Slicing Analysis and Indirect Access to Distributed Arrays

Raja Das,
Joel Saltz,
Reinhard von Hanxleden

CRPC-TR93319-S
June 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Slicing Analysis and Indirect Accesses to Distributed Arrays[*][†]

Raja Das[‡] Joel Saltz[†] Reinhard von Hanxleden[§]

Abstract

An increasing fraction of the applications targeted by parallel computers makes heavy use of indirection arrays for indexing data arrays. Such *irregular access patterns* make it difficult for a compiler to generate efficient parallel code. A limitation of existing techniques addressing this problem is that they are only applicable for a single level of indirection. However, many codes using sparse data structures access their data through *multiple levels of indirection.*

This paper presents a method for transforming programs using multiple levels of indirection into programs with at most one level of indirection, thereby broadening the range of applications that a compiler can parallelize efficiently. A central concept of our algorithm is to perform *program slicing* on the subscript expressions of the indirect array accesses. Such slices peel off the levels of indirection, one by one, and create opportunities for aggregated data prefetching in between. A *slice graph* eliminates redundant preprocessing and gives an ordering in which to compute the slices. We present our work in the context of High Performance Fortran, an implementation in a Fortran D prototype compiler is in progress.

1 Introduction

In distributed memory machines, large data arrays are frequently partitioned between local memories of processors. We will refer to such partitioned data arrays as *distributed arrays.* Recently there have been major efforts in developing programming language and compiler support for distributed memory machines. Based on initial projects like Fortran D [5, 10] and Vienna Fortran [1, 17], the High Performance Fortran Forum has proposed the first version of High Performance Fortran (HPF) [4], which can be thought of as Fortran 90 enhanced with data distribution annotations. Long term storage of distributed array data is assigned to specific memory locations in the distributed machine. HPF offers the promise of significantly easing the task of programming distributed memory machines and making programs independent of a single machine architecture. Current prototypes of compilers for HPF-like languages produce Single Program Multiple Data (SPMD) code with message passing and/or runtime communication primitives.

Reducing communication costs is crucial in achieving good performance on applications [7, 9]. While current systems like the Fortran D project [10] and the Vienna Fortran Compilation system [1] have implemented a number of optimizations for reducing communication costs (like message blocking, collective communication, message coalescing and aggregation), these optimizations have been developed mostly in the context of regular problems (*i.e.*, for codes having only regular data access patterns). Special effort is required in developing compiler and runtime support for applications that do not have such regular data access patterns.

In irregular problems, communication patterns depend on data values not known at compile time, typically because of some indirection in the code. Indirection patterns have to be preprocessed, and the elements to be sent and received by each processor must be precomputed in order to

- reduce the volume of communication,
- reduce the number of messages, and
- prefetch off processor data to hide communication latencies.

In earlier work, we have developed runtime support, analysis techniques, and compiler prototypes designed to handle loops where distributed arrays are accessed through a single level of indirection [2, 6, 11, 15]. Compiler transformations can handle loops with indirectly referenced distributed arrays by transforming

[*]From the *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[†]This work was supported by ARPA under contract No. NAG-1-1485 and by NSF under grant No. ASC 9213821. Additional Support for von Hanxleden was provided by an IBM graduate fellowship award.

[‡]Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, College Park, MD 20742. E-mail: {raja|saltz}@cs.umd.edu.

[§]Center for Research on Parallel Computation, Rice University, Houston, TX 77251. E-mail: reinhard@rice.edu.

```
          SUBROUTINE simple(x, y, col, m, n)

          INTEGER i, m, n, col(m)
          REAL x(n), y(n)
   !HPF$  DISTRIBUTE(BLOCK) :: col, x, y

   !HPF$  EXECUTE (i) ON_HOME x(i)
K1        FORALL i = 1, n
K2           x(i) = x(i) + y(col(i))
K3        ENDFORALL
K4        END
```

Figure 1: Kernel with single level of indirection.

```
          SUBROUTINE CSR(x, y, col, ija, m, n)

          INTEGER i, j, m, n, col(m), ija(n)
          REAL x(n), y(n)
   !HPF$DISTRIBUTE(BLOCK) :: col, ija, x, y

   !HPF$EXECUTE (i) ON_HOME x(i)
R1        FORALL i = 1, n
R2           x(i) = 0
R3           DO j = ija(i) + 1, ija(i + 1)
R4              x(i) = x(i) + y(col(j))
R5           ENDDO
R6        ENDFORALL
R7        END
```

Figure 2: CSR kernel – original version.

the loops into two constructs called an inspector and executor [12]. During program execution, the *inspector* examines the data references made by a processor and calculates what off-processor data need to be fetched and where these data will be stored once they are received. The *executor* loop then uses the information from the inspector to implement the actual computation.

An example for the class of kernels that can be handled by the techniques developed so far is the irregular kernel in Figure 1. In this example, data arrays *col*, x, and y are block distributed between processors. The i-loop iterations are partitioned using the HPF-directive **ON_HOME**, which in this case is equivalent to the *owner computes rule* that assigns the computation of an assignment statement to the processor that stores the left hand side reference. A single level of indirection arises because the data array y is indexed using the array *col* in statement K2.

While we can handle such simple indirection patterns, many application codes have code segments and loops with more complex access functions that go beyond the scope of current compiling techniques. In many cases, a chain of distributed array indexing is set up where values stored in one distributed array are used to determine the indexing pattern of another distributed array, which in turn determines the indexing pattern of a third distributed array. Such loops with multiple levels of indirection are very common and appear, for example, in unstructured and adaptive applications codes associated with particle methods, molecular dynamics, sparse linear solvers, and in some unstructured mesh CFD solvers.

This paper develops techniques that can be used by compilers to transform loops with array accesses involving more than a single level of indirection into loops where array references are made through at most one level of indirection. We present this transformation technique in the context of distributed memory machines and therefore often refer to prefetching as "communication" or "message blocking." However, our method is likely to be useful on any architecture where it is profitable to prefetch data between different levels of a memory hierarchy.

The rest of this paper is organized as follows. Section 2 gives an overview of our technique by transforming an example code that shows two levels of indirection. Section 3 introduces some terminology that used in Section 4, which gives a formal description of our algorithms and illustrates how the transformation shown in Section 2 was derived. Section 5 concludes with a brief discussion and an overview of the status of our implementation.

2 Example Transformation

This section illustrates the effect of applying our transformation to the HPF subroutine *CSR* shown in Figure 2. This code is based on a sparse matrix vector multiply kernel and uses the Compressed Sparse Row format [13]. An n by n sparse matrix is multiplied by an n-element array x, the results are stored in an n-element array y. The matrix values are all assumed to be equal to zero or one. The columns associated with non-zero entries in row i are specified by $col(j)$, where $ija(i)+1 \leq j \leq ija(i+1)$. For simplicity all distributed arrays are distributed blockwise in this example; our techniques apply equally well to other, potentially irregular decompositions. The indexing of

y by array *col* causes a first level of indirection. The dependence of the loop bounds of the inner j-loop on the distributed array *ija* causes an additional level of indirection. This double indirection becomes clear when rewriting the computation as

$$x(i) = \sum y(col(ija(i) + 1 : ija(i + 1)))$$

for $i = 1 \ldots n$.

All references to the distributed array x are indexed by the loop induction variable i. The HPF ON_HOME construct partitions the iteration space of the FORALL loop such that iteration i is performed on the processor that owns $x(i)$, so there is no communication required for referencing x. For the other three arrays, *ija*, *col*, and y, data communication is required. As already mentioned, keeping the total number of these communication steps down is key to high performance on a distributed memory machine. Therefore, we want to perform only a small number of aggregate prefetch operations, instead of communicating each reference individually. This requires a significant amount of preprocessing to determine what data need to be prefetched and in what order that has to be done. We will transform the code so that the compiler runtime support will have access to the subscripts of all elements of *ija*, *col*, and y that need to be prefetched from other processors. This information makes it possible to carry out the communications optimizations described in Section 1; *i.e.*, to reduce the volume of communication, reduce the number of messages and to prefetch off-processor data to hide communication latencies.

The transformed version of subroutine *CSR* is shown in Figure 3. For ease of presentation, we use a variation of HPF that contains additional directives **BEGIN LOCAL** and **END LOCAL** indicating *local variables*. These variables do not reside in the global name space inhabited by the other HPF variables, but instead they exist independently in the local name space of each processor. In strict HPF, such variables could be emulated by either adding another dimension of size *n$proc* (the total number of processors) and referencing this dimension with *my$proc* (the id of each processor), or by manipulating them only through so called extrinsic functions. Except for these local variables, the whole code is presented in global name space, and for simplicity it is here assumed that all global to local address translations will be handled by the HPF compiler. Note, however, that this index translation in the presence of indirect addressing and perhaps further complications like irregular decompositions is a nontrivial task; the code actually generated by our implementation assists in this process.

In our example, the distributed array *ija* is distributed conformable to the array x. Since the reference $ija(i)$ in statement R3 occurs in a FORALL loop whose iteration space is aligned to the index space of x, this reference does not generate any communication. It is also assumed that the back end compiler recognizes the use of induction variable i in this reference and does not require any preprocessing for performing the global to local name space conversion.

The references $ija(i{+}1)$, $col(j)$, and $y(col(j))$, however, may require preprocessing. In general, for a reference of the form $arr(sub_{ast})$, this preprocessing may perform the following:

- It has to collect all values of sub_{ast} in order to prefetch the data referenced in $arr(sub_{ast})$ en bloc. Preprocessing may also reduce communication volume by recognizing duplicated references in sub_{ast}.

- It has to provide a mechanism to access the prefetched data in the actual computation.

Here sub_{ast} stands for the Abstract Syntax Tree (AST) index of the subscript. Note that while this index is different for each reference in the program, the value numbers of these references may be identical, even for subscripts that might textually look different.

In the transformed code, the statements proceeding the actual computation (in E1. . .E12) perform this preprocessing. Statements S8, S20, and S32 indicate opportunities for aggregated prefetching of the data required for references $ija(i{+}1)$, $col(j)$, and $y(col(j))$, respectively. For our *CSR* kernel, we assume that subscript reuse is relatively low. Therefore we perform the prefetching and indexing via temporary trace arrays that store global indices and are themselves indexed through counters that are incremented with each reference. Alternative mechanisms are described in Section 4.2.

The first prefetch statement, S8, brings in all $ija(i{+}1)$'s referenced by statement R3 in the original code. Statements T1. . .T5 and S1. . .S7 perform the preprocessing necessary for S8. Since in our example we are basing the prefetching mechanism on temporary trace arrays that have to be allocated dynamically, we first have to determine the size of the trace, *i.e.*, the number of references. This size is computed into *v4* by statements T1. . .T5. Statement S1 then allocates the local array *v1arr*, which has been declared ALLOCATABLE. Statements S2. . .S7 generate and

```
          SUBROUTINE CSR(x, y, col, ija, m, n)

          INTEGER i, j, m, n, col(m), ija(n)
          REAL x(n), y(n)
    !HPF$ DISTRIBUTE(BLOCK) :: col, ija, x, y

    !HPF$ BEGIN LOCAL
          INTEGER v4, v5
          INTEGER, ALLOCATABLE(:) ::
        . v1arr, v2arr, v3arr
    !HPF$ END LOCAL

    C     COUNTING SLICE D
    C     Count local iterations of outer loop
    C     to determine size of v1arr.
T1        v4 = 0
T2  !HPF$ EXECUTE (i) ON_HOME x(i)
T3        FORALL i = 1, n
T4          v4 = v4 + 1
T5        ENDFORALL

    C     COLLECTING SLICE A
    C     Collect "i + 1" into v1arr(1:v4).
S1        ALLOCATE (v1arr, v4)
S2        v4 = 0
S3  !HPF$ EXECUTE (i) ON_HOME x(i)
S4        FORALL i = 1, n
S5          v4 = v4 + 1
S6          v1arr(v4) = i + 1
S7        ENDFORALL
S8  C     Prefetching ija(v1arr(1:v4)) goes here

    C     COUNTING SLICE E
    C     Count local iterations of inner loop to
    C     determine size of v2arr and v3arr.
T6        v4 = 0
T7        v5 = 0
T8  !HPF$ EXECUTE (i) ON_HOME x(i)
T9        FORALL i = 1, n
T10         v4 = v4 + 1
T11         DO j = ija(i) + 1, ija(v1arr(v4))
T12           v5 = v5 + 1
T13         ENDDO
T14       ENDFORALL

    C     COLLECTING SLICE B
    C     Collect "j" into v2arr(1:v5).
S9        ALLOCATE (v2arr, v5)
S10       v4 = 0
S11       v5 = 0
S12 !HPF$ EXECUTE (i) ON_HOME x(i)
S13       FORALL i = 1, n
S14         v4 = v4 + 1
S15         DO j = ija(i) + 1, ija(v1arr(v4))
S16           v5 = v5 + 1
S17           v2arr(v5) = j
S18         ENDDO
S19       ENDFORALL
S20 C     Prefetching col(v2arr(1:v5)) goes here

    C     COLLECTING SLICE C
    C     Collect "col(j)" into v3arr(1:v5).
S21       ALLOCATE (v3arr, v5)
S22       v4 = 0
S23       v5 = 0
S24 !HPF$ EXECUTE (i) ON_HOME x(i)
S25       FORALL i = 1, n
S26         v4 = v4 + 1
S27         DO j = ija(i) + 1, ija(v1arr(v4))
S28           v3arr(v5) = col(v2arr(v5))
S29           v5 = v5 + 1
S30         ENDDO
S31       ENDFORALL
S32 C     Prefetching y(v3arr(1:v5)) goes here

    C     ACTUAL COMPUTATION
E1        v4 = 0
E2        v5 = 0
E3  !HPF$ EXECUTE (i) ON_HOME x(i)
E4        FORALL i = 1, n
E5          x(i) = 0
E6          v4 = v4 + 1
E7          DO j = ija(i) + 1, ija(v1arr(v4))
E8            v5 = v5 + 1
E9            x(i) = x(i) + y(v3arr(v5))
E10         ENDDO
E11       ENDFORALL
E12       END
```

Figure 3: CSR kernel – transformed version.

store the trace into *v1arr*. Finally, the prefetching operation in S8 brings in all the non-local data and stores them in the right locations of the array *ija*. This might require resizing the array *ija* to store the off-processor data. For the purpose of this example, we assume that storing of the off-processor data in the resized *ija* array is such that they can be referenced in global coordinates.

The next potential communication is generated by the prefetching statement S20, which collects on each processor the off-processor references to $col(j)$ in statement R4. Statements S10...S19 collect the trace of the values j indexing the array *col* into the local array *v2arr*. Note that in the expression for the upper bound of the j-loop, array *ija* is no longer indexed by $(i+1)$ but by the trace vector *v1arr* generated in statements S4...S7. The statements T6...T14 in Figure 3 compute the size of the array *v2arr* into the local scalar *v5*. The array *v2arr*, that like *m1arr* has been declared to ALLOCATABLE, is allocated in statement S9.

The values of y that are required on each processor at statement R4 are communicated in the prefetching statement S32. The trace of the values that index y is done in statements S22...S31, it is stored in the dynamic local array *v3arr*. Note that the number of references to $y(col(j))$ is the same as the number of references to $col(j)$, therefore the size of *v3arr* is the same as the size of *v2arr*. Hence we do not need any additional code to find out the size of *v3arr*, instead we can reuse the already computed local variable *v5* that stores the size of *v2arr*. Note also that in statement S28 the array *col* is referenced by the local array *v2arr* which stores global indices, instead of being referenced by j. After the execution of the statement S32, all processors have the required values of y in their local memories.

The actual loop computation is performed in statements E1...E11. During this computation no communication is required because everything that is necessary on each processor has already been fetched. To summarize, the original code shown in Figure 2 has been transformed into the code in Figure 3 that does all the necessary data communication in phases after several preprocessing steps. Within the different loops in the transformed code, all distributed arrays are referenced by at most one level of indirection and require no data communication.

3 Definitions

This section introduces some concepts that will be used in the algorithms in Section 4.

A *Slice* is a tuple

$$s = (s_{vn}, s_{target}, s_{code}, s_{ident}, s_{dep\_set}[, s_{cnt\_vn}])$$

that contains a value number s_{vn}, a designated program target location s_{target}, a sequence of statements s_{code}, an identifier s_{ident}, a dependence set s_{dep_set}, and optionally another value number s_{cnt_vn}. Slices come in two flavors:

- A *collecting slice* stores the sequence of values (trace) that will be assigned to a subscript during the execution of the program in some data structure identified by s_{ident}. The type of the data structure is determined by the degree of subscript reuse within the trace of the subscript, as described in Section 4.2.
- *Counting slices* are created from the collecting slices, they calculate the size of the subscript trace that will be generated during the execution of the collecting slice. Counting slices are needed if the collecting slices have to know the sizes of the traces they have to record, for example for preallocating a data structure to store the trace.

Each of the slices has the following properties with respect to the original program P:

- Inserting s_{code} at s_{target} in P is legal; *i.e.*, it does not change the meaning of P. The s_{code} is similar to a dynamic backward executable slice [16].
- After executing s_{code}, s_{ident} will have stored the values of s_{vn}.
- If s is a *collecting slice*, then s_{vn} will be the value number of a subscript s_{ast} of a nonlocal array reference $arr(sub_{ast})$ in P, and s_{ident} will store the sequence of all subscripts occurring during the run of P. Note that the length of this sequence depends on our point of view, which is given by s_{target}. For example, if s_{target} is the statement of the reference itself, then the sequence consists of only a single subscript. If s_{target} is the header of a loop enclosing the reference, then the sequence contains the subscripts for all iterations of the loop.
- If we compute counting slices, then s_{cnt_vn} will be the value number of the counter indexing s_{ident} after execution of s_{code} is finished; *i.e.*, the value

of s_{cnt_vn} will be the size of the subscript trace computed in s_{ident}.

- If s is a *counting slice*, then there exists a collecting slice t for which $s_{vn} = t_{cnt\_vn}$ and $s_{target} = t_{target}$ hold. s_{ident} will store the size of the subscript trace computed in t_{ident}. Since s_{ident} corresponds to a single value, s_{cnt_vn} will be the value number corresponding to the constant "1." Note: $s_{target} = t_{target}$ because otherwise we might count too many (for s_{target} preceding t_{target}) or too few (for s_{target} succeeding t_{target}) subscripts.

- The s_{dep_set} stored in each slice is a set of AST indices of subscripts of references of that need runtime processing. Only the references in s_{code} that require runtime processing are considered when the s_{dep_set} is created.

A *Slice Graph* is a directed acyclic graph

$$G = (S, E)$$

that consists of a set of slices S and a set of edges E. For $s, t \in S$, an edge $e = (s, t) \in E$ establishes an ordering between s and t. The presence of e implies that t_{code} contains a direct or indirect reference to s_{ident} and therefore has to be executed after s_{code}. G has to be acyclic to be a valid slice graph. Note that the edges in the slice graph not only indicate a valid ordering of the slices, but they also provide information for later optimizations. For example, it might be profitable to perform *loop fusion* across slices; the existence of an edge between slice nodes, however, indicates that these slices cannot be fused.

A *Subscript Descriptor*

$$sub = (sub_{vn}, sub_{target})$$

for the subscript sub_{ast} of some distributed array reference consists of the value number of sub_{ast}, sub_{vn}, and the location in P where a slice generated for sub should be placed, sub_{target}. Our algorithm will generate a slice for each unique subscript descriptor corresponding to a distributed array reference requiring runtime preprocessing. Identifying slices by subscript descriptors is efficient in that it allows a slice to be reused for several references, possibly of different data arrays, as long as the subscripts have the same value number. It is conservative in that it accounts for situations where different references might have the same subscript value number but different constraints as far as prefetch aggregation goes, which corresponds to different target nodes.

4 The Algorithm

This section gives a description of the algorithm to do the transformation shown in Section 2. The algorithm comes in two parts. The first part, described in Section 4.1, analyses the program and generates the slices and the slice graph. The second part, described in Section 4.3, uses the slice graph to do the code generation.

4.1 Slice Graph Construction

The procedure **Generate_slice_graph()** shown in Figure 4 is called with the program P and the set of subscripts R of the references that need runtime preprocessing, *i.e.*, the irregular references. It returns a slice graph consisting of a set of slices S and edges E. This procedure first generates all the necessary slices and then finds the edges between these slices.

The Foreach statement in A4...A8 computes a subscript descriptor (sub_{vn}, sub_{target}) for each subscript AST index sub_{ast}. We assume that P has an associated value number table that maps AST indices to value numbers. **Lookup_val_number()** uses this table to compute sub_{vn} from sub_{ast}. **Gen_target()** maps the AST index sub_{ast} to the target node sub_{target} for the slice generated starting from that AST index. The constraints on sub_{target} are the following.

- In the Control Flow Graph (CFG), sub_{target} predominates the reference sub_{ast}; *i.e.*, it is guaranteed that sub_{target} will be executed before sub_{ast} is used to reference its data array *arr*.

- There are no modifications of the data array *arr* between sub_{target} and sub_{ast}.

- Any code inserted at sub_{target} is executed as infrequently as possible.

Gen_target() implements these constraints using a Tarjan interval tree and array MOD information; starting at the node corresponding to the reference, it walks the interval tree upwards and backwards until it reaches a modification of *arr*.

The next Foreach statement in A9...A13 iterates through the subscript descriptors $sub \in U$ and generates for each subscript descriptor both the collecting slice s and, if needed, the counting slice t. **Gen_slice()** takes a subscript descriptor $sub = (sub_{vn}, sub_{target})$ and generates for location sub_{target} the slice that computes the values corresponding to

```
     Procedure Generate_slice_graph($P$, $R$)

     // $P$: Program to be transformed
     // $R$: AST indices of subscripts of references
     //      that need runtime preprocessing

A1   $S := \emptyset$    // Slices
A2   $E := \emptyset$    // Slice ordering edges
A3   $U := \emptyset$    // Subscript descriptors

     // Compute subscript descriptors.
A4   Foreach $sub_{ast} \in R$
A5       $sub_{vn}$ := Lookup_val_number($sub_{ast}$)
A6       $sub_{target}$ := Gen_target($sub_{ast}$)
A7       $U := U \cup \{(sub_{vn}, sub_{target})\}$
A8   Endforeach

     // Compute slices.
A9   Foreach $sub \in U$
A10      $s$ := Gen_slice($sub$)
A11      $S := S \cup \{s\}$

         // The following steps are executed
         // iff counting slices are required.
O1       $t$ := Lookup_slice($S$, ($s_{cnt\_vn}$, $s_{target}$))
O2       If $t = \emptyset$ Then
O3           $t$ := Gen_slice($s_{cnt\_vn}$, $s_{target}$)
O4           $S := S \cup \{t\}$
O5           $E := E \cup \{(t, s)\}$
O6       Endif

A12      Endif
A13  Endforeach

     // Compute edges resulting from
     // dependence sets of slices.
A14  Foreach $s \in S$
A15      Foreach $sub_{ast} \in s_{dep\_set}$
A16          $sub_{vn}$ := Lookup_val_number($sub_{ast}$)
A17          $sub_{target}$ := Lookup_target($sub_{ast}$)
A18          $t$ := Lookup_slice($S$, ($sub_{vn}$, $sub_{target}$))
A19          $E := E \cup \{(t, s)\}$
A20      Endforeach
A21  Endforeach

A22  Return $(S, E)$
```

Figure 4: Slice graph generation algorithm.

sub_{vn}. The slice generation function uses the program's CFG and the SSA (Static Single Assignment). Roughly speaking, Gen_slice() follows the use-def and control dependence chain starting in sub_{ast} until it reaches sub_{target}.

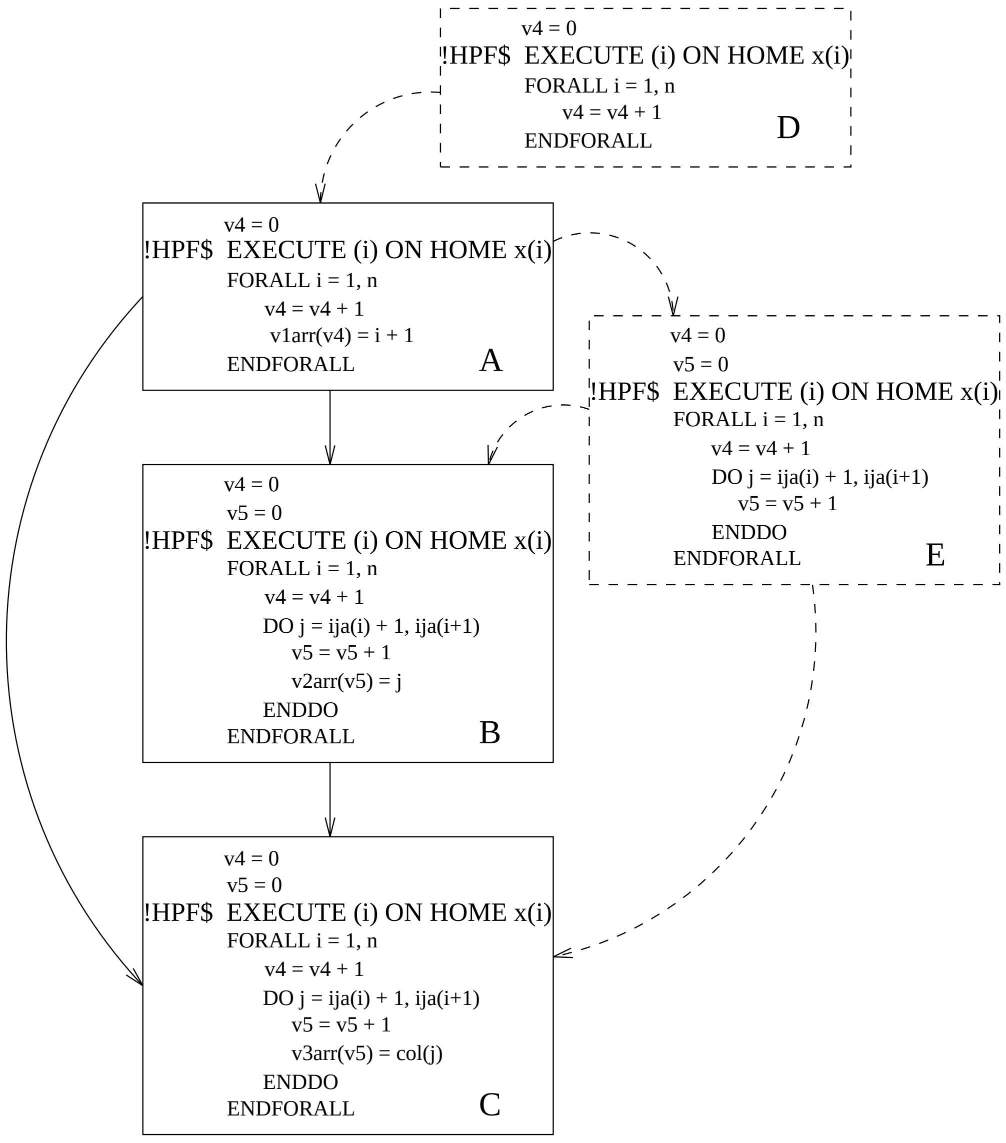
If we are interested in the size of the subscript trace recorded in s (*e.g.*, for allocating trace arrays), then we compute in statements O1…O6 a counting slice t for each s. However, different collecting slices can share a counting slice if they have the same counter value number sub_{cnt_vn} and target location sub_{target}. Therefore, we first examine the set of already created slices. **Lookup_slice()** takes as input a set of slices S and a subscript descriptor sub and returns the slice $t \in S$ corresponding to sub if there exists such an t; otherwise, it returns $\emptyset$. If a counting slice has not been created yet, a new counting slice t is generated. Since the counting slice t must be executed before the collecting slice s, a directed edge (t, s) is added to the edge set E.

The nested Foreach statements in A14…A21 are used to find the directed edges resulting from the dependence sets in each slice. The outer Foreach iterates through the slices s, the inner one loops through the references sub_{ref} stored in the dependence set s_{dep_set} of s. All the relevant information has already been generated previously, therefore these loops only have to consult tables to complete the set of edges.

The slice graph corresponding to the transformation example done in Section 2 is shown in Figure 5. There are five nodes in the slice graph, out of which nodes A, B, and C contain collecting slices, while nodes D and E contain counting slices. Note that the collecting slices B and C share the counting slice E, which reflects that the number of references to $y(col(j))$ is the same as the number of references to $col(j)$.

4.2 Trace Management Schemes

The discussion so far was mostly concerned with where to precompute which subscript traces in what order. Before actually generating code, however, we have to decide what data structures to use for first recording the traces to prefetch nonlocal data and then accessing these prefetched data. The example presented in Section 2 used temporary trace arrays for performing both of these operations. It turns out, however, that this is just one of several options, and there are different tradeoffs involved depending on the characteristics of the subscript traces. Therefore, when generating the statements s_{code} of a slice s, we do not include the code for manipulating these



```
       v4 = 0
!HPF$  EXECUTE (i) ON HOME x(i)
       FORALL i = 1, n
          v4 = v4 + 1
       ENDFORALL
```

```
       v4 = 0
!HPF$  EXECUTE (i) ON HOME x(i)
       FORALL i = 1, n
          v4 = v4 + 1
           v1arr(v4) = i + 1
       ENDFORALL
```

```
       v4 = 0
       v5 = 0
!HPF$  EXECUTE (i) ON HOME x(i)
       FORALL i = 1, n
          v4 = v4 + 1
          DO j = ija(i) + 1, ija(i+1)
             v5 = v5 + 1
          ENDDO
       ENDFORALL
```

```
       v4 = 0
       v5 = 0
!HPF$  EXECUTE (i) ON HOME x(i)
       FORALL i = 1, n
          v4 = v4 + 1
          DO j = ija(i) + 1, ija(i+1)
             v5 = v5 + 1
             v2arr(v5) = j
          ENDDO
       ENDFORALL
```

```
       v4 = 0
       v5 = 0
!HPF$  EXECUTE (i) ON HOME x(i)
       FORALL i = 1, n
          v4 = v4 + 1
          DO j = ija(i) + 1, ija(i+1)
             v5 = v5 + 1
             v3arr(v5) = col(j)
          ENDDO
       ENDFORALL
```

Figure 5: Example of a Slice Graph.

data structures; *i.e.*, we do not include counter initializations and increments or the assignments into trace arrays. Instead, we include place holders for these operations and delay the generation of these statements until the slice instantiation phase during the final code generation.

Let T be the size of the trace, *i.e.*, the number of times a subscript is evaluated with respect to the target location of the slice. Let R be the number of unique elements in T, and let N be the global size of the subscripted array, *i.e.*, the number of different subscripts possible. Note that $R \leq N$, $R \leq T$ must hold.

4.2.1 Case 1: Low subscript reuse

In this case, which is characterized by $R \approx T$, each subscript typically appears at most once in the trace produced by the slice. A possible example is the CSR kernel described in Section 2. Here it is reasonable to use a *dynamically allocated array* that is indexed through a counter incremented with each reference. This array can be used both for precomputing the subscripts and for looking them up during the actual computation. Since we have to store each subscript individually, the space requirements are $\mathcal{O}(T)$. We also usually have to generate counting slices to perform the dynamic allocation of the arrays. The time per access, however, is only $\mathcal{O}(1)$.

4.2.2 Case 2: High subscript reuse

This case is characterized by $R \ll T$, each subscript typically appears several times in the trace produced by the slice. An example of this is the pair list used for the non-bonded force kernel in molecular dynamics applications. Since each atom interacts with many other atoms, it appears many times in the pair list. Here some set representation, like a hash table, which collects subscripts and stores each of them at most once, would be an appropriate trace recording mechanism. Using a hash table to store off-processor data values was first introduced in [8]. The space requirements are only $\mathcal{O}(R)$, and we also do not need any counting slices. The time per access, however, will be $\mathcal{O}(\log(R))$ for most common set representations.

As a subscripting mechanism in the actual computation, we can use some dictionary representation, like a hash table (of a different kind than the one used for representing sets), that maps global indices to local indices. This typically requires space $\mathcal{O}(N)$ and $\mathcal{O}(\log(N))$ time per access.

An alternative subscripting mechanism is a "Global shuffle," where – roughly speaking – everything is translated to local coordinates, including the subscripting arrays themselves. The space requirements would be at most $\mathcal{O}(N)$, depending on how many data a processor needs locally and whether things can be shuffled in place or not. The time per access would be $\mathcal{O}(1)$.

4.3 Code generation

The code generation algorithm is shown in Figure 6. The procedure **Gen_code()** takes as input the original program P and the slice graph consisting of slices S and their ordering E. Gen_code() traverses the program and changes the subscripts of all the references that required runtime preprocessing. The function **Instantiate_program()** takes the program P and the set of slices S and replaces in P subscripts on which preprocessing has been performed with accesses of data structures defined in the preprocessing phase. This program instantiation depends on what type of data structure was used to store the trace of subscripts in the collecting slices, as discussed in Section 4.2.

Topological_sort() performs a topological sort of the slice graph so that the partial order given by the directed edges in E is maintained during generat-

```
Procedure Gen_code($P$, $S$, $E$)

C1 Instantiate_program($P$, $S$)
C2 Topological_sort($S$, $E$)
C3 Foreach $s \in S$
C4     Instantiate_slice($s$, $S$)
C5     Insert_code($P$, $s_{code}$, $s_{target}$)
C6 Endforeach

C7 Return $P$
```

Figure 6: Code generation algorithm.

ing code for the slices in S. The Foreach statement in C3...C6 iterates through the slices S. **Instantiate_slice()** is similar to Instantiate_program(), but instead of a program P it takes a slice s. However, it not only replaces subscript references, but it also adds the code mentioned in Section 4.2 for collecting the subscript that s is slicing on. Therefore this instantiation, like the program instantiation, depends on the type of data structure that was used to store the subscript traces of the references that affect the computation in this slice. After s has been instantiated, **Insert_code()** inserts s_{code} into the program at the target location s_{target}. The transformed program is returned to the calling procedure.

In the CSR example in Section 2, we assume that the subscript traces are stored in dynamically allocatable arrays. The instantiation routines will add the code for maintaining and referencing these arrays to the slices of the graph presented in Figure 5. A topological sort on the graph yields the node order to be D, A, E, B, and C; this is the same order in which the slices appear in the transformed code in Figure 4. For each of the slices, the subscripts of the references requiring runtime preprocessing present in the slice are changed to the local array that stores a trace of the subscript. At runtime the trace must already have been generated because there must exists an edge from the node where the trace was created to the node where it is being used. The slice is next substituted in the program before the slice target node. Note that the topological sort order is unique; this indicates, for example, that there is no loop fusion possible in the example. Note also that the transformed code in Figure 3 would be equally valid without having the subscripts of the references $ija(i+1)$, $col(j)$, and $y(col(j))$ replaced with references to trace arrays. However, this replacement makes the subsequent task of translating global indices to local indices simpler;

instead of having to modify user declared variables and subscript arrays, it is sufficient to translate the trace arrays.

5 Conclusions

The scheme that we presented in this paper can be used by a compiler to generate parallel code for irregular problems. In previous work we had presented a dataflow framework that determines the right placement of communication routines such that maximum reuse of off-processor data is possible [6]. This dataflow analysis in combination with the transformation presented in this paper will allow to generate efficient parallel code for any problem where irregular data access patterns exist. We have also carried out extensive research on the design of runtime support to support the communication requirements associated with indirectly accessed distributed arrays. One of the key optimizations is to reduce the communication volume associated with a prefetch by recognizing duplicated data references [14, 3]. While this paper did not focus on these runtime support issues, we have implicitly assumed the existence of such runtime support.

We are implementing the algorithm presented in Section 4 in the Fortran D compiler being developed at Rice University. At present we handle loops with a single level of indirection. We have also implemented a program slicer to be used to generate the slice graph.

6 Acknowledgements

We would like to thank Ken Kennedy and Chuck Koebel for their constant support of this project and also for providing us access to the Fortran D compiler being developed at Rice University. We would also like to thank Alan Sussman for many useful discussions.

References

[1] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[2] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems – data copy reuse and runtime partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pages 185–220. Elsevier, 1992.

[3] R. Das, J. Saltz, D. Mavriplis, and R. Ponnusamy. The incremental scheduler. In *Unstructured Scientific Computation on Scalable Multiprocessors*, Cambridge Mass, 1992. MIT Press.

[4] D. Loveman (Ed.). Draft High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

[5] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[6] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[7] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the Sixth International Conference on Supercomputing.* ACM Press, July 1992.

[8] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12:415–422, August 1991.

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings Supercomputing '91*, pages 86–100. IEEE Computer Society Press, November 1991.

[10] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[12] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[13] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Report 90-20, RIACS, 1990.

[14] J. Saltz, R. Das, R. Ponnusamy, D. Mavriplis, H Berryman, and J. Wu. Parti procedures for realistic loops. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, Oregon, April-May 1991.

[15] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. Technical Report 90-59, ICASE, NASA Langley Research Center, September 1990.

[16] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[17] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran — a language specification, version 1.1. Interim Report 21, ICASE, NASA Langley Research Center, March 1992.