NASA Contractor Report 189664
ICASE Report No. 92–22

ICASE

COMPILER ANALYSIS FOR IRREGULAR PROBLEMS IN FORTRAN D

Reinhard von Hanxleden
Ken Kennedy
Charles Koelbel
Raja Das
Joel Saltz

Contract No. NAS1–18605
June 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

COMPILER ANALYSIS FOR IRREGULAR PROBLEMS IN FORTRAN D

Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel
Center for Research on Parallel Computation
Rice University
Houston, TX 77251

Raja Das[1] and Joel Saltz[1]
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

ABSTRACT

We developed a dataflow framework which provides a basis for rigorously defining strategies to make use of runtime preprocessing methods for distributed memory multiprocessors.

In many programs, several loops access the same off-processor memory locations. Our runtime support gives us a mechanism for tracking and reusing copies of off-processor data. A key aspect of our compiler analysis strategy is to determine when it is safe to reuse copies of off-processor data. Another crucial function of the compiler analysis is to identify situations which allow runtime preprocessing overheads to be amortized. This dataflow analysis will make it possible to effectively use the results of interprocedural analysis in our efforts to reduce interprocessor communication and the need for runtime preprocessing.

[1]Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1 Introduction

We present a dataflow framework that can be employed to systematically make use of runtime preprocessing methods aimed at loops in which some array references are made through a level of indirection. The dataflow framework we present here pertains to collections of loops with no loop-carried data dependences or with only accumulation type dependencies. Such loops are often referred to as *data-parallel* loops, and are the primary target of the Fortran D compiler [4]. The type of irregular loops we are trying to handle are typically found in unstructured mesh explicit and multigrid solvers, molecular dynamics codes, and some sparse iterative linear systems solvers.

In distributed memory machines, large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called *distributed arrays*. Long term storage of distributed array data is assigned to specific memory locations in the distributed machine. In many irregular problems, we can reduce the amount of data to be communicated by using a partitioning algorithm that individually assigns each array element to a specific processor. Furthermore, these machines often have a non-trivial communications latency or startup cost. Therefore, efficiency demands that information to be transmitted should be collected into relatively large messages; this in turn implies that the elements to be sent and received by each processor should be precomputed. In irregular problems, the communications pattern depends on the input data, typically because of some indirection in the code. In this case, it is not possible to predict at compile time what data must be prefetched.

We treat this lack of information by transforming the original parallel loop into two constructs called an inspector and executor [6, 7]. During program execution, the *inspector* examines the data references made by a processor, and calculates what off-processor data needs to be fetched and where that data will be stored once it is received. The *executor* loop then uses the information from the inspector to implement the actual computation. The Fortran D compiler now under development at Rice University performs these tasks by calls to the PARTI library built at ICASE.

The PARTI primitives (Parallel Automated Runtime Toolkit at ICASE) were designed to ease the implementation of irregular computational problems on parallel architecture machines by relieving the user or compiler writer of having to deal with many low-level issues. These procedures

1. Coordinate interprocessor data movement,
2. Manage the storage of and access to copies of off-processor data, and
3. Support a shared name space by building a distributed translation table [9] to store the local address and processor number for each distributed array element.

This functionality can be used directly to generate inspector/executor pairs. Each inspector produces a communications *schedule*, which is essentially a pattern of communication for gathering or scattering data. Hash tables are used to avoid repeatedly communicating the same array elements. The executor has embedded PARTI primitives to gather or scatter data. The primitives are designed so that the final parallel code remains as close as possible to the

original sequential code. The primitives issue instructions to gather, scatter or accumulate (e.g. scatter followed by add) data according to a specified schedule. The latency or start-up cost is reduced by packing several small messages with the same destinations into one large message. Significant work has gone into optimizing the gather, scatter and accumulation communication routines for the iPSC/860. It is not the purpose of this paper to describe the design and implementation of PARTI in great detail; information on this can be found elsewhere [1].

Our runtime support makes it possible to track and reuse off-processor data copies [2]. We generate *incremental* communications schedules to obtain only those off-processor data not requested by a given set of pre-existing schedules. This gives us the runtime support we need to combine and hoist gather, scatter and accumulate procedures. Removal of duplicates is achieved by using a hash table. In a mesh solver, for example, the off-processor data to be accessed by the edge schedule is first hashed using a simple hash function. Next all the data to be accessed during the face_loop is hashed. At this point the information that exists in the hash table allows us to remove all the duplicates and form the incremental schedule.

The data flow framework to be described here aims at providing good information about what data are needed at which points in the code, along with information about what live off-processor data are available. At *compile time*, we compute global flow information about the communication characteristics of the loops around a flow graph. This framework bears similarities to classical techniques such as common subexpression elimination, loop invariant code motion, and dead code elimination. The framework provides a basis for determining at compile time

- Where communication schedules are to be generated,
- Where gather, scatter, and accumulate operations are to be placed, and
- When incremental schedules may be employed.

In our data flow analysis, some of the variables reflect inherent properties of the analyzed program, while others calculate the results of heuristics we employ in order to 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 and hoisting gather, scatter and accumulate procedures.

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 data flow variables for a program example, which is a simplified version of a mesh solver for which Section 8 gives concrete experimental results illustrating the effect of exploiting data flow information. Section 9 contains concluding remarks.

2 Basics of the Framework

This section describes the scope of the framework developed in this paper and defines some concepts used in later sections.

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 $\langle array \rangle(\langle index\_array \rangle(\langle loop$ 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 which are never referenced indirectly are assumed to be analyzed using other methods [3] 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 which are accessed indirectly. We assume that each reference r to some $v \in V$ is contained in some loop(s). Let L be the set of loops which *directly* enclose an occurrence of some $v \in V$. We assume that no $l \in L$ encloses any other $m \in L$. One set of data flow 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 $l' \in L$. Furthermore, we assume l_{entry} (l_{exit}) to be executed before (after) l iff l has at least one iteration. 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 which does not directly contain any irregular array references but which 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 elements 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 [3]. Finally, this paper only discusses the case where the summarized loops have no data dependences, except for commutative and associative reductions which are handled specially.

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 allows 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 bit vectors, each bit representing one array portion. The length of these bit vectors is bounded by the number of indirect array references (*i.e.*, it is linear in program size), and all equations given here are *rapid* [5]. Therefore, using bit vectors for the analysis gives us 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.

We assume that all indirect array references are identified in a previous pass over the program text and construct bit vectors/matrices accordingly. For the analysis we also assume that a (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, $\tilde{}$, obtains

$$\tilde{\tilde{A}} = x(1,3,4),\ \tilde{\tilde{B}} = x(1,3,4),\ \tilde{\tilde{C}} = x(1,3).$$

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

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,

$$X \setminus Y \ \supseteq\ X \tilde{\setminus} Y \ \supseteq\ X \tilde{\tilde{\setminus}} Y.$$

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

$$\begin{aligned}
SET^{*} &= \{p \mid p \text{ has same } \textit{array} \text{ as some } q \in SET\},\\
SET^{\cup} &= \{p \mid SET \text{ might } \textit{affect } p\}\\
&= \{p \mid p \tilde{\tilde{\cap}} SET \neq \emptyset \text{ cannot be disproven}\}\\
&\subseteq SET^{*},\\
SET^{\sqcap} &= \{p \mid SET \textit{ contains } p\}\\
&= \{p \mid p \tilde{\tilde{\subseteq}} SET \text{ can be proven}\}\\
&\supseteq SET,\\
SET^{\circ} &= \{p \mid SET \text{ might } \textit{partially touch} \text{ part of } p\}\\
&= SET^{\cup} \setminus SET^{\sqcap}.
\end{aligned}$$

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 $SET^\cup$, $SET^\cap$, and SET° directly, 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.

For example, let the portions p, q, r be defined as above, and let $D = \{q\}$. Assuming no compile time knowledge at the element or reference level, we can 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 sees different parts of X for p and q. (This applies to lhs occurrences as well, since we apply the *owner computes rule* based on index array ownerships, *not* on data array ownerships; otherwise we would not need a SCATTER operation). In this case there has communication to 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 share of each processor.

Furthermore, we have to keep different distributions of arrays and index arrays in mind for the analysis. For example, we cannot reuse a schedule between two portions which 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

We define the *local flow variables* to be the components of the data flow equations which are determined by local analysis of each loop. In the following,

- l stands for an arbitrary loop,
- p denotes a portion `x(ia(lb:ub))`,
- an *occurrence* of p is either a use of p or a definition of p, and
- *variable* or *flow variable* stand for data flow 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.

Formally:

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

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 which 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, and

BUF(l): the portions which 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 the index set of that portion 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.

Formally:

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

4 The Global Flow Variables

The computation of the *global flow variables* constitutes the meat of the data flow 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. As already mentioned, all equations given here are *rapid*, so we can expect to solve them efficiently using simple iterative techniques. 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 which are needed in l or in all/any of the following loops.

BUFFD(l): the portions which are already available when entering l. Here we assume that buffers are not flushed unless the data in them may be invalid, because either the data array or the index array has been assigned to;

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

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

The equations:

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

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 GATHER'ed 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 (even if that might be advantageous in some cases, for example in saving schedules). 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. 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 which should be scattered in a later loop, or which are dead on exit.

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

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

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 which 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). Here we want to take previous and succeeding loops and their communication requirements into account as well.

5.1 Fetches

Similarly to *FETCH*, *GATH*(l) describes which portions have to be in local memory before entering l. However, it excludes portions which 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 $\texttt{x(ia(i}_{min}\texttt{:i}_{max}\texttt{))}$ 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 $\texttt{x(ia(j}_{min}\texttt{:j}_{max}\texttt{))}$ 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; recall that $A \tilde{\setminus} B$ contains exactly those references which 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, and applying it repeatedly can lead to an explosion of the number 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 which 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.

The equation:

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

5.2 Stores

The *SCATT* variables are derived from the *STORE* variables, except that we eliminate unnecessary scatters by excluding portions which either will be scattered later, or which 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. Furthermore, this can be done for either both variables or just one of them, since they do **not** rely on each other, but merely on the loop properties.

5.3 Schedules

The framework described so far gives an accurate description of which schedules are needed where. Critical for the overall cost associated with our communications is also the *generation* of these schedules, in particular *where* the schedules are generated. However, once we know the communication requirements, schedule computation placement appears to be relatively

straightforward. Therefore, we currently use the simple heuristic of generating schedules as soon as possible, *i.e.*, as soon as the necessary index arrays are available. This seems to work well in the codes we have considered so far.

6 Reduction Variables

As indicated earlier, the framework developed so far can be extended to take advantage of reduction statements as well. The portions exclusively appearing in reduction statements can be treated differently from other defs 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:

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

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

$$\begin{aligned} RED(l) &= ADD(l) \cup MULT(l), \\ GET(l) &= REF(l) \setminus RED(l), \\ PUT(l) &= DEF(l) \setminus RED(l). \end{aligned}$$

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

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

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

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.

Like for the non-reduction framework, we can override the result variable selectively with their local counterpart, which is here *ADD*. Note that the flow equations for ADD 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*).

```
GATHER(z(nf1(1:nf)),z(nf2(1:nf)))
do iii = 1, itime
  GATHER(y(ie1(1:ne)),y(ie2(1:ne)),
       y(nf1(1:nf)),y(nf2(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(nf1(j)) = x(nf1(j)) + y(nf2(j)) + z(nf2(j))
    x(nf2(j)) = x(nf2(j)) + y(nf1(j)) + z(nf1(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(nf1(1:nf)),x(nf2(1:nf)))
  do l = 1, nn
    y(l) = x(l)
  enddo
enddo
```

Figure 1: Example code, communication is already inserted as derived by the framework.

7 Example

Figure 1 shows an example code. In this program, we have

- four inner loops, l_1, l_2, l_3, and l_4;
- one entry and one exit pad, l_0 and l_5;
- 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 $s_5 = identity(1:nn)$;
- 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.

The corresponding flow graph is shown in Figure 2.

The bit matrices of the resulting local flow variables are shown in Figure 3. 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,

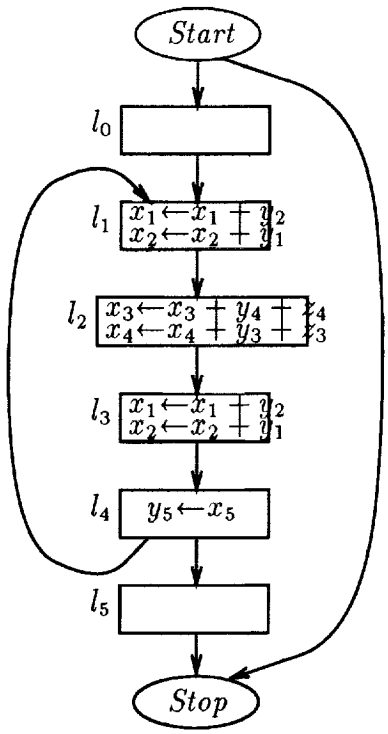

Figure 2: Flow graph for example code.

"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).

Figure 4 shows the global and result variables. Figure 5 shows the variables for the ADD framework. The result variables, *i.e.*$GATH$ and $SCATT$, determine where the GATHER and SCATTER operations should be placed. If the bit representing a portion is set in the $GATH$ set, then a GATHER operation for that portion 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. The first step can be done by inspection, while the second is a simple application of reaching definition analysis.

8 Experimental Results

We summarize the results of some of the experiments we have carried out to evaluate the performance impact of our optimizations. The experiments employed an explicit unstructured mesh solver of the three dimensional Euler equations which comprise a non-linear system of

	l_0	l_1	l_2	l_3	l_4	l_5
REF	_____	11___	__11_	11___	____1	_____
	_____	11___	__11_	11___	_____	_____
	_____	_____	__11_	_____	_____	_____
DEF	_____	11___	__11_	11___	_____	_____
	_____	_____	_____	_____	____1	_____
	_____	_____	_____	_____	_____	_____
ADD	_____	11___	__11_	11___	_____	_____
	_____	_____	_____	_____	_____	_____
	_____	_____	_____	_____	_____	_____
RED	_____	11___	__11_	11___	_____	_____
	_____	_____	_____	_____	_____	_____
	_____	_____	_____	_____	_____	_____
GET	_____	_____	_____	_____	____1	_____
	_____	11___	__11_	11___	_____	_____
	_____	_____	__11_	_____	_____	_____
PUT	_____	_____	_____	_____	_____	_____
	_____	_____	_____	_____	____1	_____
	_____	_____	_____	_____	_____	_____
BUF	_____	_____	_____	_____	____1	_____
	_____	11___	__11_	11___	____1	_____
	_____	_____	__11_	_____	_____	_____
KILL	_____	__111	11__1	__111	_____	_____
	_____	_____	_____	_____	1111_	_____
	_____	_____	_____	_____	_____	_____
FLUSH	_____	__111	11__1	__111	1111_	_____
	_____	__111	11__1	__111	_____	_____
	_____	_____	11__1	_____	_____	_____

Figure 3: Local flow variables for example code.

five differential equations. The calculation consists of a sequence of loops over edges, boundary faces and nodes of an unstructured mesh. The code was originally developed by Dimitri Mavriplis. The program was ported to the Touchstone Delta using Parti primitives [1, 2] and the code was run to simulate a variety of aircraft configurations under a range of test conditions. While the port was carried out by hand, the strategy used to place the PARTI primitives was the same as the strategies that would result from our dataflow framework. We executed a number of different versions of the parallel Euler solver to give an idea of how the code generated using the dataflow framework would affect performance.

The reader should note that the simple test loops presented in Section 7 were for ease of exposition, our experimental work involved a full unstructured explicit Euler solver. The example code shown in Figure 1 depicts loops which are motivated by the actual Euler solver. The loop structure of the example code can be derived from the actual solver by inlining the function calls. The main loop in the example is analogous to a time-stepping loop. Arrays

	l_0	l_1	l_2	l_3	l_4	l_5
$LIVE^{all}$	_____ 1111_ __11_	_____ 1111_ __11_	_____ 1111_ __11_	_____ 11___ _____	____1 _____ _____	_____ _____ _____
$LIVE^{any}$	_____ 1111_ __11_	_____ 1111_ __11_	_____ 1111_ __11_	_____ 11___ __11_	____1 _____ __11_	_____ _____ _____
$BUFFD$	_____ _____ _____	_____ 11___ _____	_____ 1111_ __11_	_____ 1111_ __11_	____1 ____1 __11_	____1 ____1 __11_
$HOIST$	_____ _____ _____	_____ _____ __11_	_____ 1111_ __11_	_____ 1111_ __11_	_____ 1111_ __11_	____1 ____1 __11_
$FETCH$	_____ _____ __11_	_____ 1111_ __11_	_____ 1111_ __11_	_____ 11___ _____	____1 _____ _____	_____ _____ _____
$GATH$	_____ _____ __11_	_____ 1111_ _____	_____ _____ _____	_____ _____ _____	____1 _____ _____	_____ _____ _____
$HOUT^{all}$	1111_ 11111 11__1	1111_ 11111 11__1	1111_ __111 _____	1111_ _____ _____	_____ _____ _____	_____ _____ _____
$HOUT^{any}$	1111_ 11111 11__1	1111_ 11111 11__1	1111_ 11111 11__1	1111_ 11111 11__1	1111_ 11111 11__1	_____ _____ _____
$DELAY$	11___ 11___ 11111	__11_ __11_ __11_	11___ _____ __11_	____1 _____ __11_	11___ 11___ 11111	11111 11111 11111
$STORE$	_____ _____ _____	_____ _____ _____	_____ _____ _____	_____ _____ _____	_____ ____1 _____	_____ _____ _____
$SCATT$	_____ _____ _____	_____ _____ _____	_____ _____ _____	_____ _____ _____	_____ ____1 _____	_____ _____ _____

Figure 4: Global flow variables and result flow variables for example code.

	l_0	l_1	l_2	l_3	l_4	l_5
GET_{ADD}	_____	_____	_____	_____	____1	_____
	_____	11___	__11_	11___	_____	_____
	_____	_____	__11_	_____	_____	_____
$FLUSH_{ADD}$	_____	_____	_____	_____	1111_	_____
	_____	__111	11__1	__111	_____	_____
	_____	_____	11__1	_____	_____	_____
$HOUT^{all}_{ADD}$	1111_	1111_	1111_	1111_	_____	_____
	11111	11111	__111	_____	_____	_____
	11__1	11__1	_____	_____	_____	_____
$HOUT^{any}_{ADD}$	1111_	1111_	1111_	1111_	1111_	_____
	11111	11111	11111	11111	11111	_____
	11__1	11__1	11__1	11__1	11__1	_____
$DELAY_{ADD}$	11111	11111	11111	____1	11___	11111
	11___	__11_	_____	_____	11___	11111
	11111	__11_	__11_	__11_	11111	11111
$STORE_{ADD}$	_____	11___	1111_	1111_	_____	_____
	_____	_____	_____	_____	_____	_____
	_____	_____	_____	_____	_____	_____
$SCATT_{ADD}$	_____	_____	_____	1111_	_____	_____
	_____	_____	_____	_____	_____	_____
	_____	_____	_____	_____	_____	_____

Figure 5: Flow variables for ADD framework of example code.

in the actual Euler solver required GATHER'ing and SCATTER'ing at different levels, as illustrated by the y and z arrays in the example. Unlike our example, the arrays in the actual code are multidimensional; incorporating this into our framework only requires treating the additional dimension as a regularly-accessed array. (This dimension has a size of five, and all elements in that dimension are set together.) There are three different kinds of loops present: two over the edges (l_1 and l_3), one over the boundary faces (l_2) and one over the nodes (l_4). These are typical of the variety of loop types found in the actual code. In summary, Figure 1 abstracts many of the complex parts of the actual code (such as control flow and irregular computations), while ignoring the straightforward parts (such as short vectors in other dimensions).

The test case we report here involves the computation of a highly resolved flow over a three-dimensional aircraft configuration. The mesh contained a total of 804,056 points and approximately 4.5 million tetrahedra. We believe this to be the largest unstructured grid Euler solution attempted to date. In Figure 6, we depict one of the meshes used in our experimentation (we do not show the 804K mesh due to printing and resolution limitations). For this case, the freestream Mach number is 0.768 and the incidence is 1.16 degrees. We employed the recursive spectral partitioning algorithm to carry out partitioning [8, 10]. Partitioning was performed on a sequential machine as a preprocessing operation.

We present timings that result from four different implementations of the Euler solver. In

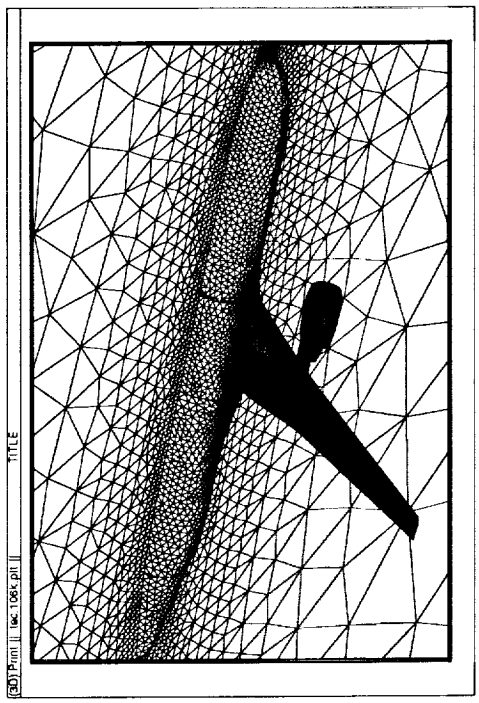

Figure 6: Coarse Unstructured Mesh about an Aircraft Configuration with Single Nacelle; Number of Points = 106,064, Number of Tetrahedra = 575,986.

two cases, we employ incremental scheduling and aggressively combine and hoist gather and accumulate procedure calls. In the two other cases, we do not make use of the knowledge we would obtain through global dataflow analysis in placement of gather, scatter and accumulate procedure calls. In these versions we place gather procedures immediately before loops that contain irregular array references. The use of incremental scheduling leads to a large amount of live data reuse. For instance, one step of Runge Kutta integration in the experimental code uses the flow variables in a sequence of three loops over edges followed by a loop over boundary faces. The flow variables are only updated at the end of the entire integration, rather than after each loop. We can obtain all of the off-processor flow variables needed at the beginning of the step.

The indirection arrays (in Figure 1 the arrays `ie1, ie2, if1, if2`), with which we form our schedules, usually do not get written into in non-adaptive calculations. In such situations one can form the communication schedules before any actual computation begins, *i.e.*, in the preprocessing stage. In our experimental code, the edges and the boundary faces of the mesh are fixed throughout the computation. Hence, we can very easily move the formation of the schedule outside the main iteration loop, as the indirection arrays are live throughout loop body. The only way the generation of the schedules (*i.e.*, the inspectors) can be moved outside a block of code is if it can be ascertained that the indirection array, with which the schedule is being formed, is live inside the whole block. This can only be determined through global dataflow analysis.

We present timings that result from generating schedules as soon as the necessary index arrays are generated and timings that result from generating a new schedule for each

Scheduling Method	Time per Iteration (seconds)	Performance (Mflops)	Preprocessing Time (seconds)
Not incremental, inside of main loop	6.91	573	273
Not incremental, outside of main loop	4.18	947	2.73
Incremental, inside of main loop	5.64	702	299
Incremental, outside of main loop	2.65	1496	2.99

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

invocation of each gather, scatter or accumulate procedure call.

Table 1 depicts:

- The time required per iteration,
- The computational rate in Mflops, and
- The preprocessing time needed per iteration for generating all communication schedules.

Comparing the times for scheduling inside the loop and scheduling only once in the computation, we see performance improvements ranging from 65% to over 100% The preprocessing time increases only modestly when we use incremental scheduling and is roughly equal to the cost of a single parallelized iteration. Once we have hoisted schedule generation outside the main iteration loop, use of incremental scheduling leads to an additional 58% reduction in total time.

9 Conclusions

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 the first part of the problem, namely for determining where to find which data and for efficient data exchange. The dataflow framework presented in this paper is designed for attacking the second part of the problem, namely enabling the compiler to make good use of these primitives without further advice by the user. We believe our approach to be effective for a wide range of interesting problems, as illustrated for an explicit unstructured mesh solver.

References

[1] 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.

[2] 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.

[3] 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.

[4] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

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

[6] 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.

[7] 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.

[8] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11:430–452, 1990.

[9] 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(4):303–312, April 1990.

[10] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications.* Permagon Press, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1992	Contractor Report

4. TITLE AND SUBTITLE

COMPILER ANALYSIS FOR IRREGULAR PROBLEMS IN FORTRAN D

5. FUNDING NUMBERS

C NAS1-18605

WU 505-90-52-01

6. AUTHOR(S)

Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel Saltz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER

ICASE Report No. 92-22

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR-189664
ICASE Report No. 92-22

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: Michael F. Card
Final Report

Submitted to 5th Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, Aug. 92

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited

Subject Category 59, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

We developed a dataflow framework which provides a basis for rigorously defining strategies to make use of runtime preprocessing methods for distributed memory multiprocessors.

In many programs, several loops access the same off-processor memory locations. Our runtime support gives us a mechanism for tracking and reusing copies of off-processor data. A key aspect of our compiler analysis strategy is to determine when it is safe to reuse copies of off-processor data. Another crucial function of the compiler analysis is to identify situations which allow runtime preprocessing overheads to be amortized. This dataflow analysis will make it possible to effectively use the results of interprocedural analysis in our efforts to reduce interprocessor communication and the need for runtime preprocessing.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parti, high performance fortran, Fortran D, sparse, irregular compiler	21
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102