Exploiting Existing Software in Libraries: Successes, Failures, and Reasons Why

William Gropp*

Abstract

The PETSc (Portable Extensible Tools for Scientific computing) library arose from research into domain decomposition methods which require combining many different solutions in a single application. The initial efforts tried to use existing numerical software but had limited success. The problems include everything from faulty assumptions about the computing environment (e.g., how many processes there are) to implicit (yet deadly) global state. More recently, PETSc and PVODE have found a way to cooperate, and new techniques that exploit dynamically linked libraries offer a more general approach to interoperable components. The paper highlights some of the issues in building sharable component software and discussing mistakes still often made in designing, maintaining, documenting, and testing components.

1 Introduction

The goal of writing software components[1] (once called libraries) is to allow users to build reliable and useful applications from well-crafted parts, just as virtually any physical artifact is built from a collection of smaller parts. Yet applications still tend to look on components with skepticism; many applications in scientific computing are still built from the ground up without using anything other than special function and I/O libraries. This paper provides some insight on why this is so, based on experiences building the PETSc [4] library for solving partial differential equations. PETSc provides a good stage for investigating this issue because it is simultaneously a provider and a consumer of software components. PETSc has also been successfully used in applications, and the steps necessary to accomplish such applications are informative. Finally, PETSc was chosen as the example because it is the system with which we are most familiar; but there are many other successful numerical libraries whose experiences are probably similar to those of PETSc.

At the very highest level, there is, fortunately, a single guiding principle. A software component is built to be used (or consumed) by someone else. If the component does not match the needs of that consumer, it won't be used. Success of a component is measured by what the intended consumers think of it. This seems obvious, but often software components measure their success against a requirements list (for example, functionality in an area of computational linear algebra) instead.

The first step is thus deciding who the customer is. All too often, the answer is "everyone." This is the first step toward failure. Any software component will have to make tradeoffs; understanding the consequences of these is important. For example, an

*Mathematics and Computer Science Division, Argonne National Laboratory

[1]The term "component" currently has an ensemble of precise yet different definitions, along with the colloquial definition. In this paper, a component is simply a piece of software with a documented interface.

implementation of a particular algorithm and data structure for solving a system of linear equations is not what many applications trying to solve a linear system want; the details of the algorithm and method, while necessary to the expert, are distracting and confusing for the application programmer. Worse, they may constrain the application writer to a particular method for solving the problem by making it difficult to choose a different method should this one fail to meet the other application requirements for correctness or performance. As another example, consider the level 2 and 3 BLAS [8, 7]. These strive to provide a convenient user interface for a collection of related operations (e.g., operations with transposes of some or all of the arguments). Unfortunately, this adds overhead that makes these routines inefficient for smaller problem sizes, such as arise in blocked sparse matrices. This is not a design flaw in the BLAS as long as the intended customer is one with large matrices.

Once the customer has been identified, it is necessary to learn what the customer actually needs. The request for a "routine to invert a matrix" is a well-known example of the customer asking for the wrong component, yet all too often lists of customer requirements just like this lead to poor software components. The solution to this problem is well known: maintain a dialog with the users. If there is one factor responsible for the success of PETSc, it is this.

The rest of this paper discusses the history of PETSc, lessons learned, and challenges that software components will be facing. This paper does not discuss object design issues; these are already recognized as an issue and receive attention elsewhere. A note of warning: this paper is a subjective and personal view of component software; its goal is to help improve the use of software components in scientific computing by drawing attention to critical but underappreciated needs.

2 PETSc History

The PETSc library for solving partial differential equations has been successfully used in a variety of applications[1, 6, 19, 15]. PETSc did not, however, start out as a project to develop a numerical library. Rather, it grew out of a need for modular numerical software in support of research into domain decomposition methods for massively parallel computers. Because we intended to use these methods to demonstrate the advantages of parallel computing, as well as the scalability of the methods, it was important that the software deliver high performance.[2]

Domain decomposition methods subdivide a domain into smaller pieces and then solve smaller problems defined on these subdomains. The process may be recursive, using domain decomposition to solve the subproblems. In addition, on a massively parallel computer, each subdomain may be spread across several processors; alternatively, there may be several subdomains, all of different sizes and shapes, on each processor. In our research, we were particularly interested in domain decomposition methods as preconditioners for Krylov-based iterative methods. Thus, we needed software that provided Krylov methods for parallel computers, along with preconditioning methods both to compare against and to use as subdomain solvers. While this seemed like a simple requirement, we could not find any appropriate software in 1990, when this project started. In addition, most libraries provided only part of the solution. For example, most libraries for sparse matrix operations did not (and many still do not) provide routines for assembly of the matrix; assembly is

[2] The easiest way to make software scalable is to make it sequentially inefficient.

awkward in the sequential case and quite difficult to accomplish with any performance in the parallel case.

There were many reasons that we were unable to use many of the existing numerical libraries. For example, many implementations of Krylov methods used reverse communication and assumed that vectors were stored in contiguous memory (otherwise all vector operations would also need reverse communication). This would not work on massively parallel, distributed memory computers. Most numerical libraries of the time did not support parallelism; those that did rarely allowed use of less than all processes. In addition, even for the sequential parts, few libraries were re-entrant (prohibiting use in nested domain decomposition). Finally, some libraries had separate setup and usage phases, but did not allow more than one problem at a time, making it impossible to use with several subdomains per process. More details on the problems in using existing numerical libraries in PETSc may be found in [14].

Nevertheless, PETSc (version 1) was able to make use of several numerical libraries. All of these provided basic services on uniprocessors. Even here, there were problems.

BLAS The BLAS provide a low-level, basic functionality with simple data structures. They are all stateless (no initialization or setup parts) and are almost one routine per function (as opposed to having a number of internal functions called to perform a function). The level 2 and 3 BLAS tend to have a relatively high overhead when applied to relatively small problems. The use of Fortran `character` data made it awkward to use from C, both because of the different calling conventions for character strings in C and Fortran and because of the need for support from the Fortran runtime library for character comparisons.

Most vendors provide an optimized version of the BLAS[3], but finding this version of the library is often difficult. One of the problems with any component solution is resource discovery; even finding the BLAS is beyond current practice.

Finally, the BLAS, large as they are, do not provide all of the routines that we needed. There are two operations very common in iterative methods but not in direct methods for solving linear equations:

$$y \leftarrow x + \alpha y$$
$$w \leftarrow x + \alpha y$$

Another critical operation for some methods is a multiple dot product; this can be simulated as a matrix-vector product in some cases, but PETSc often needed this operation for a collection of vectors not necessarily organized as columns of a matrix.

IBM ESSL The ESSL library [17] is a large collection of routines for IBM systems. Because ESSL is a proprietary and extra-cost item, PETSc could not depend on users having it available. This situation led to a strategy for conditionally loading components in PETSc version 1 that was precursor of the dynamic, runtime mechanism in PETSc version 2.

ESSL also presented problems. Most seriously, the ESSL implementations of the LAPACK routines `dgeev` and `zgeev` were incompatible with the LAPACK definitions

[3] If compilers could actually optimize code for memory hierarchies, such optimized libraries would not be necessary. This situation tells something about how hard it is to generate efficient code for memory hierarchies.

in [2]. While PETSc could work around this limitation, any application that linked with ESSL and used either of these routines would fail.

Another problem was with the routines that supported the solution of sparse linear systems. We were able to use some of these routines but not others. The ones that we could not use had unique data structures for a particular operation. While such an approach was useful for an application that chose those data structures, putting them into PETSc required either converting to and from their specific data structures every time they were used (thus eliminating their performance advantage) or providing all of the support operations that PETSc provides for defining and manipulating matrices. In other words, the component (solving a sparse linear system with a special data structure) was incomplete; because it did not have routines to aid in assembly and basic operations like matrix-vector product, it placed an unnecessary burden on the application user.

FFT Using FFT libraries such as FFTPAK or a vendor-specific implementation illustrates another problem. FFT libraries are often optimized for applying the FFT of a fixed length to successive vectors. This may be accomplished with a setup routine and a separate routine to apply the FFT. In some cases, in an attempt to make the FFT routines easier to use, the setup routine initializes some internal storage, rather than a user-provided work array. Unfortunately, this internal state makes it impossible to use such an implementation (efficiently) in the domain decomposition case where there are domains of different sizes.

SPARSPAK PETSc version 1 was able to use the Netlib version of the SPARSPAK [10] routines for generating matrix orderings. These routines use a common sparse matrix format, return error codes when problems are detected, and are stateless.

None of these routines were re-entrant, but it did not matter since they were stateless and could be used as atomic operations (even in a threaded environment, they would only cause performance problems, not deadlock).

Version 2 of PETSc, released in 1995, was a complete rewrite, based on our growing understanding of how the PETSc components should be designed. One part of this was to cleanly separate the interface to the user from the implementation. This separation was particularly important in PETSc because a single interface (e.g., `SLESSolve` for solving linear systems of equations) may invoke one of many different implementations (e.g., parallel or sequential, different Krylov methods and preconditioners and different matrix storage formats). More effort was put into user support features, such as tracing, debugging, and performance measurement. PETSc version 2 uses LAPACK and BLAS, along with ordering routines from MINPACK and SPARSPAK. It also provides access to optional components such as BlockSolve95, IBM's ESSL, PVODE, SPAI [16, 5], and ParMETIS [20]. Each of these latter components is used to provide a specific capability (e.g., ESSL is used for fast factorization of sparse matrices). More details can be found on the PETSc Web page [3].

3 What PETSc Taught Us about Components

PETSc provided an excellent opportunity to learn what did and did not work in components for numerical software. PETSc is a very demanding customer of such components, requiring great interoperability. The components that we have been able to use have the following features.

1. The components must be well documented. If the component is distributed as source code, it must be portable. If it is executable-only, there must be a reliable way to determine whether the component is available.

2. Components that implement a single algorithm must have well defined data structures and provide error returns as an option (as opposed to printing messages).

3. Components that implements a single task (such as solving a symmetric linear system) must have the same properties as components that implement a single algorithm. In PETSc version 2, the BlockSolve95 [18] and PVODE [21] libraries are examples of such components.

PETSc has tried to use other components and failed for several reasons:

1. Nonportability was due to everything from using non-standard language "extensions" to invalid assumptions about datatype sizes. C and C++ code often also suffers from namespace pollution and poorly defined header files.

2. Libraries for parallel machines that assume that all processors will call the routine. Code using message passing that does not correctly use MPI communicators (without which the code is unsafe) also was a cause of failure.

3. Obscure or inappropriate data structures. For example, a code may require a data structure that is unnatural for the application (block cyclic matrix decompositions are an example) and in addition be vague about the exact decomposition in the hard (less regular) cases.

4. Slavish object-oriented design at the expense of performance.

5. Lack of modularity (see FFT example above).

4 What Customers Taught Us about PETSc

The most important part of a successful component is the design. For simple operations, such as dense matrix operations, the design is fairly obvious. For more complex operations, such as solving systems of nonlinear equations, there are far more choices to make. How is the problem described? What are the performance tradeoffs? What is the target computing environment and how does it affect the design? What components should be allowed for subproblems such as solving linear systems within the nonlinear solvers? How important is storage minimization? The PETSc approach for addressing these issues is to interact with our users (customers); in some cases, we are the customer in the role of PETSc user for our other research; in other cases, our customers may be colleagues or even strangers who have downloaded and installed PETSc.

PETSc has also benefited enormously from its customers. In broad outline, the following are key lessons that we learned. All of these are equally important.

1. Respond to questions and bug reports. This is more than "just support." Questions give vital feedback on the design of a component and the needs of users. Bug reports, besides saving the designer time finding a bug, can point out unstated assumptions in the design and implementation of a component.

2. Documentation and examples. Without these, the components are not useful. Many researchers complain that documentation and examples take too much time, but this need not be the case (see below). Further, if writing the software and running tests are like conducting a computational experiment, producing the documentation is like writing a paper. It is well established that credit for research depends on publishing a paper; shouldn't credit for developing a component depend on adequate documentation?

3. Performance. In scientific computing, performance is important. Even in research on numerical methods, it is vital to achieve good performance; otherwise, there is a real danger of developing an intuition about the problem that is incorrect.

   Performance is also a fairly subtle area. Few people measure performance carefully; the PETSc User Manual [4] includes an entire chapter on performance measurement (this is an example of training the customer).

4. The curse of orthogonality. Orthogonality of concepts is critical; without it, there is too much for the user to learn. But orthogonality of presentation (i.e., the interface) is crippling. In other words, the concepts must be few in number and reasonably independent. At the same time, the software developer must not slavishly refuse to create a new routine because the same thing can be done with a sequence of existing routines. Instead, the benefit of having a new routine (simplicity for the user) must be weighed against the cost (extra routine to document and test).

5. Learning curve. This is an area where most component libraries, including PETSc, need much more work. For PETSc, we have developed online tutorials and have given classes; we also have a large collection of example programs (separate from the test programs) that can be used to start a project. Even with these aids, the learning curve for PETSc remains too steep.

5 Building Components for Use by Others

Based on the experience with PETSc, a number of issues must be addressed when building a component for others to use. These may seem obvious, but software that follows these suggestions still seems to be the exception rather than the rule.

1. Portability. Language standards exist for a reason—use them. Just as one would not submit a research paper written in slang, one should not use dialects of C, C++, or Fortran unless it is impossible to accomplish the work otherwise. (System specific dialects are acceptable for a system specific tool, but only in that case.)

   A more subtle problem is name space pollution, that is, either internal or external routine names that conflict either with other user names or other libraries' names. The C++ namespace offers one solution to this problem. In C and Fortran, names must be more carefully selected. In particular, no common names should be used. C users can use the `static` declaration on local functions.

2. Avoidance of global state. This is actually harder than it sounds; users often think in those terms. In fact, it is unavoidable for some actions such as error handling. Both PETSc and MPI have what is essentially global state; in MPI, it is in the properties (in particular, the error handler) associated with `MPI_COMM_WORLD`.

3. Interoperability and composability. The designer should ensure that a component does not need to be in control. Ensuring that a component can interoperate has consequences in the handling of errors, the scope of parallelism (communicators should be used in MPI, for example), and initialization.

4. Documentation, examples, and support.

5.1 Mistakes Yet to Be Made

Successful components must work not only today but over the lifetime of an application and, to a lesser extent, over the lifetime of the customer, even while the nature of computing evolves. Perhaps the next "gotcha" for components is the lack of thread safety or an assumption that all threads in the process are participating in the component.

5.2 Mistakes Still Being Made

The following is a list of common mistakes made by developers of component software.

1. Ignorance of standards
2. Mandating interactive input
3. Requirement to be master
4. Requiring that all processors be used
5. Printing error messags and/or exiting the program
6. Makefiles for particular systems
7. Lack of portability in general
8. No documentation
9. No testing
10. No examples
11. Name space pollution
12. Failure of components to exceed lifetime of consumer applications
13. Monolithic library

One other common mistake requires some discussion. One approach to making software work with applications is to provide source code and have the end user modify the code to work with the application. There are many things wrong with this approach. For example, the end user may not fully understand the code; as a result, their changes may introduce bugs. Further, as soon as the code is modified, it becomes difficult to track improvements and fixes to the the original component source code. It is also a sign that the interfaces to the components are not correct. Finally, and perhaps most important, this approach denies the authors of the component valuable feedback on the design of software. As mentioned above, one of the sources of PETSc's success was a dialog with the users. An approach that transfers the source code to the user without encouraging feedback sacrifices much valuable input.

5.3 Help in Writing a Component

Many tools help automate the process of developing component software. These can remove much of the drudgery of documentation, testing, and porting codes. The following is a short list of tools used by PETSc; other projects have used these or similar tools. A more detailed discussion of these in the context of the MPICH implementation of MPI may be found in [13].

1. Standards. (C, C++, Fortran, MPI, POSIX, etc.) This is in the "do it right the first time" category. Designers should Get copies of the standards and read them.

2. Strict compilation flags. Most compilers will help identify both suspect usage (variables undeclared or used before defined) as well as violations of the standards.

3. Configure. The GNU Autoconf tool produces a shell script that can help identify system- and site-specific features of an environment. Basically, it automates the process that one would end up using to check whether, for example, the C compiler handled prototypes properly, whether `perl` was version 4 or 5, or whether the size of a Fortran integer was 4 or 8 bytes.

4. Doctext [12]. This tool takes structured comments from the source files and produces documentation in Unix man (nroff), Web (HTML), and LaTeX form. A similar tool, bfort [11], is used to provide Fortran interfaces for the C routines in PETSc.

5. Problem reporting. We use `req` [9] to manage questions and bug reports. This is based on a simple e-mail interface. We've tried tools like the GNU GNATS, but found that their more complex interface discourages users.

6. Integrate. The best help—and the one often ignored—comes form trying to integrate with someone else's code!

Acknowledgments

The author thanks the other members of the PETSc team, Satish Balay, Lois McInnes, and Barry Smith, for the hard work and commitment to users that has made PETSc possible.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] M. Adams and J. Demmel, *A parallel maximal independent set algorithm*, Technical Report CSD-98-993, University of California, Berkeley, Mar. 11, 1998.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide, 2nd ed.*, SIAM, Philadelphia, 1995.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *PETSc home page.* http://www.mcs.anl.gov/petsc, July 1997.

[4] ——, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, Apr. 1998.

[5] S. T. Barnard and R. L. Clay, *A portable MPI implementation of the SPAI preconditioner in ISIS++*, Tech. Rep. NAS-97-002, NASA Ames Research Center, Jan. 1997.

[6] A. J. Beaudoin, *Use of polycrystal plasticity theory in assessing material performance*, in Simulation of materials processing: Theory, methods, and applications - Proceedings of The Sixth International Conference, NUMIFORM'98, J. Huétink and F. P. T. Baaijens, eds., Rotterdam, Netherlands, June 1998, A. A. Balkema Publishers. To appear.

[7] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, *A set of level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1–17.

[8] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.

[9] R. Evard, *Managing the ever-growing to do list*, in USENIX Proceedings of the Eighth Large Installation Systems Administration Conference, 1994, pp. 111–116.

[10] A. George and J. W.-H. Liu, *User guide for SPARSPAK: Waterloo sparse linear equations package*, Tech. Rep. CS-78-30, Department of Computer Science, University of Waterloo, 1978.

[11] W. Gropp, *Users manual for bfort: Producing Fortran interfaces to C source code*, Tech. Rep. ANL/MCS-TM-208, Argonne National Laboratory, Mar. 1995.

[12] ——, *Users manual for* `doctext`*: Producing documentation from C source code*, Tech. Rep. ANL/MCS-TM-206, Argonne National Laboratory, Mar. 1995.

[13] W. Gropp and E. Lusk, *Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing*, The International Journal of Supercomputer Applications and High Performance Computing, 11 (1997), pp. 103–114.

[14] W. D. Gropp, *Why we couldn't use numerical libraries for PETSc*, in Proceedings of the IFIP TC2/WG2.5 Working Conference on the Quality of Numerical Software, Assessment and Enhancement, R. F. Boisvert, ed., Chapman & Hall, 1997, pp. 249–254.

[15] W. D. Gropp, D. Keyes, L. McInnes, and M. Tidriri, *Parallel implicit PDE computations: Algorithms and software*, in Proceedings of Parallel CFD'97, Elsevier, 1997, pp. 333–344.

[16] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM Journal on Scientific Computing, 18 (1997), pp. 838–853.

[17] IBM, *Engineering and Scientific Subroutine Library Version 2 Guide and Reference*, release 2 ed., 1994. SC23-0526-01.

[18] M. T. Jones and P. E. Plassmann, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, Tech. Rep. ANL-95/48, Argonne National Laboratory, Dec. 1995.

[19] D. Kausik, D. Keyes, and B. Smith, *On the interaction of architecture and algorithm in the domain based parallelism of an unstructured grid incompressible flow code*, in Domain Decomposition Methods 10, AMS, 1998, pp. 287–295.

[20] *ParMETIS: Parallel graph partitioning & sparse matrix ordering.* http://www-users.cs.umn.edu/~karypis/metis/parmetis/main.html.

[21] *PVODE: A parallel solver for ordinary differential equations.* http://www.llnl.gov/CASC/PVODE.