LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-JRNL-205459

The Design and Implementation of *hypre*, a Library of Parallel High Performance Preconditioners

R. D. Falgout, J. E. Jones, and U. M. Yang

July 23, 2004

Submitted as a chapter contribution for: Numerical Solution of Partial Differential Equations on Parallel Computers

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

The Design and Implementation of *hypre*, a Library of Parallel High Performance Preconditioners

Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551, USA
rfalgout@llnl.gov,jjones@llnl.gov,umyang@llnl.gov

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.

The *hypre* library [14, 17] 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 most current generation solver libraries [9]. *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 and 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 implementation of the conceptual interfaces in *hypre*. The paper is organized as follows. The first two sections are of general interest. We begin in Section 2 with an introductory dis-

cussion of conceptual interfaces and point out the advantages of matching the linear solver interface with the natural concepts (grids, stencils, elements, etc.) used in the application code discretization. In Section 3, we discuss *hypre*'s object model, which is built largely around the notion of an operator. Sections 4–7 discuss specific conceptual interfaces available in *hypre* and include various examples illustrating their use. These sections are intended to give application programmers an overview of each specific conceptual interface. We then discuss some implementation issues in Section 8. This section may be of interest to applications programmers, or more likely, others interested in linear solver code development on large scale parallel computers. The next two sections are aimed at application programmers potentially interested in using *hypre*. Section 9 gives a brief overview of the solvers and preconditioners currently available in *hypre*, and Section 10 contains 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 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 different conceptual views of the problem being solved (*hypre*'s so-called conceptual 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. 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 [4], 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.

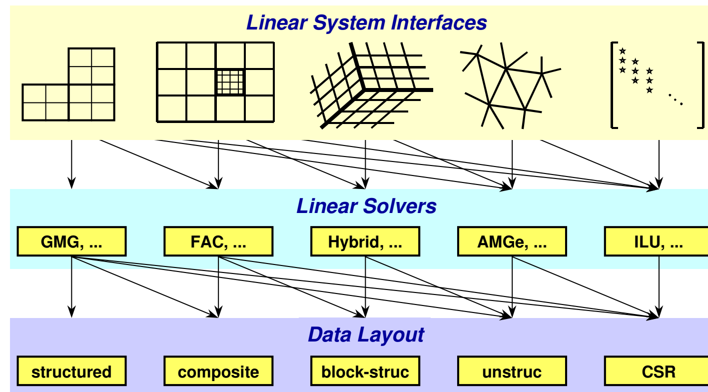

Fig. 1. Graphic illustrating the notion of conceptual interfaces.

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 [17] 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 Object Model

In this section, we discuss the basic object model for *hypre*, illustrated in Figure 2. This model has evolved since the version presented in [9], but the core design ideas have persisted. We will focus here on the primary components of the *hypre* model, but encourage the reader to see [9] for a discussion of other useful design ideas also utilized in the library.

Note that, although *hypre* uses object-oriented (OO) principles in its design, the library is actually written in C, which is not an object-oriented language. The library also provides an interface for Fortran, still another non-OO language. In addition, the next generation of interfaces in *hypre* are being developed through a tool called Babel [2], which makes it possible to generate interfaces automatically for a wide variety of languages (e.g., C, C++, Fortran77, Fortran90, Python, and Java) and provides support for important OO concepts such as polymorphism. An object model such as the one in Figure 2 is critical to the use of such a tool.

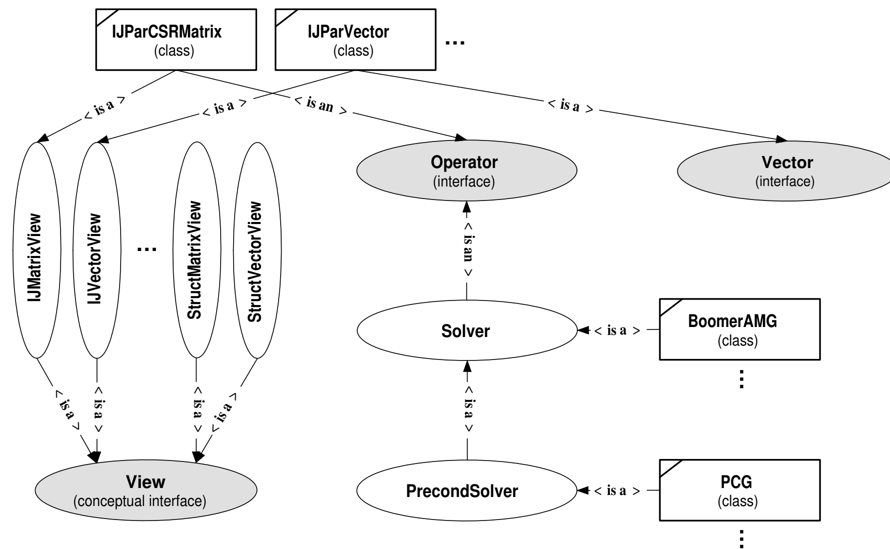

Fig. 2. Illustration of the *hypre* object model. Ovals represent abstract interfaces and rectangles are concrete classes. The `View` interface encapsulates the conceptual interface idea described in Section 2.

Central to the design of *hypre* is the use of multiple inheritance of interfaces. An *interface* is simply a collection of routines that can be invoked on an object (e.g., a matrix-vector multiply routine). The base interfaces in the *hypre* model are: `View`, `Operator`, and `Vector` (note that we use slightly different names here than those used in the *hypre* library). The `View` interface

represents the notion of a conceptual interface as described in Section 2. This interface serves as our means of looking at (i.e., "viewing") the data in matrix and vector objects. Specific views such as `IJMatrixView` are inherited from, or *extend*, the base interface. That is, `IJMatrixView` contains the routines in `View`, plus routines specific to its linear-algebraic way of looking at the data.

The `Vector` interface is fairly standard across most linear solver libraries. It consists of basic routines such as `Dot()` (vector inner product) and `Axpy()` (vector addition).

The `Operator` interface represents the general notion of operating on vectors to produce an output vector, and primarily consists of the routine `Apply()`. As such, it unifies many of the more common mathematical objects used in linear (and even nonlinear) solver libraries, such as matrices, preconditioners, iterative methods, and direct methods. The `Solver` and `PrecondSolver` interfaces are extensions of the `Operator` interface. Here, `Apply()` denotes the action of solving the linear system, i.e., the action of "applying" the solution operator to the right-hand-side vector. The two solver interfaces contain a number of additional routines that are common to iterative methods, such as `SetTolerance()` and `GetNumIterations`. But, the main extention is the addition of the `SetOperator()` routine, which defines the operator in the linear system to be solved. The `PrecondSolver` extends the `Operator` interface further with the addition of the `SetPreconditioner()` routine.

One novel feature of the *hypre* object model is the extensive use of the `Operator` interface. In particular, note that both `Solver` and `PrecondSolver` are also `Operator` interfaces. Furthermore, the `SetOperator()` interface routine takes as input an object of type `Operator`, and the `SetPreconditioner()` routine takes as input an object of type `Solver` (an `Operator`). The latter fact is of interest. In *hypre*, there is currently no specific preconditioner type. Instead, preconditioning is considered to be a *role* played by solvers.

Another fundamental design feature in the *hypre* library is the separation of the `View` and `Operator` interfaces in matrix classes such as `IJParCSRMatrix`. In this particular class, both interfaces are present, and the underlying data storage is a parallel compressed sparsed row format. This format is required in solvers like `BoomerAMG`. However, the inherited `IJMatrixView` interface is generic, which allows users to write generic code for constructing matrices. As a result, the underlying storage format can be changed (giving access to potentially different solvers) by modifying only one or two lines of code.

The separation of the `View` and `Operator` interfaces also makes it possible for objects to support one functionality without supporting the other. The classic example is the so-called matrix-free linear operator (or matrix-free preconditioner), which is an object that performs matrix-vector multiplication, but does not offer access to the coefficients of the matrix. This has proven to be a useful technique in many practical situations.

4 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 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). 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.

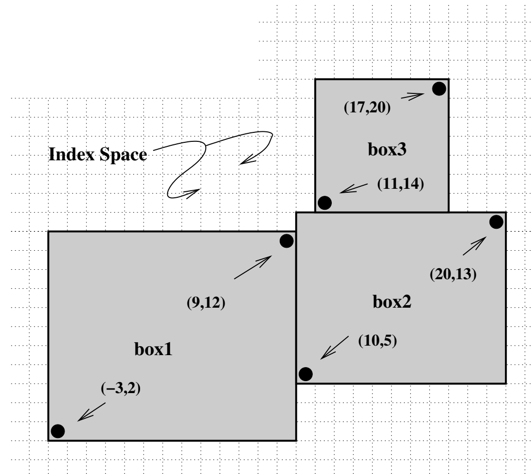

Fig. 3. A box is a collection of abstract cell-centered indices, described by its minimum and maximum indices. Here, three boxes are illustrated.

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

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

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.

5 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, 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 4) 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 for an illustration in 2D.

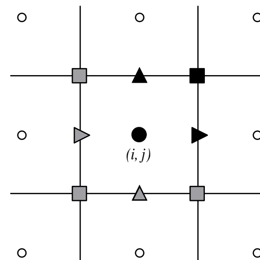

Fig. 4. Grid variables in *hypre* are referenced by the abstract cell-centered index to the left and down in 2D (and analogously in 3D). So, in the figure, index (i, j) is used to reference the variables in black. The variables in grey, although contained in the pictured cell, are not referenced by the (i, j) index.

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

5.1 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

$$-\nabla \cdot (D \nabla u) + \sigma u = f \tag{2}$$

on the block-structured grid in Figure 5, where D is a scalar diffusion coefficient, and $\sigma \geq 0$. The discretization [22] 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 6. 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 5, and defining the stencils in Figure 6. The grid is defined in terms of five separate logically-rectangular parts as shown in Figure 7, and each part is given a unique label between 0 and 4. Each part consists of a single box with lower index $(1, 1)$ and upper index $(4, 4)$ (see Section 4), 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). 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 process 3 needs to describe the data pictured in Figure 8.

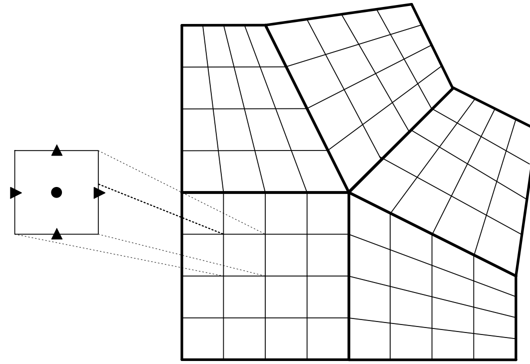

Fig. 5. Block-structured grid example with five logically-rectangular blocks and three variables types: cell-centered, x-face, and y-face.

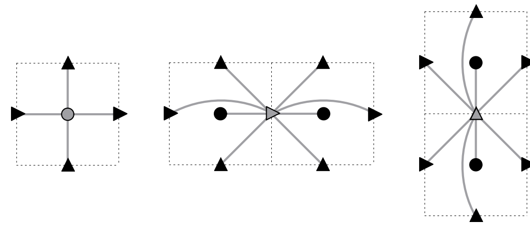

Fig. 6. Discretization stencils for the cell-centered (left), x-face (middle), and y-face (right) variables for the block-structured grid example in Figure 5.

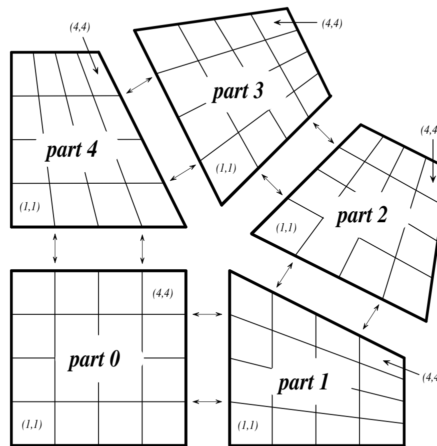

Fig. 7. Assignment of parts and indices to the block-structured grid example in Figure 5. In this example, the data for part p lives on process p.

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.

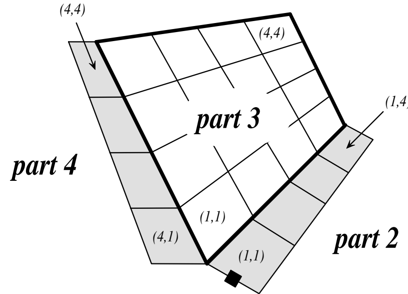

Fig. 8. Grid information given to the `semiStruct` interface by process 3 for the example in Figure 5. The shaded boxes are used to relate part 3 to parts 2 and 4.

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 8 also correspond to boxes on parts 2 and 4. To do this, two calls are made to the `GridSetNeighborbox()` routine. With this additional neighbor information, it is 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 8. 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 in Figure 6. 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 9. 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 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.

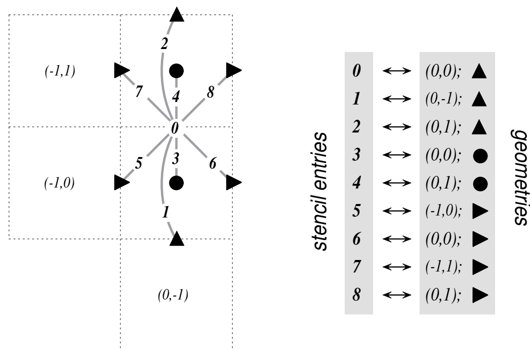

Fig. 9. Assignment of labels and geometries to the y-face stencil in Figure 6. Stencil geometries are described relative to the $(0, 0)$ index for the "center" of the stencil.

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 4 and 5.2. See the *hypre* documentation [17] for details.

An alternative approach for describing the above problem through the interface is to use the `GraphAddEntries()` routine. In this approach, the five parts are explicitly "sewn" together by adding non-stencil couplings to the matrix graph (see Section 5.2 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 8 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.

5.2 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 10. 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 10) 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 [21] 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).

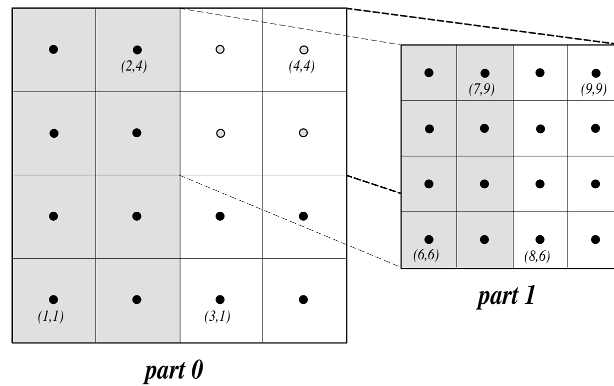

Fig. 10. Structured AMR grid example. Shaded regions correspond to process 0, unshaded to process 1. The grey dots are dummy variables.

In the example, parts are distributed across the same two processes with process 0 having the "left" half of both parts. 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 (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 at any point 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 the illustration in Figure 11. Following the discretization procedure above results in an equation for the variable at cell $(6,6)$ involving 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.

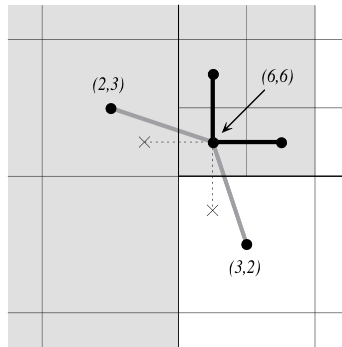

Fig. 11. Coupling for equation at corner of refinement patch. Black lines (solid and broken) are stencil couplings. Gray line are non-stencil couplings.

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.

6 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. Though this interface is provided in *hypre*, its definition was determined elsewhere [10]. 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 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.

For a more detailed description with specific function call definitions, the user is referred to [10], 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 [10].

7 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:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}, \tag{3}$$

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.

Figure 12 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 12.

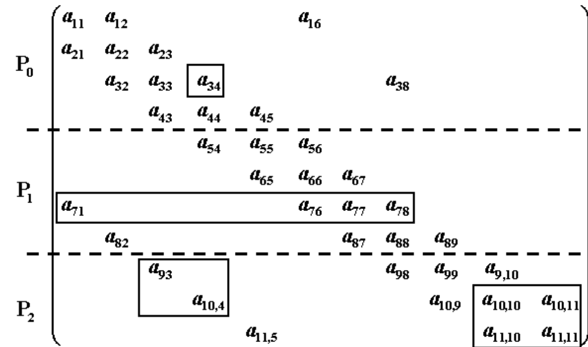

Fig. 12. An example of a ParCSR matrix, distributed across 3 processes.

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` = $[a_{93}, a_{10,4}, a_{10,10}, a_{10,11}, a_{11,10}, a_{11,11}]$.

8 Implementation

This section discusses implementation issues of the various interfaces. It describes the data structures and discusses how to obtain neighborhood information. The focus is on those issues which impact performance on a large scale parallel computer. A more detailed analysis can be found in [12].

8.1 IJ Data Structure and Communication Package

Currently only one data structure, the ParCSR matrix data structure, is available. It is similar to the parallel AIJ matrix format in PETSc [3]. It is based on the sequential compressed sparse row (CSR) data structure.

A *ParCSR matrix* consists of p parts A_k, $k = 1, \ldots, p$ (see (3)), where A_k is stored locally on processor k. Each A_k is split into two matrices D_k and O_k. D_k is a square matrix of order $n_k \times n_k$, where $n_k = r_{k+1} - r_k$ is the number of rows residing on processor k. D_k contains all coefficients a^k_{ij}, with $r_k \leq i, j \leq r_{k+1} - 1$, i.e. column indices pointing to rows stored locally. The second matrix O_k contains those coefficients of A_k, whose column indices j point to rows that are stored on other processors with $j < r_k$ or $j \geq r_{k+1}$. Both matrices are stored in CSR format. Whereas D_k is a CSR matrix in the usual sense, in O_k, which in general is extremely sparse with many zero columns and rows, all non-zero columns are renumbered for greater efficiency. Thus, one needs to generate an array of length n_{O_k} that defines the mapping of local to global column indices, where n_{O_k} is the number of non-zero columns of O_k. We denote this array as COL_MAP_O_k.

An example of an 11×11 matrix that illustrates this data structure is given in Figure 13. The matrix is distributed across 3 processors, with 4 rows on Processor 1 and Processor 2, and 3 rows on Processor 3. The 4×4 matrices D_1 and D_2 and the 3×3 matrix D_3 are illustrated as boxes. The remaining coefficients are compressed into the 4×3 matrix O_1 (with COL_MAP_O_1 = (5,6,8)), the 4×4 matrix O_2 (with COL_MAP_O_2 = (1,2,4,9)) and the 3×4 matrix O_3 (with COL_MAP_O_3 = (3,4,5,8)). Since often O_p is extremely sparse, efficiency can be further increased by introducing a row mapping that compresses zero rows by renumbering the non-zero rows.

For parallel computations it is necessary to generate the neighborhood information, which is assembled in a communication package. The communication package is based on the concept of what is needed for a matrix-vector multiplication. Let us consider the parallel multiplication of a matrix A with a vector $\mathbf{x}$. Processor k owns rows r_k through $r_{k+1} - 1$ as well as the corresponding chunk of $\mathbf{x}$, $\mathbf{x}_k = (x_{r_k}, \ldots, x_{r_{k+1}-1})^T$. In order to multiply A with $\mathbf{x}$, Processor k needs to perform the operation $A_k\mathbf{x} = D_k\mathbf{x}_k + O_k\tilde{\mathbf{x}}_k$, where $\tilde{\mathbf{x}}_k = (x_{col_map_O_k(1)}, \ldots, x_{col_map_O_k(n_{O_k})})^T$. While the multiplication of D_k and $\mathbf{x}_k$ can be performed without any communication, the elements of $\tilde{\mathbf{x}}_k$ are owned by the receive processors of k. Another necessary piece of information is the amount of data to be received by each processor. In general processor k owns elements of $\mathbf{x}$ that are needed by other processors. Consequently processor k needs to know the indices of the elements that need to be sent to each of its send processors.

In summary, the communication package on processor k consists of the following information:

- the IDs of the receive processors
- the size of data to be received by each processor

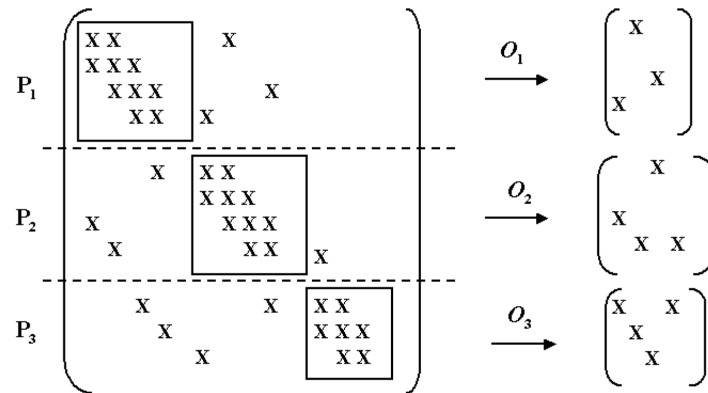

Fig. 13. An example of a ParCSR matrix, distributed across 3 processors. Matrices with local coefficients, D_1, D_2 and D_3, are shown as boxes within each processor. The remaining coefficients are compressed into the matrices O_1, O_2 and O_3.

- the IDs of the send processors
- the indices of the elements that need to be sent to each send processor

Recall that each processor by design has initially only local information available, i.e. its own range and the rows of the matrix that it owns. In order to determine the communication package, it needs to get more information from other processors. There are various ways of dealing with this situation. The simpler (and currently implemented) way is to allow each processor to own the complete partitioning information, which can easily be communicated via an MPI_Allgather. From this it can determine its receive processors. This is a reasonable approach when dealing with computers with a few thousand processors. However, when using a supercomputer with 100,000 or more processors, this approach is very expensive, requiring communication between all processors and memory usage of order $O(p)$. A better approach for this case would be to use a so-called distributed directory algorithm [23], which is a rendezvous algorithm that utilizes a concept we refer to an assumed partitioning [12]. The idea is to assume the partitioning is known by all processors, requires only $O(1)$ storage, and can be queried by an $O(1)$ function. In general, this function does not describe the actual partitioning, although one would hope that it is not too different. This approach consists of three steps. First, each processor determines where the data that it is responsible for in the assumed partitioning is stored in the actual partition. Then it computes the processors

that are responsible for the data it needs to receive according to the assumed partitioning, and sends this information to them. At the same time it receives such information from other processors. It can now compare this information with the actual ownership information that it obtained in the first step and return corrections or confirmations to the processors it received the information from. In the final step, the assumed receive processor information is updated by the actual receive processor information.

A more detailed analysis of the two approaches can be found in [12].

If matrix A has a symmetric structure, the receive processors are also send processors, and no further communication is necessary. However, this is different in the non-symmetric case. In the current implementation, the IDs of the receive processors and the amount of data to be obtained from each receive processor are communicated to all processors via a MPI_Allgatherv. For a moderate number of processors, even up to 1000, this is a reasonable approach, however it can become a potentially high cost if we consider 100,000 processors. It can be avoided by the use of MPI_Iprobe and the use of a distribution detection algorithm based on a binary tree [12].

When each processor knows its receive and send processors, the remaining necessary information can be sent directly to the receive processors, which know now its send processors as well the amount of data to be received. Since the number of neighbors and the amount of data is independent of p, the computational complexity and the storage requirement is of order $O(1)$.

8.2 `Struct` Data Structure and Neighborhood Information

The underlying matrix data structure, *Struct matrix*, contains the following.

- *Struct grid*: describes the boxes owned by the processor (local boxes) as well as information about other nearby boxes. Note that a box is stored by its “lower” and “upper” indices, called the box’s *extents*.
- *Struct stencil*: an array of indices defining the coupling pattern in the matrix.
- *data*: an array of doubles defining the coupling coefficients in the matrix.

The corresponding vector data structure is similar except is has no stencil and the data array defines the vector values. In both the vector and matrix the data array is stored so that all values associated with a given box are stored contiguously. To facilitate parallel implementation of a matrix-vector product, the vector data array includes space for values associated with a box somewhat larger than the actual box; typically including one boundary layer of cells or ghost cells (see Figure 14). Assuming that the boxes are large, the additional storage of these ghost cells is fairly small as the boundary points also take only a small percentage of the total number of points. Some of these ghost cells may be part of other boxes, owned by either the same or a different processor. Determining communication patterns for updating ghost cells is the major task in implementing the `Struct` interface in a scalable manor.

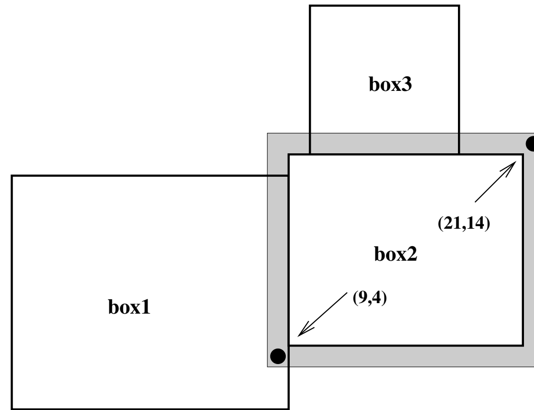

Fig. 14. For parallel computing, additional storage is allocated for cells nearby a box (ghost cells). Here, the ghost cells for BOX2 are illustrated.

Recall that in the interface, a given processor k is passed only information about the grid boxes that it owns. Determining how to update ghost cell values requires information about nearby boxes on other processors. This information is generated and stored when the Struct grid is assembled. Determining which processors own ghost cells is similar to the problem in the `IJ` interface of determining the receive processors. In the `IJ` case, this requires information about the global partitioning. In the `Struct` case, it requires information about the global grid.

The algorithm proceeds as follows. Here we let p denote the number of processors and b denote the total number of boxes in the grid (note $b \geq p$). First we accumulate information about the global grid by each processor sending the extents of its boxes to all other processors. As in the `IJ` case, this can be done using MPI_ALLGATHER with $O(\log p)$ operations. Memory usage is of order $O(b)$, since the global grid contains b boxes.

Once the global grid is known, each local box on processor k is compared to every box in the global grid. In this box-by-box comparison a *distance index* is computed which describes the spatial relationship in the index space of one box to another. This comparison of each local box to every global box involves $O(b)$ computations. Once the comparison is done, all global boxes within a specified distance (typically 2) from a local box are stored as part of a *neighborhood* data structure on processor k. Boxes not in this neighborhood can be deleted from processor k's description of the global grid. The storage requirement for the neighborhood is independent of p.

To perform the matrix-vector product, processor k must have up-to-date values in all ghost cells that will be "touched" when applying the matrix stencil at the cells owned by processor k. Determining these needed ghost cells is done by taking each box owned by the processor, shifting it by each stencil entry and finding the intersection of the shifted box with boxes in

the neighborhood data structure. As an example, consider the same layout of boxes as before with each box on a different processor (see Figure 15). If the matrix has the 5-pt stencil (1), then shifting BOX2 by the "north" stencil entry and intersecting this with BOX3 produces one of the dark shaded regions labeled as a receive box. Using this procedure, a list of receive boxes and corresponding owner processors is generated. The procedure for determining

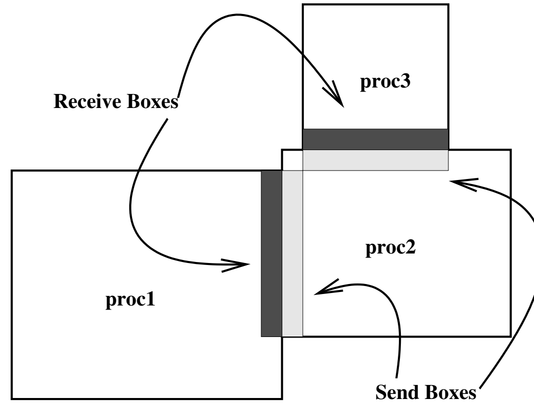

Fig. 15. The communication package for processor 2 contains send boxes (values owned by processor 2 needed by other processors) and receive boxes (values owned by other processors need by processor 2.)

the cells owned by processor k that are needed to update ghost cells on other processors is similar.

The current `Struct` interface implementation shares some of the same drawbacks as the current `IJ` interface implementation. The storage requirement in generating the neighborhood structure is $O(b)$ as the global grid is initially gathered to all processors and the box-by-box comparison to determine neighbors involves $O(b)$ operations, again note $b \geq p$. One possible approach to eliminate these drawbacks would be similar to the assumed partitioning approach described in Section 8.1. The idea is to have a function describing an assumed partitioning of the index space to processors and have this function available to all processors. Unlike the one-dimensional `IJ` partitioning, this partition would be d-dimensional. A processor would be able to determine its neighbors in the assumed partition in $O(1)$ computations and storage. A multi-phase communication procedure like that previously described for the `IJ` case could be used to determine the actual neighbors with $O(\log p)$ complexity.

8.3 `semiStruct` Data Structure

The `semiStruct` interface allows the user to choose from two underlying data structures for the matrix. One option is to use the ParCSR matrix data type discussed in Section 8.1. The second option is the *semiStruct matrix* data type which is based on a splitting of matrix non-zeros into structured and unstructured couplings $A = S + U$. The S matrix is stored as a collection of Struct matrices and the U matrix is stored as a ParCSR matrix. In our current implementation, the stencil couplings within variables of the same type are stored in S, all other couplings are stored in U. If the user selects the ParCSR data type, then all couplings are stored in U (i.e. $S = 0$.)

Since the `semiStruct` interface can use both Struct and ParCSR matrices, the issues discussed in the previous two sections impact its scalability as well. The major new issue impacting scalability is the need to relate the the semi-structured description of unknowns and the global ordering of unknowns in the ParCSR matrix, i.e. the mapping M(PART,VAR,INDEX) = GLOBAL_RANK. The implementation needs this mapping to set matrix entries in U. The global ordering of unknowns is an issue internal to the `semiStruct` implementation; the user is not aware of this ordering, and does not need to be.

In our implementation of the semi-structured grid we include the concept of BoxMap to implement this mapping. There is a BoxMap for each variable on each part; the purpose is to quickly compute the global rank corresponding to a particular index. To describe the BoxMap structure we refer to Figure 16. By cutting the index space in each direction by lines coinciding with boxes in the grid, the index space is divided into regions where each region is a either empty (not part of the grid) or is a subset of a box defining the grid. The data structure for the BoxMap corresponds to a d-dimensional table of BoxMapEntries. In three dimensions, BoxMapEntry$[i][j][k]$ contains information about the region bounded by cuts i and $i+1$ in the first coordinate direction, cuts j and $j + 1$ in the second coordinate direction, cuts k and $k + 1$ in the third coordinate direction. Among the information contained in BoxMapEntry is the first global rank (called *offset*) and the extents for the grid box which this region is a subset of. The global rank of any index in this region can be easily computed from this information.

The mapping M(PART,VAR,INDEX) = GLOBAL_RANK is computed by accessing the BoxMap corresponding to PART and VAR, searching in each coordinate direction to determine which cuts INDEX falls between, retrieving the offset and box extents from the appropriate BoxMapEntry, and computing GLOBAL_RANK from this retrieved information. This computation has $O(1)$ (independent of number of boxes and processors) complexity except for the searching step. The searching is done by a simple linear search so worst case complexity is $O(b)$ since the number of cuts is proportional to the number of boxes. However, we retain the current position in the BoxMap table, and in subsequent calls to the mapping function, we begin searching from this position. In most applications, subsequent calls will map indices nearby the

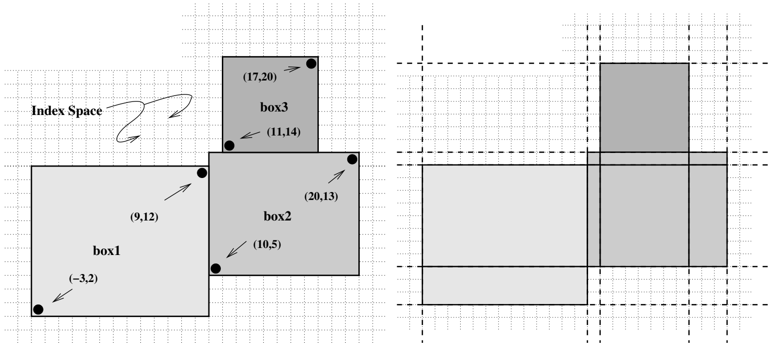

Fig. 16. The BoxMap structure divides the index space into regions defined by cuts in each coordinate direction.

previous index and the search has $O(1)$ complexity. Further optimization is accomplished by retrieving BoxMapEntries not for a single index but for an entire box of indices in the index space.

The BoxMap structure does allow quick mapping from the semi-structured description to the global ordering of the ParCSR matrix, but it does have drawbacks: storage and computational complexity of initial construction. Since we store the structure on all processors, the storage costs are $O(b)$ where b is the global number of boxes (again b is at least as large as p, the number of processors). Constructing the structure requires knowledge of all boxes (accomplished by the MPI_Allgather with $O(\log p)$ operations and $O(b)$ storage as in the `Struct` case), and then scanning the boxes to define the cuts in index space (requiring $O(b)$ operations and storage.) As in the `IJ` and `Struct` cases, it may be possible to use the notion of an assumed partitioning of the index space to remove these potential scalability issues.

9 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 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.

Solvers	Conceptual Interfaces			
	`Struct`	`semiStruct`	`FEI`	`IJ`
Jacobi	x			
SMG	x			
PFMG	x			
Split		x		
MLI			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 1. Current solver availability via *hypre* conceptual interfaces.

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 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 [25, 6] and PFMG [1, 13] 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) [21] will also be made available in the future. MLI [5] is a parallel implementation of smoothed aggregation [26]. *BoomerAMG* [16] is a parallel implementation of algebraic multigrid with various coarsening strategies [24, 11, 15] and smoothers (including the conventional pointwise smoothers such as Jacobi, as well as more complex smoothers such as

ILU, sparse approximate inverse and Schwarz [27]). ParaSails [7, 8] is a sparse approximate inverse preconditioner. PILUT [20] and Euclid [18, 19] 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.

10 Additional Information

The *hypre* library can be downloaded by visiting the *hypre* home page [17]. 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.

11 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. Work continues on the library on many fronts. We highlight two areas: providing better interfaces and solvers for structured AMR applications and scaling up to 100,000's of processors.

As in the example in Section 5.2, the current `semiStruct` interface can be used in structured AMR applications. However, the user must explicitly calculate the coarse-fine coupling coefficients which are typically defined by the composition of two equations: a structured grid stencil coupling and an interpolation formula. A planned extension to the `semiStruct` interface would allow the user to provide the these two equations separately, and the composition would be done inside the *hypre* library code. This extension would

make the `semiStruct` interface more closely match the concepts used in AMR application codes and would ease the coding burden for potential users. We are also exploring the addition of a FAC [21] solver to the library. This is a efficient multigrid solver specifically tailored to structured AMR applications.

Another area of work is ensuring good performance on very large numbers of processors. As mentioned previously, the current implementations in *hypre* are appropriate for thousands of processors but do have places where, say, the storage needed is $O(p)$. These potential bottlenecks may be of real importance on machines with 100,000 processors. The crux of the problem is that the interfaces only provide local information and determining neighboring processors requires global information. We have mentioned the assumed partitioning approach as one way we are trying to overcome this hurdle.

Acknowledgments

This paper would not have been possible without the many contributions of the *hypre* library developers: Alison Baker, Edmond Chow, Andy Cleary, Van Henson, Ellen Hill, David Hysom, Mike Lambert, Barry Lee, 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. Ashby, S. F. and Falgout, R. D.: A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, (1996). Also available as LLNL Technical Report UCRL-JC-122359
2. Babel: A language interoperability tool. www.llnl.gov/CASC/components/
3. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W., Knepley, M., McInnes, L., Smith, B., and Zhang, H.: PETSc users manual. anl-95/11-revision 2.2.0. Technical report, Aronne National Laboratory
4. Brezina, M., Cleary, A. J., Falgout, R. D., Henson, V. E., Jones, J. E., Manteuffel, T. A., McCormick, S. F., and Ruge, J. W.: Algebraic multigrid based on element interpolation (AMGe). *SIAM J. Sci. Comput.*, 22(5):1570–1592, (2000). Also available as LLNL technical report UCRL-JC-131752
5. Brezina, M., Tong, C., and Becker, R.: Parallel algebraic multigrid for structural mechanics. *SIAM Journal of Scientific Computing*, submitted, (2004). also available as Lawrence Livermore National Laboratory technical report UCRL-JRNL-204167
6. Brown, P. N., Falgout, R. D., and Jones, J. E.: 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

7. Chow, E.: A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(5):1804–1822, (2000). Also available as LLNL Technical Report UCRL-JC-130719 Rev.1
8. Chow, E.: Parallel implementation and practical use of sparse approximate inverses with a priori sparsity patterns. *Int'l J. High Perf. Comput. Appl.*, 15:56–74, (2001). Also available as LLNL Technical Report UCRL-JC-138883 Rev.1
9. Chow, E., Cleary, A. J., and Falgout, R. D.: Design of the *hypre* preconditioner library. In Henderson, M., Anderson, C., and Lyons, S., editors, *Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 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
10. Clay, R. L., Mish, K. D., Otero, I. J., Taylor, L. M., and Williams, A. B.: An annotated reference guide to the finite-element interface (fei) specification: version 1.0. Sandia National Laboratories report SAND99-8229, (1999)
11. Cleary, A. J., Falgout, R. D., Henson, V. E., and Jones, J. E.: Coarse-grid selection for parallel algebraic multigrid. In *Proc. of the Fifth International Symposium on: Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, pages 104–115, New York, (1998). Springer–Verlag. Held at Lawrence Berkeley National Laboratory, Berkeley, CA, August 9–11, 1998. Also available as LLNL Technical Report UCRL-JC-130893
12. Falgout, R., Jones, J., and Yang, U. M.: Pursuing scalability for hypre's conceptual interfaces. *ACM Transaction on Mathematical Software*, submitted, (2003). Also available as Lawrence Livermore National Laboratory technical report UCRL-JP-200044
13. Falgout, R. D. and Jones, J. E.: Multigrid on massively parallel architectures. In Dick, E., Riemslagh, K., and Vierendeels, J., 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
14. Falgout, R. D. and Yang, U. M.: *hypre*: a library of high performance preconditioners. In Sloot, P., Tan., C., Dongarra, J., and Hoekstra, A., editors, *Computational Science - ICCS 2002 Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer–Verlag, (2002). Also available as LLNL Technical Report UCRL-JC-146175
15. Gallivan, K. and Yang, U. M.: Efficiency issues in parallel coarsening schemes. Technical Report UCRL-ID-513078, Lawrence Livermore National Laboratory, (2003)
16. Henson, V. E. and Yang, U. M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, (2002). Also available as LLNL technical report UCRL-JC-141495
17. *hypre*: High performance preconditioners. http://www.llnl.gov/CASC/hypre/
18. Hysom, D. and Pothen, A.: Efficient parallel computation of ILU(k) preconditioners. SC99, ACM, (1999). published on CDROM, ISBN #1-58113-091-0, ACM Order #415990, IEEE Computer Society Press Order # RS00197
19. Hysom, D. and Pothen, A.: A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, (2001)

20. Karpis, G. and Kumar, V.: Parallel threshold-based ILU factorization. Technical Report 061, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 5455, (1998)
21. McCormick, S. F.: *Multilevel Adaptive Methods for Partial Differential Equations*, volume 6 of *Frontiers in Applied Mathematics*. SIAM Books, Philadelphia, (1989)
22. Morel, J., Roberts, R. M., and Shashkov, M. J.: A local support-operators diffusion discretization scheme for quadrilateral *r*-*z* meshes. *Journal of Computational Physics*, 144:17–51, (1998)
23. Pinar, A. and Henderson, B.: Communication support for adaptive communication. In *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific computing*, (2001)
24. Ruge, J. W. and Stüben, K.: Algebraic multigrid (AMG). In McCormick, S. F., editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, (1987)
25. Schaffer, S.: 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)
26. Vaněk, P., Mandel, J., and Brezina, M.: Algebraic multigird based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, (1996)
27. Yang, U. M.: 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)