Communication in data parallel languages

Bryan Carpenter
NPAC at Syracuse University
Syracuse, NY 13244
dbc@npac.syr.edu

January 2, 2000

Abstract

We start by discussing the patterns of communication needed to implement various constructs in High Performance Fortran. Then several communication libraries that have been developed to support these kinds of communications will be reviewed. Finally, as a case study a, scheme for implementing the array remapping operation used in general array assignments is described in detail.

Contents

1 Patterns of communication

In this section we will discuss some patterns of communication that arise in when a language like HPF is translated to a SPMD program. First we will cover the regular communication patterns that arise from simple array assignments, nearest-neighbour problems, and array intrinsics. Then the irregular communication patterns implied by data-parallel statements with non-linear subscript expressions will be introduced. Finally we cover some essentially task-parallel codes that can be translated most effectively in terms of "one-sided communication".

1.1 Array assignments

In HPF, variants of the innocent-looking assignment,

```
A = B
```

hide a variety of interesting communication patterns. The terms `A` and `B` may be any conforming arrays. Let's run through some examples.

As a first example consider:

```
!HPF$ PROCESSORS P(4)

      REAL A(50), B(50)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      A (1:49) = B (2:50)
```

The assignment variable and expression are sections of arrays with the same distribution. But the alignments of their elements are shifted by one place relative to each another. In translating this assignment, some communication between neighbouring processes will be needed. The situation is illustrated in Figure 1. In a translation to a SPMD program, the communications might be implemented using the point-to-point primitives `MPI_SEND` and `MPI_RECV`, or in a single call using `MPI_SENDRECV`.

The "shift" pattern of communication is actually quite common in practical situations, and the next subsection will discuss it in more detail. But it is slightly contrived as an example of an array assignment. Usually the right- and left-hand-side arrays will not have such a simple alignment relationship. Here is another example involving array sections[1]:

```
!HPF$ PROCESSORS P(4)

      REAL A(50,50), B(50)
!HPF$ DISTRIBUTE A(*, BLOCK) ONTO P
```

[1] As a matter of fact, any communication pattern in HPF involving array sections can be reproduced using whole arrays. One just has to give those arrays suitable alignments to templates. We choose to work with sections because they seem more concrete.

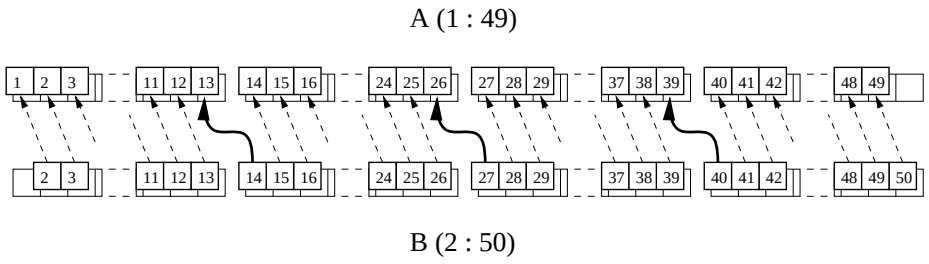

Figure 1: Assignment involving a shift in alignment. Heavy arrows represent interprocessor communication.

```
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      A (:, 1) = B
```

In this case the first dimension of `A` is collapsed, so the target of the assignment is a section that lives entirely on one processor. The communication pattern is visualized in Figure 2 You may recognize this as being essentially the communication pattern implemented by the MPI collective operation `MPI_GATHER` (or `MPI_GATHERV`). If the direction of the assignment had been reversed:

```
      B = A (:, 1)
```

then the communication pattern would correspond to the MPI operation `MPI_SCATTER` or `MPI_SCATTERV`. If the assignment target had a *replicated*, collapsed alignment:

```
      REAL A(50,50), B(50), C(50)
!HPF$ DISTRIBUTE A(*, BLOCK) ONTO P
!HPF$ DISTRIBUTE B(BLOCK) ONTO P
!HPF$ ALIGN C(I) WITH A(I, *)

      C = B
```

the communication pattern would be to broadcast all elements of the distributed array `B` to all processors—we recognize this as being like the MPI collective `MPI_ALLGATHER` or `MPI_ALLGATHERV`.

Finally we can easily write assignments that behave like `MPI_ALLTOALL` or `MPI_ALLTOALLV`. As an exercise the student may wish to verify that the following example qualifies:

```
      REAL A(50,50), B(50,50)
!HPF$ DISTRIBUTE A(*, BLOCK) ONTO P
!HPF$ DISTRIBUTE B(BLOCK, *) ONTO P

      A = B
```

as does:

```
      REAL A(50), B(50)
```

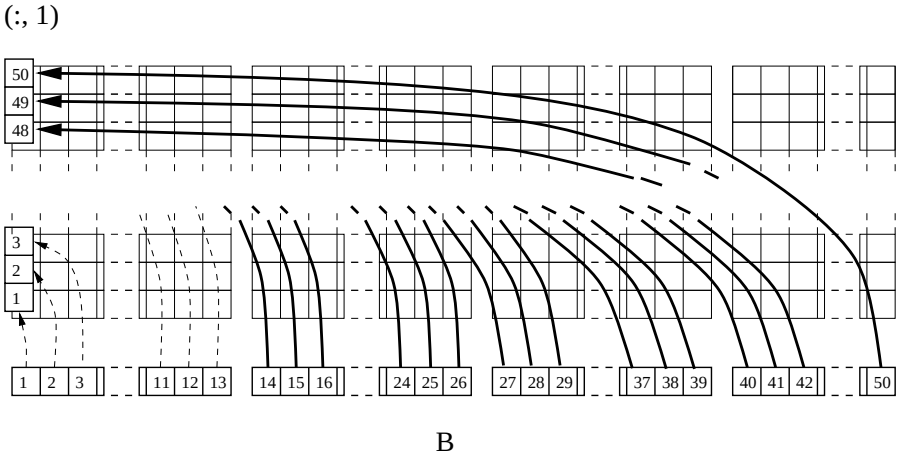

Figure 2: Assignment involving gathering array data to a single processor.

```
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ DISTRIBUTE B(CYCLIC) ONTO P

      A = B
```

On the one hand these examples illustrate the power of the HPF language. The effect of a complicated collective call in MPI can often be expressed very concisely in HPF as a simple array assignment. On the other hand, we get an idea of how complicated the communication patterns implied by simple-looking HPF statements can be.

1.2 Stencil problems and ghost areas

Stencil updates, in which each element of an array is updated in terms of some fixed footprint or "stencil" of neighbouring elements, have been an important target for parallel computing. They are quite common in practise, arising for example in solution of partial differential equations, simulation of cellular automata, and image processing applications. This situation is certainly amenable to massive parallelism, but it does require some communication.

These problems *can* be reduced to the kinds of array assignment discussed in Section 1.1. The well-known example:

```
      FORALL (I = 2:N-1, J = 2:N-1)
     &    U(I,J) = 0.25 * (U(I,J-1) + U(I,J+1) + U(I-1,J) + U(I+1,J))
```

can be recast as:

```
     U(2:N-1, 2:N-1) = 0.25 * (U(2:N-1, 1:N-2) + U(2:N-1, 3:N) +
    &                          U(1:N-2, 2:N-1) + U(3:N, 2:N-1))
```

The compiler could translate the array assignment code by introducing a series of temporary arrays `T1`, ..., `T1` all aligned with the section `U(2:N-1, 2:N-1)`:

```
T1 = U(2:N-1, 1:N-2)
T2 = U(2:N-1, 3:N)
T3 = U(1:N-2, 2:N-1)
T4 = U(3:N, 2:N-1))
U(2:N-1, 2:N-1) = 0.25 * (T1 + T2 + T3 + T4)
```

Due to the alignment of the temporary arrays, the last statement is purely local computation; the first four assignments absorb all communication. If the the arrays here have been distributed in the BLOCK style—usually optimum for this kind of problem—these communications follow a pattern similar to Figure 1.

In terms of the volume of interprocessor communication this is a reasonable approach. But it has two big drawbacks. First it uses a lot of temporary storage—four whole arrays. Secondly, although the first four assignments involve no computation and only a modest amount of communication at the edges, they introduce a lot of extra memory-to-memory copying.

A sequential version of the original loop has good locality properties in terms of its use of memory and cache. Because the elements on the left-hand-side and the operands in the right-hand-side of each assignment are typically clustered together in memory (spatial locality), and because elements are often reused in consecutive iterations, or iterations not far apart in iteration space (temporal locality), a straightforward transcription of the algorithm is likely to make good use of the cache of a modern microprocessor [3]. But if the program is split into consecutive array assignments as indicated above, much of this locality will be lost. Generally whole arrays will not fit in cache memory. Cache will be loaded from main memory and flushed back for each of the five loops associated with the five array assignments.

In fact this is a generic problem for the "array syntax" style of programming encouraged by Fortran 90. As we have seen in preceding lectures, this style evolved in large part to support a generation of SIMD and vector processors. On contemporary microprocessors, memory access costs usually dominate over the costs of arithmetic. Ironically a compiler may end up working quite hard to fuse the array assignments of a Fortran 90 program into sequential loops that have good memory locality.

A better approach than copying whole arrays of neighbours is to allocate the local segments of the distributed array `U` with a small amount of extra space around the edges—so-called *ghost regions*. The translation of the FORALL statement above could be something like:

```
REAL U(0 : BLK_SIZE1 + 1, 0 : BLK_SIZE2 + 1)
REAL T(BLK_SIZE1, BLK_SIZE2)

... Update ghost regions of U with values from neighbours

DO L1 = 1, BLK_COUNT1
    DO L2 = 1, BLK_COUNT2
        T(L1, L2) = 0.25 * (U(L1, L2 - 1) + U(L1, L2 + 1)
&                         + U(L1 - 1, L2) + U(L1 + 1, L2))
```

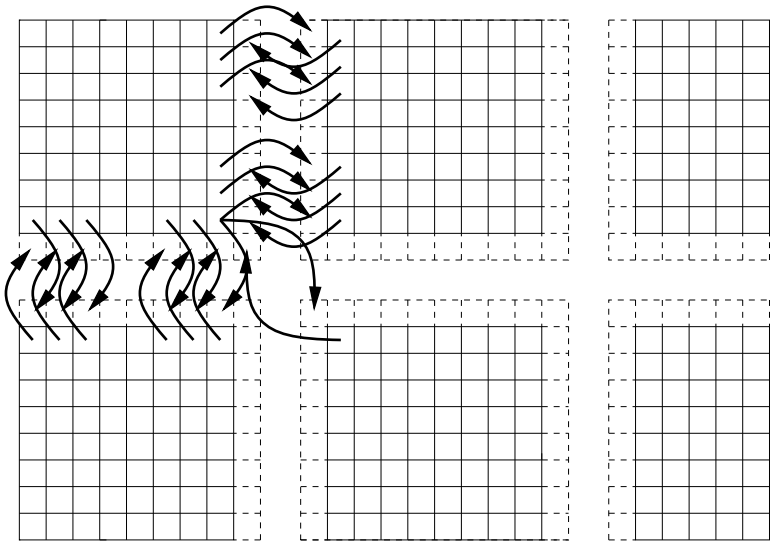

Figure 3: Communications needed to update ghost regions. Only communications involving the top left-hand processor are shown.

```
    ENDDO
ENDDO

... Copy T to interior part of U (local copying)
```

The communications required to update the ghost regions are illustrated in Figure 3. For the sake of definiteness we will refer to the interior part of the local segment—the non-ghost part—as the *physical* segment. The communication pattern illustrated here updates the cells in the ghost regions with the values from the corresponding cells in the physical segments.

1.3 Reductions and other transformational intrinsics

Fortran implementations come with a set of built-in functions and subroutines called the *intrinsics*. The *transformational intrinsics* are a subset of the intrinsics that peform functions on whole arrays, sometimes returning whole array results. In HPF these functions can operate on distributed arrays, implying new patterns of communication.

Some of the transformational intrinsics just reshuffle elements of arrays in regular ways. For example if the `CSHIFT`, `EOSHIFT`, `TRANSPOSE` and `SPREAD` intrinsics are applied to distributed arrays, they imply patterns of data remapping that are quite similar to patterns in the array assignments of section 1.1. The reduction intrinsics, including `SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `ALL`, `ANY` and several others, introduce qualitatively different communication patterns. They reduce an array either to a scalar or a lower-rank array by arithmetically combining el-

ements along some or all dimensions. Typically the associated communications occur on some spanning tree. In MPI terms they are related to, and might be implemented in terms of, `MPI_REDUCE`.

Parallel prefix operations can also be expressed. The HPF standard includes them as standard library functions (rather than intrinsics). They can also be expressed in terms of FORALL statements:

```
FORALL (I = 1 : N)  RES (I) = SUM(A (1 : I))
```

The ith element of the result contains the sum of all elements in `A` up to and including its ith element. The pattern of communication needed to implement the prefix algorithm *efficiently* is different to the one used in global reduction operations[2]. In MPI terms parallel prefixes are related to, and might be implemented in terms of, `MPI_SCAN`.

1.4 General subscripting in array parallel statements

Quite often a parallel program needs to execute some code like

```
FORALL (I = 1 : 50)  RES (I) = A (IND (I))
```

where `A` is one distributed array, and `IND` is some other distributed array. The special feature is that some subscript expressions (the subscripts for `A` in this case) are not a simple linear functions of the index variables. This operation cannot be reduced to the kind of a simple array assignment described in Section 1.1. Even in the most favourable case, when the arrays `RES` and `IND` are identically aligned, the data movement cannot be reduced to a single MPI collective operation.

Assuming `RES` and `IND` *are* aligned, the owner of the element `RES(I)` also owns the subscript `IND(I)`. So the processor that holds the target variable of the assignment can compute where the required element of `A` lives. This is generally on some other processor. Unfortunately that other processor—the one that owns the `A` element—does *not* have the information to tell the value is required by the first processor. It seems that there must be at least two phases of communication—one in which the target processors send out requests to the owners of the required data, and one in which the owners return the data. The situation is illustrated in Figure 4 for the case

```
!HPF$ PROCESSORS P(4)

      REAL A(50)
!HPF$ DISTRIBUTE A(CYCLIC) ONTO P

      REAL RES(50)
      INTEGER IND(50)
!HPF$ DISTRIBUTE RES(BLOCK) ONTO P
!HPF$ DISTRIBUTE IND(BLOCK) ONTO P

      IND = (/ 5, 41, 7, ... /)
```

[2]The CM Fortran compiler, for example, could recognize the kind of FORALL statement above as a parallel prefix and translate it appropriately.

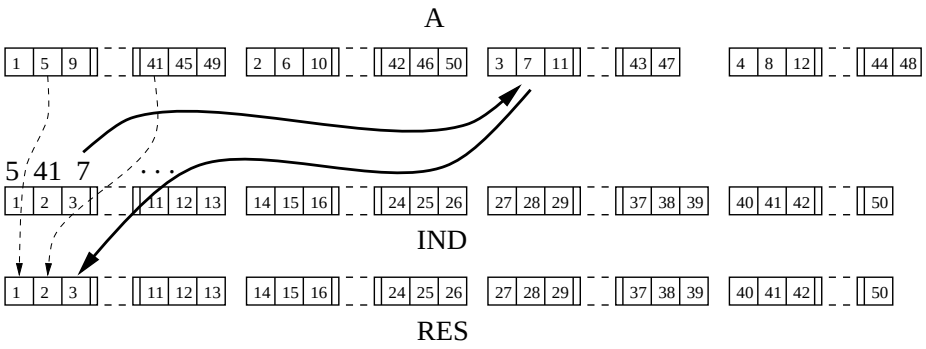

Figure 4: Assignment involving an indirect reference.

The first processor can deal with the assignment of the values `A(5)` and `A(41)` to the first two elements of `RES` locally. But the element `A(7)` lives on the third processor, and initially the third processor has no way to know that this element is needed by the first. Some two-way communication is needed.

There are many generalizations of this problem. The non-linear subscript expressions may appear on the left-hand-side rather than the right-hand-side:

```
FORALL (I = 1 : 50)  RES (IND (I)) = A (I)
```

(More generally, non-linear subscripts may appear on *both* sides.) The non-linear subscript may be a function or expression rather than an indirection array. The irregularly subscripted arrays may be multidimensional:

```
FORALL (I = 1 : 50)  RES (I) = A (IND1 (I), IND2(I))
```

As a special case it may be that the subscripts are actually be linear, but because the subscripts in orthogonal dimensions are not independent, the assignment can still not be reduced to a simple array assignment:

```
FORALL (I = 1 : 50)  RES (I) = A (I, I)
```

The loop itself may be multidimensional:

```
FORALL (I = 1 : 50, J = 1 : 50)  RES (I, J) = A (IND (I, J))
```

These examples are special cases of two (rather complicated) general families:

FORALL (i_1 = 1 : n_1, ..., i_R = 1 : n_R)
& RES ($i_1, \ldots, i_R$) = SRC (IND$_1$ ($i_1, \ldots, i_R$), ..., IND$_S$ ($i_1, \ldots, i_R$))

where the `IND` arrays are aligned with the `RES` array, and

FORALL (i_1 = 1 : n_1, ..., i_S = 1 : n_S)
& RES (IND$_1$ ($i_1, \ldots, i_S$), ..., IND$_R$ ($i_1, \ldots, i_S$)) = SRC ($i_1, \ldots, i_S$)

where the `IND` arrays are aligned with the `SRC` array. We will call these *generalized gather* and *generalized scatter* operations respectively[3]. A large class of

[3]Note that these operations are *not* related to the must simple `MPI_GATHER` and `MPI_SCATTER` operations—MPI's use of the terms "gather" and "scatter" is slightly unconventional.

FORALL statements can be reduced to a series of generalized gather and scatter operations, interspersed with some ordinary array assignments (although there is no guarantee that this is always the most *efficient* way to translate a FORALL statement).

1.5 Accessing remote data in task-parallel code

All the previous examples considered patterns of communication occurring in array parallel statements—array assignments or FORALL statements. These communication patterns are quite naturally treated as generalized collective operations. But there are situations in HPF—and in general SPMD programming—where this approach is not readily applicable.

One example is the INDEPENDENT DO loop of HPF, which takes the form:

```
!HPF$ INDEPENDENT
      DO i = 1, 10
          ...
      END DO
```

The INDEPENDENT directive asserts that there are no data dependences between individual iterations of the following loop, and the iterations may therefore be executed in parallel[4]. Unlike the FORALL statement, which explicitly limits the code executed in parallel to simple assignments, the body of an INDEPENDENT DO can involve any Fortran construct, including conditionals, loops and procedure calls. So the patterns of access to remote data inside parallel "iterations" may vary in unpredictable ways from one iteration to the next. It may become difficult to do any advance orchestration of data exchanges. An HPF the compiler is free to ignore the INDEPENDENT directive if it decides the loop is too complex to parallelize. But this may deprive the programmer of one of the few options in HPF for expressing the task-farming style of parallelism.

Actually there is at least one other way to express task parallelism in HPF. A user-defined procedure with the PURE attribute can be called from within a FORALL statement:

```
      PURE REAL FUNCTION FOO(INTEGER I)
      ...
      END


      ...
      FORALL (I = 1 : N)  RES (I) = FOO(I)
```

There are quite strict restrictions on PURE procedures, but nothing to prevent a procedure from reading elements of global distributed data—a distributed array in a COMMON block, for example. Unfortunately this makes it difficult or impossible for the compiler to determine at the point of call of FOO exactly what remote variables it will access. For example, the actual behaviour of the program might be similar to the first example of Section 1.4:

[4] Bear in mind that this independence at the logical variable level implies nothing about the home processors of accessed variables, and thus nothing about whether interprocessor communications are needed to translate statements.

```
      PURE REAL FUNCTION FOO(INTEGER I)

      REAL RES(50)
      INTEGER IND(50)
!HPF$ DISTRIBUTE RES(BLOCK) ONTO P
!HPF$ DISTRIBUTE IND(BLOCK) ONTO P
      COMMON /GLOBALS/ RES, IND

      RETURN RES (IND (I))
      END
```

But by the time `RES(IND(I))` is accessed, instances of the function `FOO` have already been dispatched to execute independently across the available set of processors. In a real sense, once inside FOO processors are no longer sharing a single "loosely synchronous" thread of control. It is difficult to see how the parallel invocations of FOO can behave collectively. In particular if the underlying model is MPI point-to-point communication it is difficult to see how the owner of a particular array element can always be ready to send an element when its value is accessed by a peer processor.

INDEPENDENT DO loops have similar problems, compounded because they do not have the restrictions on PURE procedures that prevent them from *writing* to global variables. If this sort of code is to be compiled to run in parallel the most practical approach is probably to assume the availability of *one-sided communication*. The MPI 2 standard added this functionality to MPI, but it is still not widely implemented.

2 Libraries for distributed array communication

As we have seen, the communication patterns implied by languages like HPF can be complex. Often it is impractical for a compiler to generate all the low-level message passing instructions needed to execute these communications. Instead the compiler may choose to emit code with calls to higher-level libraries for manipulating distributed array data. By analogy with the run-time support libraries used for memory management (and so on) in sequential languages, the data parallel compiler's library for handling distributed arrays is often called its *run-time library*.

In this section we will discuss some libraries that have been used, or could be used, in this role.

2.1 The PARTI primitives

The CHAOS/PARTI series of libraries was developed at the University of Maryland.

The original PARTI library was designed to deal with irregular scientific computations. A classic example is a physical problem discretized on an unstructured mesh. A characteristic inner loop for this kind of problem might look something like:

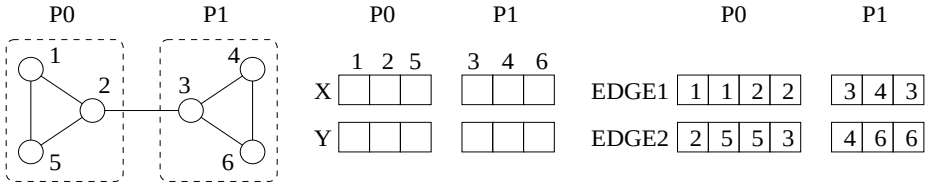

Figure 5: An irregular problem graph, exhibiting locality of reference.

```
DO I = 1, NEDGE
    Y(EDGE1(I)) = Y(EDGE1(I)) + F(X(EDGE1(I)), X(EDGE2(I)))
    Y(EDGE2(I)) = Y(EDGE2(I)) + G(X(EDGE1(I)), X(EDGE2(I)))
ENDDO
```

We assume the problem is defined on some graph (or mesh). The arrays `X` and `Y` are defined over the nodes of the graph. The `I`th edge in the graph connects the nodes `EDGE1(I)` and `EDGE2(I)`. The value of `Y` at a node is a sum of terms, each depending on the value of `X` at the ends of an edge connected to the node.

PARTI is particularly optimized for problems that have some "locality of reference". It assumes that a mapping of nodes to processors can be chosen such that *most* edges connect pairs of nodes on the same processor. The situation is illustrated in Figure 5. The local loops will be over the edges held on each processor. Here only one the edge (2, 3) held on processor P0 causes non-local references.

Figure 5 is copied from one of the illustrations in [2]. It assumes an *irregular* distribution of `X` and `Y`: elements 1, 2, 5 of the data arrays are stored on the first processor and elements 3, 4, 6 are stored on the second processor. This property is not particularly important to the discussion here. We could assume that the node numbering is permuted so that arrays like `X` and `Y` have a block distribution.

In any case the important point is that there is a class of problems with the following property: edges and nodes of the problem graph can be partitioned in such a way that most references become local. To be specific, they can be partitioned so that the *majority* of locally held elements of indirection vectors like `EDGE1`, `EDGE2` reference locally held elements of data arrays, like `X` and `Y`.

Based on this observation, PARTI assumes irregular loops are parallelized by a technique similar to the method of ghost regions discussed for regular stencil problems in section 1.2. First the indirection vectors are preprocessed to convert global index values to *local subscripts*. The locality property of the partition implies that the majority of these local subscripts refer to locally held data elements. If the global index actually referenced a data element held on another processor, the local subscript references an element in a "ghost extension" of the local data segment. These ghost regions are filled or flushed by suitable PARTI primitives: collective communication routines, called outside the local processing loop.

Here is a simpler sequential loop with irregular accesses:

```
C Create required schedules (Inspector):

      CALL LOCALIZE(DAD_X, SCHEDULE_IA, IA, LOCAL_IA, I_BLK_COUNT,
                    OFF_PROC_X)

      CALL LOCALIZE(DAD_Y, SCHEDULE_IB, IB, LOCAL_IB, I_BLK_COUNT,
                    OFF_PROC_Y)

C Actual computation (Executor):

      CALL GATHER(Y(Y_BLK_SIZE + 1), Y, SCHEDULE_IB)

      CALL ZERO_OUT_BUFFER(X(X_BLK_SIZE + 1), OFF_PROC_X)

      DO L = 1, I_BLK_COUNT
          X(LOCAL_IA(I)) = X(LOCAL_IA(I)) + Y(LOCAL_IA(I))
      ENDDO

      CALL SCATTER_ADD(X(X_BLK_SIZE + 1), X, SCHEDULE_IA)
```

Figure 6: PARTI code for simple irregular loop.

```
      DO I = 1, N
          X(IA(I)) = X(IA(I)) + Y(IB(I))
      ENDDO
```

A parallel version (from [1]) is given in Figure 6.

The first call to the subroutine `LOCALIZE` deals with the `X(IA(I))` terms. It does two things. It translates the `I_BLK_COUNT` global subscripts in `IA` to local subscripts, returned in the array `LOCAL_IA`, and it sets up a *communication schedule*. A handle to this data structure is returned in `SCHEDULE_IA`.

A communication schedule is created by analysing the requested set of accesses, sending lists of accessed elements to the processors that own them where necessary, detecting appropriate aggregations and redundancy eliminations, and so on. The end result is some digested list of messages that must be sent and received, including the local sources and destinations of the data in those messages.

Another input parameter to `LOCALIZE` is the distribution of the data array—`DAD_X` in the first call. Another output parameter is the number of elements in the ghost region that will actually be needed—`OFF_PROC_X` here.

The second call to `LOCALIZE` performs a similar analysis for the term `Y(IB(I))`.

Together these calls comprise what is called the *inspector phase* for the loop. It is followed by the *executor phase*, in which results are actually computed and data is actually communicated.

The collective call to `GATHER` communicates necessary element values from physical segments of `Y`—the second argument—into the target ghost regions for `Y`, which starts at `Y(Y_BLK_SIZE + 1)`—the first argument. The third argument

is the communication schedule for this operation. The call to `ZERO_OUT_BUFFER` just sets all elements in the ghost region of `X` to zero.

The main loop is self-explanatory. Contributions to locally owned `X(IA)` elements are accumulated directly into the local physical segment of `X`. Contributions to non-locally owned elements are accumulated into the ghost region of `X`.

Finally the call `SCATTER_ADD` sends the values in the ghost region of `X` to the relevant owners, where they are added to the appropriate elements in the physical region of the array segment. This is an example of a *combining scatter* operation.

Besides the PARTI software, a major contribution here is the elaboration of the *inspector-executor model*, and the insight that construction of communication schedules should be separated from execution of those schedules. One immediate benefit of this separation arises in the common situation where the form of the inner loop (the pattern of subscripting) is constant over many iterations of some outer loop. The same communication schedule can be reused many times—in other words the inspector phase can be lifted out of the main loop.

The importance of the inspector-executor model and the idea of communication schedules are not tied to the details of the PARTI primitives. For example they are not dependent on the particular assumptions about locality, or the special use of ghost regions, or even particularly specific to *irregular* computations.

2.2 Multiblock PARTI

2.3 The Global Array Toolkit

3 Adlib

3.1 Background and functionality

3.2 Case study: implementation of the `Remap` schedule

References

[1] Raja Das, Mustafa Uysal, Yuan-Shin Hwang, and Joel Salz. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22, 1994.

[2] Ravi Ponnusamy, Yuan-Shin Hwang, Raja Das, Joel H. Salz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions using data-parallel languages. *IEEE Parallel and Distributed Technology*, Spring, 1995.

[3] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.