An Interface for Efficient Vector Scatters and Gathers on Parallel Machines

Barry F. Smith

Abstract— **Scatter and gather type operations form the heart of most communication kernels in parallel partial differential equation solvers. This paper introduces a simple interface for defining and applying scatter and gather operations on both distributed and shared memory computers. A key feature of the interface is that it allows efficient implementations that can take advantage of underlying structure in the indexing of the scatters and gathers, such as strided indexing or indexing of blocks of data. A discussion of an implementation using MPI in the software package PETSc (the Portable, Extensible Toolkit for Scientific computation) is included. The interface is fully usable from Fortran 77, C, and C++.**

Keywords— **scatters, gathers, parallel communication, MPI, ghost points, PETSc**

I. Introduction

The parallel solution of partial differential equations (PDEs) is most often achieved by using data decomposition (sometimes called domain decomposition, perhaps more appropriately called grid decomposition). The underlying finite element, finite difference, or finite volume grid is decomposed among the various processors; each processor is then responsible for the numerical computation involving its portion of the grid. Information from neighboring portions of the grid "owned" by other processors must be communicated to the given processor; this communication is usually done through the concept of ghost nodes (or ghost cells); see Fig. 1. The communication required can be formulated as a generalization of scatters and gathers. This is true regardless of whether explicit or implicit methods are used to discretize the PDE, for both linear and nonlinear problems as well as time-dependent and steady-state problems. In addition, these data decomposition approaches are appropriate for problems solved on both structured and unstructured grids. Thus, efficient scatters, gathers, and related operations are crucial for the efficient solution of PDEs on parallel computers.

Consider vectors x and y of lengths N_x and N_y and two sets of non-negative integers ix and iy of length N_{ix}. We define a generalized scatter of x to y by

$$y[iy[i]] = x[ix[i]] \qquad i = 1, ..., N_{ix}.$$

We allow N_{ix} to be of different length than N_x and the arrays $iy[]$ and $ix[]$ may contain duplicate indices, and/or skip indices completely. In this paper we introduce a software interface for defining and applying such scatters efficiently on distributed and shared memory parallel computers.

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844. Electronic mail address: bsmith@mcs.anl.gov. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

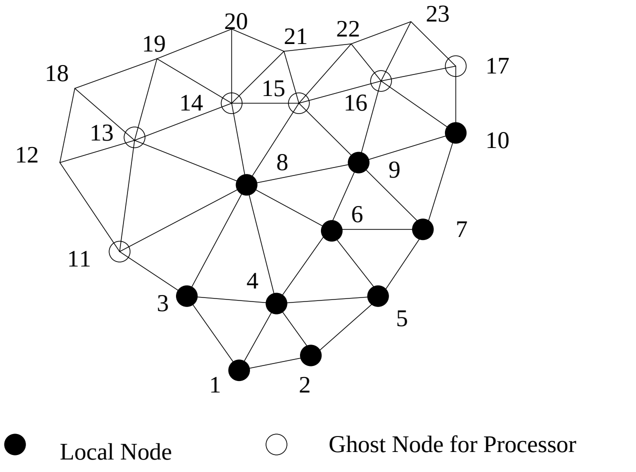

Fig. 1. Ghost Points for Processor

In Section II we provide the motivation, demonstrating how the communicational kernels in many PDE computations may be viewed as a scatter operation. Section III introduces the interface as used by the PETSc (Portable, Extensible Toolkit for Scientific computation) package [4], [3], [5], and Section IV explains the various optimizations that the interface allows (by taking advantage of structure in the indexing) to achieve the fastest possible communication kernels.

The interface introduced in this paper uses the inspector/executor ideas as proposed in the PARTI work of Saltz [6], [10]. For communication of data on regular rectangular arrays and groups of regular rectangular arrays, there is related work by Quinlan (A++/P++) [9], [12], Baden (KeLP, formerly LPARX) [2], [7], Saltz (multiblock PARTI) [1], and Parashar (DAGH) [11].

II. Scatters in Parallel PDE Computations

Communication in most parallel finite element, finite difference, and finite volume PDE computations comes in two forms: nearest neighbor and global reductions. This paper discusses the communication kernels involving neighbor communication. Consider a grid as depicted in Fig. 1 or Fig. 2. For explicit (in time) methods, one computes

$$u^{n+1} = F(t^{n+1}, u^n, ...),$$

where u^n contains the discrete solution at, say, the grid points. The function $F()$ is local in the sense that

$$u_i^{n+1} = F_i(t^{n+1}, u^n, ...)$$

depends only on u_j^n for nodes j that are near (in the grid) to node i.

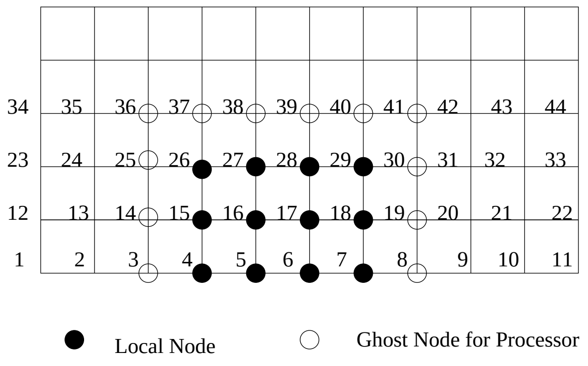

Fig. 2. Ghost Points for Processor on Regular Grid

For implicit methods, one needs to solve

$$G(u^{n+1}, t^{n+1}, u^n, ...) = 0$$

for u^{n+1}, where, again, $G()$ is local in that $G_i()$ depends only on u_j^{n+1} and u_j^n, for nodes j near node i. In addition, for implicit methods, one often wants (when using, for example, Newton's method) to solve linear systems involving some approximation to J, the Jacobian of $G()$. Fortunately, the Jacobian is sparse and entails the same coupling of nodes as the function $G()$. Thus, neighbor communication in implicit methods usually involves

- evaluation of a "sparse" function, $G()$,
- evaluation of a sparse Jacobian, J, and
- application of sparse matrix-vector products.

Let's consider in more detail an implementation to compute

$$u^{n+1} = F(u^n),$$

where the dependence on t and other parameters is hidden for simplicity. We present the interface three times, going from the abstract to the concrete.

Abstract Level

Function input: a "parallel" vector.
output: a "parallel" vector.

Abstract Distributed-Memory Level

Function input: an array containing the local values of a "parallel" vector.
output: an array containing the local values of a "parallel" vector.

More Concrete Distributed-Memory Level

Function input: `double inarray[]` and a data structure containing needed communication information.
output: `double outarray[]`.

`double work[]`, local work array.
Communicate ghost values from other processors into `work` array.
Copy local values from `inarray` to `work` array.
Compute purely local function
$$unp = F_{local}(work).$$

As an alternative to allocating the large work array to hold the local portion of the input vector as well as its ghost values, one could instead allocate a "padded" parallel vector, that, on each processor, contains extra slots to hold the required ghost values. This offers the slight advantage of not having to copy the local values to the work space (since they are already stored there).

For ease of application programming one would like to use any arbitrary global numbering of the nodes (or degrees of freedom), for example, the "natural" numbering in Fig. 2, and define the needed ghost locations in that global numbering. The proposed approach advocates exactly this.

III. An Efficient Interface for Defining Parallel Scatters

Before discussing the interface for defining the scatters and gathers, we need to define one abstract concept. We define an **index set** as an abstract way of denoting a set of indices. The reason to use an abstract object to represent indexing is that it allows memory and speed optimizations for important special cases. For example, in performing a gather of n items, the indices indicating the resulting locations consist of {0, 1, 2, ..., n}. Clearly, allocating an integer array to contain these indices and then manipulating each index directly is not desirable.

In PETSc a distributed index set is denote by `IS`. There are currently three representations of index sets:

- a basic set of integers (where each processor contains a subset of the list),
- a set of integers where each integer represents a block of indices of fixed size (this is particularly useful for efficient computation on multicomponent PDEs), and
- a strided set of integers, represented by the first value, a stride and a number of entries.

Another natural candidate would be a strided set representing blocks of indices.

To make this more concrete, we give the PETSc calling sequences to generate these three index sets.

```
ISCreateGeneral(MPI_Comm comm, int n,
                int *indices, IS *result);
ISCreateBlock(MPI_Comm comm, int blocksize,
              int n, int indices, IS *result);
ISCreateStride(MPI_Comm comm, int n,
               int first, int step, IS *result);
```

All of the processors that are to be involved in the scatter/gather would generate their portion of the `IS` by calling the appropriate `ISCreateXXX()` routine.

The index sets only define the "from" and "to" indices of the scatter/gather operations in some numbering scheme; they themselves do not contain the detailed information required to actually perform the data movement via message passing or shared memory. In PETSc we create a scatter context, denoted by `VecScatter`, that contains the detailed information that is used to actually communicate the appropriate vector values (for example, message buffers, processor lists).

A "parallel" vector in PETSc (denoted by `Vec`) is essentially represented by an array in each processor's memory, plus additional information about the layout of the values across the processors (e.g., the number of values each processor stores in its array.) For coding simplicity, indexing of vectors in PETSc is always by processor: the first n_0 elements are stored on processor zero, the next n_1 are on processor one, etc. Since this is often not the case for the application code, PETSc provides simple tools to map between other vector element numbering schemes. Thus, one need only define the scatters in terms of the PETSc vector element numbering schemes; see Fig. 3.

```
Processor 1

3  4  8  9 <- Application numbering
0  1  2  3 <- PETSc vector numbering

                    Processor 2

Application numbering  ->  0  2  1  5  6  7
PETSc vector numbering ->  4  5  6  7  8  9
```

Fig. 3. Mapping between Numbering Schemes

Once the vector layout and "to" and "from" index sets are defined, we are ready to create the vector scatter context, `VecScatter`. The scatter context is a data object that contains all information required to actually move the data, when a ghost up date (for example) is needed.

In PETSc the scatter context is created via

```
VecScatterCreate(Vec vfrom,IS from,Vec vto,
                 IS to,VecScatter *scatterctx);
```

The vector values are actually moved via the pair of commands

```
VecScatterBegin(Vec vfrom,Vec vto,
                INSERT_VALUES,SCATTER_FORWARD,
                scatterctx);
/* possibly other commands */
VecScatterEnd(Vec vfrom,Vec vto,INSERT_VALUES,
              SCATTER_FORWARD,scatterctx);
```

This is an example of the inspector/executor model for data communication. The inspector routine, `VecScatterCreate()`, determines the communication patterns (what messages have to be packed, sent, received, and unpacked), while the executor actually does the packing, sending, receiving, and unpacking of the messages containing the actual vector values. In general, for PDE computations, the inspector is called once, while the executor is called many times.

The third and fourth arguments to `VecScatter-Begin/End()` may be `INSERT_VALUES` or `ADD_VALUES` and `SCATTER_FORWARD`, or `SCATTER_REVERSE` respectively. The `SCATTER_REVERSE` indicates that the roles of the "to" and "from" vectors are reversed. In a message passing system, the sends become receives and conversely the receives become sends.

We now give a more detailed example of the use of the interface than was given in the preceding section.

```
PETSc Distributed Memory Level
    Setup Phase:

    Create vectors with given parallel layout.
    Create local work vectors.
    Determine numbering of ghost nodes from grid
            information and create the "to" and
            "from" index sets.
    Create the vector scatter context.

    Function input: Vec input
                    VecScatter scatterctx.
             output: Vec output.

    Communicate ghost values from other processors
            into workinput vector using the
            scatter context.
    Compute purely local function
            $workoutput = F_{local}(workinput)$.
    Communicate ghost values back to other processors
            from workoutput vector using the
            scatter context, if required.
```

Fortran 77 does not support structures or pointers. Therefore, to provide this interface in Fortran, we represent the fundamental objects `Vec, IS,` and `VecScatter` in Fortran as integers that "point to" the underlying datastructures in C.

IV. Optimization of the Scatter Operations

The index set concept has two functions:

- increase readability and clarity of the application code and
- allow for optimization of important special cases.

For strided and blocked indices, one clearly gets an immediate potential savings, since the individual indices (that would be required if they were simply enumerated) need not be stored. This reduces both memory usage and the loading and storing times of the indices. The strided index set is also valuable for the automatic detection of simple copies, etc.

This interface allows a variety of special scatter/gathers automatically to be detected and optimized. PETSc currently takes advantage of many of these.

- No parallel communication required
  - gather, where the "from" indices are local
  - scatter, where the "to" indices are local
  - strided copy
  - general local copy, (see Fig. 4).
- When the mapping is one to one.
- When there are no duplicates (i.e. aliasing problems) in the indices.

- Everything to everybody. Can take advantage of `MPI_Alltoall()` communication on message passing systems.
- Everything to one processor.
- From one processor to all processors. Can take advantage of `MPI_Bcast()` communication on message passing systems.
- Efficient packing and unpacking of buffers for blocked or strided indexing, see Fig. 4.
- On message passing systems, use of MPI persistent sends and receives that allow MPI to optimize for sets of communications it knows will be repeated, see Fig. 5.
- Use of MPI ready-receiver mode.

In Fig. 4, we demonstrate how the use of strided or blocked information can be used to reduce the number of loads required during a scatter/gather operation. Though this may seem like a trivial savings, it does have a direct impact on performance.

- General indexing (no optimization)
```
for ( i=0; i<n; i++ )
  y[iy[i]] = x[ix[i]];
```
- Stride one in x
```
for ( i=0; i<n; i++ )
  y[iy[i]] = x[xstart+i];
```
- Stride one in x and y
```
for ( i=0; i<n; i++ )
  y[ystart+i] = x[xstart+i];
/* or */
memcpy(y+ystart,x+xstart,n*sizeof(double));
```
- Block indexing (size 4)
```
for ( i=0; i<n; i++ ) {
  iyt = iy[i]; ixt = ix[i];
  y[iyt]     = x[ixt];
  y[1 + iyt] = x[1 + ixt];
  y[2 + iyt] = x[2 + ixt];
  y[3 + iyt] = x[3 + ixt];
}
```

Fig. 4. Examples of Optimizations in Packing/Copying

In the use of persistent sends and receives, (Fig. 5), note that all the sends can be initiated in a single MPI call. The underlying MPI implementation already knows the destinations and sizes of all messages so it can quickly route them to the correct destination.

Code demonstrating the scatter creation with persistent sends and receives.

```
/* create the structures for the receives */
for (i=0; i<nrecvs; i++ ) {
  ... /* define buffer, cnt, etc */
  MPI_Recv_init(buffer,cnt,MPI_DOUBLE,src,tag,
                comm,&rwaits[i]);
}
/* create the structures for the sends */
for (i=0; i<nends; i++ ) {
  ... /* define buffer, cnt, etc */
  MPI_Send_init(buffer,cnt,MPI_DOUBLE,dest,tag,
                comm,&swaits[i]);
}
```

Code demonstrating the scatter application

```
/* post all the receives */
MPI_Startall(nrecvs,rwaits);
```

```
/* pack and post all sends */
/* Block indexing, block size 5 */
for ( i=0; i<slen; i += 5 ) {
  idx    = *indices++;
  val[0] = xv[idx];
  val[1] = xv[1 + idx];
  val[2] = xv[2 + idx];
  val[3] = xv[3 + idx];
  val[4] = xv[4 + idx];
  val     += 5;
}
MPI_Startall(nsends,swaits);
```

Fig. 5. Example Persistent Operations Usage

V. Conclusion

We have described an abstract interface for defining vectors and performing scatters and gathers and their generalizations on parallel machines. The interface is simple, requiring only three abstract objects: a vector (denoted by `Vec` in PETSc), an index set (denoted by `IS` in PETSc), and a vector scatter context (denoted by `VecScatter` in PETSc). However, the definition of the interface allows optimization for many important special cases. A key advantage of this approach is that it enables users always to work with the same basic objects, independent of their particular communication patterns. For example, on message-passing machines, when moving vector values between processors, the user can always employ the vector scatter routines and expect high performance, rather than worrying about having to use non-blocking sends and receives, blocking sends and receives, or `MPI_AllScatter()`, or `MPI_Scatter()`, or `MPI_Gather`, etc. to obtain high performance.

We conclude by mentioning one particular unstructured grid application code that utilizes the vector scatter interface discussed above. This code, written originally by W. K. Anderson, (NASA Langley), in Fortran 77 for sequential computers was ported to parallel computers using PETSc by Dinesh Kaushik, [8] . The code solves the steady-state incompressible Euler equations using a second-order Roe scheme. In Table I, we present performance numbers of the total solution process for a 2,761,774 vertex grid, amounting to 11,047,096 degrees of freedom on the Cray T3E-900 at NERSC.

TABLE I
Performance on T3E

Procs	Time	Efficiency	Gigaflops	Scatter Time
128	6,048	–	8.5	3%
256	3,242	93%	16.6	3%
512	1,811	83%	32.1	4%

Note the relatively small percentage of the time spent in the neighbor communication. The loss in efficiency on 512 processors can be traced to a work load imbalance among the processors, since the grid was partitioned to equalized vertices, not the work related to the vertices.

We emphasis some of the key reasons for the suggested approach for nearest neighbor communication.

- The abstraction in the interface allows optimizations "under the covers".
- The form of the arguments (`IS, Vec`) allow the library to detect potential optimizations.
- Some of the optimizations that can be detected would be extremely difficult for a compiler to detect.
- The collective definitions of the IS and VecScatter allow for use of "collective" system routines such as `MPI_Bcast()` and `MPI_Scatter()`.
- The model does not require a message-passing system and can be implemented directly in shared memory, etc.

Acknowledgements

I thank the rest of the PETSc team (Satish Balay, William Gropp, and Lois Curfman McInnes) for making PETSc possible. PETSc is freely available from `http://www.mcs.anl.gov/petsc/petsc.html` and may be used from Fortran, C, or C++. I also thank Dinesh Kaushik and David Keyes for developing the parallel Euler code and Kyle Anderson for generously providing the original source code.

References

[1] G. Agrawal, A. Sussman, and J. Saltz, *An integrated runtime and compile-time approach for parallelizing structured and block structured applications*, IEEE Transactions on Parallel and Disttibuted Systems, (to appear).

[2] S. Baden, *KeLP home page*. http://www-cse.ucsd.edu/groups-/hpcl/scg/kelp.html, July 1997.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[4] ———, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11 - Revision 2.0.21, Argonne National Laboratory, Nov. 1997.

[5] ———, *PETSc home page*. http://www.mcs.anl.gov/petsc, July 1997.

[6] R. Das, M. Uysal, J. Saltz, and Y. S. Hwang, *Communications optimizations for irregular scientific computations on distributed memory architectures*, Journal of Parallel and Distributed Computing, 22 (1994), pp. 462–479.

[7] S. J. Fink, S. B. Baden, and S. R. Kohn, *Flexible communication mechanisms for dynamic structured applications*, in Irregular '96, 1996.

[8] D. Kausik, D. Keyes, and B. Smith, *On the interaction of architecture and algorithm in the domain based parallelism of an unstructured grid incompressible flow code*, in Proceedings of 10th International Symposium on Domain Decomposition Methods in Science and Industry, AMS, 1998. submitted.

[9] M. Lemke and D. Quinlan, *P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications*, in CONPAR/VAPP V, in Lecture Notes in Computer Science, Springer Verlag, 1992.

[10] S. S. Mukherjee, S. D. Sharmann, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz, *Efficient support for irregular applications on distributed memory machines*, in PPoPP 95, 1995.

[11] M. Parashar and J. C. Browne, *DAGH: A data-management infrastructure for parallel adaptive mesh refinment techniques*, tech. rep., Department of Computer Science, University of Texas at Austin, 1995.

[12] R. Parsons and D. Quinlan, *A++/P++ array classes for architecture independent finite difference computations*, in Proceedings of the Second Annual Object-Oriented Numerics Conference, 1994.