Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries

Satish Balay
William D. Gropp
Lois Curfman McInnes
Barry F. Smith [1] [2]

ABSTRACT Parallel numerical software based on the message-passing model is enormously complicated. This paper introduces a set of techniques to manage the complexity, while maintaining high efficiency and ease of use. The PETSc 2.0 package uses object-oriented programming to conceal the details of the message passing, without concealing the parallelism, in a high-quality set of numerical software libraries. In fact, the programming model used by PETSc is also the most appropriate for NUMA shared-memory machines, since they require the same careful attention to memory hierarchies as do distributed-memory machines. Thus, the concepts discussed are appropriate for all scalable computing systems. The PETSc libraries provide many of the data structures and numerical kernels required for the scalable solution of PDEs, offering performance portability.

1 Introduction

Currently the only general-purpose, efficient, scalable approach to programming distributed-memory parallel systems is the message-passing model. Other approaches, based on parallel languages or compiler directives, have worked well on shared-memory computers, particular hardware platforms (e.g., CM-5) [Thi93], or specific problems but have never been able to demonstrate general applicability. The chief drawbacks to the message-passing model have been

(1) lack of portability due to varying syntax for message passing or inefficient and poorly designed portable systems, and
(2) the difficulty experienced by end users in writing complicated message-passing code.

Fortunately, with the development of the Message Passing Interface (MPI) [GLDS96b], [MPI94], [GLS94], [SOHL+95], drawback (1) is no longer a problem. MPI is an efficient, robust standard to which the major vendors are adhering. In addition, several high-quality implementations are freely available [BDV94], [GLDS96a]. Another advantage of MPI is that it is fully usable from Fortran 77, C, and C++; this feature allows programmers to use the language that is most appropriate for a particular task or with which they are most comfortable. Another important aspect of MPI is that it provides specific mechanisms to support the development of portable software libraries that most previous message-passing systems did not provide.

---

[1] To appear in **Modern Software Tools in Scientific Computing**, E. Arge, A. M. Bruaset and H. P. Langtangen, Ed. Birkhauser Press, 1997.
[2] Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844. balay@mcs.anl.gov, gropp@mcs.anl.gov, curfman@mcs.anl.gov, bsmith@mcs.anl.gov, `http://www.mcs.anl.gov/petsc/petsc.html`.

Drawback (2) is far more challenging; however, the implementation of high-quality parallel numerical libraries using modern software engineering practices can ease this difficulty enormously. This paper discusses specific techniques used in PETSc 2.0 (the Portable, Extensible Toolkit for Scientific computation) to allow end users, with relative ease, to write sophisticated parallel application codes that involve the numerical solution of partial differential equations (PDEs) through the use of portable, efficient numerical libraries. Thus, we assert that the combination of

the message-passing model
+
carefully designed and implemented parallel numerical libraries

is the solution to the problem of efficiently using large-scale distributed-memory, as well as clustered and NUMA (non-uniform memory access) shared-memory computers. This approach enables us to face the explicit tradeoffs that must be made to balance the code's performance (computational efficiency) and ease of use (programmer efficiency). Most important, this combination allows the gradual process of improving performance by the addition of new computational kernels, while retaining the remainder of the correctly working libraries and application code.

Over the past fifteen years, various government funding agencies world-wide have invested heavily to make parallel computing usable for more than very special high-budget custom projects. Aside from hardware development, much of the research funding has been directed toward (1) basic computer science research in parallel programming paradigms and languages and (2) basic mathematical research in parallel algorithm development and analysis. These two research thrusts have often been orthogonal, with neither leveraging the corresponding advances in the other field. Thus, the resources devoted to the high-performance communication and computation (HPCC) community unfortunately have not led to the advances in the usability of parallel computing that many had anticipated. Developing efficient applications for massively parallel processors (MPPs) and clusters is still difficult and requires a large commitment from application scientists.

The approach used in PETSc is to encapsulate mathematical algorithms using computer science developments in object-oriented programming and message passing. Object-oriented programming techniques enable us to manage the complexity of efficient numerical message-passing codes. All the PETSc software is freely available and used around the world in a variety of application areas [BGMS96].

Our approach does not attempt to completely conceal parallelism from the application programmer. Rather, the user initiates combinations of sequential and parallel phases of computations, but the library handles the detailed (data-structure dependent) message passing required during the coordination of the computations. This provides a good balance between ease of use and efficiency of implementation. In this article we discuss six of our main guiding design principles; the first four focus on allowing the application programmer to achieve high performance, while the last two focus on ease of use of the libraries.

- Performance
  - overlapping communication and computation,
  - determining within the library the details of various repeated communications, and optimizing the resulting message passing code (similar to the inspector/executor model of PARTI/Chaos [ASS93]),
  - allowing the user to dictate exactly when certain communication is to occur, and
  - allowing the user to aggregate data for subsequent communication
- Ease of use
  - allowing the user to work efficiently with parallel objects without specific regard for what portion of the data is stored on each processor, and

– managing communication whenever possible within the context of higher-level operations on a parallel object or objects instead of working directly with lower-level message-passing routines.

Note that the first four principles are chiefly related to reducing the number of messages, minimizing the amount of data that needs to be communicated, and hiding the latency and limitations of the bandwidth by sending data as soon as possible, *before* it is required by the receiving processor. The six guiding principles, embedded in a carefully designed object-oriented library, enable the development of highly efficient application codes, without requiring a large effort from the application programmer. We note that PETSc is not intended to be a complete parallel mathematical software library like the Thinking Machines' Scientific Software Library; rather, PETSc focuses on components required for the solution of PDEs and related problems.

Another strength of the approach of message passing combined with numerical libraries is that application codes written with this model will also run well on shared-memory computers—often as well as codes custom written for a particular machine. This translation occurs because even shared-memory machines have a memory hierarchy that message-passing programs must inherently respect. For the small number of code locations where taking explicit advantage of the shared memory can lead to improved performance, alternative library routines that bypass the message-passing system may easily be provided, thus retaining a performance-portable library.

Other researchers are also investigating object-oriented techniques and programming strategies for large-scale numerical software. A few of the projects that are most closely related to PETSc in the problems they address include Diffpack [BL96] (a collection of uniprocessor libraries for solving PDEs), Aztec [HST95] (a library for iteratively solving linear systems in parallel), and POOMA [RCH+96] (a framework for parallel scientific simulation). The unique aspect of PETSc compared with other packages is the complete integration of the six guiding principles throughout its design.

This article is organized as follows. In Section 2 we introduce the message-passing programming model and discuss why it can result in highly efficient programs but why programming with raw message passing is difficult for most numerical applications. Section 3 introduces the concept of a parallel distributed object (for example, a matrix) and explains how it is managed in PETSc. Section 4 briefly explains the importance of each of the six conceptual principles introduced above. Section 5 introduces the design of several fundamental PETSc objects and for each explains how the six guiding principles are related. This section discusses several important components and operations within numerical libraries for PDEs, namely

- vector assemblies,
- vector scatters and gathers,
- matrix assemblies,
- parallel matrix-vector products,
- parallel computation of Jacobian matrices, and
- linear and nonlinear solvers.

We discuss our implementation techniques for balancing efficiency and ease of use. We conclude the section by explaining how all three principles of object-oriented programming are crucial to the management of complexity in the PETSc design. Section 6 demonstrates the performance possibilities of an application code written using PETSc by presenting results for a three-dimensional, fully implicit Euler simulation.

This article is not intended as a users guide or introduction to the use of PETSc; for that information we refer readers to the PETSc users manual [BGMS95]. Rather, this article discusses in some technical detail several specific aspects that are important in the design of PETSc. In fact, users of PETSc do not have to understand the technical details discussed here in order to use PETSc effectively.

2 The Message-Passing Model for Programming Distributed-Memory Parallel Systems

Hardware for parallel computers has been designed in many ways, which can be distinguished by memory layout and interconnection schemes. The main spectrum includes common memory and bus shared by all processors, common memory connected to all processors through a switch, separate memory "owned" by one processor but directly accessible to all processors, and separate memory accessible only to its local processor [HJ88]. Each approach has both advantages and disadvantages. The common memory approach is limited by the ability of the memory banks to serve all processors efficiently, while the distributed-memory approach is limited by the need of all processors to share data. Thus, even moderately scalable systems (and single-processor systems as well) have a hierarchy of local and remote memory that is managed directly by the hardware.

At the programmer's level, of course, the details of the memory systems are well hidden. The programmer uses an abstract memory model (or parallel programming model) that is somehow related, through system software and hardware, to the physical machine.

2.1 Flat Address Space

In the simplest parallel programming model, the application programmer works with a *flat memory structure*; all processors share the same address space and are free to change data at any location in memory. This model's inherent simplicity is countered by two main drawbacks.

- The user must ensure that two (or more) different processes do not generate inconsistent values in memory. For example, two processes simultaneously incrementing the same memory location by one may actually increase the value by only one, rather than the intended two. While techniques for this are well understood in theory, including locks and monitors, it does require care on the part of the user to prevent programming errors and hot-spots (bottlenecks in the program where several processes are waiting on the same locks).

- A flat address space is not scalable; even with very sophisticated hardware only extremely carefully written code can completely utilize more than a few processors. To achieve good performance, even recent machines such as the SGI/Cray ORIGIN2000 will require libraries, such as PETSc, that truly acknowledge and respect the memory hierarchy of the machine. In fact, such carefully tuned shared-memory code strongly resembles message-passing code in that chunks of data are moved among "local" memories in a very controlled way.

Parallelizing compilers have been postulated as the cure for these two problems, and on a small scale they have been quite successful. But even on systems for which parallelizing compilers work well, they are often limited to highly structured code for which the compiler can detect parallelism (e.g., double loops and rectangular array operations). Compilers that can handle general sparse matrices, for example, are only at the beginning research stages, while these are exactly the types of matrices that applications scientists need to use on a regular basis. Even if parallelizing compilers vastly improve, it seems highly unlikely that they will ever be able to compile complex sequential application codes into even moderately efficient parallel codes. In fact, few do well even in entirely sequential codes for sparse matrix operations.

2.2 Message Passing

The other standard parallel programming model is *message passing*; in this model each process can directly access only its own memory and must explicitly communicate with other processes to access the data in their memories. The communication is done through the *send* and *receive* operations. Thus, both the sending and receiving processors must be involved whenever a remote memory location is accessed. For example, if process 1 wanted to add to its local variable **x** the value **y** from processor 0, the code for the two processes could look like the following:

```
Process 0 code          Process 1 code
--------------          --------------
MPI_Send(y,......);     MPI_Recv(mess,....);
                        x += mess; /* Add the remote data to x */
```

The Message Passing Interface (MPI) standard contains a wide variety of basic communication routines, including reductions, scatters, and broadcasts [MPI94],[GLS94], [SOHL+95]. But these routines are predicated on both the sending and receiving processors being aware of the data's origin and destination. Consequently, writing complicated message-passing codes is tedious and prone to error. To illustrate this point, we consider a specific example: sparse matrix-vector multiplication, $y = A * x$. This operation occurs in most iterative linear solvers, scalable eigenvalue solvers, etc.

Why Writing Message-Passing Code Is Tedious

Given a parallel sparse matrix A and a parallel vector x, we wish to write a routine that scalably and efficiently computes $A * x$. We assume that A is distributed by rows among a collection of processors; that is, each processor contains (for simplicity) an adjacent series of rows of the matrix and the corresponding elements of the vector. See the matrix depicted below for an example division among three processors.

One could easily code a naive sparse matrix-vector product using MPI. Each processor could broadcast its elements of the vector x to all other processors with the command

```
MPI_Allgatherv(local,localsize,MPI_DOUBLE,global,localsizes,...);
```

Here each processor contributes its piece (of length `localsize`, called `local`) of the entire vector to a copy of the entire vector (called `global`), which lies in its entirety on each processor. The advantages of this approach are that the communication call is simple, and every processor knows exactly what messages it must send and receive. One disadvantage is that this code is not scalable; the amount of communication grows as $O(n)$, where n is the number of columns in the matrix. In addition, memory is wasted since each processor must store a complete copy of the vector x. Note that even in this simple case each processor must know the amount of data to expect from all other processors, as determined by prior communication (in a setup phase) and given by the array `localsizes`.

To discuss how we can take advantage of matrix sparsity, we consider the following matrix, which is partitioned by rows among three processors so that processors zero, one, and two "own" submatrices consisting of rows 0–2, 3–5, and 6–7, respectively. The corresponding vector is partitioned accordingly.

$$\left(\begin{array}{cccccccc} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\ 9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\ \hline 13 & 0 & 0 & 15 & 16 & 17 & 0 & 0 \\ 0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\ 0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\ \hline 25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\ 30 & 0 & 0 & 0 & 0 & 33 & 0 & 34 \end{array}\right) \left(\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \hline x_3 \\ x_4 \\ x_5 \\ \hline x_6 \\ x_7 \end{array}\right)$$

Note that each processor's local submatrix contains certain columns with all zero elements, so that when computing a matrix-vector product, $y = A * x$, each processor requires only a subset of the entries of x. In this example, processor zero does not need x_4, x_5, and x_7; processor one does not need x_2 and x_7; and processor two does not need x_3 and x_4. To minimize communication, processor zero should distribute x_0, x_1, and x_2 to processor two but only x_0 and x_1 to processor one. Likewise, processor one need only distribute x_3 to processor zero and x_5 to processor two.

Meanwhile, processor two needs to distribute x_6 to both processors zero and one. Clearly, for this small problem the communication reduction achieved by this organization is not worth the coding difficulties, but for large sparse problems (e.g., $n = 1,000,000$) for which the communication can potentially drop to 500 from $1,000,000$, such reduction is very important. What makes this problem nontrivial is that no processor knows *a priori* what components of the vector other processors will require.

Since sparse matrix-vector products are only a small part of a large application code, it is unrealistic to require an application programmer not highly trained or interested in message-passing programming to write all the code required to perform efficient parallel sparse matrix-vector products. In later sections, we discuss how PETSc provides efficient implementations of these fundamental, low-level parallel routines, in a format immediately useful for an application code.

Why Writing Correct Message-Passing Code Is Difficult

Not only is writing message-passing code tedious, it also is technically difficult, since rather subtle issues become extremely important for guaranteeing robust, *correct* libraries. Writing such code requires expert knowledge that most application programmers have neither the time nor interest to master.

We now present a specific example for which a naive but quite reasonable implementation can result in unexplained deadlock, while a more sophisticated implementation will perform efficiently and correctly for all problem sizes and machines. Consider the case of dealing with finite sizes for the internal system buffers used in message passing. If two processors both initiate blocking sends to each other at the same time, the data to be transferred must be moved away from the sending processors' buffers before the sends can complete and return to the user's code. The data is usually placed in an intermediate buffer (the details of which vary among machines) until the receiving processor can accept the data. If the messages exceed the amount of available buffer space, then deadlock can occur. The likelihood of deadlock depends on the past history of the computation and even the order in which packets are switched through the computer. Thus, seemingly random deadlock can occur for different runs of the same code.

In more complicated applications that use blocking sends, it is not always easy to determine whether a possibility of deadlock exists. This situation arises because each processor may be running different portions of the code when the deadlock occurs. Also, locating the cause of the deadlock is often difficult due to its random appearance. There are, of course, techniques to prevent the occurrence of deadlock, including the use of nonblocking communications and explicit library control of buffering, but these require a more sophisticated and deeper understanding of parallel programming than most application programmers have time to master.

Another example illustrates degradation of a program's performance, due to a naive, yet seemingly reasonable, implementation. In this case a collection of processes all simultaneously send data to the processor to the right (the last processor does not perform a send). In Figure 1 this situation is depicted with eight processors. During the first stage, processor 6 actually transfers its message to processor 7 while processors 1 through 5 wait for their right-hand neighbors to post a receive. During the second stage, processor 5 transports its message to processor 6 while processors 1 through 4 continue to wait for their neighbors to post a receive. Thus, the entire communication requires seven stages, while the underlying hardware may have been able to perform the required communication in one or at most two stages. Again, the proper use of nonblocking techniques would alleviate this problem.

Understanding such details in message passing is similar to the necessity of understanding the numerical effects of working with finite-precision calculations when computing. Most of the time one can simply use common sense and generate the correct results, but catastrophic failure can occur if one is not grounded in the fundamentals of numerical analysis. Thus, for both message passing and numerical programming, we can encapsulate the needed functionality in software libraries that allow all users to take advantage of the experts' knowledge.

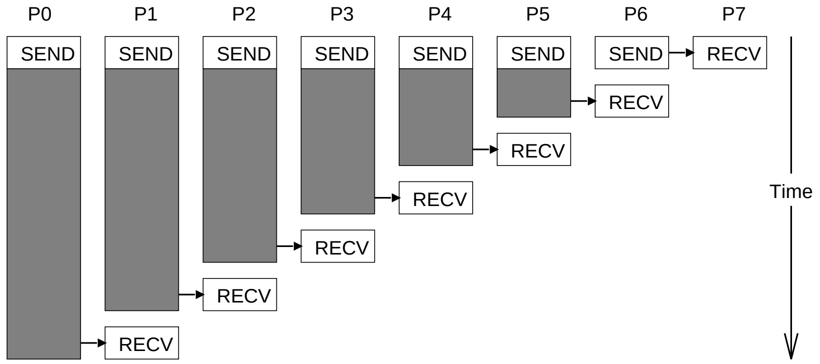

FIGURE 1. Blocking Sends Effect on Performance

We remind the reader that the solution to PDEs at any point is determined mostly by input data that is geometrically near that point. Thus, for many application problems involving PDEs, a geometric decomposition of the solution domain among the processors is most appropriate. This leads immediately to data locality on the computer, an ideal situation for any NUMA parallel machine, including distributed-memory processors programmed with message passing. Since the bulk of the computation involves local data, with careful coding, the computation does not become limited by the need for massive amounts of communication among the processors. So, for the class of problems PETSc is intended, scalable computing is at least theoretically achievable.

3 Distributed Computational Objects

PETSc is built around a variety of data structures and algorithmic objects, some of which are depicted in Figure 2. The application programmer works directly with these objects rather than concentrating on the underlying (rather complicated) data structures.

The three basic abstract data objects in PETSc are index sets, vectors, and matrices. An index set is an abstraction of a list of integer indices, which is used for selecting, gathering, and scattering subsets of elements. A PETSc vector is an abstraction of an array of values (e.g., coefficients for the solution of a PDE), and a matrix represents a linear operator that maps between vector spaces. Each of these abstractions has several representations in PETSc. For example, PETSc currently provides three sequential sparse matrix data formats, four parallel sparse matrix data structures, and a dense representation. Each is appropriate for particular classes of problems. Several data distribution examples for particular PETSc objects are given in Section 5.

Built on top of this foundation are various classes of solver objects, including linear, nonlinear, and timestepping solvers. These solver objects encapsulate virtually all information regarding the solution procedure for a particular class of problems, including the local state and various options. Details are discussed in Section 5.

In general, the data for any PETSc object (vector, matrix, grid, linear solver, etc.) is distributed among several processors. The distribution is handled by an MPI communicator (called `MPI_Comm` in MPI syntax), which represents a group of processors. When a PETSc object is created, for example with the commands

```
VecCreate(MPI_Comm comm,int m,Vec* vector);
MatCreate(MPI_Comm comm,int m,int n,Mat *matrix);
SLESCreate(MPI_Comm comm,SLES *linear_solver);
```

the first argument specifies the communicator, thus indicating which processes share the object. The creation routines are collective over all processors in the communicator; thus, when creating a PETSc object, all processors in the communicator *must* call the creation routine.

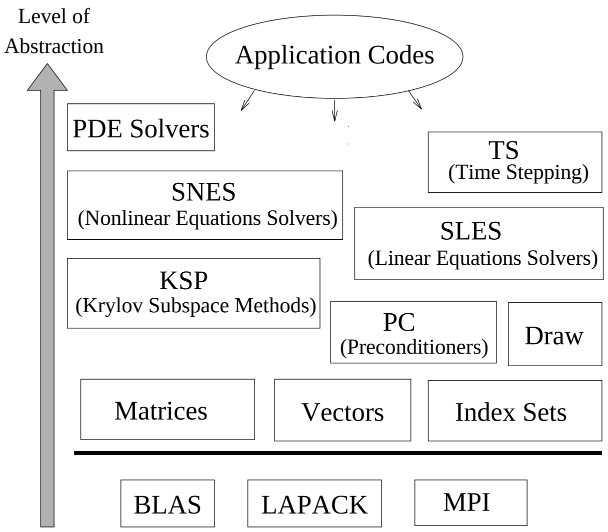

FIGURE 2. Organization of the PETSc Library

The use of communicators in parallel software libraries is extremely important, since it enables all communication for a particular operation (e.g., a matrix-vector product) to be isolated from communication in other parts of code. Such encapsulation eliminates the problem of colliding tags (for example, when two libraries inadvertently use the same tag on different messages, one library may incorrectly receive a message intended for the other library), which was a serious limitation of older message-passing systems.

The underlying communicators in PETSc objects ensure that communications for different computations are separate. We achieve this segregation upon object creation by immediately duplicating via `MPI_Comm_dup()` (an MPI function that makes a copy of a given communicator) any communicator that is not already a "PETSc communicator" and then denoting it as such by inserting an MPI attribute via `MPI_Attr_put()`. An MPI attribute is simply any collection of data a user chooses to attach to a communicator. This PETSc attribute essentially contains a tag number that is assigned to the PETSc object. The tag number is then decremented to ensure that each PETSc object that shares a common communicator has a unique tag (or tags) for use in its internal communication.

4 Six Guiding Principles

As introduced in Section 1, the six guiding principles in the development of the parallel PETSc software are strongly interrelated. This section discusses each principle, while the following section describes their integration into the PETSc design.

4.1 Managing the Communication in the Context of Higher-Level Operations on Parallel Objects

Raw message-passing code is often extremely difficult to understand and debug because, unless the code is very carefully documented, it is often unclear what specific message is associated with a particular operation or data structure in the code. PETSc is designed so that application programmers generally need not worry about writing individual message-passing calls. Instead, they can direct communication as part of higher-level operations on a parallel object or objects. For example, the matrix-vector product interface routine, given by

```
MatMult(Mat A,Vec x,Vec y);    /*  y = A*x  */
```

institutes a collection of MPI calls to manage the underlying communication required for the particular data structures being used (see Section 5.2 for details). Additional examples of encapsulating complicated lower-level communication sequences are the PETSc vector scatters/gathers as well as matrix and vector assembly routines.

The ability to encapsulate all details of communication inside a PETSc object/operation is vital for building a system that is relatively easy to use. Such organization also facilitates the design of higher-level modules, such as linear and nonlinear solvers, which can then focus more clearly on mathematical abstractions rather than being cluttered by excessive communication details.

4.2 Overlapping Communication and Computation

On virtually all modern computers, the data communication required in implementing an algorithm is often more of a limiting factor in its performance than the actual floating-point operations. To use a computer efficiently in numerical computations, it is important to limit data movement and, whenever possible, to perform numerical computation on one set of data while another set of data is in motion.

Within MPI, nonblocking operations support overlapping the communication of certain data with computation on other data. These nonblocking routines initiate communication but may return to the calling routine immediately, before the communication is complete. For example, a nonblocking send for double-precision data of length `count` can be handled with the code fragment

```
MPI_Request request;  int  count, proc, tag;
MPI_Status  status;   void *buffer;
MPI_Comm    comm;
MPI_Isend(buffer,count,MPI_DOUBLE,proc,tag,comm,&request);
... /* Do some other computation, etc. */
MPI_Wait(&request,&status);
```

Here `buffer` is the initial address of the send buffer, `proc` is the rank (number, where the processors are numbered from 0 to `size-1`) of the destination processor, `tag` is the message tag, `comm` is the communicator, and `request` is the communication request. Likewise, a basic nonblocking receive can be handled as follows:

```
MPI_Request request;  int  count, proc, tag;
MPI_Status  status;   void *buffer;
MPI_Comm    comm;
MPI_Irecv(buffer,count,MPI_DOUBLE,proc,tag,comm,&request);
... /* Do some other computation, etc. */
MPI_Wait(&request,&status);
... /* Now use the data in buffer */
```

Clearly, programmers working directly with message-passing routines can themselves institute the overlap of computation and communication. More important, PETSc provides this opportunity within many of the higher-level operations mentioned in the preceding section for encapsulating complicated communication patterns. This makes all the optimizations in the communication transparent to the user.

4.3 Precomputing Communication Patterns

In many aspects of the numerical solution of PDEs (for example, iterative solution of linear systems and explicit timestepping), the same communication is repeated many times on essentially the same data structures. If, each time the communication had to occur, one redetermined what sends and receives had to be instituted (and just this determination requires its own communication), this process would be very inefficient. Fortunately, it is possible to precompute exactly what messages need to be sent and received and the amount of data that will be transferred. In fact, MPI even explicitly supports precomputing through *persistent* send and receive objects.

If one is aware that a particular receive is to occur multiple times, in MPI one can initialize the set of receives by calling

```
MPI_Request request;
MPI_Recv_init(buffer,count,MPI_DOUBLE,proc,tag,comm,&request);
```

In this case, we are receiving double-precision data of length `count`; `buffer` is the initial address of the receive buffer; `proc` is the rank of the receive processor; `tag` is the message tag; `comm` is the communicator; and `request` is the communication request. Then every time the receive is required, one can simply call

```
MPI_Start(&request);
/* Do other computations */
MPI_Wait(&request,&status);
/* Use the data in the buffer */
```

There is analogous syntax for repeated sends. In addition, MPI directly supports the use of persistent communication on a series of related messages.

As discussed in Section 2.2, it is often not known *a priori* by a receiver who is sending it data. Thus a set-up phase must be performed to even know what communication needs to take place. For similar, repeated communications, a clean mechanism is required to manage this set-up phase, so the same set-up need not be repeated. This is discussed in detail for vector scatters in Section 5.1.

It is natural to encapsulate information for potentially repeated communication patterns in the objects that define higher-level operations. As will be demonstrated in the following sections, we do exactly this for operations such as matrix-vector products and vector scatters/gathers.

4.4 Programmer Management of Communication

Certain parallel programming paradigms (for example, HPF) attempt to conceal completely from the user knowledge of when communication takes place. Although this approach to make parallel programming "easier" is very appealing, it can have serious effects on both efficiency and the user's knowledge of what is actually occurring in an application code. In PETSc, the user can explicitly initiate communication by calling specific PETSc routines. For example, to perform a parallel vector scatter/gather (discussed in Section 5.1), the user calls the routines

```
VecScatterBegin(Vec x,Vec y,InsertMode im,ScatterMode sm,
                VecScatter scattercontext);
VecScatterEnd(Vec x,Vec y,InsertMode im,ScatterMode sm,
                VecScatter scattercontext);
```

Thus, within an application code the user can dictate the time that communication takes place with respect to computations. In addition, the user often can arrange the overlap of communication and computation by placing code unrelated to the message passing between calls to multiphased routines such as the vector scatters/gathers. This situation is analogous to the use of prefetching for hierarchical memory.

4.5 Working Efficiently with Parallel Objects without Regard for Details of Data Location

When assembling a distributed object that requires a large amount of user-provided data (for example, a vector or matrix), one needs an efficient mechanism for transferring data from the application code to its correct location in the library data structures. This mechanism must be easy to use; otherwise, the application programmer will be tempted simply to bypass the library and assemble the data structures manually.

Within PETSc we allow the user to insert data into global PETSc objects without regard for the particular processor on which the data is destined ultimately to be stored. To require the user to generate all data on the "owner" processor is simply too much of a burden within most applications and is, in fact, unnecessary. Rather, PETSc internally retains (called stashing in PETSc) those values destined for another processor until an explicit communication phase is initiated by the user (an example of the concept of the application programmer determining when communication occurs, as discussed above). This topic is discussed in detail for vectors in Section 5.1 and for matrices in Section 5.2. This same stashing technique is commonly used and, in fact, is vital for obtaining good performance in I/O systems, where it is referred to as write caching.

4.6 Aggregation of Data for Communication

The overhead of initiating message transition is quite high relative to the cost of sending a segment of data. For efficient parallel programming, whenever possible one should pack relatively large amounts of data for a single subsequent aggregate transition instead of sending several smaller messages. PETSc incorporates this strategy in various communication phases throughout the library, for example in the transfer of user-supplied nonlocal data within parallel matrix and vector assembly routines. What is important in the PETSc design is that it allows this type of optimization without requiring an effort from the application programmer.

5 PETSc Design of Fundamental Objects

In this section we discuss several of the fundamental PETSc objects and their implementations in the context of the six guiding principles introduced in Section 4. We briefly summarize this information, which is given in full detail below.

- Vectors are perhaps the simplest PETSc objects and are used, for example, to store the coefficients of the solutions of PDEs. The main parallel operations that we perform on vectors are
  - vector assembly, which (1) allows the user to insert vector entries without regard to their processor ownership, and (2) aggregates the values until the user directs that communication can begin; and
  - vector scatters/gathers, which (1) allow overlap of the communication and computation, (2) reuse precomputed communication patterns, and (3) conceal all communication within the context of the higher-level abstraction of a scatter/gather.
- Matrices, another class of fundamental PETSc data objects, represent linear operators. Often, matrices are obtained as Jacobians of an underlying nonlinear algebraic system arising from the discretization of a PDE. We discuss
  - matrix assembly, which is similar to the vector assembly mentioned above; and
  - sparse matrix-vector products, which demonstrate (1) the overlap of communication and computation by dividing the calculation into sections that are purely local and that require off-processor data, (2) precomputation of a repeated communication pattern, and (3) management of communication within the context of a higher-level operation (the matrix-vector product).

- We discuss the efficient parallel computation of numerical Jacobians using coloring and finite differences. This topic demonstrates (1) the use of aggregation to reduce the amount of distinct communications required, (2) precomputation of communication patterns (including the coloring), and (3) the encapsulation of the entire process within an abstract PETSc operation.
- Finally, we discuss the linear and nonlinear solvers. In this section we demonstrate how all communication required during the solution process is managed directly in the context of a higher-level "solver" object.

The remainder of this section discusses these operations in detail.

5.1 Vectors

In the numerical solution of PDEs, a vector may often be thought of as a set of coefficients representing a function on a grid. A vector may also be thought of as a set of elements of R^N, the usual Euclidean vector space. It is, however, too limiting to think of a vector as merely a one-dimensional array of floating-point numbers in memory, since the components of a vector in a parallel machine will generally not be stored in contiguous memory locations.

In PETSc one can create a parallel vector with the command

```
VecCreateMPI(MPI_Comm comm,int nlocal,int nglobal,Vec *vector);
```

As mentioned in Section 3, all processors in the communicator `comm` must call this routine, because vector creation is an aggregate operation. The arguments `nglobal` and `nlocal` indicate, respectively, the total size of the vector and the number of elements to be represented locally on a particular processor. Either one, but not both, of the arguments `nglobal` and `nlocal` may be set to `PETSC_DECIDE` to allow PETSc to determine the value. Upon this call, PETSc allocates memory to store the vector entries and sets up any data structures required for manipulating the vector.

Vector Assembly

Although the components of PETSc vectors are ultimately distributed among the various processors, it is highly desirable to allow users to assemble the vector components easily, without regard to the elements' final storage locations. That is, we would like to separate completely the vector data storage formats from the application codes. This capability simplifies the application codes and allows library writers to provide highly tuned data structures without imposing a burden on users. This facet of design is an example of data encapsulation, which is a very basic requirement for flexible libraries.

To illustrate the parallel vector assembly process, we consider three cases of constructing the right-hand side of a linear system: by the finite difference method, by the finite element method, and by a special case for a grid-based nonlinear system arising from a finite difference discretization of a nonlinear PDE. In all cases, the resulting parallel vector is distributed among the processors, so that each vector component is owned by exactly one processor.

Case 1: Simple finite difference discretization in one dimension. The user creates a parallel vector and partitions it among the processors by calling `VecCreateMPI()`, as discussed above. To evaluate the vector entries, one can use a routine such as the following:

```
Vec    F;                    /* global vector */
int    istart, iend, i, N;
double f;
  ...
VecCreateMPI(comm,PETSC_DECIDE,N,&F);
VecGetOwnershipRange(F,&istart,&iend);
/* Loop over local entries, inserting vector elements */
for ( i=istart; i<iend; i++ ) {
```

```
    /* Assign f to be some function of the grid node */
    VecSetValues(F,1,&i,&f,INSERTVALUES);
  }
  /* Complete the vector assembly process */
  VecAssemblyBegin(F);
  VecAssemblyEnd(F);
```

In this simple case each processor generates contributions only for its local part of the vector, `F`, and inserts these values one at a time with `VecSetValues()`; no parallel communication is required. In this example, the routines `VecAssemblyBegin()` and `VecAssemblyEnd()` do essentially nothing except verify that the user has completed inserting entries into the vector on all processors. However, in most serious applications the situation is not this simple, as demonstrated by Case 2.

Case 2: Finite element discretization of the right-hand side of a PDE. The vector is calculated by the integrals

$$F_i = \int_{\mathrm{supp}(\phi)} f(x)\phi(x)dx,$$

where x can be one-, two-, or three-dimensional, and $\mathrm{supp}(\phi)$ denotes the support of ϕ. In most finite element codes these integrals are calculated numerically, element by element. Thus, certain vector components, F_i, receive contributions from more than one processor.

We demonstrate this situation with an example in two dimensions using piecewise linear finite elements. Assume that we have generated a parallel vector, `F`, and in addition have distributed the finite elements (in this case triangles) among the processors. That is, each processor has a list of its local elements (triangles) and, for simplicity, the locations of the element nodes. For demonstration purposes the data structures representing the grid in the following code fragment are very simple and are not necessarily the best choices for an actual code.

```
  Vec    F;         /* global vector */
  int    gidx[3];  /* work array of vector indices */
  double f[3];      /* work array of vector contributions */
  int    Nt, *tri, i;
  double *X, *Y, x0, y0, x1, y1, x2, y2;
  /* Loop over the local elements */
  for ( i=0; i<Nt; i++ ) {
    /* Load the vertices of the element into local variables */
    x0 = X[3*i];   y0 = Y[3*i];
    x1 = X[3*i+1]; y1 = Y[3*i+1];
    x2 = X[3*i+2]; y2 = Y[3*i+2];
    /* Compute the local element vector in a work array, f,
       with an application-specific routine */
    CalculateElementIntegral(n,x0,x1,x2,y0,y1,y2,f);
    /* Load global indices for the nodes of the element into gidx */
    gidx[0] = tri[3*i]; gidx[1] = tri[3*i+1]; gidx[2] = tri[3*i+1];
    /* Add the local vector contribution, f, into the global vector,
       F, in the rows indicated by gidx */
    VecSetValues(F,3,gidx,f,ADD_VALUES);
  }
  /* Complete the vector assembly process */
  VecAssemblyBegin(F);
  VecAssemblyEnd(F);
```

Here we insert three elements into the parallel vector, `F`, with each call to `VecSetValues()`. Generally, it is desirable to set multiple values simultaneously; otherwise, the overhead for function calls increases. This is an example of aggregation. In most true applications, aggregating the insertions is natural and easy.

In this example, as for most problems discretized with parallel finite elements, some contributions to the vector components are generated by a different processor from their owner, because each finite element is assigned a unique processor and each vector element similarly is assigned a unique processor. Note that these contributions are added to the appropriate global vector component, as indicated by the flag `ADD_VALUES` within `VecSetValues()`. The vector assembly routines perform the required message passing to move the values to their correct locations. As discussed below, separating the assembly process into two stages, `VecAssemblyBegin()` and `VecAssemblyEnd()`, is critical for allowing application programmers to overlap communication and computation. Although the procedure was not done in this example, a careful user will attempt to generate the majority of the vector entries on the correct processor. (Note that there are tricks for reducing the cost of integral evaluation that also may exploit locality.) This example indicates that trying to generate all vector entries on the correct processor can lead to a unnecessarily messy application code, because certain finite elements would have to be dealt with several times on multiple processors.

Application programmers can, in fact, manage such message passing themselves. However, this task should be done only by experienced parallel programmers who wish to develop a custom implementation that optimizes performance for a particular problem. Since most users have neither the time nor the experience to finely tune their assembly routines, their special-purpose codes generally would not perform as efficiently as well-written library routines.

Since the application code is free to insert components into a vector from any processor, regardless of their ultimate storage location, there must be a mechanism to store the values locally until they are transferred to the processor that owns them. In PETSc, these values are stored in a stash until they are transferred to the correct processor. The stash can be implemented in many ways; the one tricky point is correctly handling values that are inserted multiple times for the same component. Our current implementation is fairly naive; we simply retain an array of values and their indices. A simple optimization might be to sort the list by indices; more sophisticated implementations might use a hash table.

The final movement of the stashed values to their proper home processor is done in several stages:

- By making calls to `MPI_Allreduce()`, each processor determines the maximum size delivery to expect, as well as a count of the number of deliveries.
- Each processor allocates enough buffers to accommodate the arrival of all stashed values and posts corresponding nonblocking receives.
- Each processor posts nonblocking sends of all its elements to be distributed.
- Each processor waits for its receives and inserts the received values into the correct locations as indicated by the received indices.
- Finally, all processors wait for their nonblocking sends.

The first three steps are done during the `VecAssemblyBegin()` stage and the last two during the `VecAssemblyEnd()` phase.

The process of setting values into a vector demonstrates several of our six guiding principles:

- Aggregation:
  - The user aggregates several vector values into a single call to `VecSetValues()`.
  - The "off-processor" values are aggregated in the stash before being moved *en masse* to the correct processor.
- Values are stashed locally until the user explicitly initiates communication by calling `VecAssemblyBegin()` and `VecAssemblyEnd()`.
- The communication is managed within a vector object, not as a series of seemingly unrelated calls to `MPI_Send()` and `MPI_Recv()`.

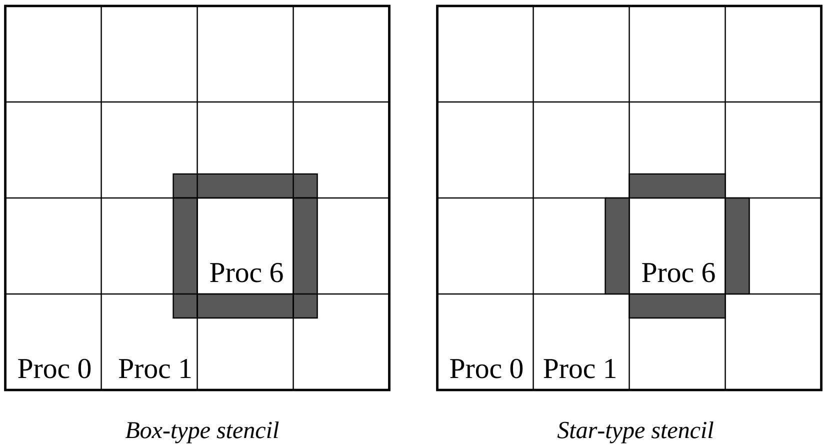

FIGURE 3. Ghost Points for Two Stencil Types on Processor Six

- Communication and computation can be overlapped between the two assembly calls.
- The user can insert values into the vector without specific regard for the processor on which the data will ultimately reside. But through the use of `VecGetOwnershipRange()`, the user can ensure that most vector entries are generated on the "correct" processor, thus allowing efficiency without requiring complicated application code including application-specific message passing.

Distributed Arrays: Managing Ghost Points

Next, we consider a more complicated scenario, requiring vector communication before local function evaluation can occur. In this example, we introduce another PETSc construct, the distributed array (called `DA` in PETSc). These data structures manage the communication required for *ghost points*, which are the bordering portions of the vector that are owned by neighboring processors. Figure 3 illustrates the ghost points for processor six of a two-dimensional, regular, parallel grid. Each box represents a processor; the ghost points for processor six's local part of a parallel array are shown in gray.

Two types of distributed arrays can be created: one based on star-type stencils and one based on box stencils. In both cases the ghost points are identical, the only difference being that with star-type stencils certain ghost points are ignored, potentially decreasing substantially the number of messages sent. In two dimensions a `DA` object of stencil type `st` and periodicity type `pt` can be created with the command

```
DACreate2d(MPI_Comm comm,DAPeriodicType pt,DAStencilType st,
           int M,int N,int m,int n,int w,int s,DA *da);
```

Here `w` indicates the number of degrees of freedom per node, and `s` is the stencil width. The arguments `M` and `N` are the global dimensions of the array, while `m` and `n` indicate the partition of the array among the processors. One should either use `PETSC_DECIDE` for `m` and `n` or ensure that `m*n` equals the number of processors in the communicator. We employ two-dimensional distributed arrays in the following example.

Case 3: Consider the classic Bratu problem [ACM91],

$$-\triangle u - \lambda e^u = 0, \qquad 0 < x, y < 1,$$

with boundary conditions

$$u = 0 \text{ for } x = 0, x = 1, y = 0, y = 1,$$

which is discretized by finite differences on a uniform m by m grid. The resulting nonlinear algebraic equation for an interior node is given by

$$4u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} - h^2 \lambda e^{u_{i,j}} = 0,$$

where $h = 1/(m-1)$ is the grid size. The PETSc code for evaluating this function (`F(X)`) in parallel, for use in a Newton-based method within SNES (the nonlinear solvers component), is given by the following routine:

```
/* SNES - nonlinear solver context */
/* X    - input vector */
/* F    - output vector (nonlinear function) */
/* user - user-defined context with application-specific data */
int EvaluateFunction(SNES snes,Vec X,Vec F,ApplicationCtx user)
{
  DA      da = user->da;         /* distributed array */
  Vec     localF = user->localF; /* local work vector */
  Vec     localX = user->localX; /* local work vector */
  int     i,j,row,xs,ys,xm,ym,Xs,Ys,Xm,Ym,m = user->m;
  double  h,hh,ut,ub,ul,ur,u,uxx,uyy,*x,*f,lambda = user->param;
  h       = 1.0 / (double)(m-1);
  hh      = h*h;
  /* Transfer local portion (including ghost points) of global
     vector, X, into a local work vector, localX */
  DAGlobalToLocalBegin(da,X,INSERTVALUES,localX);
  DAGlobalToLocalEnd(da,X,INSERTVALUES,localX);
  /* Get the indices of the lower left corner of the region
     owned by this processor; get the corresponding indices
     for the region including ghost points. */
  DAGetCorners(da,&xs,&ys,0,&xm,&ym,0);
  DAGetGhostCorners(da,&Xs,&Ys,0,&Xm,&Ym,0);
  /* Directly access arrays (x and f) containing vector data */
  VecGetArray(localX,&x);
  VecGetArray(localF,&f);
  /* Loop over local region owned by processor, performing
     application operations */
  for (j=ys; j<ys+ym; j++) {
    for (i=xs; i<xs+xm; i++) {
      row = i - Xs + (j - Ys)*Xm;
      if (i == 0 || j == 0 || i == m-1 || j == m-1 ) {
        f[row] = x[row]; continue;
      }
      u = x[row]; ub = x[row - Xm]; ul = x[row - 1];
      ut = x[row + Xm]; ur = x[row + 1];
      uxx = (-ur + 2.0*u - ul); uyy = (-ut + 2.0*u - ub);
      f[row] = uxx + uyy - hh*lambda*exp(u);
    }
  }
  /* Place values in global vector */
  DALocalToGlobal(da,localF,INSERTVALUES,F);
  return 0;
}
```

The application portion of the routine (within the double `for` loop) appears exactly like a sequential code, except that the loop indices range over only the local portion of the distributed array. The indices of the lower left corner of the local portion of the array as well as the local array size are obtained with the commands `DAGetCorners()` and `DAGetGhostCorners()`. The first version excludes any ghost points, while the second version includes them and deals with the fact that subarrays along boundaries of the problem domain have ghost points only on their interior edges, but not on their boundary edges.

To evaluate its local function, each processor requires its portion of the vector **x** as well as its ghost points. The calls to `DAGlobalToLocalBegin()` and `DAGlobalToLocalEnd()` handle all of the detailed message passing to retrieve the needed ghost points from neighboring processors.

The distributed arrays in PETSc demonstrate several of our guiding design principles:

- Overlapping communication and computation.
- Precomputing repeated communication patterns.
- Managing the communication in the context of a PETSc object, in this case the `DA`.
- Allowing the user full control of the time that communication occurs, via calls to `DAGlobalToLocalBegin()` and `DAGlobalToLocalEnd()`.
- Aggregation, in that from the user's point of view a single communication of all the ghost values occurs, while internally a variety of message-passing activities (which have been organized for efficiency) take place.

Index Sets and Vector Scatters

The previous example illustrates a very special type of vector scatter and gather. PETSc also includes general-purpose scatters and gathers, which we illustrate below with a contrived example.

To facilitate vector scatters and gathers, PETSc employs the concept of an *index set.* An index set, which is a generalization of a set of integer indices, is used to define various operations on vectors and matrices. The following command creates a sequential index set, `is`, based on a list of integers of length `n` in the array `indices`:

```
ISCreateGeneral(MPI_Comm comm,int n,int *indices,IS *is);
```

Another standard index set, which is defined by a starting point (`first`), a stride (`step`), and a length (`n`), can be created with the command

```
ISCreateStride(MPI_Comm comm,int n,int first,int step,IS *is);
```

We use strided index sets in the following vector scatter/gather example. In this case, one vector is half the length of the other; we wish to gather every second element of the longer vector, `x_from`, into the shorter, `x_to`.

```
VecScatter ctx;
IS         is_from, is_to;
ISCreateStride(MPI_Comm comm,n/2,0,1,&is_to);
ISCreateStride(MPI_Comm comm,n/2,0,2,&is_from);
VecScatterCreate(x_from,is_from,x_to,is_to,&ctx);
ISDestroy(is_to); ISDestroy(is_from);
VecScatterBegin(x_from,x_to,INSERT_VALUES,SCATTER_ALL,ctx);
  /* Do other useful work */
VecScatterEnd(x_from,x_to,INSERT_VALUES,SCATTER_ALL,ctx);
```

The scatter context can be saved and reused whenever the same scatter is required.

As discussed below, the vector scatters are performed in several stages to enable reusing communication patterns and overlapping communication and calculation. In the first scattering stage, which is initiated by calling `VecScatterCreate()`, the destination and source of each component are determined. First, each processor decides the number of components it needs from the other processors; it then calls `MPI_Allreduce()` with these quantities to determine the number of requests to expect and the maximum length of any request. Next, each processor posts a suitable number of nonblocking receives with suitable buffer sizes, followed by posts of all of its requests with nonblocking sends. Each processor then waits for its requests. As the requests arrive, the processor creates the `VecScatter` data structure that contains information regarding all the sends it must perform during the scatter. Once this has been completed, the processor creates an analogous data structure with information about all of its expected receives and the storage location of the values. Finally, the processor waits for the posted nonblocking sends. Note that the construction of the `VecScatter` data structure is modeled on the inspector/executor ideas in PARTI/Chaos [ASS93].

We have decided to hide the creation of the `VecScatter` data structure in a single stage. To allow even more overlap of application computation and communication with the nonblocking sends and receives, one could break the context creation into two or three stages. In the interest of simplicity we have not done so, believing that it is an overoptimization, since the scatter context is created only once and then used repeatedly.

The actual vector scattering begins with the `VecScatterBegin()` stage, when the processor posts all of its nonblocking receives, followed by posting its nonblocking sends. The `VecScatter` contains the required MPI communicator, all information regarding what must be posted, and locations to store the `MPI_Request` data structure.

The `VecScatterEnd()` phase waits for the receives, inserts the arriving data into the correct vector locations, and waits for the posted sends. In our implementation, we do not use the MPI scatter construct to do the scattering. Instead, we manually gather the data into a buffer (hidden in `VecScatter`) and send the buffer with a persistent nonblocking send. Similarly, we receive the data into a buffer and then scatter the buffer into the actual vector. We wrote the code this way to support a scatter-add operation, which MPI does not directly support.

This implementation facilitates the overlap of calculation and communication by allowing the application code to perform work unrelated to the scatters while the messages are in transition (that is, between the calls to `VecScatterBegin()` and `VecScatterEnd()`). However, we ignore several additional possibilities for overlap. For instance, the final wait-on-sends could constitute a separate stage at the scatter's conclusion. Such modifications could provide additional efficiency, though at the cost of a steeper learning curve and more complicated application and library codes. We have tried to strike a healthy balance between the ability to achieve the important overlap of communication and calculation with little added complexity.

An observant reader may have noticed that since we keep MPI request objects as well as temporary buffers in the `VecScatter`, it may be used for only a single scatter at a time. To prevent programming errors resulting from accidental concurrent reuse, the scattering context contains a flag that indicates whether it is currently involved in a scatter.

The vector scatter construction in PETSc demonstrates several of our six guiding principles:

- Overlapping communication and computation.
- Precomputing repeated communication patterns during the routine `VecScatterCreate()`, which are then reused for several scatters.
- Managing the communication in the context of a PETSc object, in this case the `VecScatter`.
- Allowing the user complete control of the time that communication takes place, via calls to `VecScatterBegin()` and `VecScatterEnd()`.

In a later section we discuss the use of the basic vector scatter to implement some of our parallel sparse matrix-vector products.

5.2 Matrices

PETSc provides a variety of matrix data structures and many of the matrix operations required for the solution of PDEs. In this section we focus on two specific facets of our matrix implementation (matrix assembly and matrix-vector products) that reflect our guiding design principles.

Matrix Assembly

Matrix assembly is organized similarly to vector assembly, as demonstrated by the following routine for forming a finite element stiffness matrix. The call to `MatSetValues()` adds a 3×3 block to the stiffness matrix. Again, as for the vector assembly, it is more efficient to insert elements by blocks, rather than individually.

```
Mat    K; /* Global stiffness matrix */
int    Nt, *tri, i, gidx[3];
double *X, *Y, x0, y0, x1, y1, x2, y2, k[9];
/* Loop over local elements */
for ( i=0; i<Nt; i++ ) {
  /* Load the vertices of the element into local variables */
  x0 = X[3*i];   y0 = Y[3*i];
  x1 = X[3*i+1]; y1 = Y[3*i+1];
  x2 = X[3*i+2]; y2 = Y[3*i+2];
  /* Compute the element stiffness matrix with an application-
     specific routine that stores it in k */
  CalculateElementStiffness(x0,x1,x2,y0,y1,y2,k);
  /* Load global indices for nodes of the element into gidx */
  gidx[0] = tri[3*i]; gidx[1] = tri[3*i+1]; gidx[2] = tri[3*i+1];
  /* Add the local stiffness matrix, k, to the global stiffness
     matrix, K, in the rows and columns indicated by gidx */
  MatSetValues(K,3,gidx,3,gidx,k,ADD_VALUES);
}
MatAssemblyBegin(F,FLUSH_ASSEMBLY);
MatAssemblyEnd(F,FLUSH_ASSEMBLY);
```

Certain internal data structures (for instance, data structures needed for a matrix-vector product) must be created once during sparse matrix assembly, after all of the matrix elements have been inserted. The second argument of `MatAssemblyBegin()` and `MatAssemblyEnd()` indicates whether the assembly is complete, `FINAL_ASSEMBLY`, or only partial (so that values will be added/changed later), `FLUSH_ASSEMBLY`. Only after calling the `FINAL_ASSEMBLY` variant of the assembly routines is the parallel matrix ready for use. The `FLUSH_ASSEMBLY` is to allow the local stashes to be unloaded to the correct processor, without setting up the rest of the data structures required to make the matrix ready for use.

PETSc provides a variety of parallel matrix data structures that are distributed by rows, including (block) compressed sparse row, (block) diagonal, and dense formats. The matrix assembly process for all of these implementations is analogous to that for vectors, discussed in Section 5.1. Specifically, stashing is employed to enable the application code to insert any components into a matrix, regardless of their ultimate storage location. The assembly phase automatically handles the movement of the elements into their final locations. The matrix assembly design reflects the same guiding principles as the vector assembly.

Matrix-Vector Products

One of the most computationally expensive operations within iterative methods for sparse systems of linear equations is the matrix-vector product. We next describe the use of the vector scatters introduced above to compute the matrix-vector product for one of our simple matrix implementations. This section continues the discussion at the end of Section 2.2.

The default sparse matrix representation supported in PETSc is the compressed row storage format (also called the Yale sparse matrix format or AIJ storage). This format employs three arrays: the matrix nonzeros (ordered by rows), their respective column indices, and pointers to the beginning of each row. The internal data representation is compatible with standard Fortran 77 storage. In PETSc these three arrays, as well as other useful information, are encapsulated in the internal part of the `Mat` datatype.

In our implementation, parallel AIJ matrices are distributed by rows among the processors, so that each processor owns a submatrix consisting of all of the nonzeros within its assigned rows. Each processor's local matrix is divided into diagonal and off-diagonal parts. For a square global matrix we define each processor's diagonal portion to be its local (owned) rows and the corresponding columns (a square submatrix); each processor's off-diagonal portion encompasses the remainder of the local matrix (a rectangular submatrix). As described in the following paragraph, this subdivision enables the overlap of computation and communication during the matrix-vector products. We indicate below the pertinent parts of the parallel AIJ data structure for our discussion:

```
typedef struct {
Mat        A;     /* diagonal submatrix */
Mat        B;     /* off-diagonal submatrix */
Vec        lvec;  /* local vector to receive scattered elements */
VecScatter Mvctx; /* scatter context for matrix-vector product */
....              /* additional data */
} Mat_MPIAIJ;
```

After all elements have been inserted into a parallel matrix, we create the data structures needed for the matrix-vector product within the `MatEndAssembly()` phase. These data structures are generally formed a single time for a matrix; however, they must be recomputed when the nonzero structure of the matrix changes.

Each processor first compresses its off-diagonal submatrix, renumbering the column numbers from one through nc, where nc denotes the number of nonzero columns in the entire off-diagonal submatrix. Each processor also creates a mapping from the compressed to the actual column numbers (needed for such actions as printing a matrix) and forms a local vector of dimension nc that will be used to receive scatters of nonlocal vector elements during the matrix-vector product. Then each processor forms a vector scatter context with `VecScatterCreate()`, so that *only* the vector elements needed by the off-diagonal part of the local matrix will be scattered to that processor during a matrix-vector product.

After this setup phase, the matrix-vector product computations are then extremely simple, as indicated by the routine given below for computing $y = A * x$:

```
int MatMult_MPIAIJ(Mat A,Vec x,Vec y)
{
Mat_MPIAIJ *aij = (Mat_MPIAIJ *) A->data;
VecScatterBegin(x,aij->lvec,INSERT_VALUES,SCATTER_ALL,aij->Mvctx);
MatMult(aij->A,x,y);
VecScatterEnd(x,aij->lvec,INSERT_VALUES,SCATTER_ALL,aij->Mvctx);
MatMultAdd(aij->B,aij->lvec,y,y);
return 0;
}
```

We initiate scattering with `VecScatterBegin()` and compute the local part of the matrix-vector product, thus overlapping calculation and communication. We then complete the scattering with `VecScatterEnd()`. Finally, we perform the off-diagonal part of the matrix-vector product.

The matrix-vector product exemplifies several of the guiding principles:

- Overlapping communication and computation.
- Managing the computation in the context of a PETSc object, in this case a matrix.
- Precomputing communication patterns inside the vector scatter.

5.3 Jacobian Computation via Finite Differences

PETSc provides a variety of scalable nonlinear solvers. At the heart of these is a need to provide (approximate) Jacobian information. In the context of finite element and finite difference methods, these Jacobians are usually extremely large sparse matrices that for realistic applications are difficult to compute analytically. PETSc provides the infrastructure to compute Jacobians via finite differences efficiently and scalably. Since this is an important example of the use of our six guiding principles, we discuss it in detail.

The Jacobian, J, of a function $F(x)$ is a matrix whose columns are the derivatives of F with respect to each of the components of x. Thus,

$$\text{column}_j(J) = \nabla_{x_j} F \approx \frac{F(x+dx_j) - F(x)}{dx_j},$$

where dx_j represents a small perturbation to the jth component of x. Once $F(x)$ has been evaluated, one can compute each column of the Jacobian at the cost of one function evaluation. In addition, one can determine the nonzero pattern of each column by locating the nonzeros generated during this computational process.

Unfortunately, this approach is completely nonscalable because, if the Jacobian is n by n, this method requires n function evaluations and n^2 checks to locate the nonzeros in the Jacobian. Fortunately, for the class of problems that PETSc tackles, both of these problems can be eliminated.

Determining the nonzero pattern: For PDEs discretized on finite element or finite difference grids, it is generally straightforward to determine the nonzero structure of the Jacobian matrices. The nonzero structure arises from the underlying coupling between nodes used on the grid (which corresponds to the stencil for finite difference methods), and in general this is determined by cell or node neighbors. Most important, it is determined locally and thus is easily parallelized.

Efficient computation of the Jacobian entries: Consider the function

$$F(x_0, x_1, x_2, x_3, x_4) = \begin{pmatrix} x_0 + x_1 * x_0 \\ x_1 + x_4^2 \\ 3 * x_2 + x_0 \\ x_3 + x_0 \\ x_4 + 2 * x_1 \end{pmatrix}.$$

Its Jacobian is easily computed to be

$$\begin{pmatrix} 1 + x_1 & x_0 & & & \\ & 1 & & & 2x_4 \\ 1 & & 3 & & \\ 1 & & & 1 & \\ & 2 & & & 1 \end{pmatrix}.$$

Note that columns 1, 2, and 3 share no common rows. Consequently, one could compute all three columns with a single function call

$$\begin{gathered} \text{column}_1(J) + \text{column}_2(J) + \text{column}_3(J) \\ \approx \\ \frac{F(x+dx_1)-F(x)}{dx_1} + \frac{F(x+dx_2)-F(x)}{dx_2} + \frac{F(x+dx_3)-F(x)}{dx_3} \\ = \end{gathered}$$

$$\begin{pmatrix} \frac{F_0(x+dx_1)-F_0(x)}{dx_1} \\ \frac{F_1(x+dx_1)-F_1(x)}{dx_1} \\ \frac{F_2(x+dx_2)-F_2(x)}{dx_2} \\ \frac{F_3(x+dx_3)-F_3(x)}{dx_3} \\ \frac{F_4(x+dx_1)-F_4(x)}{dx_1} \end{pmatrix}$$
$$\approx$$
$$\begin{pmatrix} \frac{F_0(x+dx_1+dx_2+dx_3)-F_0(x)}{dx_1} \\ \frac{F_1(x+dx_1+dx_2+dx_3)-F_1(x)}{dx_1} \\ \frac{F_2(x+dx_1+dx_2+dx_3)-F_2(x)}{dx_2} \\ \frac{F_3(x+dx_1+dx_2+dx_3)-F_3(x)}{dx_3} \\ \frac{F_4(x+dx_1+dx_2+dx_3)-F_4(x)}{dx_1} \end{pmatrix}.$$

Similarly, columns 0 and 4 may be computed simultaneously. From this example, one sees a very general mechanism for computing sparse Jacobians efficiently.

First, one *colors* the columns of the sparse matrix so that no two columns of the same color share a common row. Then, the number of function evaluations required to approximate the Jacobian drops from n to the number of colors. In most applications the number of colors required can vary between 4 and 50.

The use of coloring to compute Jacobians efficiently has been known for many years and is widely used on sequential machines, for example in the library MINPACK [MSGH84]. We discuss briefly how efficient Jacobian computation is performed in PETSc.

- Compute the nonzero structure of the Jacobian; as indicated above, this is essentially a local computation.
- Generate a coloring of the resulting matrix. This may be done in one of two ways:
  - color the underlying grid directly (this again is essentially a local operation), or
  - use a parallel graph coloring algorithm (for example, [JP93]), to color the parallel sparse matrix directly.
- Determine all the local computation and the communication that will be required for each color in the actual Jacobian approximation. This is an inspector step, and in PETSc it generates a data structure called `MatFDColoring`. Note that the actual Jacobian calculations will (generally) require communication of the scaling factors dx_j, since a column change on one processor will (in general) affect rows on other processors (though usually only a few).

A code fragment that demonstrates this process is given below.

```
ISColoring    iscoloring;
MatFDColoring fdcoloring;
Mat           J;

DAGetColoring2dBox(da,&iscoloring,J);
MatFDColoringCreate(J,iscoloring,&fdcoloring);
```

The PETSc data structure `ISColoring` contains the information about the column coloring, and the routine `DAGetColoring2dBox()` is a PETSc utility routine that colors two-dimensional grids using a nine-point stencil.

The routine `SNESDefaultComputeJacobianWithColoring()` that actually computes the approximate Jacobian can be expressed in pseudo-code as follows:

```
Loop over all colors {
  For each local column, j, of that color {
    Compute an appropriate dx[j] perturbation and add it to x
  }
  Evaluate the function at the new x
  Communicate the required scaling factors dx[j]
  For each local column of that color {
    Loop over known nonzero rows {
      Scale Jacobian entry by appropriate dx[j]; insert in matrix
    }
  }
}
```

The data structure `MatFDColoring` contains all the precomputed information required in implementing this pseudo-code.

The efficient computation of Jacobians via coloring as implemented in PETSc exemplifies several of our guiding principles:

- Managing the computation in the context of a PETSc object, in this case a `MatFDColoring` object.
- Precomputing communication and computation patterns, required for efficiently determining nonzero rows and scalings.
- Aggregating the computation, by updating all columns of the same color simultaneously.

5.4 Linear and Nonlinear Solvers

We next provide an overview of the techniques used in PETSc for coordinating various types of algorithms (e.g., linear, nonlinear, and timestepping solvers) by focusing on the the simplest case—linear solvers. We build these solvers on the foundation of the previously discussed matrix and vector data structures.

Most modern linear iterative solvers may be viewed as the combination of a preconditioner (simple stationary iterative solver) and a Krylov subspace accelerator. (Even multigrid methods may be categorized in this manner if we view the Richardson iteration as playing the role of a Krylov subspace method.) Since its earliest days PETSc has provided uniform and simple access to a variety of parallel Krylov subspace accelerators. These methods are relatively easy to code (since they merely require a matrix-vector product and several vector operations), and the data-structure-neutral implementations within PETSc are independent of underlying matrix and vector data structures [SG96]. We emphasize that this data-structure-neutral approach exposes mathematical details in a uniform fashion, without making unnecessary assumptions about the representation of mathematical objects. In particular, the parallelism within the Krylov subspace methods is handled completely within the vector and matrix modules of PETSc. Such organization is crucial in managing the complexity of the parallel software.

In contrast to the Krylov subspace techniques, most preconditioner implementations must be data-structure-dependent, so that each preconditioner must be explicitly coded for use with each new (parallel, sparse) matrix data structure. In addition, the source code and parallelism for matrix-based preconditioners are generally much more complicated than those for Krylov subspace methods. Thus, the numerical kernel of most preconditioners within PETSc is within the matrix module of the library, where such actions as incomplete factorization, triangular solves, and extraction of matrix subblocks occur.

We coordinate the solvers by using a context data type (called `SLES` for linear solvers and `SNES` for nonlinear solvers), to store all information about the solution process, including the right-hand side, convergence tolerances, options, parameters, etc. All of the computations and communications related to a particular solution process are managed in the solver context variable. These solvers, as well as the lower levels of code on which they are built, of course also employ the previously discussed software design principles.

To demonstrate the easy-to-use interface that this approach enables, we consider a simple example of solving a linear system. We first create and assemble the matrix `A` and right-hand-side vector `b`; then we create a solver context, set some solver parameters, and finally solve the system.

```
Vec      x, b;
Mat      A;
SLES     sles;
MPI_Comm comm;
...
MatCreate(...,&A);
/* ... Assemble the matrix ... */
VecCreate(...,&b);
/* ... Assemble the vector ... */
SLESCreate(comm,&sles);   /* Create a linear solver context */
SLESSetFromOptions(sles); /* Set solver parameters at runtime */
SLESSetOperators(sles,A,A,SAME_NONZERO_PATTERN);
/* Set the matrix that defines the linear system and an
   optionally different preconditioning matrix.  The flag
   SAME_NONZERO_PATTERN indicates the matrix A will have
   the same nonzero structure for several distinct linear
   solves; this flag allows reuse of information during
   successive solves to reduce the computation required
   in constructing the preconditioner for the next linear
   system. */
SLESSolve(sles,b,x);      /* Solve the system */
```

All communication required by the linear solver is handled internally within the SLES routines (and the routines that they call) and uses the MPI communicator set in the call to `SLESCreate()`. This communication includes

- passing matrix information during the preconditioner construction,
- computing vector inner products and norms within Krylov subspace codes, and
- performing vector scatters in parallel matrix-vector products.

The advantage of this organization is that the communication is always intrinsically associated with a higher-level object or objects and the operations performed on them, rather than using simply a jumble of sends, receives, and other MPI calls for which it is difficult to understand the connection between a particular message-passing call and the actual mathematical operations on the data. Consequently, parallel application codes, including the linear, nonlinear, and timestepping solvers within PETSc, are much easier to develop, maintain, and understand.

Through its design, PETSc provides a uniform interface to all of its linear solvers. These solvers include a variety of Krylov subspace accelerators such as

- the conjugate gradient method,
- GMRES,
- CGS,
- Bi-CG-stab, and
- transpose-free QMR,

and a growing family of preconditioners, such as

- block Jacobi,
- overlapping additive Schwarz,

- ILU(0) and ICC(0) (through an interface to BlockSolve95 [JP92]), and
- ILU(k) (currently sequential only).

There are three main parallel sparse matrix formats:

- CSR (compressed sparse row),
- block CSR, and
- storage by block diagonals.

PETSc provides two main approaches to solving nonlinear algebraic systems (in parallel): line search techniques and trust region methods. Both support the use of any of the linear solvers in a truncated Newton algorithm. Recently, we have added support for using backward Euler timestepping schemes. We plan to include additional higher order timestepping schemes in the future. All the solvers are introduced in the PETSc users manual [BGMS95].

5.5 Object-Oriented Features of PETSc

This article has touched on several object-oriented features of PETSc without providing a complete overview of the object-oriented design. In this section we focus on how object-oriented techniques are used to organize the overall software package.

Object-oriented programming is often defined by the three complementary principles:

- data encapsulation,
- polymorphism, and
- inheritance.

Data encapsulation (also sometimes called data hiding) refers to writing objects (data structures) so that application code does not directly access the underlying data in the object. Rather, the application code can affect the data only by making subroutine calls that change the data. In C++, these routines are called member functions, and they are the only part of the code that can change data that is "private" to an object.

Polymorphism refers to techniques that allow one to call the same function from the application level of the code to perform a specific operation regardless of the underlying data structure used internally to store the data. An example of polymorphism in PETSc is the use of the routine `MatMult()` to perform a matrix-vector product regardless of the particular matrix and vector formats being used. In C++, polymorphism is usually accomplished by using virtual functions.

Inheritance is the process of defining new objects by either combining properties of several different types of objects or adding properties to an object that is already defined.

PETSc uses all three of the defining principles of object-oriented programming. PETSc uses strong data encapsulation in both the vector and matrix data objects. Application access to the vector and matrix data is obtained through function calls such as `VecSetValues()` and `MatMult()`. All of the PETSc operations are supported via polymorphism. The user need not call routines specific to a particular data structure from the application code. Instead, the user calls a generic interface routine that correctly calls the underlying data-specific routine. This process is handled in PETSc through the use of structures of function pointers, quite similar to the way virtual functions are handled in C++.

Surprisingly, PETSc also uses inheritance in its design. All PETSc objects are derived from an abstract base object. From this fundamental object an abstract base object is defined for each PETSc object; examples include the `IS` (index set) object, the `Vec` (vector) object, the `Mat` (matrix) object, and the `PC` (preconditioner) object. Each of these abstract base objects then has a variety of instantiations. For example, there are at least five different matrix storage formats. No attempt is made to use inheritance directly inside these instantiations. For example, the CSR and block CSR objects do not share any common code. For specific details on how the inheritance is managed (e.g., for vectors), one may refer to the files `petsc/include/phead.h` and `petsc/src/vec/vecimpl.h` as well as `petsc/src/vec/impls/dvecimpl.h`.

Another level of inheritance has been incorporated in the newest component of PETSc, the `GVec` (Grid Vector) package. Grid vectors are vectors with additional information about an underlying grid and discretization. Since PETSc is coded in C, which does not provide direct support for inheritance, we manage inheritance by using techniques similar to those used for handling attributes in MPI.

Thus, although PETSc is written in C, which is not an object-oriented programming language, it is still possible to take advantage of the three underlying principles of object-oriented programming to manage the complexity of such a large package. Clearly, a software package similar in design to PETSc could be implemented in C++ with virtually a one-to-one mapping between the underlying data structures and routines in the two packages. We have chosen to develop PETSc in C to ensure complete portability across a wide range of machines, while C++ has continued to evolve over the years.

6 Sample Performance Results

To demonstrate the type of parallel performance one can expect using PETSc, we consider here a three-dimensional structured Euler simulation on a C-type grid using a finite volume discretization. The case for which the following data was generated is transonic flow over an ONERA M6 wing, a standard test case originally coded (for a serial computer) by David Whitfield of Mississippi State University in a Newton-iterative formulation with low-CFL, pseudo-transient continuation and explicit enforcement of boundary conditions [WT91]. As part of the conversion to PETSc, we have replaced the explicit boundary conditions with a fully implicit variant [Tid95].

The solution technique used for this problem is a Newton-Krylov approach with pseudo-transient continuation and adaptive advancement of the CFL number. Much of the approximate Jacobian matrix has block-band structure corresponding to the three-dimensional, seven-point stencil, with five degrees of freedom per node (three-dimensional momentum, internal energy, and density). The fully implicit boundary conditions spoil the strictly banded structure due to the C-type grid wrap-around of the mapped grid in the wake region. We use the PETSc matrix format for block, compressed, sparse rows (block CSR) to exploit this structure.

We present timings on an IBM SP2 for the matrix-vector product, the entire linear solve, and a fixed number of iterations (29) of the nonlinear solve. The times for the matrix-vector products and linear solves are for the entire solution process. Thus, for each matrix-vector product in the table, the times are given for between 110 and 150 products, while the linear solve times are for 29 complete linear solves. To put these numbers in perspective, we note that the peak performance of one processor of the IBM SP2 is 264 Mflops, the LINPACK 100 benchmark produces 130 Mflops, and a sparse matrix-vector product that uses the standard compressed sparse row format (CSR) attains 27 Mflops.

Tables 1.1 and 1.2 are for a fixed grid size of $98 \times 18 \times 18$, so that the Jacobian matrix has a dimension of 158,760 with 4,636,200 nonzero elements. The linear Newton systems are solved using GMRES(30) and block Jacobi preconditioning, where each processor has one block that is solved with ILU(0). The speedup over two processors is given in parentheses in the tables. Note that the entire problem does not fit on one processor when explicitly forming the Jacobian matrix and using ILU preconditioning.

We next consider similar runs for the same problem on a refined mesh of size $194 \times 34 \times 34$, which produces a system that is roughly eight times as large as the previous one, having 1,121,320 rows and 35,742,760 nonzeros. These results are given in Tables 1.3 and 1.4, where the speedup over sixteen processors for each phase is given in parentheses.

For the final problem having 1,121,320 unknowns, the computation rate on sixteen processors for the matrix-vector product was 1,281 Mflops, while the complete linear solve achieved 1,011 Mflops.

TABLE 1. Performance on Small Grid Problem: Computation Time (Seconds)

Number of Processors	Mat-Vec Products		Linear Solves		Nonlinear Solve	
	Time	Speedup	Time	Speedup	Time	Speedup
2	5.67	–	17.46	–	73.63	–
4	3.03	(1.9)	9.02	(1.9)	39.22	(1.9)
8	1.65	(3.4)	4.74	(3.7)	21.17	(3.5)
16	0.92	(6.2)	2.62	(6.7)	11.50	(6.4)
32	0.58	(9.8)	1.62	(10.8)	6.70	(11.0)
64	0.40	(14.2)	1.14	(15.3)	4.87	(15.1)

TABLE 2. Performance on Small Grid Problem: Computation Rate (Mflops)

Number of Processors	Mat-Vec Products		Linear Solves	
	Mflops	Speedup	Mflops	Speedup
2	179	–	147	–
4	337	(1.9)	286	(2.0)
8	620	(3.5)	540	(3.7)
16	1137	(6.4)	994	(6.8)
32	2038	(11.4)	1785	(12.1)
64	3003	(16.8)	2546	(17.3)

TABLE 3. Performance on Large Grid Problem: Computation Time (Seconds)

Number of Processors	Mat-Vec Products		Linear Solves		Nonlinear Solve	
	Time	Speedup	Time	Speedup	Time	Speedup
16	6.38	–	20.04	–	83.08	–
32	3.29	(1.9)	9.38	(2.1)	42.15	(2.0)
64	1.94	(3.3)	5.34	(3.8)	25.44	(3.3)

TABLE 4. Performance on Large Grid Problem: Computation Rate (Mflops)

Number of Processors	Mat-Vec Products		Linear Solves	
	Mflops	Speedup	Mflops	Speedup
16	1281	–	1011	–
32	2483	(1.9)	2154	(2.1)
64	4217	(3.3)	3744	(3.7)

On sixty-four processors the matrix-vector product ran at 4,217 Mflops, while the complete linear solve achieved 3,744 Mflops. Note that for this problem we solved a single linear system in .18 seconds on sixty-four processors using a general-purpose object-oriented numerical library.

For representative parallel computation rates for a similar problem, we consider use of the block tridiagonal (BT) method in a three-dimensional Navier-Stokes simulation within category two of the NAS Parallel Benchmarks (NPB 2) [NAS96]. This method solves three sets of uncoupled systems of equations, each of which is block tridiagonal with 5×5 blocks. For a problem size of 102^3 (which is comparable to our large grid problem mentioned above), the BT code achieves 924 Mflops on sixteen processors of an IBM SP2; the sixty-four processor case achieves 3,487 Mflops. While the numerical methods used in this benchmark differ from those employed in our Euler code, this comparison does illustrate that the PETSc libraries achieve computation rates in line with those of similar sparse applications.

7 Conclusion

We have outlined six of the guiding principles that are used to efficiently manage the communication and computation in a parallel, large-scale, numerical object-oriented software library. These techniques are appropriate for distributed as well as NUMA-based shared-memory computers. In fact, the underlying PETSc libraries can deliver high performance on all modern scalable machines. The guiding principles are appropriate for all scalable computers, not merely those requiring message-passing programming.

We have shown how it is possible to organize a library around two conflicting goals—*ease of use* and *high efficiency*—while maintaining a good balance between the two and allowing users to move along the spectrum in either direction depending on their needs.

The complete PETSc distribution is freely available via our home page, `http://www.mcs.anl.gov/-petsc/petsc.html`.

Acknowledgments: The work of the first author was supported by the Applications Technology Research Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. The work of the third author was supported by the National Science Foundation under ECS-9527169, through the Old Dominion University Research Foundation. The second and fourth authors were 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.

8 References

[ACM91] Brett M. Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-150, Argonne National Laboratory, 1991.

[ASS93] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for unstructured and block structured problems. In *Proceedings of Supercomputing '93*, pages 578–587, 1993.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[BGMS95] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, November 1995.

[BGMS96] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc home page. http://www.mcs.anl.gov/petsc/petsc.html, December 1996.

[BL96] A. M. Bruaset and H. P. Langtangen. *A Comprehensive set of Tools for Solving Partial Differential Equations: Diffpack*. Birkhauser, 1996.

[GLDS96a] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performace, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[GLDS96b] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. MPICH home page. http://www.mcs.anl.gov/mpi/mpich/index.html, December 1996.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2*. Adam Hilger, 1988.

[HST95] Scott A. Hutchinson, John N. Shadid, and Ray S. Tuminaro. Aztec user's guide version 1.1. Technical Report SAND95/1559, Sandia National Laboratories, October 1995.

[JP92] Mark T. Jones and Paul E. Plassmann. BlockSolve v1.1: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-92/46, Argonne National Laboratory, 1992.

[JP93] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[MPI94] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.

[MSGH84] Jorge J. Moré, Danny C. Sorenson, Burton S. Garbow, and Kenneth E. Hillstrom. The MINPACK project. In Wayne R. Cowell, editor, *Sources and Development of Mathematical Software*, pages 88–111, 1984.

[NAS96] NAS Parallel Benchmarks home page. http://www.nas.nasa.gov/NAS/NPB/-index.html, December 1996.

[RCH+96] J. V. W. Reynders, J. C. Cummings, P. J. Hinker, M. Tholburn, M. Srikant S. Banerjee, S. Karmesin, S. Atlas, K. Keahey, and W. F. Humphrey. *POOMA: A FrameWork for Scientific Computing Applications on Parallel Architectures*, chapter 14. 1996.

[SG96] Barry F. Smith and William D. Gropp. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. *Scientific Programming*, 5:329–336, 1996.

[SOHL+95] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[Thi93] Thinking Machines Corporation. *Users Manual for CM-Fortran*. Thinking Machines Corporation, 1993.

[Tid95] M. D. Tidriri. Krylov methods for compressible flows. Technical Report 95-48, ICASE, June 1995.

[WT91] D. Whitfield and L. Taylor. Discretized Newton-relaxation solution of high resolution flux-difference split schemes. In *Proceedings of the AIAA Tenth Computational Fluid Dynamics Conference*, pages 134–145, 1991.