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*

June 2, 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 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

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

†A version of this report appears in the proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing

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.

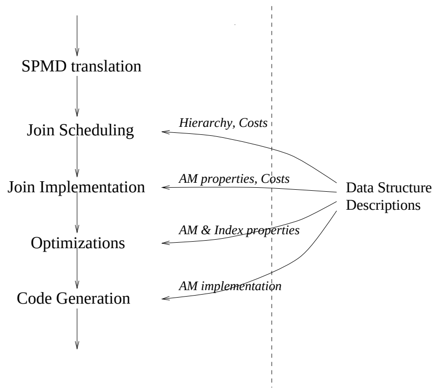

FIG. 1. *Architecture of the Compiler*

- *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 $j = 1, n$
    $Y(i) = Y(i) + A(i,j) * X(j)$

We assume that the matrix $\mathbf{A}$ is sparse and the vectors $\mathbf{X}$ and $\mathbf{Y}$ are dense. It is shown in [5] that execution of this loop can be viewed as evaluation of a relational query:

$$Q(i,j,v_A,v_y,v_x) = \sigma_{\mathcal{P}}\Big(I(i,j) \bowtie A(i,j,v_A) \bowtie Y(i,v_y) \bowtie X(j,v_x)\Big) \tag{1}$$

with the predicate $\mathcal{P}$ defined as:

$$\mathcal{P} \stackrel{\text{def}}{=} NZ(A(i,j)) \wedge NZ(X(j)) \tag{2}$$

In this query $I(i,j)$ is the relation that represents the set of the iterations of the loop, $A(i,j,v_A)$ is the relation that stores the values v_A of the matrix $\mathbf{A}$, X stores $\mathbf{x}$, Y stores $\mathbf{y}$. Conceptually, the relations store both zeros and non-zeros. Although it might be the case that only non-zeros are physically stored. The predicates $NZ(A(i,j))$ and $NZ(X(j))$ test if a particular array element is indeed physically stored and has to be assumed to have a non-zero value. Notice that the NZ predicate is true for all indices of a dense array, even if some values are numerically zero.

This query (1) 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 $\mathbf{A}$ 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 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 $\mathbf{A}$ 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.

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*

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 $\mathbf{A}$ 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

Assume that the matrix is a relation with three fields named I, J and V where the I field corresponds to rows, the J field corresponds to columns and the V field is the value. Table 1 illustrates specification of the hierarchy of indices for a variety of formats.

Name	Type
CRS	$T_{\mathrm{CRS}} = I \succ J \succ V$
CCS	$T_{\mathrm{CRC}} = J \succ I \succ V$
COORDINATE	$T_{\mathrm{coord}} = (I, J) \succ V$
DENSE	$T_{\mathrm{dense}} = I \times J \succ V$
INODE	$T_{\mathrm{i-node}} = INODE \succ_{\not\cap} (I \times J) \succ V$
ELEMENT	$T_{\mathrm{FE}} = E \succ_{+} (I \times J) \succ V$
PERMUTATION	$T_{perm} = (I \succ I') \bigcup (I' \succ I)$

TABLE 1

Hierarchy of indices for various formats

In this notation the $\succ$ operator is used to indicate the nesting of the fields within the structure. For example, $I \succ J \succ V$ in the Compressed Row Storage (CRS) format [7] indicates that we have to access a particular row before we can enumerate the column indices and values; and that within a row, we can search on the column index to find a particular value. The notation (I, J) in the specification of the coordinate storage indicates that the matrix is stored as a "flat" collection of tuples.

The $\times$ operator indicates that the indices can be enumerated independently, as in the dense storage format.

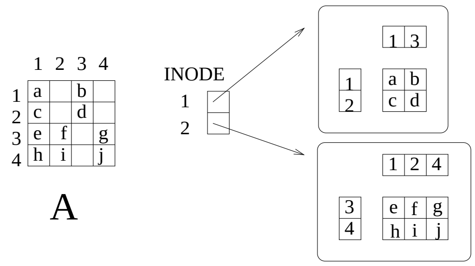

FIG. 4. *I-node storage format*

Consider the i-node storage format shown in Figure 4. This format is used in the BlockSolve library ([3]). It 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. How do we specify such format? The problem here is that a new *INODE* field is introduced in addition to the row and column fields. Fields like inode number which are not present in the dense array are called *external* fields. An important property that we need to convey is that inodes partition the matrix into disjoint pieces. We denote it by the $\not\cap$ symbol subscript in $T_{\mathrm{i-node}}$. This will differentiate the i-node storage format from the format often used in Finite Element analysis [8]. In this format the matrix is represented as a sum of element matrices. The element matrices are stored just like the inodes, and the overall type for this format is T_{FE} in Table 1, where E is the field of element numbers. Our compiler is able to recognize the cases when the matrix is used in additive fashion and does not have to be explicitly constructed.

Some formats can be seen as providing several alternative index hierarchies. Suppose we have a permutation relation $P(i, i')$ which is stored using two integer arrays: `PERM` and `INVP`. $i' = \mathtt{PERM}(i)$ is the value of the permuted index. $\mathtt{INVP}(i')$ provides the inverse of the permutation. The index hierarchy for such format is shown in the table.

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

Below is the grammar for building index hierarchy specifications:

$$T := V \;\Big|\; F \;\Big|\; F \succ_{\text{op}} T \;\Big|\; F \times F \times F \times \ldots \succ T \;\Big|\; (F, F, F, \ldots) \succ T \;\Big|\; T \bigcup T \tag{3}$$

where the terminal V indicates an array value field, and F indicates an array index field.

2.3 Access Methods

Access methods describe how the indices can be searched and enumerated for each term in the hierarchy: $F = P \succ T$. 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 result, which can conceptually be a relation, as well. Usually, the handle is an integer offset into an underlying array or a pointer.

Notice that we can separately search and enumerate the components of the product term. For example, an implementation of the i-node 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 P 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 Indices and Access Methods

Notice that dense format and format for individual i-nodes have the same hierarchy $I \times J$. One difference that we need to capture is that searches into a dense matrix are cheaper. Also, a dense matrix stores *all* (i, j) pairs within the $(1 \ldots n) \times (1 \ldots n)$ 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".
- *Arity of dereference:* We differentiate between the result being a singleton or a relation.
- *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 format is represented by its index hierarchy.
- Access methods are provided for each level in the hierarchy.
- 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 sparse matrix-vector product when the i-node format (Figure 4) is used to store the matrix. The join scheduler and join implementer (described in [5]) produce an unoptimized plan

$$
\begin{array}{l}
\text{DO } \langle in, h_{\text{in}} \rangle \in \texttt{Enum}(A) \\
\quad R_{\text{inode}} = \texttt{Deref}(A, h_{\text{in}}) \\
\quad \text{DO } \langle i, h_i \rangle \in \texttt{Enum}_i(R_{\text{inode}}, in) \\
\quad\quad \text{DO } \langle j, h_j \rangle \in \texttt{Enum}_j(R_{\text{inode}}, in) \\
\quad\quad\quad v_A = \texttt{Deref}(R_{\text{inode}}, \langle h_i, h_j \rangle) \\
\quad\quad\quad v_x = \texttt{Deref}(X, \texttt{Search}(X, j)) \\
\quad\quad\quad v_y = \texttt{Deref}(Y, \texttt{Search}(Y, i)) \\
\quad\quad\quad v_y = v_y + v_A * v_x
\end{array}
$$

FIG. 5. *Unoptimized plan for the i-node storage*

for this computation 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 $1 \ldots M_{\text{in}}$. 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.

$$
\begin{array}{l}
\text{DO } \langle j, h_j \rangle \in \texttt{Enum}_j(R_{\text{inode}}, in) \\
\quad v_x = \texttt{Deref}(X, \texttt{Search}(X, j)) \\
\quad \ldots v_x \ldots \\
\text{ENDDO}
\end{array}
$$

FIG. 6. *Before 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

$$
\begin{array}{l}
\text{Allocate real array } tmp(1:M_{\text{in}}) \\
\text{DO } \langle j, h_j \rangle \in \texttt{Enum}_j(R_{\text{inode}}, in) \\
\quad tmp(h_j) = \texttt{Deref}(X, \texttt{Search}(X, j)) \\
\text{ENDDO} \\
\text{DO } h_j = 1, M_{\text{in}} \\
\quad \ldots tmp(h_j) \ldots \\
\text{ENDDO}
\end{array}
$$

FIG. 7. *After gather optimization*

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

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

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. In our approach, each access method is a macro, which is then expanded by the compiler into the actual code.

We illustrate this on an example. Consider a sparse matrix stored in CCS format using three arrays: `COLP`, `VALS` and `ROWIND`. The array section $\texttt{VALS}(\texttt{COLP}(j) \ldots (\texttt{COLP}(j+1) - 1))$ stores the non-zero values of the j-th column and the array section $\texttt{ROWIND}(\texttt{COLP}(j) \ldots (\texttt{COLP}(j+1) - 1))$ stores the row indices of the non-zero elements of the j-th column. The macros for the enumeration of row indices within a column and for the dereference of the values would expand the following fragment of a plan:

$$
\begin{array}{l}
j = \ldots. \\
R_{\text{column}} = \texttt{Deref}(A, \texttt{Search}(A, j)) \\
\text{DO } \langle i, h \rangle \in \texttt{Enum}_i(R_{\text{column}}) \\
\quad \ldots i \ldots \\
\quad v_A = \texttt{Deref}(R_{\text{column}}, h) \\
\quad \ldots v_A \ldots
\end{array}
$$

into the actual Fortran code:

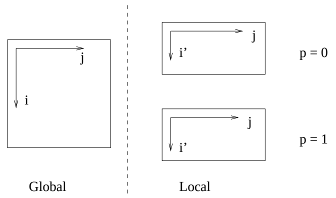

FIG. 9. *Partitioning of a matrix by row*

$j = ...$
DO ii = COLP(j), COLP($j+1$) -1
  ...ROWIND(ii)...
  v_A = VALS(ii)
  ...v_A...

This approach 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 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:

$$A(i,j,v) = \pi_{i,j,v}(\bigcup_p \delta(i,p,i') \bowtie A^{(p)}(i',j,v)) \tag{4}$$

where π is the relational algebra projection operator ([9]), $A^{(p)}$ 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 (4). 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 $F^{(p)}(i,i')$ of δ, then we write down the fragmentation equation for δ itself:

$$\delta(i,p,i') = \bigcup_p F^{(p)}(i,i') \tag{5}$$

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:

$$\delta(i,p,i') = \pi_{i,p,i'}\left(\bigcup_q \delta'(i,q,i'') \bowtie \delta^{(q)}(i'',q,i')\right) \tag{6}$$

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 ACM ICS '97, July 1997.

[5] V. Kotlyar, K. Pingali, and P. Stodghill, *A relational approach to sparse matrix compilation*, in EuroPar, August 1997. Available as Cornell Computer Science Tech. Report 97–1627 (http://cs-tr.cs.cornell.edu).

[6] ———, *Unified framework for sparse and dense spmd code generation (preliminary report).*, Tech. Rep. TR97-1625, Computer Science Department, Cornell University, 1997. http://cs-tr.cs.cornell.edu.

[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.