Compiling Parallel Sparse Code for User-Defined Data Structures *

Vladimir Kotlyar, Keshav Pingali and Paul Stodghill
Department of Computer Science
Cornell University, Ithaca, NY 14853
{*vladimir,pingali,stodghil*}*@cs.cornell.edu*

January 7, 1997

Abstract

We describe how various sparse matrix and distribution formats can be handled using the *relational* approach to sparse matrix code compilation. This approach allows for the development of compilation techniques that are independent of the storage formats by viewing the data structures as relations and abstracting the implementation details as access methods.

1 Introduction

Sparse matrix computations are at the core of many computational science algorithms. A typical application can often be separated into the *discretization* module, which translates a continuous problem (such as a system of differential equations) into a sequence of sparse matrix problems, and into the *solver* module, which solves the matrix problems. Typically, the solver is the most time- and space-intensive part of an application and, quite naturally, much effort both in the numerical analysis and compilers communities has been devoted to producing efficient parallel and sequential code for sparse matrix solvers. There are two challenges in generating solver code that has to be interfaced with discretization systems:

- Different discretization systems produce the sparse matrices in many different formats. Therefore, the compiler should be able to generate solver code for different storage formats.
- Some discretization systems partition the problem for parallel solution, and use various methods for specifying the partitioning (distribution). Therefore, a compiler should be able to produce parallel code for different distribution formats.

In our approach, the programmer does not write sparse matrix code. Rather, he writes programs as if all matrices were dense, and then provides a specification of which matrices are actually sparse, and what formats/distributions are used to represent them. The job of the compiler is the following: *given a sequential, dense matrix program, descriptions of sparse matrix formats and data and computation distribution formats, generate parallel sparse SPMD code.* [5] and [6] have introduced a *relational algebra approach* to solving this problem. In this approach, we view sparse matrices as database relations, sparse matrix formats as implementations of *access methods* to the relations, and execution of loop nests as evaluation of certain relational queries. The key operator in these queries turns out to be the *relational join*. For parallel execution, we view loop nests as distributed queries and the process of generating SPMD node programs as the translation of distributed queries into equivalent local queries and communication statements.

In this paper, we focus on how our compiler handles user-defined sparse data structures and distribution formats. The architecture of the compiler is illustrated in Figure 1.

*This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF Grant CCR 9503199 and ONR grant N00014-93-1-0103.

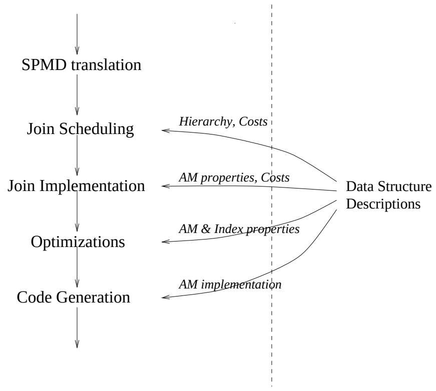

FIG. 1. *Architecture of the Compiler*

- *SPMD translation* phase takes an HPF-like parallel program description and translates it into SPMD node programs. [6] describes this process as distributed query evaluation. Section 4 shows how different distribution formats are handled by the compiler.
- *Join Scheduler* and *Join Implementer*, described in detail in [5], extract relational expressions describing the computations that must be performed, and decide how these expressions should be evaluated. The result is a high-level program, called a *plan*, in terms of abstract *access methods* (AMs) to the relations.

  Many sparse storage formats have a hierarchical structure. For example, the compressed row format ([7]) allows access first by row and then by column. The scheduler and implementer use this information along with the information about the properties of the indices stored in the matrix and the costs of the access methods in order to produce an optimized plan. Section 2 explains how this information is provided to the compiler.
- *Optimizer* further optimizes the plan. The need for optimizations influences the design of the interface between data structure descriptions and the compiler. We illustrate this on an example in Section 2.6.
- *Code Generator* translates the plan into low-level (e.g., C or Fortran) code by *instantiating* the calls to access methods. This is described in Section 3.

In the rest of the paper, we describe how these phases of the compiler interface user-defined data structures.

2 Describing Relations to the Compiler

2.1 A Motivating Example

We use sparse matrix-vector multiplication as a running example:

DO $i = 1, n$
  DO $j = 1, n$
    IF $A(i,j) \neq 0$ THEN
      $y(i) = y(i) + A(i,j) * x(j)$

We assume that the matrix **A** is sparse and the vectors **x** and **y** are dense. It is shown in [5] that execution of this loop can be viewed as evaluation of a relational query:

$$Q(i, j, v_A, v_y, v_x) = R_I(i, j) \bowtie R_A(i, j, v_A) \bowtie R_y(i, v_y) \bowtie R_x(j, v_x) \tag{1}$$

where $R_I(i, j)$ is the relation that represents the set of the iterations of the loop, $R_A(i, j, v_A)$ is the relation that stores the non-zero values v_A of the matrix **A**, R_x stores **x**, R_y stores **y**. Intuitively, the idea is that a tuple (i, j, v_A, v_y, v_x) is in the output relation only if $A(i,j)$ is non-zero; the fields v_A,v_y and v_x of such a tuple are needed to perform the computation.

This query contains two joins: on i and on j. The order of the nesting of the joins in the final code depends on how the matrix **A** is stored. For example, if the matrix were stored using compressed row format (CRS), then the outer loop of the resulting code enumerates the rows (i's) and the inner loop walks within each row and searches into the vectors (Figure 2). On the other

DO $i = 1, n$
  DO $\langle v_A, j\rangle \in A(i, *)$
    $v_y =$ search y for i
    $v_x =$ search x for j
    $v_y = v_y + v_A * v_x$

FIG. 2. *MVM for CRS format*

DO $j = 1, n$
  DO $\langle v_A, i\rangle \in A(*, j)$
    $v_y =$ search y for i
    $v_x =$ search x for j
    $v_y = v_y + v_A * v_x$

FIG. 3. *MVM for CCS format*

hand, if the matrix were stored using compressed column format (CCS) then the outer loop would run over the column index (Figure 3). Our compiler generates the appropriate code based on the fact that we can efficiently (in $O(1)$ time) search the matrix **A** for the row i in the case of CRS format or for the column j in the case of CCS format, and that the (dense) vectors can be quickly searched.

Unlike previous work ([2]), our compiler does not have the CRS or CCS formats "hard-wired". It reasons about them based on a high-level description, which consists of:

- Description of the *hierarchy* of the indices. This differentiates between compressed row and compressed column storage. All other things being equal, we prefer to enumerate rows before columns in the former case, and columns before rows in the latter.
- Description of how efficiently the data structure can be searched or enumerated. Suppose that the matrix **A** in the example above contained rows with all elements being zero. We could still store it in the "usual" CRS – that is store all n rows. Or we could store only non-zero rows (we can call this "Compressed Compressed Row Storage" or CCRS). This would save space, but require more expensive searches.
- Description of the properties of the domains of indices. For example, in CRS the row index is dense – all values from 1 to n are present. In CCRS the row index is sparse.

We now formalize these ideas.

2.2 Hierarchy of Indices

The key is to view data structures as collections of functions. For example, CRS can be seen as providing a function of the type

$$T_{\text{CRS}} = \mathcal{I} \rightarrow T_{\text{row}} \tag{2}$$

where $\mathcal{I} = \{1 \ldots n\}$ is the type of the row index. T_{row} is the type of the relation for each row:

$$T_{\text{row}} = \mathcal{J} \rightarrow \mathbb{R} \tag{3}$$

where $\mathcal{J} = \{1 \ldots n\}$ is the type of the column index and $\mathbb{R}$ is the value type (type of real values). We can combine (2) and (3) into $T_{\text{CRS}} = \mathcal{I} \rightarrow (\mathcal{J} \rightarrow \mathbb{R})$. There is still some ambiguity in this notation. Notice that the type for the CCS format looks exactly like the type for the CRS format: $T_{\text{CCS}} = \mathcal{J} \rightarrow (\mathcal{I} \rightarrow \mathbb{R})$.

To resolve the ambiguity we need to associate the indices with the types in these equations. We will write CRS as: $T_{\text{CRS}}(i, j, v) = \mathcal{I}(i) \rightarrow (\mathcal{J}(j) \rightarrow \mathbb{R}(v))$.

In general, the notation $T(i, j, k, \ldots) = T_1(i) \rightarrow T_2(j, k, \ldots)$ means that we can get a (sub)relation of the type T_2 in the indices j, k, etc by searching the relation in the i index. In this paper, we omit the names for the indices when they are clear from context.

The next question is what form can the type T_1 take? To answer it we consider two more examples: dense storage and coordinate storage. A matrix stored in dense format allows access to the value field by searching on the pair row and column indices. We write this as $T_{\text{dense}} = (\mathcal{I} \times \mathcal{J}) \rightarrow \mathbb{R}$. But we can also search a matrix in coordinate format on the row and column indices! Is the type of coordinate storage formats the same as the type of dense storage format? We would like to account for important structural differences between the two. Besides the efficiency of the search, there is an important difference between coordinate and dense storage: a dense matrix stores *all* pairs of the indices in the set $\mathcal{I} \times \mathcal{J}$, but a matrix in coordinate storage stores a *subset* of all possible pairs. We write this as $T_{\text{coord}} = (\mathcal{I}, \mathcal{J}) \rightarrow \mathbb{R}$.

In general, the notation $S \times T$ means that we are storing all pairs $\langle s, t\rangle$ where s comes from some subset of S and t comes from some subset of T. For example, the type for the storage format for an element stiffness matrix ([8]) (we call this *cartesian storage*) is the same as the type for dense storage format $T_{\text{cart}} = (\mathcal{I} \times \mathcal{J}) \rightarrow \mathbb{R}$. The difference between these two is captured later when we describe the properties of the domains of the indices. At this point we summarize the rules for building relation types:

- A *relation* type is a union of *function* types $T = F_1 \cup F_2 \cup \ldots$.
- The argument type P for the function type $F = P \rightarrow T$ is a *tuple* or a *product* of basic types: $P = (\mathcal{I}_1, \mathcal{I}_2, \ldots)$ or $P = (\mathcal{I}_1 \times \mathcal{I}_2 \times \ldots)$. The result type T is a relation type. A product or tuple type can have zero or more components. We use ϵ to denote a product (or a tuple) of zero types.

Here are some more examples. $\epsilon \rightarrow \mathbb{R}$ describes a scalar. $\mathcal{I} \rightarrow \mathbb{R}$ describes a vector (sparse or dense). $(\mathcal{I} \rightarrow \mathcal{J}) \cup (\mathcal{J} \rightarrow \mathcal{I})$ describes a permutation stored using two vectors: on for the permutation itself and one for the inverse.

2.3 Access Methods

Access methods describe how the indices can be searched and enumerated for each function type $F = P \rightarrow T$. Let P be a product type $P = (\mathcal{I}_1 \times \ldots \times \mathcal{I}_n)$. Then the following methods should be provided by an implementation:

$$\begin{aligned} \langle b, h_k\rangle &= \texttt{Search}(x_k) \\ \{\langle x_k, h_k\rangle\} &= \texttt{Enum}() \\ t &= \texttt{Deref}(\langle h_1, \ldots, h_n\rangle) \end{aligned}$$

where x_k is the value of the index of type $\mathcal{I}_k$, b is a boolean flag indicating whether the value was found or not and h_k is the *handle* used to dereference the values t the result type T. For example, a handle can be an integer offset into an underlying array or a pointer.

Notice that we can separately search and enumerate the components of a product type. For example, an implementation of the cartesian storage would provide a way to enumerate all column indices and all row indices. Also notice that the `Enum()` method returns a set of tuples.

If the argument is of tuple type $P = (\mathcal{I}_1, \ldots, \mathcal{I}_n)$ then only one set of functions need be provided:

$$\begin{aligned} \langle b, h\rangle &= \texttt{Search}(x_1, \ldots, x_n) \\ \{\langle x_1, \ldots, x_n, h\rangle\} &= \texttt{Enum}() \\ t &= \texttt{Deref}(h) \end{aligned}$$

2.4 Properties of Index Types and Access Methods

We have mentioned in Section 2.2 that dense and cartesian storage have the same types. One difference that we need to capture is that searches into a dense matrix are cheaper. Another point that a dense matrix stores *all* (i, j) pairs within the $(1 \dots n) \times (1 \dots n)$ square: we know that for a certain range of the indices the `Search()` always succeeds. We also need to know whether the indices can be enumerated in a particular order. This is used, for example, in deciding which join method to apply: merge join ([9]) requires indices to be sorted. In order to enable certain optimizations, we need to know the types of the handles. In particular, we need to know if the handles fill a particular range of integers. We also need to recognize the case when a call to `Deref()` is just an array access.

Overall, the following characteristics of the indices and access methods are required by the compiler:

- *Cost of searching:* We differentiate between $O(1)$ lookups, $O(\log n)$ binary searches and $O(n)$ linear searches.
- *Ordering of indices:* Ordered or Unordered
- *Range of the indices:* Dense or Sparse. Notice that $O(1)$ lookup does not necessarily imply that an index is dense: a sparse array can be stored using a hash table.
- *Type and range of handles:* We differentiate between a range of integers and a general "pointer".
- *Kind of dereference:* We differentiate between a "general case" and the case when `Deref()` is only an array access.

2.5 Summary

To summarize the discussion so far:

- Each relation is a union of function types.
- Access methods are provided for each function type.
- Access methods and indices are characterized by the cost of searching and by the properties of the index and handle domains.

We now use an example to illustrate how this protocol is used in optimizations.

2.6 An Extended Example

Let us consider a simplification of the matrix storage format used in the BlockSolve library ([3]). This format exploits the fact that in many physical simulations, there are many rows of a sparse matrix with the same non-zero structure [1]. Each set of rows with identical structure is called an *inode*. The non-zero values in the rows of an inode can be stored as dense matrices, and the row and column indices are stored just once, as shown in Figure 4. Clearly, this format has a hierarchical structure. We can describe it using our notation as:

$$T_{\mathrm{BSS}}(in, i, j, v) = \mathcal{IN}(in) \rightarrow T_{\mathrm{cart}}(i, j, v) = \mathcal{IN}(in) \rightarrow (\mathcal{I}(i) \times \mathcal{J}(j) \rightarrow \mathbb{R}(v)) \tag{4}$$

The unoptimized plan for matrix-vector multiplication using this simplified BlockSolve format is shown in Figure 5.

An important optimization, which is performed in the BlockSolve package, is to *gather* the values of X used in each i-node into a dense vector. This way the computation in the inner loops becomes a dense matrix-vector multiplication and can be done very efficiently. How does our framework facilitate this optimization?

We know from the description of the access methods that for each i-node the handle h_x takes on a range of integral values, say $1 \dots M_{\mathrm{in}}$. This allows us to replace the fragment in Figure 6 by the fragment in Figure 7. The first loop *gathers* the values of X and the second loop is the copy of the original computation with the use of v_x replaced by the reference to the temporary array. The code for *scattering* X if it were assigned to in the original loop is similar.

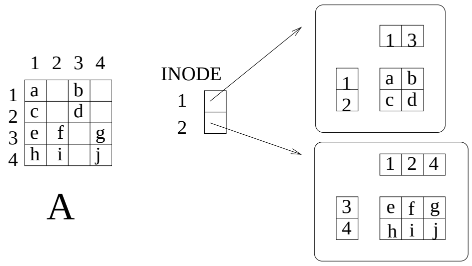

FIG. 4. *Simplified BlockSolve storage*

DO $\langle in, h_{\mathrm{in}}\rangle \in \mathtt{Enum}(R_A)$
  $R_{\mathrm{inode}} = \mathtt{Deref}(R_A, h_{\mathrm{in}})$
  DO $\langle i, h_i\rangle \in \mathtt{Enum}_i(R_{\mathrm{inode}}, in)$
    DO $\langle j, h_j\rangle \in \mathtt{Enum}_j(R_{\mathrm{inode}}, in)$
      $v_A = \mathtt{Deref}(R_{\mathrm{inode}}, \langle h_i, h_j\rangle)$
      $v_x = \mathtt{Deref}(R_x, \mathtt{Search}(R_x, j))$
      $v_y = \mathtt{Deref}(R_y, \mathtt{Search}(R_y, i))$
      $v_y = v_y + v_A * v_x$

FIG. 5. *Unoptimized plan for the simplified BlockSolve format*

DO $\langle j, h_j\rangle \in \mathtt{Enum}_j(R_{\mathrm{inode}}, in)$
  $v_x = \mathtt{Deref}(R_x, \mathtt{Search}(R_x, j))$
  $\ldots v_x \ldots$
ENDDO

FIG. 6. *Before gather optimization*

Allocate real array $tmp(1 : M_{\mathrm{in}})$
DO $\langle j, h_j\rangle \in \mathtt{Enum}_j(R_{\mathrm{inode}}, in)$
  $tmp(h_j) = \mathtt{Deref}(R_x, \mathtt{Search}(R_x, j))$
ENDDO
DO $h_j = 1, M_{\mathrm{in}}$
  $\ldots tmp(h_j) \ldots$
ENDDO

FIG. 7. *After gather optimization*

In our example, we can gather X and scatter Y to obtain the optimized plan in Figure 8. Since we know that the dereference into an inode by $\langle h_i, h_j\rangle$ is a simple dense array access, we can actually pattern match this fragment and call the best matrix-vector multiplication routine available.

3 Code Generation

How does the compiler translate the output of the Optimizer phase – the plan – into C or Fortran code? In particular, how are the calls to Search/Enumerate/Dereference access methods are translated? The simplest solution would be to write a run-time library of for each data structure and then translate the calls to access methods into actual function calls. The only complication is that one call to the `Enum()` method has to be translated into three function calls: to open the *stream* of tuples, to advance the stream and to check for the end of stream.

The problem with this solution is that we would have to rely on the ability of the underlying C or Fortran compiler to inline these function calls, because function call overhead is unacceptable. We have chosen to let the data structures generate code for themselves. This way the descriptions of the data structures become extensions of the compiler: when the compiler needs to expand a call to an access method it passes the syntactic representation of the arguments to the module for a particular data structure, and the module returns the translation of the call. This enables us to generate code which is very similar to hand-written codes.

4 Handling Data Structures for Parallel Code

The main idea of [6] is to view distributes arrays (matrices) as distributed relations which are built out of local storage *fragments*. For example, suppose we have a matrix A distributed by row. Let i be the global row index and j be the global column index. On each processor we just have a matrix, which has the same column index as the global matrix and it has a new local row index i'. This is

[1] Each row has different *values* of course.

DO $\langle in, h_{\text{in}} \rangle \in \texttt{Enum}(R_A)$
  $R_{\text{inode}} = \texttt{Deref}(R_A, h_{\text{in}})$
  Allocate $tmp_x(1 : M^j_{\text{in}})$ to gather X
  Allocate $tmp_y(1 : M^i_{\text{in}})$ to scatter Y
  DO $\langle j, h_j \rangle \in \texttt{Enum}_j(R_{\text{inode}}, in)$
    $tmp_x(h_j) = \texttt{Deref}(R_x, \texttt{Search}(R_x, j))$
  ENDDO
  DO $h_i = 1, M^i_{\text{in}}$
    DO $h_j = 1, M^j_{\text{in}}$
      $tmp_y(h_i) = tmp_y(h_i)$
        $+\texttt{Deref}(R_{\text{inode}}, \langle h_i, h_j \rangle)$
        $*tmp_x(h_j)$
    ENDDO
  ENDDO
  DO $\langle i, h_i \rangle \in \texttt{Enum}_i(R_{\text{inode}}, in)$
    $\texttt{Deref}(R_y, \texttt{Search}(R_y, i)) = tmp_y(h_i)$
  ENDDO
ENDDO

FIG. 8. *Optimized plan for the simplified BlockSolve format*

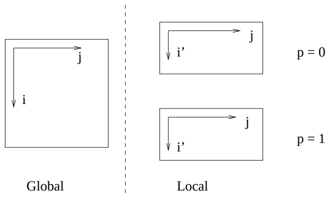

FIG. 9. *Partitioning of a matrix by row*

illustrated in Figure 9. There is a one-to-one relationship between the global index i and the pair $\langle p, i' \rangle$ of the processor index and the local row index.

If we view the global matrix as a relation with i, j and value fields and each local matrix as a relation with i', j and value fields, then the following *fragmentation* equation holds:

$$R_A(i, j, v) = \pi_{i,j,v}(\bigcup_p \delta(i, p, i') \bowtie R^{(p)}_A(i', j, v)) \tag{5}$$

where π is the relational algebra projection operator ([9]), $R^{(p)}_A$ is the relation that stores the local matrix and $\delta(i, p, i')$ captures the relationship between the local and global indices.

Now the query (1) for the computation in sparse matrix-vector multiplication becomes a distributed query with each relation distributed among the processor according to (5). The task of SPMD translation phase is to translate the *distributed* query, like (1), into a sequence of local queries and communication statements. The details can be found in [6].

In this paper, we show some examples of different distribution schemes that can be uniformly represented in our framework. Notice that the fragmentation relation δ is a distributed relation itself. In the simplest case it can be represented by a closed-form formula: this happens when we use block/cyclic distributions which can be translated into systems of linear equalities and inequalities [1].

Now let us turn to various ways of specifying irregular distributions: the relation $\delta(i, p, i')$ cannot be represented in closed form, and has to be stored. One way of storing it is to have on each processor an array F of global row indices indexed by the local index $i' : i = F(i')$. If we think of this array as storing a local fragment $R^{(p)}_F(i, i')$ of δ, then we write down the fragmentation equation for δ itself:

$$\delta(i, p, i') = \bigcup_p R^{(p)}_F(i, i') \tag{6}$$

Another way of storing $\delta(i, p, i')$ is to partition it block-wise across processors based on index i; *i.e.*, a tuple $\langle i, p, i' \rangle$ is stored on the processor $q = i/B$ for some block size B. This scheme is called *paged translation table* ([11]). We can write this scheme as:

$$\delta(i,p,i') = \pi_{i,p,i'}\left(\bigcup_q \delta'(i,q,i'') \bowtie \delta^{(q)}(i'',q,i')\right) \tag{7}$$

The fragmentation relation δ' describes the paged (blocked) partitioning of the relation δ: $\delta'(i,q,i'') \equiv q = i/B \wedge i'' = i \mod B$.

By viewing distributions as (globally distributed) relations, we can represent many different distribution formats. How is this different from the alignment/distribution specification of HPF? First of all, HPF compilers have a certain set of distribution formats "hard-wired". Second, alignment/distribution specification in HPF only specifies a map from global index to processor number. A particular local storage format (and the relationship with the local index) is implicit in this scheme. This simplifies the job of a user, but is less flexible in interfacing other codes (such as discretization and partitioning systems).

Current research compilers have different compilation paths for regular and irregular distributions ([10]). We believe that our SPMD translation algorithm can be used as a pass in an HPF compiler after the compiler has generated the specification for local storage. This way regular and irregular distributions can be handled uniformly.

5 Conclusions and Future Work

We have shown how various sparse matrix and distribution formats can be handled using the *relational* approach to sparse matrix code compilation. Currently, our techniques are only applicable to DOALL loops and loops with reductions: the compiler only needs to find the best way to enumerate over the data structures without worrying about the legality of such enumeration. We have extended these techniques to loop with dependencies for a restricted case of Compressed Hyperplane Storage ([4]), and we are currently working on generating code with dependencies for a wider range of data structures.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell, *A linear algebra framework for static hpf code distribution*, in CPC'93, November 1993. Also available at http://cri.ensmp.fr/doc/A-250.ps.Z.

[2] A. Bik, *Compiler Support for Sparse Matrix Computations*, PhD thesis, Leiden University, the Netherlands, May 1996.

[3] M. T. Jones and P. E. Plassmann, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, Tech. Rep. ANL–95/48, Argonne National Laboratory, Dec. 1995.

[4] V. Kotlyar and K. Pingali, *Sparse code generation for imperfectly nested loops with dependencies*, in Submitted to ACM ICS '97 conference, 1997.

[5] V. Kotlyar, K. Pingali, and P. Stodghill, *A relational approach to sparse matrix compilation*, in Submitted to ACM PLDI '97 conference, 1997.

[6] ———, *Unified framework for sparse* and *dense spmd code generation*, in Submitted to ACM PLDI '97 conference, 1997.

[7] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press, London, 1984.

[8] G. Strang, *Introduction to applied mathematics*, Wellesley-Cambridge Press, 1986.

[9] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, v. I and II*, Computer Science Press, 1988.

[10] R. v. Hanxleden, K. Kennedy, and J. Saltz, *Value-based distributions and alignments in Fortran D*, Tech. Rep. CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993.

[11] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani, *Distributed memory compiler design for sparse problems*, IEEE Transactions on Computers, 44 (1995). Also available from ftp://hyena.cs.umd.edu/pub/papers/ieee_toc.ps.Z.