Compiler Analysis for Irregular Problems in Fortran D

Reinhard von Hanxleden
Ken Kennedy
Charles Koelbel
Raja Das
Joel Saltz

CRPC-TR92287-S
December 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appeared in Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August 1992.

Compiler Analysis for Irregular Problems in Fortran D*

Reinhard von Hanxleden reinhard@rice.edu
Ken Kennedy ken@rice.edu
Charles Koelbel chk@rice.edu
Center for Research on Parallel Computation, Rice University, Houston, TX 77251, USA.

Raja Das raja@icase.edu
Joel Saltz jhs@icase.edu
Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, USA.

Abstract

Many parallel programs require run-time support to implement the communication caused by indirect data references. In previous work, we have developed the *inspector-executor* paradigm to handle these cases. This paper extends that work by developing a dataflow framework to aid in placing the executor communications calls. Our dataflow analysis determines when it is safe to combine communications statements, move them into less frequently executed code regions, or avoid them altogether in favor of reusing data which are already buffered locally.

1 Introduction

We present a dataflow framework that can be employed to systematically use runtime preprocessing methods for loops in which some array references are made through a level of indirection. This framework applies to collections of loops with no loop-carried data dependences except possibly those used for accumulations. Such loops are often referred to as *data-parallel* loops, and are the primary target of the Fortran D compiler [5]. Examples for the types of irregular loops we are trying to handle are found in unstructured mesh solvers and molecular dynamics codes [1, 2].

1.1 The Inspector-Executor Paradigm

In distributed memory machines, large data arrays need to be partitioned between processor memories. We call these *distributed arrays*. The pattern chosen often has a great effect on the efficiency of communication, but that topic is beyond the scope of this paper. Given a distribution pattern for these arrays, our problem is to generate efficient

*From the *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

communications calls; this usually involves collecting the information to be transmitted into relatively large messages.

In irregular problems, the communications pattern depends on the input data, typically because of some indirection in the code. This implies that messages must be aggregated at run time rather than compile time; we do this by transforming the original program into two constructs called inspector and executor [9, 10]. At run time, the *inspector* examines the data references made by a processor calculating the off-processor data to be received and allocating its storage. The *executor* then uses the information from the inspector to perform the actual computation. In the Fortran D compiler project at Rice University, we plan to perform these tasks by calls to the PARTI (Parallel Automated Runtime Toolkit at ICASE) library [2]. These primitives perform such low-level functions as coordinating interprocessor data movement, managing the storage of and access to copies of off-processor data, and supporting complex data distributions using distributed translation tables [11]. The PARTI functions are used by the inspector to produce a *schedule* describing the communications pattern, and by the executor to perform the actual gather or scatter of data. The routines are designed in the style of collective communication calls in which all processors participate; thus, there is a well-defined point in the program to insert the subroutine calls.

1.2 Optimizing Data Communication

The PARTI routines can track and reuse off-processor data copies [3]. We generate *combining* communication schedules that combine off-processor data for several indirect references, and *incremental* schedules that send only data not already requested by previous schedules. This gives the run-time support needed for combining and hoisting gather, scatter, and accumulate operations. Duplicates are removed by using a hash table.

To illustrate the importance of combining and incremental schedules, we summarize some previous hand-coded results. We ported an explicit unstructured mesh solver of the three dimensional Euler equations originally developed by Dimitri Mavriplis to the Touchstone Delta using PARTI primitives [2]. We present timings from two implementations of the program. In the first, labeled "Without Dataflow," each data-parallel loop was treated separately, resulting in gather and scatter calls before and after every loop. The second, labeled "With Dataflow," used combining/incremental scheduling, as given by the dataflow equations, to aggressively merge and hoist gather and accumulate primitives out of iterative loops. We have hand-checked that the dataflow framework produces the same placement of communications as the second version. Table 1 shows the performance of these implementations on the largest unstructured mesh we used, consisting of 804,056 points. The table gives the total time for the inspector, the time per iteration of the executor, and the execution speed of the executor. The executor performance improves by 58% when using combined scheduling, offsetting a modest increase in preprocessing time.

Implementation Strategy	Inspector (seconds)	Executor/Iteration (seconds)	Performance (Mflops)
Without Dataflow	2.73	4.18	947
With Dataflow	2.99	2.65	1496

Table 1: Explicit Unstructured Euler Solver on 804K Mesh on 512 Delta Processors

1.3 Compilation Strategy

At compile time, the dataflow framework determines for each point of the program which data are needed and which live off-processor data are available. This framework is similar to classical techniques such as Common Subexpression Elimination. The flow information determines where gather, scatter, and accumulate operations should be placed, and when combining/incremental schedules may be employed. In our data flow analysis, some of the variables reflect inherent properties of the analyzed program, while others are heuristics for producing the gather and scatter operations. Our heuristics aim to exploit situations where we can reuse communications schedules, and to remove duplicate communications by combining communications calls.

The rest of the paper is organized as follows. Section 2 provides some definitions and terminology for the framework. Section 3 introduces the local flow variables, followed by global variables in Section 4 and result variables in Section 5. Section 6 gives an extension of the framework for handling reduction operations. In Section 7, we work through the dataflow variables for a program example, which is a simplified version of a the Euler solver described above. Section 8 contains concluding remarks and an outlook on future work.

2 Basics of the Framework

2.1 The domain

Even though our implementation can handle other cases as well, we assume here for presentation purposes that all indirect references in the program text are of the form $\langle array \rangle(\langle index\_array \rangle(\langle loop\_variable \rangle))$. For many programs, this can actually be achieved by forward substituting array indices. For example, the code sequence `j=ia(i); x(j)=10` would be treated as `x(ia(i))=10`. Arrays that are never referenced indirectly are assumed to be analyzed using other methods [4] prior to this analysis. References with multiple (but bounded) levels of indirection will require more levels of complexity in the dataflow framework; we do not consider potentially unbounded indirection, as is found in linked lists.

Let V be the set of arrays that are accessed indirectly; in the example code shown in Figure 1, it is $V = \{x, y, z\}$. We assume that each reference r to some $v \in V$ is contained

in some loop(s). Let L be the set of loops that *directly* enclose an occurrence of some $v \in V$; in the example, it is $L = \{l_1, l_2, l_3, l_4\}$. We assume that no $l \in L$ encloses any other $m \in L$. One set of dataflow variables is computed for each element of a set of *nodes*, N. It is $N = L \cup P$, where P contains one *entry pad*, l_{entry}, and one *exit pad*, l_{exit}, for each loop $l \notin L$ containing some $l' \in L$. The example contains one outer time stepping loop, for which we introduce $l_{entry} = l_0$ and $l_{exit} = l_5$. This results in $P = \{l_0, l_5\}$ and $N = \{l_0, l_1, l_2, l_3, l_4, l_5\}$. Furthermore, we assume l_{entry} (l_{exit}) to be executed before (after) l iff l has *at least one iteration*; this enables us to hoist communication out of loops without risking unnecessary communication in case the loop has zero iterations. In the example, the resulting loop flow graph shown in Figure 1 therefore has an edge around the outer loop *including* l_0 and l_5.

The framework operates on a *loop flow graph* $G = (N, E)$ of the program, where the edges E are simple control flow edges. For example, if l is an outer time stepping loop that does not directly contain any irregular array references but contains a loop l' over mesh edges, then l' is represented as a node in G and l is represented as some interval in G. In the following, *loop* refers to an element of N, *i.e.*, it may denote a pad as well.

Future work will present a complete framework in which summary information is built in a bottom-up fashion similar to array kill information [4]. Furthermore, this paper only discusses the case where the summarized loops have no data dependences, except for commutative and associative reductions that are handled separately.

2.2 Array portions

Array portions are a central concept to the framework and best introduced by an example. A *portion* `x(ia(1:n))` consists of the *array* `x` and the *index set* `ia(1:n)`. This index set in turn consists of the *index array* `ia` and the *range* `(1:n)`, which has the *lower bound* `1` and the *upper bound* `n`.

Several portions may be taken from the same array or may have the same index set. The index range does not have to be known at compile time, so the bounds may contain symbolics. No assumptions are made about whether different portions taken from the same array are disjoint or whether they overlap each other partially or completely. This enables analyzing symbolic index ranges, but it requires the analysis to be conservative when using intersection and set subtraction in the equations.

The framework can be implemented using a lattice of bit vectors. Each bit vector represents a dataflow variable at a certain node in G, and each bit represents one array portion. To construct these bit vectors, an initial pass over the program has to collect all indirect references. The length of the bit vectors is bounded by the number of indirect array references an therefore linear in the program size. All equations given here are *rapid* [7]. Therefore, using bit vectors for the analysis results in good asymptotic running times. However, for our examples (and probably also in a practical implementation), it seems advantageous to represent the different flow variables as *bit matrices*. The

rows of a bit matrix correspond to the arrays of the portions represented (*e.g.*, `x` in `x(ia(1:n))`), while the columns correspond to the index sets (`ia(1:n)`). Theoretically that representation increases variable sizes from linear in program size to quadratic in program size, so the feasibility of this approach depends on how programs behave in practice. However, this representation makes potential schedule sharing, for example, very easy to recognize by determining which index set columns have more than one entry.

For the analysis we also assume that an (identity) dummy index array is inserted for all direct array references.

2.3 Operations on portions

To aid the distinction between portions, indirect array references, array elements, and sets of all these constructs, we make a short digression to introduce the conversion operators *elements-of* p (where p is some portion or set of portions), denoted $\tilde{p}$, and *references-of* p, denoted $\tilde{\tilde{p}}$. Assume we are given an array x, an index array $ia(1{:}5)$, portions $p = x(ia(p_l{:}p_u))$, $q = x(ia(q_l{:}q_u))$, $r = x(ia(r_l{:}r_u))$, and sets of portions $A = \{p, q\}$, $B = \{p\}$, $C = \{r\}$. We can reason about A, B, and C at different *levels*. For example, if the index ranges of the portions are only known symbolically, one can determine at the *portion level* that $A \supseteq B$ must hold, but no other relationships can be proven among the sets of portions. However, if we know for example that $p_l = 1, p_u = 3, q_l = 3, q_u = 5, r_l = 3, r_u = 4$, then the elements-of operator, $\tilde{}$, can be applied to the portions and to the sets thereof, to obtain $\tilde{A} = x(ia(1{:}5)), \tilde{B} = x(ia(1{:}3)), \tilde{C} = x(ia(3{:}4))$. The scope of $\tilde{}$ is extended to set operators and predicates, so we can assert at the *element level* that $A \tilde{\supseteq} B, A \tilde{\supseteq} C$. Assume furthermore that we know the values of the index array to be $ia(1{:}5) = 1, 4, 3, 1, 4$. Then the references-of operator, $\tilde{\tilde{}}$, obtains $\tilde{\tilde{A}} = \{x(1), x(3), x(4)\}, \tilde{\tilde{B}} = \{x(1), x(3), x(4)\}, \tilde{\tilde{C}} = \{x(1), x(3)\}$. With this knowledge, we conclude at the *reference level* that $A \tilde{\tilde{\supseteq}} B \tilde{\tilde{\supseteq}} C$. We can see how the set relationship predicates change over the different levels of reasoning, with $X \supseteq Y \Longrightarrow X \tilde{\supseteq} Y \Longrightarrow X \tilde{\tilde{\supseteq}} Y$.

Another interesting operation in this context is set subtraction:

- $A \setminus B = \{p, q\} \setminus \{p\} = \{q\}$, which is $\{x(1), x(3), x(4)\}$;
- $A \tilde{\setminus} B = \tilde{A} \setminus \tilde{B} = x(ia(1{:}5)) \setminus x(ia(1{:}3)) = x(ia(4{:}5))$, which is $\{x(1), x(4)\}$;
- $A \tilde{\tilde{\setminus}} B = \tilde{\tilde{A}} \setminus \tilde{\tilde{B}} = \{x(1), x(3), x(4)\} \setminus \{x(1), x(3), x(4)\} = \emptyset$.

As described in Section 5, $A \setminus B$ (and the corresponding sets at lower levels) can be viewed as a so called incremental schedule, which indicates what has to be communicated if A is needed and B is already available in local memory. We can see immediately the consequences for this incremental schedule in the example: the more we know about portions, the less we might have to communicate. Formally, for any X and Y, $X \setminus Y \supseteq X \tilde{\setminus} Y \supseteq X \tilde{\tilde{\setminus}} Y$ holds.

To aid formulating conservative equations that still offer the possibility to exploit any knowledge potentially available at compile time, we introduce some set operators that map sets of portions into sets of portions. Given some set of portions SET, we define

$SET^{*} = \{p \mid p \text{ has same } \textit{array} \text{ as some } q \in SET\}$,

$SET^{\cup} = \{p \mid SET \text{ might } \textit{affect } p\} = \{p \mid p \tilde{\tilde{\cap}} SET \neq \emptyset \text{ cannot be disproven}\}$,

$SET^{\cap} = \{p \mid SET \textit{ contains } p\} = \{p \mid p \tilde{\tilde{\subseteq}} SET \text{ can be proven}\}$,

$SET^{\circ} = \{p \mid SET \text{ might } \textit{partially touch} \text{ part of } p\} = SET^{\cup} \setminus SET^{\cap}$.

SET^{*} can be derived easily from SET by just reducing a bit matrix (array names by index sets) to a bit column (array names) using row-wise OR. From there we can conservatively approximate $SET^{\cup}$ and $SET^{\cap}$ by SET^{*} and SET, respectively, or we can employ further compile time knowledge about how portions relate to each other if available. Either way, we do not leave the portion space as given in the program; *i.e.*, we can still represent these sets with binary bit matrices. SET° can then be computed from $SET^{\cup}$ and $SET^{\cap}$ as a simple bit operation.

For example, let the portions p, q, r be defined as above, and let $D = \{q\}$. If no compile time knowledge at the element or reference level is available, then we conservatively assume that $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} = \{q\}$, and $D^{\circ} = \{p, r\}$. With knowledge at the element level, we have $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} = \{q, r\}$, and $D^{\circ} = \{p\}$. Reference level knowledge gives $D^{*} = \{p, q, r\}$, $D^{\cup} = \{p, q, r\}$, $D^{\cap} = \{p, q, r\}$, and $D^{\circ} = \emptyset$.

A point to keep in mind when reasoning about which elements are contained in which portions and how portions relate to each other is that two portions p, q might globally contain the same set of array elements of some array X, but that *locally* a given processor may see different parts of X for p and q. In this case communication must occur if for example we first define p and then use q. The important consequence is that we must apply $\tilde{}$ and $\tilde{\tilde{}}$ based on the portion share of each processor. Furthermore, the analysis has to consider the decompositions of arrays and index arrays. For example, we cannot reuse a schedule between two portions that have the same index set, but whose arrays are distributed differently. For sake of simplicity, however, we assume in this paper that all arrays are conformable.

3 The Local Flow Variables

The *local flow variables* are the components of the dataflow equations that are determined by local analysis of each loop. In the following, l stands for an arbitrary loop node and p denotes a portion `x(ia(lb:ub))`. An *occurrence* of p is either a use of p or a definition of p, and the terms *variable* or *flow variable* stand for dataflow variables.

We begin with two variables, *REF* and *DEF*, which are familiar from standard live variable analysis. A point to keep in mind, however, is that here "live" does not refer to whole arrays, but to limited portions thereof instead. Also, there may be conditionals in the loops generating the variables, which can be handled by annotating portions with

(symbolic) guards applying to whole portions or elements thereof.

For each loop l, we define

REF(l): the portions *live* on entry to l, and

DEF(l): the portions *defined* in l.

$$REF(l) = \{p \mid \textit{first} \text{ stmt containing } p \text{ in } l \text{ reads } p\},$$
$$DEF(l) = \{p \mid \text{some stmt in } l \text{ assigns to } p\}.$$

To aid the extension to reduction statements discussed in Section 6, we do not base the further development of the framework on *REF* and *DEF* directly, but replace them with *GET* and *PUT*. These variables are used to derive the portions that have to be buffered locally. We define

GET(l): the portions referenced in l from local memory (the *buffer*).

PUT(l): the portions written by l into the buffer.

BUF(l): the portions that will be buffered on exit from l.

The equations (which will be redefined in Section 6):

$$GET(l) = REF(l),$$
$$PUT(l) = DEF(l),$$
$$BUF(l) = GET(l) \cup PUT(l).$$

We also have to compute the live ranges of index sets, otherwise we might accidentally try to communicate a portion before or after the program region where its index set is available (*i.e.*, before the index set is defined or after it is overwritten with other values). We define

IND(l): the portions whose index sets may be computed (in part) by l.

KILL(l): the portions that may be made invalid by l, either because l assigns an overlapping part of the array or l reassigns the index set. GATHER operations can never be hoisted above l for these portions.

FLUSH(l): the portions that may be read by l or whose index sets may be reassigned by l. SCATTER operations can never be delayed until after l for these portions.

$$IND(l) = \{p \mid p \text{ has index set } \texttt{ia(i}_{min}\texttt{:i}_{max}\texttt{)} \text{ and } l \text{ assigns to } \texttt{ia}\},$$
$$KILL(l) = IND(l) \cup DEF^{\circ}(l),$$
$$FLUSH(l) = IND(l) \cup REF^{\circ}(l).$$

4 The Global Flow Variables

The computation of the *global flow variables* constitutes the meat of the dataflow framework. Here we actually propagate knowledge about the communication characteristics

of the loops around in the flow graph. The problems addressed here have elements from Common Subexpression Elimination, Loop Invariant Code Motion, and Dead Code Elimination. All global variables are initialized to $\emptyset$.

4.1 Fetches

The strategy for determining where to place GATHER operations is based on the following definitions:

LIVE$^{\text{any/all}}$(l): the portions that are needed in l or along any/all paths starting in l.

BUFFD(l): the portions that are already available when entering l. Here we assume that buffers are not flushed unless the data in them may be invalidated by an assignment to either the data array or the index array.

HOIST(l): the portions for which a GATHER should be hoisted ahead of l.

FETCH(l): the portions that are needed in l or in some later loop and that can be hoisted before l.

$$
\begin{aligned}
LIVE^{all}(l) &= GET(l) \cup \bigcap_{s \in succs(l)} (LIVE^{all}(s) \setminus KILL(l)),\\
LIVE^{any}(l) &= GET(l) \cup \bigcup_{s \in succs(l)} (LIVE^{any}(s) \setminus KILL(l)),\\
BUFFD(l) &= BUF(l) \cup \bigcap_{p \in preds(l)} (BUFFD(p) \setminus KILL(l)),\\
HOIST(l) &= \bigcap_{p \in preds(l)} (LIVE^{all}(p) \cup BUFFD(p)),\\
FETCH(l) &= GET(l) \cup \bigcap_{s \in succs(l)} (HOIST(s) \cap FETCH(s)).
\end{aligned}
$$

At this point, we have identified candidate locations in the program for placing GATHER's. In short, whenever a portion appears in a *FETCH*(l) set, then that portion can be gathered before l and will be used before it is assigned. The final placement will be determined by the result flow variables discussed in Section 5.

Note that we can not only distinguish the variables defined so far by whether they are local or global, but we can also classify them into either reflecting fixed properties *inherent* of the analyzed program, or being subject to *heuristics*. Furthermore, this classification can be done based either on the *definition* of the variable, *i.e.*, how it is defined in terms of other variables, or on the actual *values* of the variable.

For example, *HOIST* is currently defined so that we combine and hoist up GATHER's as much as possible, subject to the constraint that we never want to overcommunicate. If we, for example, replace the $LIVE^{all}$ in the definition of *HOIST* with $LIVE^{any}$, we could hoist up communication even further, at the expense of possibly communicating unnecessary data, but with the potential benefit of additional schedule saving. Another strategy would be to limit communication hoisting to cases where we actually reduce the size or number of messages, which can also be achieved in a straightforward manner

similar to Lazy Code Motion [8]; this might decrease the live ranges of our communication buffer with a possible savings in overall buffer storage requirements, but at the expense of reduced opportunities for hiding communication delays.

In other words, the definition of *HOIST* is a matter of heuristics, which is not the case for the other definitions so far. For other variables dependent on *HOIST* (so far, *FETCH* is the only such variable), their values become a matter of the chosen heuristics as well, but not their definition.

4.2 Stores

The high level strategy for determining where to place SCATTER operations is relatively similar to the one for placing GATHER's. Note that we do not have to scatter portions (*i.e.*, send them back to the owner) if they are used only locally, which is why we restrict our attention to GET° instead of *GET*. The definitions:

HIN$^{\text{any/all}}$(l) / HOUT$^{\text{any/all}}$(l): the portions touched by a reference on any/all of the paths starting at the entry/exit of l.

DELAY(l): the portions that should be scattered in a later loop, or are dead on exit.

STORE(l): portions that are assigned to in l, or were assigned to earlier and whose SCATTER's can be hoisted into l.

$$
\begin{aligned}
HIN^{all}(l) &= GET^\circ(l) \cup HOUT^{all}(l),\\
HOUT^{all}(l) &= \bigcap_{s \in succs(l)} HIN^{all}(s),\\
HIN^{any}(l) &= GET^\circ(l) \cup HOUT^{any}(s),\\
HOUT^{any}(l) &= \bigcup_{s \in succs(l)} HIN^{any}(l),\\
DELAY(l) &= \bigcap_{s \in succs(l)} (HOUT^{all}(s) \cup \overline{HOUT^{any}(s)}) \setminus \bigcup_{s \in succs(l)} FLUSH(s),\\
STORE(l) &= PUT(l) \cup \bigcap_{p \in preds(l)} (DELAY(p) \cap STORE(p)).
\end{aligned}
$$

Our heuristic, here defined by *DELAY*, is to combine and delay SCATTER's as much as possible, subject to the constraint that we never scatter data that are dead.

5 The Result Flow Variables

The *result flow variables* given in this section are computed after solving the equations given so far. They should accurately describe which portions have to be gathered before entering l or scattered after leaving l (possibly using reductions). This phase takes previous and succeeding loops and their communication requirements into account as well.

5.1 Fetches

Similarly to *FETCH*(l), *GATH*(l) describes which portions have to be in local memory before entering l. However, it excludes portions that must already be locally available

either by previous gathers or by previous calculations. Furthermore, we may not only exclude these available data on a portion by portion basis, but also on an element by element basis. In other words, if we know that a portion `x(ia(`i_{min}`:`i_{max}`))` is buffered, then we might not only eliminate gathers of exactly that portion, but we can also save on a gather of a potentially overlapping portion `x(ia(`j_{min}`:`j_{max}`))` by gathering only the *increment* from the first portion to the second one. For that purpose we compute *incremental schedules* using the $\tilde{\setminus}$ operator as introduced in Section 2:

$$GATH(l) = FETCH(l) \tilde{\setminus} \bigcap_{p \in preds(l)} (FETCH(p) \cup BUFFD(p)).$$

Recall that $A \tilde{\setminus} B$ contains exactly those references that appear in the portions in A but do not appear in any of the portions in B. Note that this operator, unlike the $\setminus, \cup, \cap$ used in the flow equations so far, brings us out of the fixed space of sets of portions appearing in the program text. Applying it repeatedly can lead to an explosion of the number of portions we have to be able to represent (nestings of increments of intersections of increments, etc.). Applying this operator just once, however, leads to sets that can still be represented by 3-valued "bit" vectors/matrices; in addition to *included/not included*, we also need *explicitly excluded*.

Note also that $A \tilde{\setminus} B = \emptyset$ is possible even for $A \setminus B \neq \emptyset$. This reflects for example the case where we express a mesh and its boundary as different portions of the same array; the portions are distinct, but one contains a subset of the other.

5.2 Stores

The *SCATT* variables are derived from the *STORE* variables, except that we eliminate unnecessary scatters by excluding portions that either will be scattered later or are not at least potentially live (using $\overline{HOUT^{any}}$). Again, we use the set operator $\tilde{\setminus}$ to support incremental schedules.

$$SCATT(l) = STORE(l) \tilde{\setminus} \bigcap_{s \in succs(l)} (STORE(s) \cup \overline{HOUT^{any}(s)}).$$

Note that we can still override the communication patterns obtained by global analysis for *GATH* and *SCATT* by just substituting the local counterparts *GET* and *PUT* for them, as long as this is done consistently for all loops. Furthermore, this can be done for either both variables or for just one of them, since they do not rely on each other, but merely on the loop properties.

6 Reduction Variables

As indicated earlier, the framework developed so far can be extended to take advantage of reduction statements as well. The portions appearing exclusively in reduction statements can be treated differently from other definitions and uses, since they are not necessarily brought into local memory if we use reduction operations like SCATTERADD or SCATTERMULT. However, portions appearing in different reduction operations within one

loop have to be brought into local memory, so we have to carefully separate the portions into the ones used exclusively in *ADD* reductions and the ones used only in *MULT* reductions:

$$ADD(l) = \{p \mid \text{all } q \in p^{\cup} \text{ are only added to in } l\},$$
$$MULT(l) = \{p \mid \text{all } q \in p^{\cup} \text{ are only multiplied to in } l\}.$$

We derive *RED*, the set of all portions that are used exclusively in reduction operations, and redefine *GET* and *PUT* which were introduced in Section 3:

$$RED(l) = ADD(l) \cup MULT(l),$$
$$GET(l) = REF(l) \setminus RED(l),$$
$$PUT(l) = DEF(l) \setminus RED(l).$$

The changes so far have eliminated the GATHER's and SCATTER's for portions that appear exclusively in reductions.

We now define another, separate framework, which computes *only* the SCATTER_ADD's (similarly for the other reductions). This new *ADD framework* coexists with the old non-reduction framework, which is still used to compute communication requirements for non-reduction operations. The redefined variables are:

$$GET_{ADD}(l) = REF(l) \setminus ADD(l),$$
$$FLUSH_{ADD}(l) = IND(l) \cup GET^{\circ}{}_{ADD}(l),$$
$$STORE_{ADD}(l) = ADD(l) \cup \bigcap_{p \in preds(l)} (DELAY_{ADD}(p) \cap STORE_{ADD}(p)).$$

Corresponding to these new variables, we can derive $HIN_{ADD}^{any/all}$, $HOUT_{ADD}^{any/all}$, $DELAY_{ADD}$, and $SCATT_{ADD}$ with the same equations as for the non-reduction framework. $SCATT_{ADD}$ now indicates where to place SCATTER_ADD's.

In the exact same fashion we can define a *MULT framework* by substituting *ADD* with *MULT*. Like for the non-reduction framework, we can override the result variable with their local counterpart, which is here *ADD* (*MULT* in the MULT framework). Note that the flow equations for ADD (MULT) are defined independently of other reductions. This simplifies extending the framework to other reduction operations by just adding flow variables and equations, without having to modify existing ones (except extending *RED*).

7 Example

Figure 1 shows an example program and its flow graph. In this program, we have

- four inner loops, l_1, l_2, l_3, and l_4;
- three array names, x, y, and z;
- five index sets, $s_1 = ie1(1{:}ne)$, $s_2 = ie2(1{:}ne)$, $s_3 = if1(1{:}nf)$, $s_4 = if2(1{:}nf)$, and $s_5 = identity(1{:}nn)$;

```
IF (itime ≥ 1)
  GATHER(z(if1(1:nf)), z(if2(1:nf)))
  DO t = 1, itime
    GATHER(y(ie1(1:ne)), y(ie2(1:ne)),
           y(if1(1:nf)), y(if2(1:nf)))
    DO i = 1, ne
      x(ie1(i)) = x(ie1(i)) + y(ie2(i))
      x(ie2(i)) = x(ie2(i)) + y(ie1(i))
    ENDDO
    DO j = 1, nf
      x(if1(j)) = x(if1(j)) + y(if2(j)) + z(if2(j))
      x(if2(j)) = x(if2(j)) + y(if1(j)) + z(if1(j))
    ENDDO
    DO k = 1, ne
      x(ie1(k)) = x(ie1(k)) + y(ie2(k))
      x(ie2(k)) = x(ie2(k)) + y(ie1(k))
    ENDDO
    SCATTERADD(x(ie1(1:ne)), x(ie2(1:ne)),
               x(if1(1:nf)), x(if2(1:nf)))
    DO l = 1, nn
      y(l) = x(l)
    ENDDO
  ENDDO
ENDIF
```

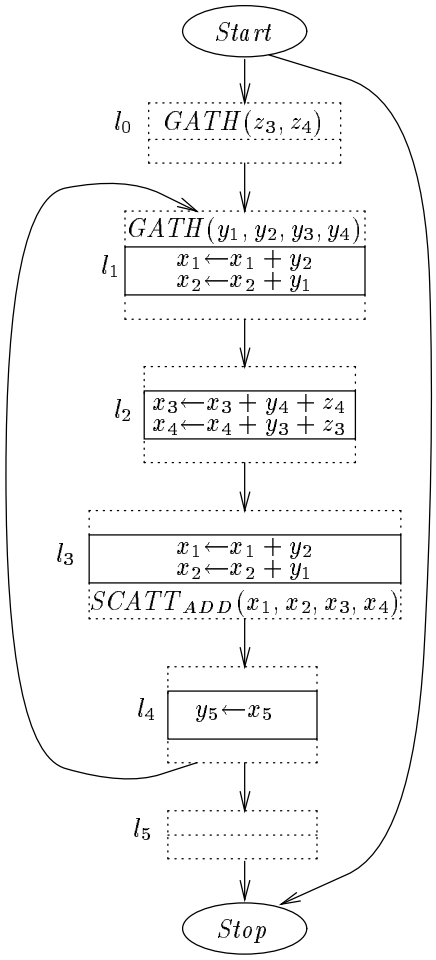

Figure 1: Example code and corresponding loop flow graph. Communication and guard statements as derived by the framework are shown in bold type.

- this spans a bit matrix of fifteen portions, $x_1 = x(s_1), x_2 = x(s_2), \ldots, z_5 = z(s_5)$, twelve of which actually occur in the program text.

Note the entry pad l_0 and the exit pad l_5 of the outer time stepping loop, which does not directly enclose any indirect references, but which contains other loops $(l_1, \ldots, l_4)$ which contain such references.

The bit matrices of some of the resulting flow variables are shown in Table 2. A matrix entry for a particular portion p and a flow variable VAR is defined as follows:

"1" – p is *included* in VAR,

"_" – p is *not included* in VAR,

"0" – p is *explicitly excluded* from VAR (as a result of the $\tilde{\setminus}$ operator; in our example, there are none such entries due to the simple control flow structure).

Arrows under the variable name indicate forward ($\Longrightarrow$) and backward ($\Longleftarrow$) problems. The result variables, *i.e.* $GATH$ and $SCATT$, determine where the GATHER and SCAT-

TER operations should be placed. If the bit representing a portion p is set in the *GATH* set, then a GATHER operation for p is placed at the *beginning* of that loop. Similarly, a set bit in the *SCATT* set results in placement of a SCATTER operation (SCATTER_ADD in the ADD framework) at the *end* of a loop. GATHER's and SCATTER's of portions with *identity* as the index array are ignored. This is valid because they represent data movement from a processor to itself.

We do not show here the optimizations needed to generate the *schedule* operations (*i.e.*, the inspectors). In general, the method is to identify the index sets used, and insert the inspectors at the birthpoints of those sets.

8 Conclusions and Future Work

Communicating the right data at the right time and place is a difficult, yet crucial task for parallelizing irregular problems. The PARTI primitives are valuable tools for managing the details of efficient data exchange. The dataflow framework presented in this paper is designed to enable the compiler to make good use of these primitives. We believe our approach to be effective for a wide range of interesting problems, and we are currently implementing the framework jointly at Rice and ICASE; the Fortran D compiler prototype will use the analysis capabilities of the framework for handling the communication needs of both irregular and regular applications.

The framework described so far gives an accurate description of which schedules are needed where. This paper did not discuss the generation of the inspectors, which is equally important in optimizing these codes. We currently use the simple heuristic of generating schedules as soon as possible, *i.e.*, as soon as the necessary index arrays are available, and are investigating a dataflow framework to handle this task.

There is still room for improvements and generalizations of the underlying theory. For example, the loop flow graph representation as described in Subsection 2.1 seems to be convenient and well suited for a first implementation, but has some unsatisfactory theoretical aspects, such as the ad hoc distinction between loops which directly enclose indirect references and other loops. A more general approach could be based on basic blocks combined with *slicing* [6]; we are already exploring this alternative in our implementation.

Another extension is to divide the communication calls into matching send/receive pairs, placing these components to overlap communication and computation when possible. Send operations could be performed where the operations are placed now, while receives could be delayed until the data are actually used.

Furthermore, one might consider relaxing the static ownership concept for data accessed via indirection arrays. For example, it might occur that a processor p computes some data which are owned by processor q, but the next use of the data is on processor r. Our framework would generate two communications calls in this situation, but a richer

Local Variables

	l_0	l_1	l_2	l_3	l_4	l_5
GET	____	____	____	____	___1	____
	____	11__	__11_	11__	____	____
	____	____	__11_	____	____	____
PUT	____	____	____	____	____	____
	____	____	____	____	___1	____
	____	____	____	____	____	____
KILL	____	__111	11__1	__111	____	____
	____	____	____	____	1111_	____
	____	____	____	____	____	____

Flow and Result Variables

	l_0	l_1	l_2	l_3	l_4	l_5
$LIVE^{all}$	____	____	____	____	___1	____
	1111_	1111_	1111_	11__	____	____
$\Longleftarrow$	__11_	__11_	__11_	____	____	____
$LIVE^{any}$	____	____	____	____	___1	____
	1111_	1111_	1111_	11__	____	____
$\Longleftarrow$	__11_	__11_	__11_	__11_	__11_	____
BUFFD	____	____	____	____	___1	___1
	____	11__	1111_	1111_	___1	___1
$\Longrightarrow$	____	____	__11_	__11_	__11_	__11_
FETCH	____	____	____	____	___1	____
	____	1111_	1111_	11__	____	____
$\Longleftarrow$	__11_	__11_	__11_	____	____	____
GATH	____	____	____	____	___1	____
	____	1111_	____	____	____	____
$\Longrightarrow$	__11_	____	____	____	____	____

ADD Framework Variables

	l_0	l_1	l_2	l_3	l_4	l_5
$HOUT^{all}_{ADD}$	11111	11111	__111	____	____	____
$\Longleftarrow$	11__1	11__1	____	____	____	____
	1111_	1111_	1111_	1111_	1111_	____
$HOUT^{any}_{ADD}$	11111	11111	11111	11111	11111	____
$\Longleftarrow$	11__1	11__1	11__1	11__1	11__1	____
	____	11__	1111_	1111_	____	____
$STORE_{ADD}$	____	____	____	____	____	____
$\Longrightarrow$	____	____	____	____	____	____
	____	____	____	1111_	____	____
$SCATT_{ADD}$	____	____	____	____	____	____
$\Longleftarrow$	____	____	____	____	____	____

Table 2: Selected flow variables for example code.

theoretical framework might generate a single optimized communication phase.

Acknowledgements

Von Hanxleden was supported by an IBM graduate fellowship award. Support for Kennedy and Koelbel was provided by the Texas Governor's Energy Office under Contract #1059. Das and Saltz were supported by National Aeronautics and Space Administration under NASA contract NAS1-18605 while they were in residence at ICASE, NASA Langley Research Center.

References

[1] T. W. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[2] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*. AIAA, January 1992.

[3] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems — data copy reuse and runtime partitioning. ICASE Report 91-73, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1991.

[4] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[5] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[7] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):159–171, January 1976.

[8] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[9] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[10] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[11] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.