig. 4.1), composite grids in structured AMR applications, and overset grids. In addition, it supports more general PDEs than the `Struct` interface by allowing multiple variables (system PDEs) and multiple variable types (e.g. cell-centered, face-centered, etc.). The interface provides access to data structures and linear solvers in *hypre* that are designed for semi-structured grid problems, but also to the most general data structures and solvers.

The `semiStruct` 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 Section 3) 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 4.2 for an illustration in 2D.

The `semiStruct` 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

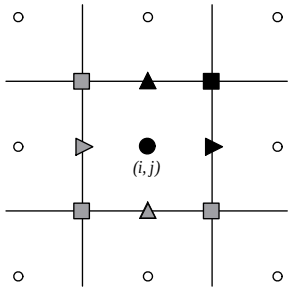

FIG. 4.2. *Grid variables in hypre are referenced by the abstract cell-centered index to the left and down in 2D (and analogously in 3D). So, in the figure, index* (i, j) *is used to reference the variables in black. The variables in grey, although contained in the pictured cell, are not referenced by the* (i, j) *index.*

suited for block-structured grid problems. Several examples are given in the following sections to illustrate these concepts.

4.1. Block-Structured Grids. In this section, we describe how to use the `semiStruct` 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

$$-\nabla \cdot (D\nabla u) + \sigma u = f \tag{4.1}$$

on the block-structured grid in Figure 4.3, where D is a scalar diffusion coefficient, and $\sigma \geq 0$. The discretization [19] introduces three different types of variables: cell-centered, x-face, and y-face. The three discretization stencils that couple these variables are given in Figure 4.4. The information in these two figures is essentially all that is needed to describe the nonzero structure of the linear system we wish to solve. Traditional linear solver interfaces require that this information first be translated to row-column entries of a matrix by defining a global ordering of the unknowns. This translation process can be quite complicated, especially in parallel. For example, face-centered variables on block boundaries are often replicated on two different processes, but they should only be counted once in the global ordering. In contrast, the `semiStruct` interface enables the description of block-structured grid problems in a way that is much more natural.

Two primary steps are involved for describing the above problem: defining the grid (and its associated variables) in Figure 4.3, and defining the stencils in Figure 4.4. The grid is defined in terms of five separate logically-rectangular parts as shown in Figure 4.5, 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 3), 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 3.1). Also note that the `semiStruct` 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.

For simplicity, we restrict our attention to the interface calls made by process 3. Each process describes through the interface only the grid data that it owns, so

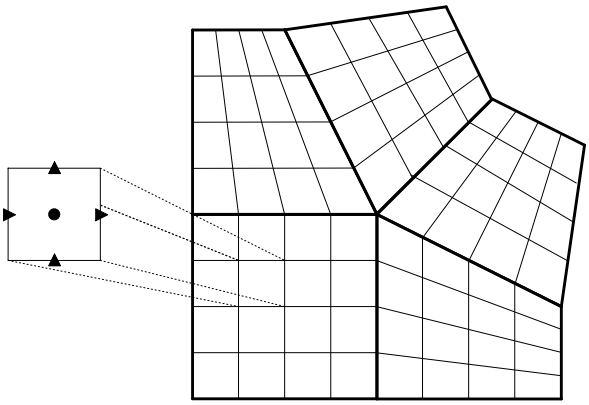

FIG. 4.3. *Block-structured grid example with five logically-rectangular blocks and three variables types: cell-centered, x-face, and y-face.*

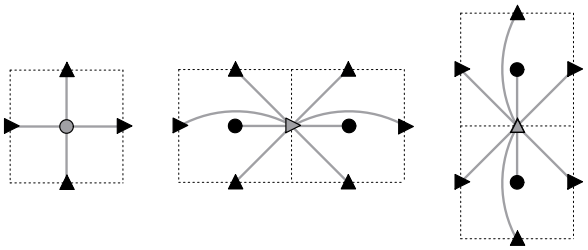

FIG. 4.4. *Discretization stencils for the cell-centered (left), x-face (middle), and y-face (right) variables for the block-structured grid example in Figure 4.3.*

process 3 needs to describe the data pictured in Figure 4.6. That is, it describes part 3 plus some additional neighbor information that ties part 3 together with the rest of the grid. To do this, the `GridSetExtents()` routine is first called, passing it the lower and upper indices on part 3, $(1, 1)$ and $(4, 4)$. Next, the `GridSetVariables()` routine is called, passing it the number of variables on part 3, 3, and the three variable types: cell-centered, x-face, and y-face.

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 Figure 4.6 also correspond to boxes on parts 2 and 4. To do this, two calls are made to the `GridSetNeighborbox()` 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 3.1). 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 `GridSetNeighborbox()` call are simply the lower and upper indices on part 3, $(1, 0)$ and $(4, 0)$, and the corresponding indices on part 2, $(1, 1)$ and $(1, 4)$. 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.

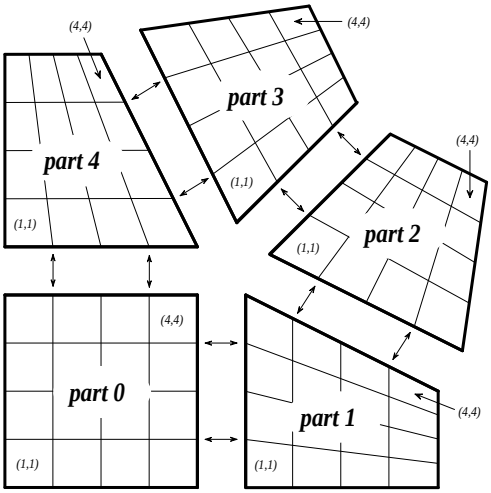

FIG. 4.5. *Assignment of parts and indices to the block-structured grid example in Figure 4.3. In this example, the data for part p lives on process p.*

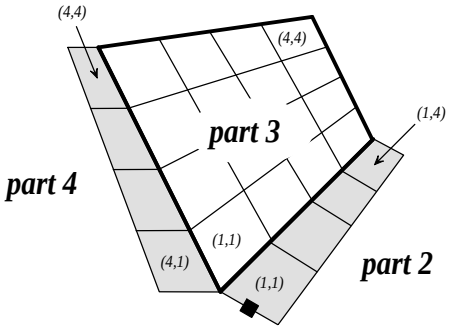

FIG. 4.6. *Grid information given to the* `semiStruct` *interface by process 3 for the example in Figure 4.3. The shaded boxes are used to relate part 3 to parts 2 and 4.*

With this 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.

An important consequence of the use of the `GridSetNeighborbox()` routine is that it can declare variables on different parts as being the same. For example, consider the highlighted face variable at the bottom of Figure 4.6. This is a single variable that lives on both part 2 and part 1. Note that process 3 cannot make this determination based solely on the information in the figure; it must use additional information on other processes. Also note that a variable may be of different types on different parts. Take for example the face variables on the boundary of parts 2 and 3. On part 2 they are x-face variables, but on part 3 they are y-face variables.

The grid is now complete and all that remains to be done is to describe the stencils

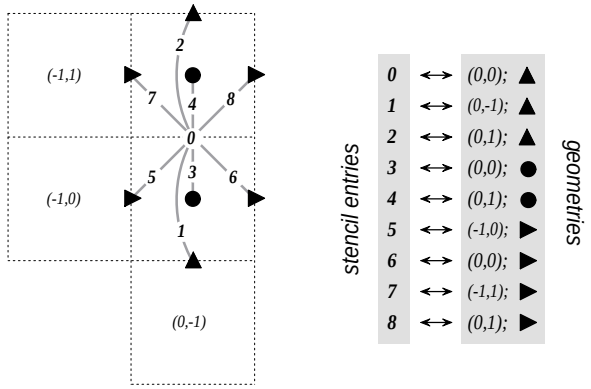

FIG. 4.7. *Assignment of labels and geometries to the y-face stencil in Figure 4.4. Stencil geometries are described relative to the* $(0, 0)$ *index for the "center" of the stencil.*

in Figure 4.4. For brevity, we consider only the description of the y-face stencil, i.e. the third stencil in the figure. To do this, the stencil entries are assigned unique labels between 0 and 8 and their "geometries" are described relative to the "center" of the stencil. This process is illustrated in Figure 4.7. Nine calls are made to the routine `StencilSetEntry()`. 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 4.2 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.

With the above, we now have a complete description of the nonzero structure for the matrix. The matrix coefficients are then easily set using the `MatrixSetValues()` routine in a manner similar to what is described in Sections 3 and 4.2. See the *hypre* documentation [13] 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 are explicitly "sewn" together by adding non-stencil couplings to the matrix graph (see Section 4.2 for more information on the use of this routine). 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, the face variable at the bottom of Figure 4.6 would now represent two different variables that live on different parts. To "sew" the parts together correctly, we 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.

4.2. Structured Adaptive Mesh Refinement. We now briefly discuss how to use the `semiStruct` interface in a structured AMR application. Consider Poisson's equation on the simple cell-centered example grid illustrated in Figure 4.8. 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

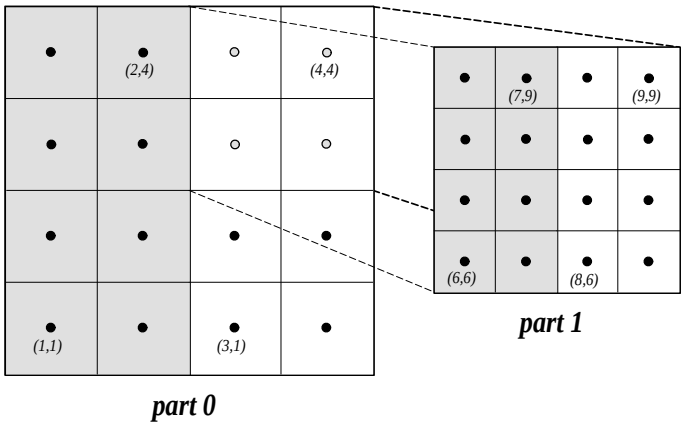

FIG. 4.8. *Structured AMR grid example. Shaded regions correspond to process 0, unshaded to process 1. The grey dots are dummy variables.*

patch (gray dots in Figure 4.8) 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 [18] for *hypre* the equations for these "dummy" unknowns should be uncoupled from the other unknowns (this can easily be done by setting all off-diagonal couplings to zero in this region).

In the example, parts are distributed across the same two processes with process 0 having the "left" half of both parts. For simplicity, we discuss the calls made by process 0 to set up the composite grid matrix. First to set up the *grid*, process 0 will make `GridSetVariables()` calls to set the number of variables, 1, and variable type, cell-centered, and `GridSetExtents()` calls to identify the portion of the grid it owns for each part: *part 0* $(1,1) \times (2,4)$, *part 1* $(6,6) \times (7,9)$. Note that in the interface there is no required rule relating the indexing on 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 (offsets $(0,0), (1,0), (-1,0), (0,1), (0,-1)$) 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 4.9. 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

$$\int_{-h/4}^{h/4} u_x(-h/4, s)ds \approx \frac{h}{2} u_x(-h/4, 0) \approx \frac{h}{2} \frac{u(0,0) - u(-3h/4, 0)}{3h/4} \tag{4.2}$$

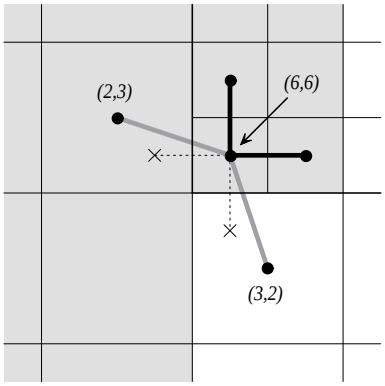

FIG. 4.9. *Coupling for equation at corner of refinement patch. Black lines (solid and broken) are stencil couplings. Gray line are non-stencil couplings.*

$$\approx \frac{2}{3}(u_{6,6} - u_{2,3}).$$

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

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.

5. The Finite Element Interface (`FEI`). The finite element interface is appropriate for users who form their systems from a finite element discretization. The interface mirrors typical finite element data structures. For an example of the finite element discretization of part of a sphere using quadrilateral elements see Figure 5.1. Though this interface is provided in *hypre*, its definition was determined elsewhere [7]. A brief summary of the actions required by the user is given below.

The use of this interface to build the underlying linear system requires two phases: initialization and loading. During the initialization phase, the structure of the finite-element data is defined. This requires the passing of control data that defines the underlying element types and solution fields; data indicating how many aggregate finite-element types will be utilized; element data, including element connectivity information to translate finite-element nodal equations to systems of sparse algebraic equations; control data for nodes that need special handling, such as nodes shared

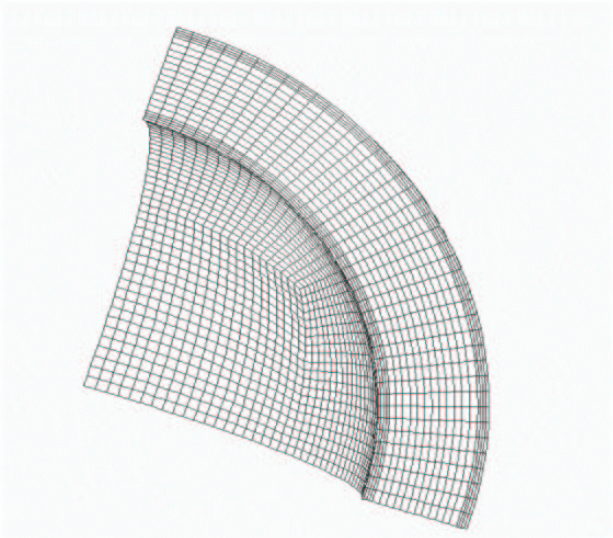

FIG. 5.1. *Finite element discretization of a sphere using quadrilateral elements.*

among processes; and data to aid in the definition of any constraint relations local to a given process. These definitions are needed to determine the underlying matrix structure and allocate memory for the load step.

During the loading phase, the structure is populated with finite-element data according to standard finite-element assembly procedures. Data passed during this step includes: boundary conditions (essential, natural, and mixed boundary conditions); element stiffness matrices and load vectors, passed as aggregate element set abstractions; and constraint relations, defined in terms of nodal algebraic weights and tables of associated nodes.

After these two phases have been completed, two further phases are necessary to obtain the solution. During the solution phase, the user needs to specify which solvers should be used and which solver parameters should be set. In the final phase, the user retrieves the solution.

For a more detailed description with specific function call definitions, the user is referred to [7], the web site *http://z.cz.sandia.goc/fei/* which contains information on newer versions of the `FEI`, as well as the *hypre* user manual. The current `FEI` version used in *hypre* is version 2.x, and comprises additional features not defined in [7].

The `FEI` differs from the `Struct` and `semiStruct` interfaces in two main aspects: the grid and the discretization type. However, note that it would be easy to augment both of these interfaces to more directly support finite element discretizations. That is, instead of describing the discretization in terms of stencils, it would be described in terms of finite-element stiffness matrices as in the `FEI`. This would allow solvers to take advantage of both the structure present in the problem as well as the finite element information. In particular, note that although the example grid in Figure 5.1 uses a finite element discretization, the grid itself is actually block-structured. Augmenting the `Struct` and `semiStruct` interfaces as just described is an area of future work of interest to the authors.

6. The Linear-Algebraic Interface (`IJ`). The `IJ` interface is the traditional linear-algebraic interface. Here, the user defines the right hand side and the matrix in the general linear-algebraic sense, i.e. in terms of row and column indices. This interface provides access only to the most general data structures and solvers and as such should only be used when none of the grid-based interfaces is applicable.

As with the other interfaces in *hypre*, the `IJ` interface expects to get the data in distributed form. Matrices are assumed to be distributed across p processes by contiguous blocks of rows. That is, the matrix must be blocked as follows:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}, \tag{6.1}$$

where each submatrix A_k is "owned" by a single process k. A_k contains the rows $n_k, n_k + 1, ..., n_{k+1} - 1$, where n_0 is arbitrary (n_0 is typically set to either 0 or 1).

First, the user creates an empty `IJ` matrix object on each process by specifying the row extents, n_k and $n_{k+1} - 1$. Next, the object type needs to be set. The object type determines the underlying data structure. Currently only one data structure, the ParCSR matrix data structure, is available. However, work is underway to add other data structures. Additional data structures are desirable for various reasons, e.g. to be able to link to other packages, such as PETSc. Also, if additional matrix information is known, more efficient data structures are possible. For example, if the matrix is symmetric, it would be advantageous to design a data structure that takes advantage of symmetry. Such an approach could lead to a significant decrease in memory usage. Another data structure could be based on blocks and thus make better use of the cache. Small blocks could naturally occur in matrices derived from systems of PDEs, and be processed more efficiently in an implementation of the nodal approach for systems AMG.

After setting the object type, the user can give estimates of the expected row sizes to increase efficiency. Providing additional detail on the nonzero structure and distribution of the matrix can lead to even more efficiency, with significant savings in time and memory usage. In particular, if information is given about how many column indices are "local" (i.e., between n_k and $n_{k+1} - 1$), and how many are not, both the ParCSR and PETSc matrix data structures can take advantage of this information during construction.

The matrix coefficients are set via the `MatrixSetValues()` routine, which allows a great deal of flexibility (more than its typical counterpart routines in other linear solver libraries). For example, one call to this routine can set a single coefficient, a row of coefficients, submatrices, or even arbitrary collections of coefficients. However, each process should only set those values that it "owns" according to the previously defined row partitioning. This is accomplished with the following parameters, describing which matrix coefficients are being set:

- `nrows`: the number of rows,
- `ncols`: the number of coefficients in each row,
- `rows`: the global indices of the rows,
- `cols`: the column indices of each coefficient,
- `values`: the actual values of the coefficients.

It is also possible to add values to the coefficients with the `MatrixAddValues()` call.

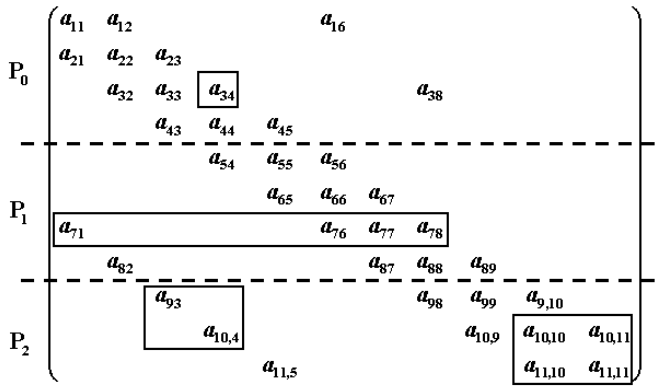

FIG. 6.1. *An example of a ParCSR matrix, distributed across 3 processes.*

Figure 6.1 illustrates this for the example of an 11×11-matrix, distributed across 3 processes. Here, rows 1–4 reside on process 0, rows 5–8 on process 1 and rows 9–11 on process 2. We now describe how to define the above parameters to set the coefficients in the boxes in Figure 6.1.

On process 0, only one element a_{34} is to be set, which requires the following parameters: `nrows` = 1, `ncols` = [1], `rows` = [3], `cols` = [4], `values` = $[a_{34}]$. On process 1, the third (local) row is defined with the parameters: `nrows` = 1, `ncols` = [4], `rows` = [7], `cols` = [1,6,7,8], `values` = $[a_{71}, a_{76}, a_{77}, a_{78}]$. On process 2, the values contained in two submatrices are to be set. This can be done by two subsequent calls setting one submatrix at a time, or more generally, all of the values can be set in one call with the parameters: `nrows` = 3, `ncols` = [1, 3, 2], `rows` = [9, 10, 11], `cols` = [3, 4, 10, 11, 10, 11], `values` = $[a_{93}, a_{10,4}, a_{10,10}, a_{10,11}, a_{11,10}, a_{11,11}]$.

7. Preconditioners and Solvers. The conceptual interfaces provide access to several preconditioners and solvers in *hypre* (we will refer to them both as solvers in the sequel, except when noted). Table 7.1 lists the current solver availability. We expect to update this table continually in the future with the addition of new solvers in *hypre*, and potentially with the addition of solvers in other linear solver packages (e.g., PETSc). We also expect to update the `Struct` interface column, which should be completely filled in.

Great efforts have been made to generate highly efficient codes. Of particular concern has been the scalability of the solvers. Roughly speaking, a method is *scalable* if the time required to produce the solution remains essentially constant as both the problem size and the computing resources increase. All methods implemented here are generally scalable per iteration step, the multigrid methods are also scalable with regard to iteration count.

The solvers use MPI for parallel processing. Most of them have also been threaded using OpenMP, making it possible to run *hypre* in a mixed message-passing / threaded

Solvers	Conceptual Interfaces			
	`Struct`	`semiStruct`	`FEI`	`IJ`
Jacobi	x			
SMG	x			
PFMG	x			
Split		x		
BoomerAMG		x	x	x
ParaSails		x	x	x
PILUT		x	x	x
Euclid		x	x	x
PCG	x	x	x	x
GMRES	x	x	x	x
BiCGSTAB	x	x	x	x
Hybrid	x	x	x	x

TABLE 7.1
Current solver availability via hypre conceptual interfaces.

mode, of potential benefit on clusters of SMPs.

All of the solvers can be used as stand-alone solvers, except for ParaSails, Euclid and PILUT which can only be used as preconditioners. For most problems, it is recommended that one of the Krylov methods be used in conjunction with a preconditioner. The Hybrid solver can be a good option for time-dependent problems, where a new matrix is generated at each time step, and where the matrix properties change over time (say, from being highly diagonally dominant to being weakly diagonally dominant). This solver starts with diagonal-scaled conjugate gradient (DSCG) and automatically switches to multigrid-preconditioned conjugate gradient (where the multigrid preconditioner is set by the user) if DSCG is converging too slowly. SMG [21, 3] and PFMG [1, 9] are parallel semicoarsening methods, with the more robust SMG using plane smoothing and the more efficient PFMG using pointwise smoothing. The Split solver is a simple iterative method based on a regular splitting of the matrix into its "structured" and "unstructured" components, where the structured component is inverted using either SMG or PFMG. This is currently the only solver that takes advantage of the structure information passed in through the `semiStruct` interface, but solvers such as the Fast Adaptive Composite-Grid method (FAC) [18] will also be made available in the future. *BoomerAMG* [12] is a parallel implementation of algebraic multigrid with various coarsening strategies [20, 8, 11] and smoothers (including the conventional pointwise smoothers such as Jacobi, as well as more complex smoothers such as ILU, sparse approximate inverse and Schwarz [22]). ParaSails [4, 5] is a sparse approximate inverse preconditioner. PILUT [17] and Euclid [14, 15] are ILU algorithms, where PILUT is based on Saad's dual-threshold ILU algorithm, and Euclid supports variants of ILU(k) as well as ILUT preconditioning.

After the matrix and right hand side are set up as described in the previous sections, the preconditioner (if desired) and the solver are set up, and the linear system can finally be solved. For many of the preconditioners and solvers, it might be desirable to choose parameters other than the default parameters, e.g. the strength threshold or smoother for *BoomerAMG*, a drop tolerance for PILUT, the dimension of the Krylov space for GMRES, or convergence criteria, etc. These parameters are

defined using `Set()` routines. Once these parameters have been set to the satisfaction of the user, the preconditioner is passed to the solver with a `SetPreconditioner()` call. After this has been accomplished, the problem is solved by calling first the `Setup()` routine (this call may become optional in the future) and then the `Solve()` routine. When this has finished, the basic solution information can be extracted using a variety of `Get()` calls.

8. Additional Information. The *hypre* library can be downloaded by visiting the *hypre* home page [13]. It can be built by typing `configure` followed by `make`. There are several options that can be used with configure. For information on how to use those, one needs to type `configure --help`. Although *hypre* is written in C, it can also be called from Fortran. More specific information on *hypre* and how to use it can be found in the users manual and the reference manual, which are also available at the same URL.

9. Conclusions and Future Work. The introduction of conceptual interfaces in *hypre* gives applications users a more natural means for describing their linear systems, and provides access to an array of powerful linear solvers that require additional information beyond just the matrix. In this paper, the design and use of these interfaces was discussed, paying particular attention to the new semi-structured grid interface, where both a block-structured grid example and a structured adaptive mesh refinement (AMR) example were given.

The *hypre* library contains a variety of scalable preconditioners and solvers. Nevertheless, as new research leads to better and more efficient algorithms, new preconditioners will be added and old preconditioners will be improved. On the list of new solvers to be made available in *hypre* is AMGe, an algebraic multigrid method based on the use of local finite element stiffness matrices [2, 16]. This method has proven to be more robust and to converge faster than classical AMG for some problems, e.g. elasticity problems. AMGe will be available directly through the `FEI` interface (and potentially through updated `Struct` and `semiStruct` interfaces; see below). FAC [18] is another solver that we would like to make available in the future. This solver is the method of choice for many structured AMR application code developers.

The `Struct` and `semiStruct` interfaces are currently stencil-based, but it is possible to augment these interfaces to more directly support finite element discretizations. This is another possible direction of future work.

Finally, although the `semiStruct` interface is currently only available in *hypre*, plans are being made to separate the interface from the library, releasing it as a stand-alone specification that other linear solver libraries can also implement and support. In addition to the specification, a partial implementation will be released that library writers can then build on.

Acknowledgments. This paper would not have been possible without the many contributions of the *hypre* library developers: Edmond Chow, Andy Cleary, Van Henson, David Hysom, Mike Lambert, Jeff Painter, Charles Tong and Tom Treadway. This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

REFERENCES

[1] S. F. Ashby and R. D. Falgout, *A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations*, Nuclear Science and Engineering, 124 (1996), pp. 145–159. Also available as LLNL Technical Report UCRL-JC-122359.

[2] M. Brezina, A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge, *Algebraic multigrid based on element interpolation (AMGe)*, SIAM J. Sci. Comput., 22 (2000), pp. 1570–1592. Also available as LLNL technical report UCRL-JC-131752.

[3] P. N. Brown, R. D. Falgout, and J. E. Jones, *Semicoarsening multigrid on distributed memory machines*, SIAM J. Sci. Comput., 21 (2000), pp. 1823–1834. Special issue on the Fifth Copper Mountain Conference on Iterative Methods. Also available as LLNL technical report UCRL-JC-130720.

[4] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1804–1822. Also available as LLNL Technical Report UCRL-JC-130719 Rev.1.

[5] E. Chow, *Parallel implementation and practical use of sparse approximate inverses with a priori sparsity patterns*, Int'l J. High Perf. Comput. Appl., 15 (2001), pp. 56–74. Also available as LLNL Technical Report UCRL-JC-138883 Rev.1.

[6] E. Chow, A. J. Cleary, and R. D. Falgout, *Design of the hypre preconditioner library*, in Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, M. Henderson, C. Anderson, and S. Lyons, eds., Philadelphia, PA, 1998, SIAM. Held at the IBM T.J. Watson Research Center, Yorktown Heights, New York, October 21-23, 1998. Also available as LLNL technical report UCRL-JC-132025.

[7] R. L. Clay, K. D. Mish, I. J. Otero, L. M. Taylor, and A. B. Williams, *An annotated reference guide to the finite-element interface (fei) specification: version 1.0*. Sandia National Laboratories report SAND99-8229, January 1999.

[8] A. J. Cleary, R. D. Falgout, V. E. Henson, and J. E. Jones, *Coarse-grid selection for parallel algebraic multigrid*, in Proc. of the Fifth International Symposium on: Solving Irregularly Structured Problems in Parallel, vol. 1457 of Lecture Notes in Computer Science, New York, 1998, Springer–Verlag, pp. 104–115. Held at Lawrence Berkeley National Laboratory, Berkeley, CA, August 9–11, 1998. Also available as LLNL Technical Report UCRL-JC-130893.

[9] R. D. Falgout and J. E. Jones, *Multigrid on massively parallel architectures*, in Multigrid Methods VI, E. Dick, K. Riemslagh, and J. Vierendeels, eds., vol. 14 of Lecture Notes in Computational Science and Engineering, Berlin, 2000, Springer, pp. 101–107. Proc. of the Sixth European Multigrid Conference held in Gent, Belgium, September 27-30, 1999. Also available as LLNL technical report UCRL-JC-133948.

[10] R. D. Falgout and U. M. Yang, *hypre: a library of high performance preconditioners*, in Computational Science - ICCS 2002 Part III, P. Sloot, C. Tan., J. Dongarra, and A. Hoekstra, eds., vol. 2331 of Lecture Notes in Computer Science, Springer–Verlag, 2002, pp. 632–641. Also available as LLNL Technical Report UCRL-JC-146175.

[11] K. Gallivan and U. M. Yang, *Efficiency issues in parallel coarsening schemes*, Tech. Report UCRL-JC-XXXXXX, Lawrence Livermore National Laboratory, 2002.

[12] V. E. Henson and U. M. Yang, *BoomerAMG: a parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177. Also available as LLNL technical report UCRL-JC-141495.

[13] *hypre: High performance preconditioners*. http://www.llnl.gov/CASC/hypre/.

[14] D. Hysom and A. Pothen, *Efficient parallel computation of ILU(k) preconditioners*, SC99, ACM, November 1999. published on CDROM, ISBN #1-58113-091-0, ACM Order #415990, IEEE Computer Society Press Order # RS00197.

[15] ———, *A scalable parallel algorithm for incomplete factor preconditioning*, SIAM J. Sci. Comput., 22 (2001), pp. 2194–2215.

[16] J. E. Jones and P. S. Vassilevski, *AMGe based on element agglomeration*, SIAM J. Sci. Comput., 23 (2001), pp. 109–133. Also available as LLNL technical report UCRL-JC-135441.

[17] G. Karpis and V. Kumar, *Parallel threshold-based ILU factorization*, Tech. Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, 1998.

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

[19] J. Morel, R. M. Roberts, and M. J. Shashkov, *A local support-operators diffusion discretization scheme for quadrilateral r-z meshes*, Journal of Computational Physics, 144 (1998), pp. 17–51.

[20] J. W. Ruge and K. Stüben, *Algebraic multigrid (AMG)*, in Multigrid Methods, S. F. McCormick, ed., vol. 3 of Frontiers in Applied Mathematics, SIAM, Philadelphia, PA, 1987, pp. 73–130.

[21] S. Schaffer, *A semi-coarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients*, SIAM J. Sci. Comput., 20 (1998), pp. 228–242.

[22] U. M. Yang, *On the use of Schwarz smoothing in AMG.* Presentation at the Tenth Copper Mountain Conference on Multigrid Methods. Also available as LLNL technical report UCRL-VG-142120, 2001.