PETSC AND OVERTURE: LESSONS LEARNED DEVELOPING AN INTERFACE BETWEEN COMPONENTS *

Kristopher R. Buschelman†
buschelm@mcs.anl.gov

William D. Gropp
gropp@mcs.anl.gov

Lois C. McInnes
curfman@mcs.anl.gov

Barry F. Smith
bsmith@mcs.anl.gov

Abstract We consider two software packages that interact with each other as components: Overture and PETSc. An interface between these two packages could be of tremendous value to application developers in that Overture provides a simple mechanism for generating the large, sparse systems of linear equations that correspond to discretizations of a PDE, and PETSc provides a powerful collection of methods for solving these systems. Two types of interfaces are discussed: the internal interface between components, and the external interface for the application developer. We compare three basic approaches to developing the internal interface between Overture and PETSc, the final one of which is a peer-to-peer model.

Keywords: Components, Interface, PETSc, Overture, Peer-to-Peer Interaction

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.
† All four authors are affiliated with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

1. INTRODUCTION

A complete application to numerically solve a partial differential equation (PDE) and analyze the results typically involves: a mechanism for creating a grid; a scheme for calculating spatial derivative approximations; a method for time advancement, which may require use of linear algebra routines including scalable linear and nonlinear equation solvers; another mechanism for visualization and analysis of the data; and, since the application may be performed on a parallel computer, routines for communicating data between processors. With our expectations of software rising as the capabilities of computers increase, writing a good implementation of any one of these tasks requires significant expertise. Indeed, the expectation that one person, or one group, could write an entire package for such a general-purpose tool is unreasonable. To create such an application, we must take advantage of the expertise of several persons or groups, each focusing on one component of the full application. We can then consider a framework in which these components can be linked together. (In this paper, we shall use the terms "component" and "framework" in a general sense as opposed to the specific definitions as set forth by the Common Component Architecture Forum [1] and other organizations.)

Before creating such a framework, however, we must learn what makes a good component. We can gain this insight by looking at successes and failures of various projects that have attempted interaction between software packages. In this paper, we shall consider two software packages that interact in this manner, Overture [5, 6] and the Portable Extensible Toolkit for Scientific computation, PETSc [2, 3, 4].

Overture is a collection of C++ classes that provide tools for solving PDEs. It contains a tool for generating composite grids (i.e., lists of structured grids that overlap) and a wide variety of operators of varying accuracy for computing derivatives via finite difference, finite volume, and spectral methods on these grids. Overture is also extensible in that application developers can create their own sets of operators. Furthermore, several equation solver libraries can be used within Overture, and new equation solvers can be added.

PETSc is a scalable library for the solution of PDEs and related problems. With PETSc, one can create complete applications, as one might within Overture, but the emphasis has been on generating a collection of solvers for linear and nonlinear systems of equations as well as lower-level infrastructure for managing the details of parallel programming.

An interface between these two packages could be of tremendous value to application developers in that Overture provides a simple mechanism for generating large, sparse systems of equations, and PETSc provides a powerful collection of methods for solving these systems. From the Overture developer's perspective, the obvious mechanism for this interface is via a PETSc equation

solver class within Overture. The development of such a class is still ongoing, but much can be learned about how to write useful components by observing this work in progress.

Two types of interfaces shall be discussed: the internal interface between components, and the external interface which the application developer will use. Three basic approaches toward developing the internal interface between Overture and PETSc have been explored. The first approach is to have Overture convert its native data structures into those that the Overture developers expect to be appropriate for linear algebra purposes, and to require any linear algebra solver to support these formats. The second is to force the linear algebra solver (PETSc) to use the native Overture data structures as vectors and matrices. The third approach is to have both Overture and the linear algebra component work together to convert Overture's native data structure into whichever data structure the linear algebra component recommends. For the external interface, one must balance simplicity with flexibility, to allow the user to develop high-performance applications without needing to learn new interfaces for well-known tasks.

2. THE INTERNAL INTERFACE

When working with multiple software components, the principal barrier toward interaction is related to the data structures involved. The best data structure for one particular task, and component, is not the best data structure for another. Clearly, all components cannot be expected to use the same data structure. An interface must be generated to determine the interaction between the components at the level of the data structures. This interface is one that, if properly implemented, is used by component developers and is not essential for the application developer to use directly. In this section, three approaches toward this internal interface are discussed.

2.1. OVERTURE CONVERTS DATA STRUCTURES TO "STANDARD" VARIETIES

The first approach is for Overture to require linear algebra solvers to support specific matrix and vector data structures that are common among linear algebra toolkits. In particular, contiguous one-dimensional arrays are used to store vectors. For storing matrices, the compressed and uncompressed sparse row formats as well as their column-based counterparts are currently supported. Overture provides a mechanism for the conversion into and out of these specific data structures. These data structures are created and destroyed by Overture, but since they are supported by other components, those components are free to manipulate them as they see fit.

From PETSc's perspective, this approach is acceptable for vectors but has only limited benefit for matrices. In particular, PETSc implements a (sequential) sparse matrix, which is entirely owned by one process (or duplicated across all processes), using the compressed sparse row format. But, if this matrix is to be distributed across several processors, the data structure must be changed rather dramatically to achieve high performance. Similarly, if a certain block structure is present in the matrix, PETSc prefers a blocked variant of the compressed sparse row format. The use of this blocked variant allows for many fewer cache misses and register loads, resulting in vastly superior performance. Neither of these two matrix formats is directly supported by Overture, so to accommodate these data structures, a second conversion would be required.

From the Overture developer's perspective, there is only one procedure for data conversion for vectors. This is quite easy to develop and maintain. Matrices are another story, however. Overture currently supports four data structures. PETSc supports nine matrix formats currently, with four more under development, but only one of these formats is supported by Overture. In fact, a draft of the Basic Linear Algebra Subprogram (BLAS) standard proposed 13 sparse data structures to be supported, including those supported by Overture, and only four which are supported by PETSc [7]. One could expect Overture to provide routines for conversion into and out of each of these additional varieties, but where would the list end? Furthermore, as new software packages are developed, new data structures with increasing complexity are bound to arise. To deal with this problem, PETSc allows additional formats that are user defined; the user can provide new data structures and overload the basic PETSc functions with appropriate implementations. How would Overture deal with these additional types?

2.2. PETSC SUPPORTS NATIVE OVERTURE DATA STRUCTURES

The presence of a user-defined type within PETSc suggests a different approach to the interface. The PETSc matrix and vector operations could simply be overloaded to use the data structures for vectors and matrices that are defined by Overture. This approach has been used successfully to interface PETSc with the Structured Adaptive Mesh Refinement Applications Infrastructure (SAMRAI) [9, 10], and other packages. It has the advantage that there is no performance overhead associated with copying elements between data structures in terms of memory or CPU usage.

Two sources of difficulty are associated with this approach, however. First, the linear equation solvers within PETSc are primarily based on Krylov subspace methods. In these methods, the most fundamental operation between

matrices and vectors is the matrix-vector product. Unfortunately, this operation is far less efficient when implemented using data structures that have not been optimized for linear algebraic operations as has been done with the PETSc data structures. The Overture data structures, although well optimized for PDE discretization, are not optimized for linear algebraic operations. As a result, the cost of copying the data into a PETSc data structure once per solution of a linear system of equations is far less than the overhead associated with using the Overture data structures for the matrix-vector product. This alone would be sufficient for not choosing this approach, but there is another challenge facing this approach.

Overture recognizes two basic linear algebra classes, vectors and matrices. In PETSc however, there is a third basic class, preconditioners. In order to achieve high performance when solving the large, sparse linear systems of equations generated by the discretization of PDEs with Krylov methods, preconditioners are essential. Since Overture has delegated the creation of preconditioners to the linear algebra component, if this approach is to be used, the mechanisms for interoperation of PETSc preconditioners with vectors and matrices defined by non-PETSc data structures must be understood.

To achieve high performance when solving systems of linear equations, many of the PETSc preconditioners place certain functionality requirements on matrix and vector data types. These attributes include the ability to extract the diagonal (block) elements of the matrix and to solve systems of equations with the locally owned portion of the matrix. Furthermore, it is assumed that one can obtain a pointer to a (locally owned) contiguous data array for each vector type. Since the Overture data structures are not stored locally as a contiguous array, the vector data must be copied into this format for use with many of the PETSc preconditioners. As a result, if PETSc users provide their own storage formats for matrices and vectors, they often provide their own preconditioners. As this is not an option for the Overture developer, we return to the concept of conversion between data structures.

2.3. PEER TO PEER INTERACTION

No developer can learn every possible data structure that could be used for matrix and vector storage and then provide all possible conversion routines. As a result, a two-step conversion process was used in early releases of Overture. However, since the Overture developer knows its data structure, and the linear algebra component has intimate familiarity with its own formats, the two packages should be able to cooperate and together carry out the conversion process in a more direct manner, despite the complexity of the exact process involved. To do so, we employ nontraditional approach.

The tradition in scientific computing software has been to gather groups of developers together and have them discuss data structures and interfaces. The intended result is a standard that other software packages can use; eventually, high-performance implementations based upon these standards could be developed and used by all. This is the model that was used when generating the BLAS and LAPACK standards. In the realm of dense linear algebra, the resulting implementations have enjoyed a great deal of success. But this is not the case for large, sparse systems generated from the discretization of PDEs as the sparse matrix standard has yet to be finalized. Furthermore, the current draft of the BLAS Standard for sparse matrices does not specify the underlying implementation of the sparse matrix, leaving that decision to the author of the particular BLAS implementation [8].

Instead of relying on the existence of a standard data structure, a generic converter can be created. To do so, one must remove the responsibility for generating the matrix and vector data structure from Overture, and share it with the linear algebra component. However, this linear algebra component cannot be expected to know how to properly traverse a data structure that it does not know. A compromise must be found; each component can be expected to perform only the tasks it knows how to perform. This means that during the conversion between the two data structures, Overture would provide two services: size information and traversal path; and the linear algebra component would provide two additional services: new data structure allocation and element definition.

In particular, Overture would provide generic linear algebra information such as the global and local matrix dimensions and a bound on the number of nonzero elements in each row of the matrix. This information would then get passed to the equation solver component via a routine called AllocateMatrix. PETSc and other components would then provide a specific implementation of AllocateMatrix that uses this information to generate an empty matrix data structure. Overture would also provide a mechanism for walking through its data structures while making calls to another routine, called SetMatrixElement, and each component would implement this routine in the most suitable manner. The implementation is quite simple in C++. Each equation solver subclass is derived from a base class that has the two required functions declared as virtual.

This approach raises several issues related to the efficiency of the conversion process. The SetMatrixElement method would need to be able to insert an element into any location in the matrix. This might require a significant amount of alteration to the data structure or might involve communication between processors. Clearly, some sort of aggregation process should be allowed, in which case a SetMatrixRow could help. But, this would not be of use if the matrix were stored in a column-based format. A generic SetMatrixElements would

clearly be prefered. To allow for greater aggregation, the component developer might also employ a stash based approach toward the setting of element values. In this approach, the elements initially get set into a private stash (perhaps a linked list), which would then get manipulated and converted into the final format. As a result, another virtual operation, AssemblyFinalize, should be added to the base equation solver class to facilitate this second step.

This has introduced a second non-traditional characteristic of this conversion process. In many linear algebra libraries, there is no concept of an invalid matrix. That is to say, once a matrix is created, it can be used. The stash based approach to setting elements in a matrix requires that this not be the case. By setting a single element, the matrix data structure becomes invalid, since information about the matrix itself is located in the temporary stash. The matrix is then made valid by performing an AssemblyFinalize step after additions to the stash are complete. It would be possible to hide the explicit call to AssemblyFinalize from the end user by placing this call within each operation that requires a valid matrix data structure, but there is a penalty for such an approach.

In PETSc, when adding elements to a matrix, a stash is used [2]. For parallel matrix formats this provides one particularly important benefit, elements can be added in one process that are to be stored as part of the local matrix in a different process. To allow the application developer to overlap the required communication with computation, PETSc divides the process into two stages: MatAssemblyBegin, which initiates the communication, and MatAssemblyEnd which terminates communication and performs the final data structure assembly. If there were no concept of an invalid matrix in PETSc, or if this concept were hidden from the application developer, the idle time that occurs during these communication stages could not be used for useful computation.

This mechanism is quite useful for achieving high performance and could be incorporated not only by dividing the Assembly Finalize into two pieces, but by dividing the entire conversion process into two parts. MatrixConversionBegin would contain allocation of memory for the matrix data structure in the equation solver, and traversal of the Overture data structure while setting each element (or row/column) of the matrix, making the matrix invalid. MatrixConversionEnd would assemble the valid matrix and perhaps provide some useful debugging options to verify that the valid matrix was assembled properly.

3. THE EXTERNAL INTERFACE

Finally, we must determine how to appropriately encapsulate the interface information to generate a well-defined minimum interface layer as well as additional convenience layers. The question to be addressed is one of exposure:

How much is enough? Should an application developer be expected to know the entire application-programming interface (API) for each individual component involved? That is to say, should an Overture user be expected to know all the details of how to use PETSc? Clearly, the answer is no; but the same question should be asked about a PETSc user. Furthermore, knowledgeable PETSc users should not feel as though they are restricted when using Overture to generate their systems of equations, nor should they be required to learn a completely new API for the PETSc aspects of their application. The minimum layer of interface must be found and adequately described, while other convenience layers may be provided so that multiple APIs do not need to be learned.

In this area, a "least common denominator" interface makes some sense. An Overture user with no experience with PETSc, or other equation solvers, would have access to a very basic set of commands: BuildMatrix, BuildRHSAndSolutionVector, and Solve. A common interface for selecting various linear equation solvers such as GMRES and CG and preconditioners such as ILU and Jacobi would provided, as well as mechanisms for selecting the different data structure conversion options. But, this alone would be a gross limitation for expert users of PETSc who wish to use their own advanced equation solvers and preconditioners built within PETSc.

Similarly, the Overture user is primarily interested in finding a numerical solution to a PDE subject to a certain discretization. Many methods for solving time dependent PDEs do not require solving a system of linear equations but do require solution of a nonlinear system of equations. Many linear algebra components do not provide a mechanism for this, but PETSc does. It would be a great disservice to application developers if this additional capability within PETSc could not be exploited.

To achieve this end, the PETSc matrix and vector data structures are public members of the class PETScEquationSolver. In this manner, once the BuildMatrix and BuildRHSAndSolutionVector routines have been called, these data structures can be extracted, and the full PETSc API can be used without limitation, if desired. Furthermore, an Overture-enhanced version of the PETSc API for solving nonlinear systems has also been generated to simplify this process, which may be incorporated into a PETScNonlinearEquationSolver class in future versions of the PETSc-Overture interface.

4. LESSONS LEARNED

From this work, we have identified various guidelines that a component developer should follow. For a component to be used, it must interact with other components. Common approaches to enable this interaction have been to mandate adherence to a standard data structure or to provide of a single data struc-

ture for all to use. For the solution of large, sparse systems of linear equations, these approaches do not allow the user sufficient flexibility to obtain high performance. This limitation in turn discourages the use of that component. One must accept the fact that data structures as well as the algorithms that use those data structures determine performance. To allow for the use of a wide variety of data structures, a data conversion process is required. For conversion processes to be successful, both sides in the conversion process need to cooperate. An appropriate division of tasks and assignment of responsibility is essential. In general, this division of tasks must allow each component to provide the services and information that it can, and must not require knowledge beyond the scope of the component. By specifying methods to be used and not implementations, we allow component writers the freedom to implement the highest performance data structures for their specific task, without placing limitations on the interaction with other components.

When making a component, two aspects to the interface require attention: the external interface detailing what it can accomplish, and the internal interface to other components. It is easy to neglect the second while focusing on the first. But, doing so can make the component much less powerful.

Ultimately the success of the component lies in the answer to one question: Do people use it with other components?

Acknowledgments

We express our thanks for the invaluable assistance of Satish Balay who helped us overcome many of the technical problems associated with software interaction and software development.

References

[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 Parallel Computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 Users Manual. Technical Report ANL-95/11 - Revision 2.0.29, Argonne National Laboratory, 2000.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc Home Page. http://www.mcs.anl.gov/petsc, 2000.

[5] D. Brown, W. Henshaw, and D. Quinlan. Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 215–224. SIAM, 1999.

[6] D. Brown, W. Henshaw, and D. Quinlan. Overture: Object-Oriented Tools for Solving CFD and Combustion Problems in Complex Moving Geometry. http://www.llnl.gov/CASC/Overture, 1999.

[7] Basic Linear Algebra Subprograms Technical Forum. DRAFT-Document for the Basic Linear Algebra Subprograms Standard, August 11, 1997: Sparse BLAS. http://www.netlib.org/utk/papers/sparse.ps, 1997.

[8] Basic Linear Algebra Subprograms Technical Forum. DRAFT-Document for the Basic Linear Algebra Subprograms Standard, May 31, 2000:

Sparse BLAS. http://www.netlib.org/cgi-bin/checkout/blast/blast.pl, 2000.

[9] R. Hornung and S. Kohn. The Use of Object-Oriented Design Patterns in the SAMRAI Structured AMR Framework. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 235–244. SIAM, 1999.

[10] S. Kohn, X. Garaizar, R. Hornung, and S. Smith. SAMRAI Home Page. http://www.llnl.gov/CASC/SAMRAI, 1999.