ANL/MCS/CP--85098
Conf-9310220--7

Scalable, Extensible, and Portable Numerical Libraries

William Gropp[*]
Mathematics and Computer Science
Argonne National Laboratory
Argonne, IL 60439

Barry Smith[†]
Department of Mathematics
University of California, Los Angeles
Los Angeles, CA 90024

Abstract

Designing a scalable and portable numerical library requires consideration of many factors, including choice of parallel communication technology, data structures, and user interfaces. The PETSc library (Portable Extensible Tools for Scientific computing) makes use of modern software technology to provide a flexible and portable implementation. This talk will discuss the use of a meta-communication layer (allowing the user to choose different transport layers such as MPI, p4, pvm, or vendor-specific libraries) for portability, an aggressive data-structure-neutral implementation that minimizes dependence on particular data structures (even vectors), permitting the library to adapt to the user rather than the other way around, and the separation of implementation language from user-interface language. Examples are presented.

1 Introduction

Numerical libraries must have certain features in order to obtain widespread use in the application community. In this paper we will discuss five of these principles and particular techniques used in PETSc to support them.

1.1 Adaptive to the User

Writing libraries of routines for scientific computing, particularly on parallel machines, involves more than identifying the appropriate algorithms to use. A library routine needs to meet the needs of applications programmers; otherwise, the programmer is likely to write their own routines. It is important that the library adapt to the application, not the other way around. This often requires a commitment of the library to use the data-structures of application. Hence the need for data-structure-neutral libraries. Otherwise the application programmer will be constantly recoding the same algorithm for a different data structure.

1.2 Portable

The portability of library routines is becoming increasingly important as codes are being required to run on several varieties of parallel computers. One part of portability that is often missed is that programs should be portable to *computers*, not just uniprocessors *or* parallel processors. The time is nearly over when just parallelizing an application will be interesting in and of itself. Portablity of libraries must also mean portable across programming languages. In order to obtain maximal reuse, the same libraries must be useable from several languages.

1.3 Extensible

Extensibility of the library to allow for new algorithms and implementations to be used in an application without rewriting the application code is also important.

1.4 Scalable

A library and its underlying implementation must allow for and encourage scalable algorithms and implementations. But most algorithms often reach maximum performance at a finite number of nodes (See 1). In order to provide optimal performance in this

[*]This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38

[†]This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38 and in part by the Office of Naval Research under contract ONR N00014-92-J1890

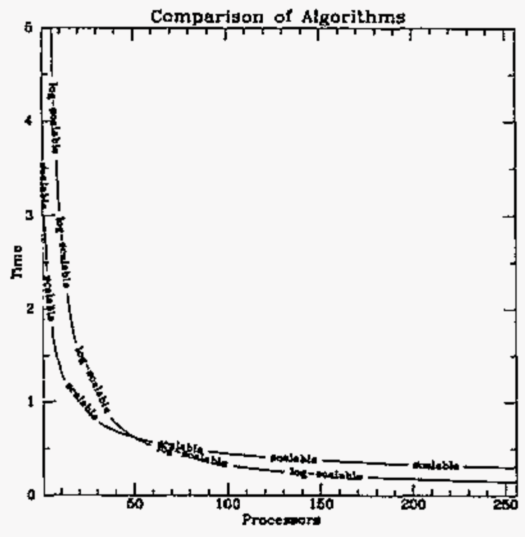

Figure 1: Comparison of scalable and partially scalable algorithms. Compares scalable iteration ($\sqrt{p}$ iterations) with domain decomposition ($\mathcal{O}(1)$ iterations) but $\log p$ cost per iteration.

case, a library must be prepared to easily use a subset of the available processors when solving a problem.

1.5 Efficient

A final but important issue is that of efficiency. There is more to this than just careful coding or using level 3 BLAS. In many cases, it is important to provide the option to aggregate operations so that they may be performed more efficiently. An example of this is in sparse-matrix assembly. In many cases, the sequence of operations that a user follows is to build a sparse matrix, adding one or more elements at a time. No other operation is performed on the sparse matrix until the user completes the assembly process. There are significant performance gains to be realized by deferring the actual update to the sparse matrix until the end of the assembly process. Thus, a library organization that encourages this sort of aggregation of operations without imposing much if any burden on the applications programmer will be more efficient than one that does not.

In the following sections, we will describe some of the techniques that we have used in our library, PETSc (Portable Extensible Tools for Scientific computing). Section 2 discusses an method for separating the mathematics of an algorithm from its implementation. Section 3 discusses how we have approached the issue of writing portable programs for distributed-memory parallel computers. Section 4 gives some examples of the use of our library. More information may be found in [1], [2], and [3].

2 Data-structure neutral design

Often, a library routine will define the data structures to be used for the input and output arguments. While this maybe acceptable in the case of common, frequently used data structures, it is becoming increasingly less useful as applications begin to use more sophisticated data structures. This is a problem for both existing and new applications.

In existing applications, the application programmer has already chosen data structures. A library that can not adapt to these data structures may put too much burden on the application programmer. The application programmer will then spend considerable time recoding the algorithms, rather than using the library.

More and more new applications are choosing data structures specific to that application; these data structures may be irregular or arranged in unexpected ways. As in the case of existing applications, the library must be prepared to work with these data structures rather than insisting on a particular arrangement.

A common argument that many programmers give for not using libraries is that the appropriate algorithm can be coded directly into the application, adapting the algorithm to the application's data structures. This, however, requires a complete detailed understanding of all incorporated algorithms, by the application programmer. With the trend toward using more sophisticated, mathematical algorithms, this is impossible. The net result is only simple, less efficient algorithms are used, when more effective ones are actually needed.

One important advantage that a library can offer is the ability to transparently make use of new algorithms. For example, a new Krylov method for the iterative solution of linear equations could be used by simply relinking the application. Another source of new algorithms (and implementations) are proprietary libraries (e.g., IBM's ESSL [4]). Applications programmers are often faced with using these libraries and sacrificing portability or not using then and sacrificing performance. A library that is extensible to new methods can offer applications programmers a good way to get the best of both worlds: with little (if any) change to the source code, the programmer can select either a proprietary or portable implementation of an algorithm.

In order to maximize the range of applications to which a library can be applied, it is crucial to exploit the fact that the mathematical expression of an algorithm does not imply a specific data structure. We emphasize that this applies not just to potentially complicated data structures (such as sparse matrices) but to such simple structures as vectors.

A key to the development of portable and extensible libraries is the use of *data-structure neutral design.* By this we mean that code should be written to operate on the natural objects (matrices, vectors, graphs) without reference to their representation as data-structures. For example, our Krylov methods package is independent of the data-structure used for the *vectors*; they can dense and contiguous in memory (the usual assumption in other Krylov methods implementations), sparse (e.g., a partial oct-tree), or distributed (among many nodes in an MPP).

In order for the data structure neutral approach to be effective it must provide all of the functionality the application programmer needs, while maintaining high efficiency. The general functionality can be achieved by providing a way to convert into and out of a canonical representation on which the less used operations will be performed. For the more common operations associated with a data-structure, code must be provided to perform these operations. We call these common operations the *primitives.* Efficiency can be achieved by providing an adequately large set of primitive operations. This is discussed below, in more detail, in the context of our implementation of sparse matrices.

Because the set of operations may be large (for example, our vector set includes more than 15 basic operations), an important implementation issue concerns how all of this information is passed to the routines that need to perform these operations. Another important, but often overlooked, issue concerns how new operations can be added in a backward-compatible fashion. Our solution to these problems is to use an encapsulation of this data into an object that we call a *context.* A context contains the data structure and the operations on it. This object hides the details of an implementation (or algorithm!) and permits the programmer to deal with the mathematical problem rather than the details of a specific implementation.

The basic approach is to create a context, modify that context (by setting various options or hints), use the context to solve a problem, and finally destroy the context in order to free any resources (such as memory) used.

Table 1: Table of required sparse matrix operations

Routine	Operation
`create`	Create a new matrix
`destroy`	Destroy a matrix (recover space)
`createsplit`	Create a structure to hold a factored form of the matrix
`gathertorow`	Insert a row into the matrix
`gatheraddtorow`	Add to a row of the matrix
`scatterfromrow`	Extract a row from the matrix

Because a context is an opaque object (an object whose contents are hidden from the application code), it is relatively easy to extend the choices of methods by adding new options to the context *without changing existing codes!* To do this, new methods and perhaps new data structures are defined. For example, to add a new Krylov method, it is necessary only to write the routine to perform the operations and to register that with the routine that creates Krylov method contexts.

We explore these abstract concepts more concretely with two examples taken from PETSc. The first is our implementation of sparse matrices. The second is our implementation of Krylov space iterative methods.

In our implementation of sparse matrices each storage format must have provided with it the matrix primitives listed in Table 1. In addition, a wide variety of operations, such as matrix-vector multiply and incomplete factorization, may be (but are not required to be) defined for each matrix type. Occassionally one may require an operation on a sparse matrix which is not one of the matrix primitives. These specialized operations may be implemented by first converting to the canical form using `SpToCanonical`, performing the needed operation and then converting back via `SpFromCanonical`. Obviously this involves some additional overhead, however for little used operations this overhead is minimal. If one of the specialized operations is used often then it will be optimized and become an additional matrix primitive.

We have followed the same approach with our Krylov-space package (KSP) for the iterative solution of linear systems. Each Krylov method is described by a context that contains contexts for vector operations (thus allowing the same Krylov code to be used for uniprocessors and MPPs), the matrix operations, the iterative state (e.g., maximum number of iterations

```
   program main
   external amult, binv
   include "iter/fiter.h"
   double precision x(100), y(100)
   integer itctx, its
c
   itctx = ITCreate( ITCG )
   call DVSetDefaultFunctions(
  +             ITVectorContext( itctx ),
  +             100 )
   call ITSetRelativeTolerance(itctx,1.d-10)
   call ITSetIterations(itctx,25)
   call ITSetAmult(itctx,amult, n)
   call ITSetBinv(itctx,binv, n)
   call ITSetSolution(itctx,x)
   call ITSetRhs(itctx,b)
   call ITSetUp(itctx)
   call its = ITSolve(itctx)
   call ITDestroy(itctx)
   end
```

Figure 2: Simple Fortran program to solve a linear system with CG

and convergence criteria), and any information specific to that method (e.g., number of direction vectors for GMRES).

Figure 2 shows a Fortran program for solving a linear system with KSP. The routines `ITSetIterations` and `ITSetRelativeTolerance` are examples of optional calls. Defaults are supplied for all parameters by `ITCreate`; these defaults may be overridden by calling routines for those specific parameters. This approach makes it easy to add additional features; in addition, it can simplify the presentation to potential users because advanced features (such as graphical display of the residual, which is available in KSP) need not be presented to beginning users.

By replacing the functions `amult`, `binv`, and the call to `DVSetDefaultFunctions` by its parallel equivalent `DVPSetDefaultFunctions`, this program becomes parallel.

3 Portable parallel code

The approach that we have taken for programming distributed-memory parallel computers is to define a message-passing interface that maps efficiently onto most existing systems. Thus, we consider our system a "meta" message-passing system that supports a variety of *transport layers* including p4, PVM, and vendor-specific systems such as Intel nx, TMC CMMD, and IBM EUI. Because this system supports a variety of transport layers, and because it can also be used to run programs written in one transport layer under a different transport layer, we call the system *Chameleon*. Chameleon also provides support for a common startup interface and correctness and performance debugging, as well as for user-defined subsets of processors called *procsets*.

```
[0] recvstart <Tag 1> [buggy.c:17]
[1] recvstart <Tag 1> [buggy.c:17]
```

Figure 3: Sample trace output from Chameleon, showing deadlock in a two processor program

An important recent development is the creation of a message-passing interface (MPI). To Chameleon, MPI is another transport layer. MPI does offer some important enhancements over existing message-passing systems, such as message communicators (separate contexts or message-passing domains) and non-contiguous data, and Chameleon needs to provide access to these functionalities.

Chameleon also addresses a number of issues that MPI explicitly avoids, such as debugging and profiling aids, the program invocation environment, and parallel I/O. For example, using a command-line switch, Chameleon can be instructed to generate a trace of all message-passing operations (Figure 3).

Chameleon offers excellent performance. The BlockSolve code of Jones and Plassman [5], winner of the 1992 Gordon Bell Prize for Speedup, is written using Chameleon.

A portable message passing system, however, is only the first step in producing application oriented software libraries. Ideally, the software libraries will come with standard parallel data-structures which the application programmer may use without directly writing message passing code. In addition, it should include mechanisms which allow the application programmer to easily construct new parallel data structures easily. By providing a hierarchy of libraries, PETSc is able to handle the most technically oriented application programmer who choses to directly program using the message passing paradigm, as well as the more result oriented programmer who would like to ignore all architectural issues, but still obtain good performance.

To demonstrate this we will consider briefly our im-

```
typedef struct {
 /* Number of local indices and their
    global values */
 int  nl, *g;
 /* Map from local to global */
 void (*ltog)(), *ltogctx;
 /* Map from global to local */
 void (*gtol)(), *gtolctx;
 /* Map from global to owner */
 void (*gtop)(), *gtopctx;
 /* Generic context */
 void *ctx;
 /* Routine to free any storage */
 void (*destroy)();
 /* Procset for all members of mapping */
 ProcSet *pset;
 } PSPMapping;
```

Figure 4: Sparse-matrix mapping data

plementation of distributed sparse matrices. Since the distributed sparse matrices are based on many of the same matrix primitives as the sequential version, application codes can easily be moved between sequential and parallel machines.

A distributed sparse matrix can be described in many ways. We will consider here only a row-oriented distribution; other distributions are similar. In a row-oriented distribution, each processor is responsible for some rows of the matrix. These rows need not be contiguous. One way to specify and to represent such a matrix is for each processor to hold a rectangular matrix, with the number of rows being the number of "local" rows and the number of columns being the number of "global" columns. When using a row-oriented sparse matrix format, this requires storage proportional to the number of non-zeros in the matrix. In addition, a mapping from local to global rows is needed.

In our package, the local rows can be represented with a sparse matrix represented in a data-structure-neutral way, allowing the choice of local representation to be made to fit the problem. In particular, special code or vendor-specific uni-processor sparse matrix implementations may be used.

To perform a parallel operation such as matrix-vector product, additional information is needed. In particular, a mapping from global row to local row on a processor is required. We handle this by defining a set of mapping operations, shown in Figure 4.

```
PSPMat      *pmat;
int         mx = 4
PSPMapping  *pmap;
...
pmap = PSPMapCreate( PSAllProcs, nl, g );
PSPMapSetDefaults( pmap );
pmat = PSPMatCreate( pmap, PSAllProcs,
             mx, mx );
PSPMatStartAsmble( pmat );
for (i=PImytid; i<mx; i+=np) {
    PSPMatAddElm( pmat, i, i, (double)i );
    if (i > 0)
        PSPMatAddElm( pmat, i, i-1, -1.0 );
    if (i < mx - 1)
        PSPMatAddElm( pmat, i, i+1, -1.0);
    }
PSPMatEndAsmble( pmat );
```

Figure 5: Creation of a distributed matrix with each processor contributing some rows

However, since providing all of these routines can be a burden on the average user, we provide a default set of routines that use a distributed data-server that operates semi-synchronously (for portability). However, if the user can provide these mappings independently, she can provide the routines for the mappings explicitly. An example is a regular mesh decomposition; in this case, all of the mappings can be easily computed without any global information or communication.

Additional tools are used to schedule and perform and communications necessary for these operations. For example, much of the matrix-vector product code is shared with parallel vector scatter/gather.

At a higher level, there are additional routines to do the specification and assembly of a distributed matrix without requiring that the "local" parts be distributed by the user. These allow a matrix to be specified element-wise, with the process of generating the matrix (involving caching of modifications and occasional communication) hidden from the user. This is sketched in Figure 5.

Since scientific computing today is done using several different programming languages it is imperative that the same libraries be usable from all of the languages used by application programmers. It is a waste of resources to provide two or more complete sets of source code in different languages. PETSc is written in C, in order to make the PETSc library available to Fortran (and in the future, perhaps other languages) we have implemented a set of routines which translate

```
worker( argc, argv )
int  argc;
char **argv;
{
PSpMat *pmat;
PSVctx *ctx;
PSVMETHOD pmethod = PSVNOPRE;
ITMETHOD  itmethod = ITGMRES;
SpMat  *mat;
int    mx, my, si, ei, sj, ej;
int    *lidx;
double *b, *x;
int    its;

/* Figure out my part of the matrix and generate it */
mx = 16;
my = 16;
SYArgGetInt( &argc, argv, 1, "-mx", &mx );
SYArgGetInt( &argc, argv, 1, "-my", &my );
pmethod + PSVJacobi;
PSVGetMethod( &argc, argv, 1, "-psv", &pmethod );
ITGetMethod( &argc, argv, 1, "-itmethod", &itmethod );
PSVGet2dStripDecomp( mx, my, &si, &ei, &sj, &ej, PSAllProcs );
mat  = FDBuildLaplacian2dBndySub( mx, my, 0.0, 0.0, 0.0, si, ei, sj, ej );
lidx = PSVGet2dMapping( mx, my, si, ei, sj, ej );
pmat = PSpCreate( mat, lidx );

ctx  = PSVCreate( pmat, pmethod );
PSVSetAccellerator( ctx, itmethod );
PSVSetUp( ctx );

/* Actually solve the problem */
b   = (double *)MALLOC( ln * sizeof(double) );      CHKPTR(b);
x   = (double *)MALLOC( ln * sizeof(double) );      CHKPTR(x);
its = PSVSolve( ctx, b, x );
if (PImytid == 0)
    printf( "[%d] Completed in %d iterations\n", PImytid, its );

PSVDestroy( ctx );
return errs;
}
```

Figure 6: Code to solve a linear system in parallel

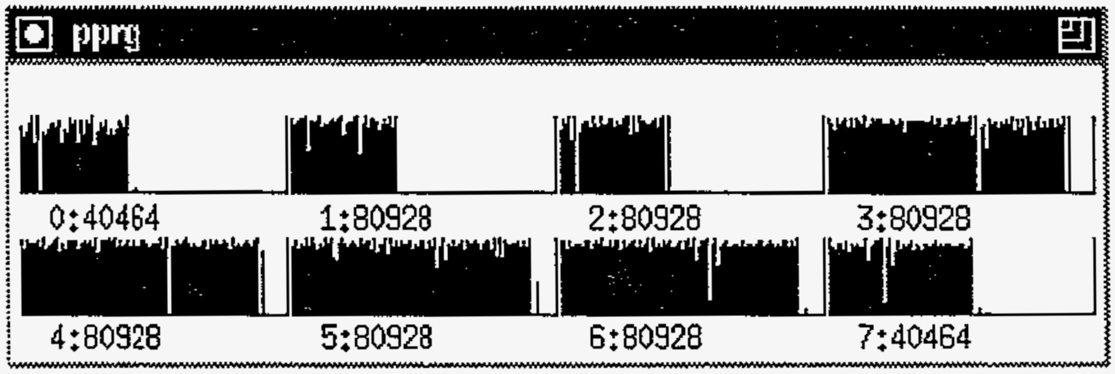

Figure 7: Display for 8 processors showing the communication activity of a running parallel program

the calling sequences of C functions into corresponding Fortran functions. Note that we do not translate the C programs, instead the user calls a Fortran stub, generated automatically at installation time, which calls the corresponding C routine. The stub handles the appropriate conversion of pointers to values, etc. With this approach any changes in the C source are immediately reflected in the Fortran version. The manual task of maintaining two interfaces is almost completely eliminated.

The construction of documentation for the software is also handled in the same way. In the source code associated with each subroutine are specially formated comments. A text processing program is used to format the comments in a variety of ways: to Unix man pages, to latex format and to emacs help format. Since the comments are in the actual source code, any changes to the source result in revised documentation. A common problem in many software packages is the the documentation always lags behind the actual library by one revision. This will not happen with the automatic generation of documentaion from the source.

4 Examples

In this section we present an example of code written using the PETSc package. These illustrate our points that a library can, with a consistent interface, provide a variety of different algorithms that solve the same mathematical problem to the applications programmer.

Figure 6 shows a parallel linear system solver that provides a wide variety of Krylov methods coupled with preconditioners. The same interface (and even code!) can handle direct and iterative methods. If a new Krylov method or preconditioning technique were discovered, this program would only need to be relinked; not a line would change (note that the program uses the command line to choose the actual method). The matrix for this example is build with `FDBuildLaplacian2dBndySub` which is a routine for building distributed sparse matrices.

One of the features that Chameleon provides is the ability to attach user-defined operations to each communication operation. Chameleon uses this feature to provide the user with runtime graphical information on the progress of a parallel program. Output for an eight-processor parallel job is shown in Figure 7. This display provides a useful diagnostic aid for developing programs; even for production programs, the overhead of this display (updated every few seconds) is relatively small.

References

[1] William D. Gropp and Barry F. Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[2] William D. Gropp and Barry F. Smith. Simplified Linear Equation Solvers users manual. Technical Report ANL-93/8, Argonne National Laboratory, March 1993.

[3] William D. Gropp and Barry F. Smith. Users manual for KSP: Data-structure-neutral codes implementing Krylov space methods. Technical Report ANL-93/30, Argonne National Laboratory, August 1993.

[4] IBM. *Engineering and Scientific Subroutine Library Version 2: Guide and Reference*, 1992.

[5] Mark T. Jones and Paul E. Plassmann. An efficient parallel iterative solver for large sparse linear systems. In *Proceedings of the IMA Workshop on Sparse Matrix Computations: Graph Theory Issues & Algorithms*, Minneapolis, 1991. University of Minnesota.