Divorcing Language Dependencies from a Scientific Software Library*

S. Kohn†, *G. Kumfert*†, *J. Painter*†, *and C. Ribbens*‡

1 Introduction

In scientific programming, the never-ending push to increase fidelity, flops, and physics is hitting a major barrier: scalability. In the context of this paper, we do not mean the run-time scalability of code on processors, but implementation scalability of numbers of people working on a single code. With the kinds of multi-disciplinary, multi-physics, multi-resolution applications that are here and on the horizon, it is clear that no single code group — nor any single organization — has all the required expertise or time available to independently create all of the software needed to solve today's cutting-edge computational problems.

Scientific programming libraries have alleviated some of this pressure in the past, but scaling problems are becoming increasingly apparent. The upshot of software libraries has been that different code groups in different organizations can bring their expertise to bear on particular sub-problems. Unfortunately, different groups and different organizations also bring with them implicit dependencies on different software development platforms, different programming languages, and different conceptual models of the problem decomposition — all of which must be resolved if the libraries they produce are to be useful in a final application. The good news is that scientific computing is not alone in these software scalability problems and several industry solutions have proven successful. The bad news

*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. UCRL-JC-140349

†Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory

‡Department of Computer Science, Virginia Tech & on sabbatical at the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory

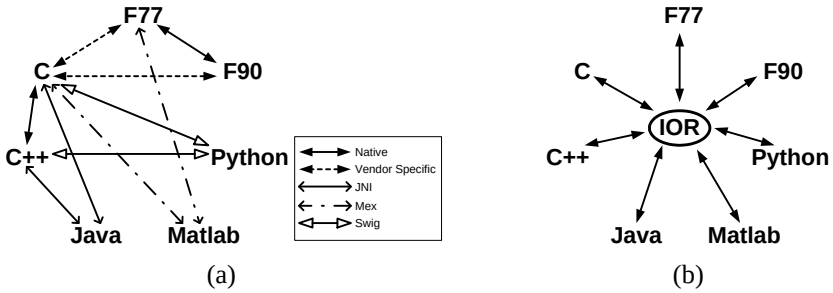

Figure 1. *Language interoperability without (a), and with (b) IDL techniques*

is that scientific computing is different enough in its nature for an "off-the-shelf" solution from industry to not quite fit the scientific computing domain.

This paper describes the ideas, process, and results of the first year in an ongoing collaboration between members of the Components Project and the Hypre Project in the Center for Applied Scientific Computing (CASC) in Lawrence Livermore National Laboratory. The Components Project has developed a tool called Babel that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation and distribution of language independent software libraries. Hypre is a parallel, scalable scientific library of linear solvers and preconditioners. By using Babel tools on Hypre in this collaboration, we found that Babel enables better software design and is an effective tool for producing language independent scientific software libraries at a negligible performance overhead.

2 SIDL

It is already very common in scientific computing to have libraries written in different languages interoperate. Consider the common case of Basic Linear Algebra Subroutines (BLAS) written in Fortran77 and invoked from C/C++. Although vendors have provided custom solutions for this problem for years, this solution has scaling problems for general libraries. First, BLAS are often tuned specifically for the target architecture. Second, glue code has to be written for C/C++ to call the Fortran subroutines. Third, the Fortran77 standard does not define the binary calling interface between C/C++ and Fortran77, so the wrappers are also vendor specific.

Many programming languages can call other languages, but only on a pairwise basis. These pairs often require significant effort (meaning wrappers or "glue code"), are not guaranteed to be portable, and may require special interconnect technology. This is illustrated in Figure 1(a). For instance, Matlab can be coaxed to run an external library written in C, but to do so means writing special Mex-Files. Getting Matlab to run a Python script natively is another matter entirely.

In large, multidisciplinary scientific applications, we are increasingly observing a need for truly language independent pieces of software. One can easily envision an application with Java or Tcl/Tk graphical displays, Python scripts driving the highest levels of

logic, Fortran linear algebra routines, solvers written in C, and the adaptive mesh refinement and time-stepping management infrastructure written in C++. Such an application would be almost impossible using the technology represented in Figure 1(a).

This problem is addressed in industry using component technologies such as CORBA and COM. In both cases, language interoperability is achieved using Interface Definition Languages (IDLs).

The Components Project has designed a Scientific Interface Definition Language (SIDL) that addresses the particular needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallelization attributes and communication directives that are required for general parallel distributed data structures, all of which are lacking from industry IDLs. SIDL also provides other features that are generally useful but not necessarily related to scientific computing, such as an object-oriented inheritance model similar to Java, name space management, and interface versioning.

SIDL is not a "lowest-common-denominator" solution between programming languages. SIDL supports full object-oriented programming, even in non object-oriented languages. It implements reference counting and dynamic type casting, even in Fortran77 which has no aliasing and limited type casting through equivalence statements.

3 Babel

The Babel tool suite takes the SIDL descriptions and a language/platform description of a software library and generates all of the glue-code on demand. It consists of a number of interrelated pieces: a SIDL parser, a code generator, a small run-time support library, and a software repository. Currently, Babel supports Fortran77, C, and C++; efforts are underway to support Java, Python, Fortran 90, and Matlab.

The Babel parser, which is available either at the command-line or through a web interface, reads SIDL interface specifications and generates an intermediate XML representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a web-based repository or a GUI development environment. XML interface descriptions are stored locally or in a shared web-based software repository called Alexandria[1]. The vision is that a scientist downloading a particular software library from the repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads XML files and generates glue code for linking from a software library to an *intermediate object representation* (IOR), and from the IOR to the application programmer's language of choice (see Figure 1(b)). This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space. The IOR used by the code generator is similar to that used by COM, CORBA's Portable Object Adaptor, or by scientific libraries such as PETSc [2, 3]. The IOR handles the virtual function dispatch for all the methods in an object's interface, maintains the object's state data, and manages some internal Babel data structures and metadata.

[1] Also developed in the Components Project, but beyond the scope of this paper.

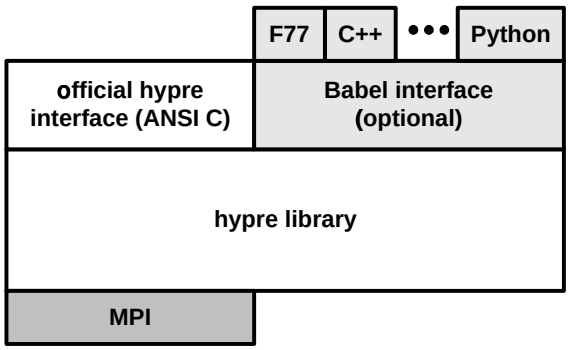

Figure 2. *The original vision of hypre and Babel*

4 Hypre-Babel Collaboration

Hypre [5] is a suite of scalable parallel linear solvers and preconditioners for the solution of large, sparse linear systems of equations on distributed-memory parallel computers. The primary algorithmic emphasis in Hypre is on robustness and scalable parallel performance. In addition, important design goals for the library include ease of use, flexibility, the rapid incorporation of new algorithms, and compatibility and interoperability with other similar libraries. These goals and emphases are driven by the needs of the most demanding scientific simulation codes, as typified by the U.S. Department of Energy's Accelerated Strategic Computing Initiative (ASCI).

The collaboration between Hypre and Babel began by identifying four primary goals and a vision of how the two projects interact. The four primary goals are:

1. The Babel team wanted to demonstrate the technology and get feedback from library developers.

2. The Hypre project had an immediate need for automatically generated Fortran bindings that would track changes in the library. Future needs for bindings to other languages (e.g., Python) was considered extremely likely. Previously, a number of different Fortran bindings were developed by various users on various platforms but fell into obsolescence with new changes to the Hypre library.

3. Hypre developers wanted to integrate software developed by other groups who had written code in C++ and Fortran.

4. The Hypre team wanted to explore new design options using object-oriented and component-based software techniques, but the team had no desire to generate and support the necessary object-oriented infrastructure by hand. This included a desire to participate in the Equation Solver Interface (ESI) working group [6], which requires working implementations to verify proposed designs.

The original vision of how Babel was to interface to Hypre is shown in Figure 2. Hypre makes a clear distinction between their "official" (meaning published and supported) pro-

```
interface Vector {
  int Clear();
  int Copy( in Vector x );
  int Clone( out Vector x );
  int Scale( in double a );
  int Dot( in Vector x, out double d );
  int Axpy( in double a, in Vector x );
};

interface Operator {
  int Apply( in Vector x, out Vector b );
};

interface LinearOperator extends Operator {
};

interface Solver extends LinearOperator {
  int GetSystemOperator( out LinearOperator op );
  int GetResidual( out Vector resid );
  int GetConvergenceInfo( in string name, out double value );
};

interface PreconditionedSolver extends Solver {
  int GetPreconditioner( out Solver precond );
};

interface RowAccess extends LinearOperator {
  int GetRow( in int row, out int size,
              out array<int,1> col_ind,
              out array<double,1> values );
};
```

Figure 3. *SIDL definition of some basic Hypre interfaces. Not all methods are shown.*

gramming interface in ANSI C and the library proper which was subject to more frequent change during the course of research. The original expectation was to supply an optional Babel interface to support other languages as they came on line.

Due to the overall size of Hypre, our initial focus was on designing and implementing a Babel interface for a representative subset of the library. We developed a SIDL file that matched the programming interfaces of this Hypre subset while adhering to SIDL's object model. We then generated the glue code between Hypre and the Babel IOR, and hand-edited the implementation details to finish the new language-independent library.

SIDL's object-model follows that of Objective-C and Java, using *classes* and *interfaces*. For C++ programmers, interfaces are similar to classes except that all methods are pure virtual, meaning they have no implementation. In this model, a class can inherit an implementation from only one class, but may inherit multiple interfaces. Figure 3 shows a SIDL definition of several key interfaces in the Hypre object hierarchy.

Table 1. *Runtime (in seconds) for a SMG multigrid solver on a* $40 \times 40 \times 40$ *structured mesh with a seven point stencil on ASCI-Blue Pacific*

	setup	solution
standard Hypre C interface	8.07	43.08
standard Hypre C interface	8.07	42.96
Babel-generated C interface	8.09	42.45
Babel-generated C interface	8.05	42.76

5 Results

We are very pleased and encouraged by the results of this collaboration between the two research groups. The performance and interoperability results were in line with expectations. Additionally there were some unexpected results that were very positive and constructive.

Negligible Runtime Overhead. Results of four runs of a standard Hypre test problem are reported in Table 1. The test problem uses Hypre's SMG multigrid solver on a Poisson equation in three dimensions, finite-differenced on a seven-point stencil, on a uniform $40 \times 40 \times 40$ structured mesh. The timings were measured using eight processors on two nodes of ASCI Blue-Pacific, a large system based on IBM RS/6000. The times reported are the sum of the times of the eight processors. Most of the manipulation through either set of interfaces is done during the setup phase. The solution phase is practically entirely within the Hypre library proper.

It is clear to see in this example that the overhead of using the Babel interface is well within the noise of the system. Moreover, it is reassuring to see that Babel can be added to existing MPI based SPMD code without ruining parallel performance.

Reduced Code Size Through Polymorphism. Some Hypre implementations proved to be unnecessary once the SIDL defined interfaces were available. For example, it was easy for the Hypre team to write generic implementations of common solvers. Given definitions of interfaces such as `Vector`, `LinearOperator`, and `RowAccess`, it is natural to implement Krylov solvers such as conjugate gradient and GMRES in terms of these interfaces. These solvers can then work with any concrete classes that implement the required interfaces. There is no longer a need to write and maintain multiple versions of common solvers, one for each matrix data type.

Originally, Hypre included eight implementations of PCG (preconditioned conjugate gradient), some of them almost identical except for how they handled the matrix-vector multiply, because of data-structure differences. To take advantage of Babel's polymorphism capabilities, we coded a PCG solver which exclusively used the Babel interface to manipulate vectors. We have Babel interfaces for two vector types so far, so this PCG solver effectively replaces two separate implemenations in the Hypre library. Likewise Hypre developers have similarly been able to reduce the number of GMRES (generalized minimal residual) solvers.

Table 2. *Runtime (in seconds) for a SMG multigrid solver on a* $10 \times 10 \times 10$ *structured mesh with a seven point stencil on Sun Sparcstation Ultra 10*

	setup	solution
standard Hypre C interface	0.14	0.26
Babel-generated F77 interface	0.14	0.27

Hypre developers involved in this collaboration feel that using Babel will allow users to get the benefits of object-oriented design without requiring object-oriented languages such as C++, which is much less portable than C.

Automatic Language Bindings. Babel was used to generate a Fortran interface to the same Hypre library (which is written in ANSI C). We ran some of the same test problems from a Fortran driver and obtained the same numerical results on a Sun workstation. This successfully demonstrated a key goal to the Hypre developers. Previous Fortran interfaces have required frequent maintenance and lacked portability.

We present in Table 2 some runtime results that again show no real difference between the performance through the Hypre and Babel interfaces. This was done on a single processor Sun workstation using a smaller version of the problem in the previous section.

Hypre developers involved in this collaboration are confident that an application code written in terms of a particular set of interfaces could use any solver or library that implements those interfaces, with virtually no change to the application code. Users could easily experiment with using different solver libraries by simply replacing one library's implementation of the required interfaces with another library's implementation.

Explore New Design Options. In addition to the basic Hypre objects defined by the interfaces shown in Figure 3, a second set of interfaces, called *builder* interfaces, were developed and plays a role of increasing importance. A builder interface is a set of methods for constructing one or more basic objects and follows the Builder design pattern [7]. These builders have no concrete analog in the Hypre library and are exclusively available through the Babel interface. A major benefit of the builders is that users are prevented from accessing partially constructed datastructures.

The most interesting examples are the `MatrixBuilder` and `SolverBuilder` interfaces. A `MatrixBuilder` can be thought of as a particular user interface through which users define problems. Each `MatrixBuilder` is accompanied by a `VectorBuilder` for building compatible vectors. A `SolverBuilder` is used to set the components and parameters that define a `Solver`. Partial SIDL definitions of builder interfaces are given in Figure 4.

SIDL as a Design Language. To generate interface code Babel requires a SIDL file defining the interfaces. This forced the Hypre developers to consider the user interface as a separate issue from the implementation, and provided an automated mechanism to keep

```
interface MatrixBuilder {
  int SetMap( in Map map );
  int Setup();
  int GetConstructedObject(out LinearOperator obj);
};

interface StructuredGridMatrixBuilder extends MatrixBuilder {
  int Start( in StructGrid grid, in StructStencil stencil,
             in int symmetric, in array<int,1> num_ghost );
  int SetValue( in array<int,1> where, in double value );
  int SetBoxValues( in Box box, in array<int,1> stencil_indices,
                    in array<double,1> values );
};

interface IJMatrixBuilder extends MatrixBuilder {
  int Start( in MPI_Com com, in int m_global, in int n_global );
  int SetLocalSize( in int m_local, in int n_local );
  int SetRowSizes( in array<int,1> sizes );
  int InsertRow( in int n, in int row,
                 in array<int,1> cols,
                 in array<double,1> values );
};

interface SolverBuilder {
  int Start( in MPI_Com comm );
  int SetParameterDouble( in string name, in double value );
  int SetParameterInt( in string name, in int value );
  int SetParameterString( in string name, in string value );
  int Setup( in LinearOperator A, in Vector b, in Vector x );
  int GetConstructedObject( out Solver obj );
};

interface PreconditionedSolverBuilder extends SolverBuilder {
  int SetPreconditioner( in Solver precond );
};
```

Figure 4. *Examples of Builder interfaces in Hypre. Not all methods are shown.*

the code consistent with the user interface design. There was no opportunity to clutter the interface with quick, one-time hacks. The result was a more stable and predictable user interface.

The simplicity of the SIDL file made it the most convenient language for Hypre developers to use to discuss user interface design. We could limit our discussion to pure interface issues while remaining confident that whatever we came up with would be practical. SIDL was an easy language to pick up and (unlike UML) was easy to write up in email and send to collaborators.

Improvements to SIDL. The Hypre interface project also provided useful feedback to the Babel project. Our experience with practical use of Babel led to several features and tools which now make Babel easy to use. One common mistake that was made was confusion over how to make a concrete class, i.e. one for which all the inherited virtual functions have an implementation to handle the calls. It was easy for classes to inherit a lot of interfaces, and the writer of the SIDL file to forget to add a single method signature that was supposed to be implemented only to find that Babel created an abstract, not concrete class.

To correct the situation, SIDL was modified in two ways. First, the keyword "abstract" was added to classes that may have unimplemented methods. If a method is left unimplemented and the class is not declared abstract, there is an error. Additionally, the keyword "implements-all" was added. If a class inherits an interface through a regular implements directive, it overrides only those methods explicitly mentioned in the class definition. If the interface is inherited through an "implements-all" directive, all the methods of the interface are expected to be overridden by the class and writing the method call in the class definition becomes redundant.

Improvements to Babel Tools. Based on observing the use of the Babel tools and interviews with the Hypre developers involved, two features were added to improve usability: automatic makefile generation, and preservation of user edits to generated code.

Even on a small SIDL file, the Babel tools can generate a surprising number of header and source files, often in various languages. The Babel code generators were modified so that a makefile fragment is generated in each directory where code is generated. These makefile fragments define macros that list the relevant filenames and are suitable for inclusion into larger makefiles.

In addition to the glue code that the Babel tools generate, they also generate so called *Impi* files with empty function bodies. Developers of new libraries may want to build their implementation directly in these files, but developers of legacy libraries use this as a place to simply dereference pointers and call their own code. We added functionality to the Babel tools so that if the SIDL file was changed incrementally, these edits to the Impl files are preserved. This improvement has saved Hypre developers a significant amount of cut-and-paste.

Revised Hypre Architecture. At the end of one year of a Hypre-Babel collaboration, a new vision is emerging about the Hypre architecture as shown in Figure 5. In this new design, the Hypre library will depend on the Babel runtime library to provide object-oriented support throughout the entire hypre library, not just the Babel interfaces. Additionally, all the published interfaces, including the ANSI C interface will be provided with Babel.

The design in Figure 5 represents a major shift in the Hypre library and has yet to be finally decided. The Babel developers are particularly pleased that though this collaboration, Hypre developers have developed so much enthusiasm for Babel tools.

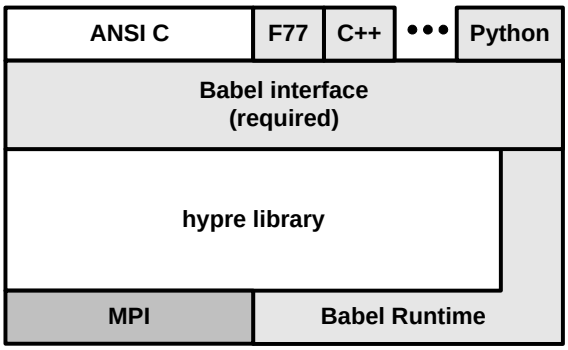

Figure 5. *The revised vision of hypre and Babel*

6 Conclusions and Future Work

Babel did the language interfacing job it had been designed for, at no cost to the Hypre user and great advantage to Hypre developers. The Hypre-Babel collaboration led to improved codes, and methodologies for both groups.

In the long term, Hypre plans to increase its reliance on the Babel tools and may eventually be distributed with pregenerated interfaces for several languages and platforms, and a BabelLite runtime library. In this configuration, it is entirely possible that the users of the library don't even have to be aware that they are using Babel as well. Members of the Hypre team also plan to continue participation in the Equation Solver Interface (ESI) [6] working group, developing standards for linear solver interfaces.

Babel continues to mature. Work is constantly being done to support additional languages and platforms. Much of the current research within the Components Project at LLNL is focused on handling parallel remote method invocations and data redistribution in a language independent manner. The Components Project also maintains close ties to a larger, grass-roots initiative called the Common Component Architecture (CCA) Forum [1, 4]. The goal of the CCA is to bring modern component technology to scientific computing. The Babel tools are targeted to provide the language independence to the CCA.

As scientific applications become more interdisciplinary the need for interoperability between different libraries and among pieces from different libraries becomes even more important. An important question is how a Babel/SIDL skin can easily be wrapped around existing libraries. The Hypre developers feel that if the existing library was reasonably well organized (even if not using an explicit OO language) the effort is reasonable, the runtime costs negligible, and the potential payoff in increased interoperability huge.

Bibliography

[1] R. ARMSTRONG, D. GANNON, A. GEIST, K. KEAHEY, S. KOHN, L. MCINNES, S. PARKER, AND B. SMOLINSKI, *Toward a common component architecture for high-performance scientific computing*, in Eighth IEEE Int'l Sym. on High-Performance Distributed Computing, Redondo Beach, CA, August 1999. also Lawrence Livermore National Laboratory technical report UCRL-JC-134475, June 1999.

[2] S. BALAY, W. D. GROPP, L. C. MCINNES, AND B. F. SMITH, *Petsc: The portable extensible toolkit for scientific computing*. `http://www.mcs.anl.gov/petsc`.

[3] ——, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, Birkhauser Press, 1997, pp. 163–202.

[4] *Common Component Architecture (CCA) Forum* . `http://z.ca.sandia.gov/~cca-forum`.

[5] E. CHOW, A. CLEARY, AND R.FALGOUT, *Design of the hypre preconditioner library*, in Object Oriented Methods for Interoperable Scientific and Engineering Computing, M. E. Henderson, C. R. Anderson, and S. L. Lyons, eds., SIAM, Philadelphia, PA, 1999, pp. 106–116.

[6] *Equation Solver Interface (ESI) Working Group* . `http://z.ca.sandia.gov/esi`.

[7] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, Addison Wesley Longman, 1995.