Design of the *hypre* Preconditioner Library *

Edmond Chow Andrew J. Cleary Robert D. Falgout†

Abstract

We discuss the design of *hypre*, an object-oriented library for the solution of large sparse linear systems on parallel computers. The mathematical emphasis of *hypre* is on modern powerful and scalable preconditioners. The design of *hypre* allows it to be used as both a solver package and a framework for algorithm development. The object model used for *hypre* is more general and flexible than the current generation of solver libraries. The design of *hypre* is proceeding in parallel with the standardization efforts of the DOE/academia Linear Equation Solver Interface forum.

1 Introduction

In this paper, we discuss the design of *hypre*, an object-oriented library for the solution of large sparse linear systems on parallel computers. The need for *hypre* is motivated by the demands of computationally challenging applications such as those from the Accelerated Strategic Computing Initiative (ASCI) in the Department of Energy. The linear systems that arise in these applications are extremely large and difficult to solve, and require a new generation of solvers and massively parallel computers for their solution. Current solver technologies are insufficient by themselves for solving these problems, though they can provide components out of which more advanced solvers can be built.

This paper represents work-in-progress. As of this writing, *hypre* reflects its origin as a collection of subroutines more than a designed library. This paper documents our plans to evolve *hypre* into an integrated, flexible, and coherent object-oriented library. We note that the design of *hypre* is proceeding in parallel with the standardization efforts of the DOE/academia Linear Equation Solver Interface forum [5].

We first review parallel linear solver libraries in Section 2. In Section 3, we discuss the design goals for *hypre* in more detail. In Section 4, we describe the two key elements of *hypre* that provide the framework for meeting the *hypre* design goals, and Section 5 finishes with a summary.

2 Review of the state of the art

Many application developers code their own linear solvers directly into their application codes. This is possible with simple solvers and sequential computer architectures, but for more complicated situations, most applications utilize linear solver libraries. The older linear solver libraries are generally organized as sets of monolothic, stand-alone subroutines, and are procedural in nature. To use such a library, an application developer has to decide

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. The work was supported by the DOE2000 program and the ASCI PSE program.

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

ahead of time on a particular algorithm from a particular package, and then code the application in terms of the solver's data structure. This process is particularly difficult for parallel sparse data structures, making the required investment of effort so large that application developers rarely try more than one solver. However, it is very difficult *a priori* to know which specific solver will work best for a particular application, meaning that the ability to experiment with different solvers is very important. Also, the adoption of new algorithms is often slow due to the investment necessary to switch solvers.

Increasingly, more modern solver libraries have adopted object-oriented techniques that make them easier and more extensible to use. The object-oriented solver libraries in widest use within the DOE are the PETSc library [2], and the ISIS++ library [7]. Although there are several other object-oriented libraries in the wider community, in this paper we will concentrate on these two libraries as representative of the state of the art.

The core object models for both libraries are similar. The model consists of base classes capturing the mathematics, including `Matrix`, `Vector`, `Solver`, `LinearSystem`, `KrylovMethod`, and `Preconditioner`, as well as an auxiliary class that encapsulates information about the parallel computer and the distribution of objects to distributed memories, generally called a `Map` class. Note that the names we use here do not necessarily match exactly the names in these libraries but have obvious counterparts.

- The `Vector` class provides functions for inserting elements into the vector, as well as the standard set of linear algebraic functions such as dot product. In both PETSc and ISIS++, its implementation is straightforward so that it can be used efficiently within critical operations such as matrix-vector products.

- The `Matrix` class has several uses. It is used in the `LinearSystem` class to define problems to be solved. Its matrix-vector multiply member function is used by the `KrylovMethod` class to implement Krylov-based algorithms. In PETSc, building preconditioners is the responsibility of the matrix class. In ISIS++, preconditioners can be implemented in this way, but another paradigm is supported as well. Here, subclasses of `Matrix` introduce *access functions* that provide abstractions for accessing the underlying data structure, and preconditioners can be written in terms of these access functions. In both libraries, interfaces are also defined for inserting coefficients into matrices.

- The `LinearSystem` class is basically a composition of a `Matrix` and two vectors, representing the equation $Ax = b$.

- In both libraries, the `Solver` class is essentially a composition of a `KrylovMethod` object and one (PETSc) or two (ISIS++) `Preconditioner` objects, along with parameters such as convergence tolerances.

- The `KrylovMethod` class generalizes algorithms such as Generalized Minimum Residual (GMRES) and Conjugate Gradient (CG), of which there are many varieties. This class is used by the `Solver` class, and has the functionality of providing the next iteration of the Krylov method based on the history of previous iterates and the action of the preconditioner.

- The `Preconditioner` class includes iterative methods that are not Krylov algorithms, such as simple iterative schemes like Jacobi's method and Gauss-Seidel iterations, along with more sophisticated algorithms such as incomplete factorizations and sparse approximate inverses. Objects in this class use `Matrix` objects, as discussed above under the `Matrix` class.

We note that while ISIS++ is written in C++, PETSc is written in C, and yet it is entirely object-oriented, with encapsulation, inheritance, and polymorphism. Arguably, implementing object-oriented code in C is more difficult and less natural than in C++ or other object-oriented languages, but it does illustrate the point that the object-oriented design is far more important than the choice of language in which that design is implemented. For this reason, we do not stress any particular language in this paper. Indeed, the choice of language of implementation for *hypre* is not entirely settled, and in the end, different parts of *hypre* may be implemented in different languages. In [8], a technique for achieving language interoperability for object-oriented libraries is presented, and it is our intention to leverage this project to allow flexibility in both the calling language and the implementation language for *hypre*.

3 Design goals

We now present the design goals for *hypre* and compare them with the designs of the current generation of object-oriented solver libraries.

One major difference is that the mathematical emphasis of *hypre* is on modern powerful and scalable preconditioners, whereas the traditional emphasis has been on Krylov-type solvers, splitting-based iterative methods like Jacobi's method and Gauss-Seidel, and other high level preconditioners like polynomial preconditioning. Libraries like PETSc and ISIS++ are notably weak in parallel, scalable preconditioners, though this is more a function of the paucity of such preconditioners rather than a design choice.

To fill this void, we are designing *hypre* so that it can be used in two different modes: like existing libraries, it can be used as a stand-alone linear solver library, but in addition, it can be used as an *add-on package* to existing solver libraries. The first mode allows us to leverage existing libraries by allowing *hypre* to use Krylov solvers, basic iterative methods, and concrete matrix classes from other libraries, but all from a *hypre* interface. The second mode allows parallel scalable preconditioners from *hypre* to be used from other libraries through their own interfaces. This dual-mode goal presents many new interoperability challenges. For instance, the scalable preconditioners in *hypre* require access to the coefficients of matrices to build data structures that implement the preconditioning step, whereas the majority of algorithms in current libraries do not need significant access to coefficients. Meeting this dual-mode goal requires relaxation of the standard library model in which a library assumes it "owns the world" to a "peer model" where each library is equal.

Another area that impacts our design is the emergence of "physics-based interfaces", e.g., the Finite Element Interface [6]. For the most part, solver libraries have a linear-algebraic interface that may not be particularly well matched to an application developer's view of a problem. Linear-algebraic interfaces are more appropriate for solver developers than for application developers. The concept behind a physics-based layer is to allow application developers to describe their linear system in the language of their problem domain. For a problem discretized by the finite element method, for example, it is much more natural for the application to describe the problem in terms of element stiffness matrices and element connectivities than to describe the problem in terms of matrix columns and rows. An implementation of the finite element interface performs the mapping from the finite element interface to a data structure that is appropriate for the underlying solver. These physics-based interfaces, to the extent they can be standardized, can provide a

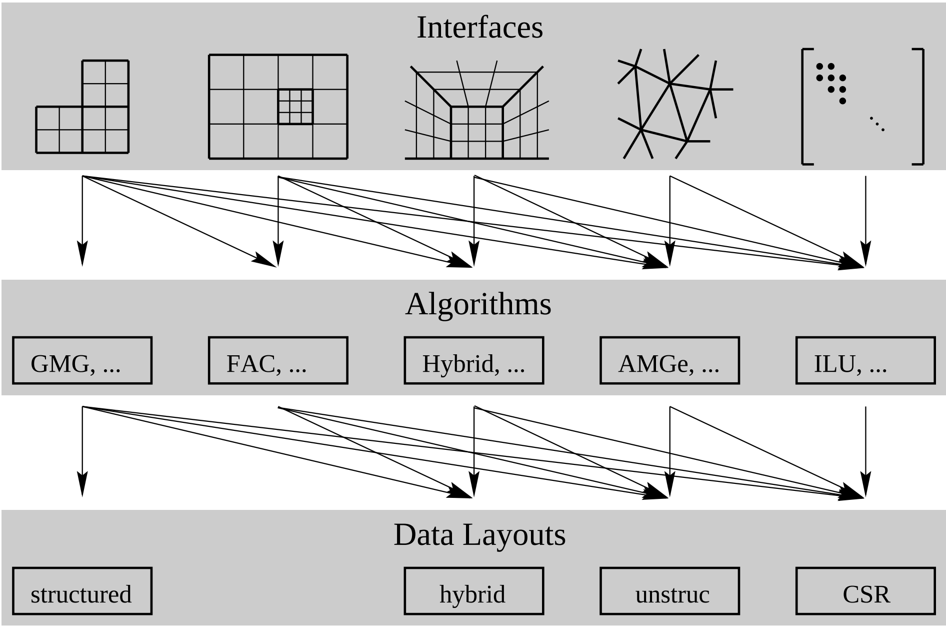

FIG. 1. *Schematic of physics-based interfaces, with specialization and performance decreasing from left-to-right.*

single user-level entry point to multiple solver libraries, allowing application developers to experiment with different libraries.

The physics-based interfaces are more than just conveniences for applications developers, however. As problems grow in size and difficulty, it becomes increasingly necessary for solvers to have more information about the problem than what is encapsulated in the traditional matrix. Multigrid solvers, for instance, often need to know information about the grids that defined the problem, and finite element-specific solvers like AMGe [3] must know the element stiffness matrices in unassembled form. Some solvers are not even defined for general matrices, for instance, alternating direction methods only make sense on structured grids where "direction" is meaningful. The physics-based interfaces provide a mechanism by which information other than just the matrix can be passed into solvers.

Figure 1 illustrates this concept. The level of generality moves from left-to-right in this figure. On the left are specific interfaces with algorithms and data structures that take advantage of this specificity. On the right are the more general interfaces, algorithms, and structures. The arrows represent the fact that many of the more specific interfaces can be mapped to more general algorithms and interfaces, and this property provides access to a wide variety of general algorithms. However, there is a subsequent loss of performance, both in the algorithms and the data structures, that results from the discarding of information. The linear-algebraic interface is the most general and interoperable, but also the least efficient. For example, we have defined an interface for finite differences on structured grids. One implementation of this interface creates a structured-grid data structure very close in definition to the interface, and this structure is used by a semi-coarsening multigrid algorithm [4] designed specifically for problems defined on these types of grids. However, another implementation of the interface creates PETSc matrices. For most problems, the multigrid code is fastest, but the PETSc implementation provides access to a much richer variety of algorithms for cases in which the structured multigrid code fails to converge. Little or no source code change is required to switch from one implementation to the other.

Recent trends have introduced another evolution in requirements for linear solver libraries that we are incorporating into the *hypre* design. Many of the newer solvers do not conform to the current solver model of a Krylov method plus a preconditioner, as we now illustrate. Parallelism, through domain partitioning, is encouraging the development of hierarchical solvers that are really solvers within solvers, e.g., a Krylov solver on the outside, block Jacobi as a preconditioner, a Krylov method to solve the individual blocks, and an ILU method to precondition the block solves. Several research groups have developed "inner-outer" iterations that consists of at least two nested algorithms. Some libraries (e.g. ISIS++) provide *composed preconditioners* that are defined as several different preconditioners applied in sequence. Coarse-grid correction schemes can be composed with domain-decomposition preconditioners to form two-grid solvers. These examples illustrate the fact that there is a trend towards algorithms that encompass an increasingly general set of possible combinations of solvers, preconditioners, and matrices, and thus a new and more flexible model that allows combining of methods is needed.

This viewpoint is important because *hypre* is intended for use by two mostly separate groups: applications developers will use *hypre* for solving linear systems, while solver developers will use *hypre* as a framework for developing new algorithms. It should be noted that PETSc and ISIS++, as well as other libraries, are also intended to be used in both of these modes and that they succeed in this endeavor to a certain extent. The more general object model in *hypre* is designed to provide increased flexibility for this purpose.

Though it is our intent that *hypre* supports and promotes object-oriented programming for linear solvers, the fact is that most new algorithms are developed by mathematicians who are not trained in this paradigm. Therefore, it is a requirement that we provide a relatively lightweight mechanism by which such a code can be included in *hypre*, while still allowing for the inclusion of more interoperable and flexible solvers and matrices. To achieve these goals, *hypre* has been designed to support an arbitrary level of interoperability of each component, where the level is decided by the component developer rather than dictated by *hypre*.

4 Object Model

In this section, we present the basic object model of *hypre*. Central to the design is the use of multiple inheritance of interfaces for allowing solver developers a flexible system for mixing and matching exactly the interfaces that they want to support for their objects (note that we adopt the Java model of multiple inheritance in which at most one of the inherited base classes can be concrete). The core object model is designed to be a generalization of the model used in existing libraries, so that their classes are supportable through derivation of appropriate classes in the core model of *hypre*.

Our core model consists of two layers, and is illustrated in Figure 2. The top layer consists of three base abstract classes representing the mathematics of linear solver libraries, and a `Map` class analogous to that in current libraries. This top layer's main function is to provide type safety and polymorphism. The second level consists of a myriad of separate *interfaces* (where we define interface to mean a collection of functions that can be invoked on an object). In C++ terms, these interfaces are pure virtual classes. The function of the second layer is to provide a "menu" of interfaces out of which other classes can build their more complete interfaces through multiple inheritance. This mix-and-match paradigm is vital for supporting a wide variety of concrete classes developed by varied development teams.

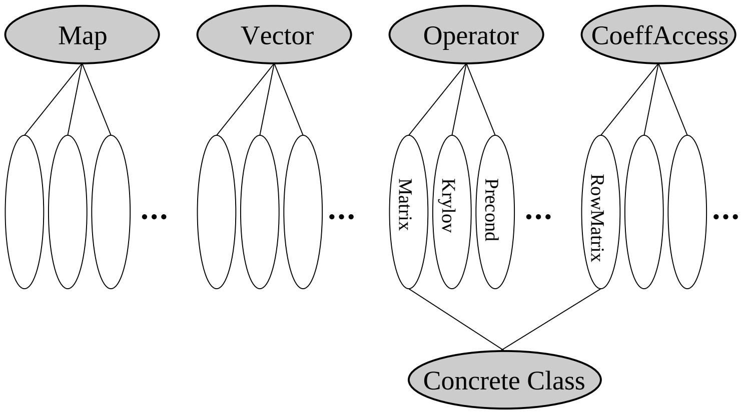

FIG. 2. *Core object model for hypre.*

The classes in the top layer are `Map`, `Vector`, `Operator`, and `CoefficientAccess`. The `Map` and `Vector` classes are essentially unchanged from existing libraries, and thus we will not discuss them any further. The other two classes are less obvious, and in particular, there is a notable absence of standard objects such as matrices and solvers from our core model. Our design builds these more standard objects from the latter two classes, and thus it is important to justify their existence as peer classes. We use the matrix to illustrate the intent of these classes.

As described in Section 2, the traditional matrix class supports both the linear-algebraic matrix-vector multiply operation, as well as being a container for the matrix coefficients. Fundamental to the *hypre* design is the notion that these are orthogonal functionalities, and that it should be possible for objects to support one of these functionalities without supporting the other. The `Operator` class in *hypre* contains (a more general form of) matrix-vector product functionality, while the `CoefficientAccess` class represents container functionality. Specific classes are built by multiply inheriting from subclasses of the `Operator` and `CoefficientAccess` classes.

The classic example of an object that performs matrix-vector multiply but does not offer access to coefficients is the so-called *matrix-free linear operator* (inlcuding matrix-free preconditioners), which by definition can perform matrix-vector multiply through some procedure (differencing formulas in non-linear methods, for example) without actually using a matrix data structure. The matrix-free matrix is already supported in current libraries through user-defined matrix operations. However, the classes that supply coefficient access functionality inherit from the `Matrix` class in current libraries, and this precludes having objects that provide coefficient access without providing matrix-vector multiply.

A coefficient-based preconditioner (i.e. one that must access coefficients and cannot be implemented with just matrix-vector multiplications, e.g., incomplete factorization), needs the coefficient container functionality from the input matrix object but does not care whether that object can perform a matrix-vector multiply operation. Under the standard model, the matrix used to form the preconditioner is also used to define an outer Krylov method, in which case the matrix has to provide matrix-vector multiply. However, there are occasions where it is appropriate to use iterative methods without Krylov wrappers, in which case the matrix-vector multiply functionality is unnecessary, as we now illustrate.

Direct methods are the most straightforward example, as they clearly do not need to be wrapped in a Krylov solver. A direct method like Gaussian elimination is not usually considered an iterative method, but in the context of *iterative refinement*, it can be considered as an iterative method that almost always converges in a single step. There is precedent for this: PETSc treats direct methods as special cases of iterative methods. Even if direct methods are not particularly practical for solving huge sparse problems, they still have many uses as components for building algorithms, e.g. direct methods are often used to solve the coarsest grid problem in multigrid.

The separation of the operator and coefficient access functionalities also provides a natural place for implementations such as functions that return coefficients based on some formula but do not explicitly store a matrix data structure, or databases that can be used to access element stiffness matrices without implementing a matrix-vector multiply operation. The Finite Element Interface has coefficient access functionality, in that it is used to input coefficients, but it does not support a matrix-vector multiply operation.

In general, it is our claim that there is a spectrum of objects that support these two functionalities in fairly arbitrary combinations, that is, some classes support (flavors of) one or the other or both. Requiring the two interfaces to go hand in hand is too restrictive.

4.1 `CoefficientAccess` Class

The `CoefficientAccess` class is an abstraction of matrix data structures. As a simple example, consider that many of the parallel solver libraries in use today have a matrix data structure that consists of a contiguous block of rows of the parallel matrix stored in each processor's memory in a compressed sparse row (CSR) format. These data structures are very similar and codes that implement a particular algorithm for these different data structures are logically the same. Nonetheless, because of the minor differences, separate code for each data structure is necessary. A solution that allows more re-use of algorithmic code is to write algorithms on top of abstracted data structures, and then wrap each compatible data structure in terms of the abstract data structure. Not all data structures are compatible with all coefficient access patterns, but using an abstraction layer allows interoperability of those data structures that are functionally equivalent in terms of the access patterns that they support.

An alternate strategy is to provide conversion functions for transforming one data structure to another. This solution is workable for a small number of data structures, but the number of conversion routines grows as the square of the number of matrix structures and quickly becomes unmanageable. It might make sense to provide conversion routines for particularly common pairs of data structures, though this still breaks encapsulation and thus makes it difficult to modify the data structures without introducing errors.

There are two basic dimensions of coefficient access, input and output (or "put" and "get"). Again, we claim that these are orthogonal interfaces in that derived classes may mix and match input and output coefficient access interfaces in arbitrary combinations through multiple inheritance.

Input coefficient access is already fairly common. The finite element interface has an input interface which is defined in the language of the finite element method. To the user of this interface, the object looks like a matrix data structure that supports input of coefficients in blocks corresponding to element stiffness matrices. This allows the user to build a matrix through the abstractions of the FEI rather than building a particular data structure. Likewise, PETSc's base matrix class has a member function called MatSetValues. Users build their matrix data structures through this member function, and PETSc transparently does the work of placing coefficients into the requested data structure.

Generally, building a matrix is done only once and is a lower-order cost than solving a linear system. This low cost means that these functions do not have to be particularly efficient, which in turn means that input patterns can have a fairly loose coupling to the underlying data structures.

Output coefficient access is more problematic. For the parallel CSR-based data structures described above, the access is random to locally stored rows. This access can be abstracted by providing an abstract class with a GetRow function. Given a row number, this function returns the given row in a sparse format. The exact semantics need to be decided, that is, whether the function should return a pointer, whether it should use copy semantics, which format the row should be in, etc. The concept remains the same, however: an algorithm written for one data structure in terms of GetRow will work with other matrices that support the same function.

Note that we leave the decision about the level of interoperability of solvers up to the developer of the solver. There is obviously some tradeoff between interoperability and performance, and the developer should be allowed to make the design decision of where to draw this line. Some developers consider performance too crucial for their preconditioners to risk losing speed to abstraction layers.

Another way to look at output coefficient functionality is as a generalized Iterator. In this context, an Iterator is a construct that allows the elements of complex data structures to be accessed in some sequence. This concept has proven very powerful in allowing *generic algorithms* for certain problem domains. Unfortunately, while some basic iterative methods can be written with only serial access to the matrix, most sophisticated preconditioners cannot. The idea behind `CoefficientAccess` is to establish other access patterns besides pure serialization in which preconditioning algorithms can be expressed. Expressing algorithms in terms of access patterns is more natural than it seems at first: there is *no* sparse matrix data structure that supports efficient random access to its coefficients, and therefore there is *no* preconditioner that accesses the coefficients randomly.

The obvious benefit of the coefficient access classes is re-use of algorithmic code, but there are other benefits. As stated in Section 2, one of our design goals is to use our core object model as a framework for developing complicated algorithms out of components developed across development teams. Assume that such an algorithm wants to use `Operator` A, B, and C (the use of the `Operator` class to build algorithms is explained more fully below). This use of multiple operators can only work if the input matrix data structure is compatible with all of the operators. In an environment with many diverse data structures, this will not be possible very often. However, with the coefficient access concept, it is only necessary that the input data structure supports access patterns that are compatible with all of the operators. It is hoped that there will be far fewer access patterns than data structures, making this much more likely.

The output coefficient access concept may also be useful in capturing data locality, much the same as the BLAS capture data locality for the LAPACK dense matrix library [1]. As illustrated by the GetRow example, the return value of an access function does not have to be floating point numbers representing coefficients. It is perfectly natural for some access functions to return other coefficient access objects. Generally, these will encapsulate some smaller portion of the matrix. At the right level of granularity, these intermediate portions can be used to build efficient kernels that can be used as building blocks for codes that are both efficient and interoperable.

The level to which these concepts are practical is still an area of research. While we have worked through several examples as part of *hypre*'s design, it is an open question as to what the ratio of algorithms-to-access patterns is across the spectrum of preconditioners, and the concept can only be useful if this ratio is large. Likewise, it is not clear that patterns can be found that are both expressive and efficient.

4.2 `Operator` Class

We discuss the `Operator` class, which is the concept that unifies many of the more commonly used classes. The unification comes from the observation that the fundamental action of most mathematical objects in linear solver libraries is operating on vectors to produce an output vector. This includes the classical notions of matrices, preconditioners, Krylov methods, other iterative methods, direct methods, etc. We believe that generalizing all of these classes into subclasses of an `Operator` base class not only encapsulates common behavior, but more importantly, it allows the flexibility for building complicated solvers from nontrivial combinations of `Operator` objects. We illustrate with examples.

The `Solver` class in PETSc is composed of a `KrylovMethod`, a `Preconditioner`, and optionally, a separate `PreconditionerMatrix`. It is not our intention to review the mathematics of Krylov solvers in depth, but we point out that mathematically, a preconditioner to a Krylov method is a matrix M such that $M^{-1}A$ is better conditioned than A. Mathematically, then, any linear operator (subject to algorithm-specific restrictions such as symmetry) can be used as a preconditioner, since a matrix and a linear operator are equivalent. Since our goal is to provide a flexible framework for solver development, we let the preconditioner be any linear operator rather than dictating that it be a "preconditioner". This obviously encompasses the standard preconditioner model: the user inputs the `Matrix` to a `LinearOperator` object (`LinearOperator` is a restricted class inheriting from `Operator`) that provides the action of the preconditioner, and then composes that `Operator` object with the `Preconditioner` to form the solver. There is no need for the `Solver` to know anything about the form or implementation of the preconditioner. It only needs to know that it is a `LinearOperator`.

The concept of a preconditioning matrix is useful for applications that have a simpler representation (a lower order discretization, for example, or the symmetric part of a mildly nonsymmetric matrix) of the problem to be solved. In PETSc, there is special code and special semantics surrounding the `PreconditionerMatrix`, namely, if one is supplied, the "preconditioner" is the result of an iterative method applied to the preconditioning matrix. In our model, a user or developer who wants to use this method first inputs the preconditioning matrix to an appropriate `LinearOperator` like an iterative method or an incomplete factorization method, and then composes this `Operator` with the `KrylovMethod` within the `Solver`. No special code or semantics are necessary, since the core model encompasses this combination of objects.

Should this user have a matrix that somehow approximates the inverse of A directly, this matrix could be composed with the `KrylovMethod` as a preconditioner directly. The key concept is that if the mathematical requirement of a preconditioner is that it should be a linear operator, than we would prefer to let *any* linear operator be allowed to fill this role. It is difficult to anticipate all possible future algorithmic developments, so we allow full generality wherever it makes sense. The `Operator` class is our mechanism for providing this generality.

5 Summary

We have presented the preliminary design of *hypre* as a framework for developing and providing advanced linear solver algorithms for extremely large systems. We have reviewed the standard object-oriented models for linear solver libraries. We presented a core object model that replaces many of the common classes such as matrix, and preconditioner with two classes, the `CoefficientAccess` class and the `Operator` class. We showed how the former class supports physics-based interfaces, allows generic programming of algorithms, and eases interoperability between modules developed independently, while the latter class supports innovative algorithm development as well as interoperability. The fundamental technique used to provide this framework is the use of multiple inheritance and a flat inheritance tree to allow developers the freedom to implement objects with exactly the interfaces that they think are appropriate for their objects.

Acknowledgements:
The design of *hypre* has been heavily influenced by two forums within the DOE and academia. The Linear Equation Solver Interface forum ([5]) is attempting to define a standard object model and standard interfaces for linear solvers, and the *hypre* design is proceeding in parallel, influencing and being influenced by this effort. Much of the object model presented in this work mirrors the object model being developed in this forum. The other group is the Common Component Architecture forum, which is attempting to standardize a component architecture for high performance scientific computing. The *hypre* model of multiply inheriting desired interfaces is influenced in large part by the fact that it maps one-for-one *query-interfaces* (or *introspection*), which is at the core of most component models.

References

[1] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen, *LAPACK Users' Guide*, SIAM, 2nd ed., 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[2] S. Balay, W. Gropp, L. C. McInnes, and B. Smith, *Petsc 2.0 user's manual*, Tech. Rep. ANL-95/11, Argonne National Laboratory, Nov. 1995.

[3] 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)*. Submitted to the SIAM Journal on Scientific Computing special issue on the Fifth Copper Mountain Conference on Iterative Methods, 1998.

[4] P. N. Brown, R. D. Falgout, and J. E. Jones, *Semicoarsening multigrid on distributed memory machines*. Submitted to the SIAM Journal on Scientific Computing special issue on the Fifth Copper Mountain Conference on Iterative Methods. Also available as Lawrence Livermore National Laboratory technical report UCRL-JC-130720, 1998.

[5] R. Clay, *Linear equation solver interface (ESI) standards multi-lab working group and interface design effort*. http://z.ca.sandia.gov/esi/, 1998.

[6] R. Clay, K. Mish, I. Otero, L. Taylor, and A. Williams, *A proposed general finite element/solver interface specification for use with scalable algebraic solver packages*. http://z.ca.sandia.gov/fei/, 1998.

[7] R. Clay and A. Williams, *ISIS++: Iterative scalable implicit solver (in C++)*. http://z.ca.sandia.gov/isis/, 1998.

[8] A. Cleary, S. Kohn, S. Smith, and B. Smolinski, *Language interoperability mechanisms for high-performance scientific applications*, in SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. Henderson, C. Anderson, and S. Lyons, eds., Philadelphia, PA, 1998, SIAM.