Compiling Parallel Code for Sparse Matrix Applications

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

August 18, 1997

Abstract

We have developed a framework based on relational algebra for compiling efficient sparse matrix code from dense DO-ANY loops and a specification of the representation of the sparse matrix. In this paper, we show how this framework can be used to generate parallel code, and present experimental data that demonstrates that the code generated by our *Bernoulli* compiler achieves performance competitive with that of hand-written codes for important computational kernels.

Keywords: parallelizing compilers, sparse matrix computations

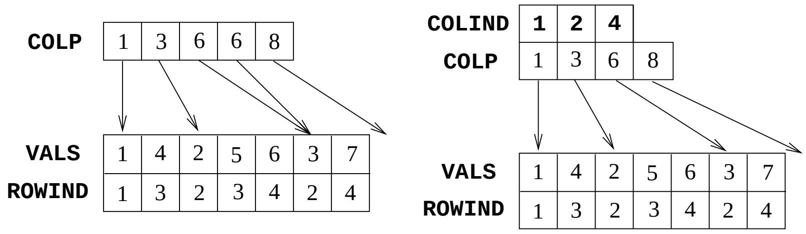
1 Introduction

Sparse matrix computations are ubiquitous in computational science. However, the development of high-performance software for sparse matrix computations is a tedious and error-prone task, for two reasons. First, there is no standard way of storing sparse matrices, since a variety of formats are used to avoid storing zeros, and the best choice for the format is dependent on the problem and the architecture. Second, for most algorithms, it takes a lot of code reorganization to produce an efficient sparse program that is tuned to a particular format. We illustrate these points by describing two formats — a classical format called Compressed Column Storage (CCS) [10] and a modern one used in the BlockSolve library [11] — which will serve as running examples in this abstract.

CCS format is illustrated in Fig. 1. The matrix is compressed along the columns and is stored using three arrays: COLP, VALS and ROWIND. The values of the non-zero elements of each column j are stored in the array section $\mathtt{VALS}(\mathtt{COLP}(j) \ldots (\mathtt{COLP}(j+1)-1))$. The row indices for the non-zero elements of the column j are stored in $\mathtt{ROWIND}(\mathtt{COLP}(j) \ldots (\mathtt{COLP}(j+1)-1))$. This is illustrated in Fig. 1(b). If a matrix has many zero columns, then the zero columns are not stored, which results in what is called Compressed Compressed Column Storage format (CCCS). In this case, another level of indirection is added (the COLIND array) to compress the column dimension, as well (Fig. 1(c)).

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 \\ 0 & 6 & 0 & 7 \end{pmatrix}$$

(a) An example matrix



(b) CCS format (c) CCCS format

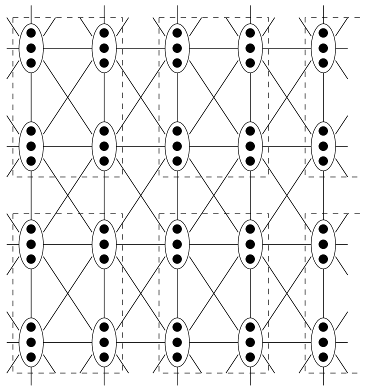
Figure 1: Illustration of Compressed Column Storage format

This is a very general and simple format. However, it does not exploit any application specific structure in the matrix. The format used in the BlockSolve library exploits structure present in sparse matrices that arise in the solution of PDE's with multiple degrees of freedom. Figure 2(a) (adapted from [11]) illustrates a grid that would arise from 2-D, linear, multi-component finite-element model with three degrees of freedom at each discretization point. The degrees of freedom are illustrated by the three dots at each discretization point. The stiffness matrix for such model would have groups of rows with identical column structure called *i-nodes* ("identical nodes"). Non-zero values for each i-node can be gathered into a dense matrix as shown in Fig. 2(c).

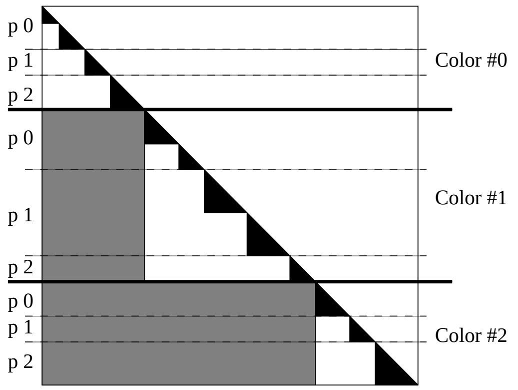
Such matrices are also rich in cliques (a partition into cliques is shown in Fig. 2(a) using dashed rectangles). The library colors the contracted graph induced by the cliques and reorders the matrix as shown in Fig. 2(b). For symmetric matrices, only the lower half is stored together with the diagonal. Black triangles along the diagonal correspond to dense matrices induced by the cliques. Gray off-diagonal blocks correspond to sparse blocks of the matrix (stored using i-nodes). Notice that the matrix is stored as a collection of smaller dense matrices. This fact helps reduce sparse storage overhead and improve performance of matrix-vector products.

For parallel execution, each color is divided among the processors. Therefore each processor receives several blocks of contiguous rows. On each processor, the off-diagonal blocks are actually stored by column (in column i-nodes). When performing a matrix-vector product, this storage organization makes the processing of messages containing non-local values of the vector more efficient. In addition, this allows the overlap of computation and communication by separating matrix-vector product into a portion which accesses only local data and one that deals with non-local data in incoming messages.

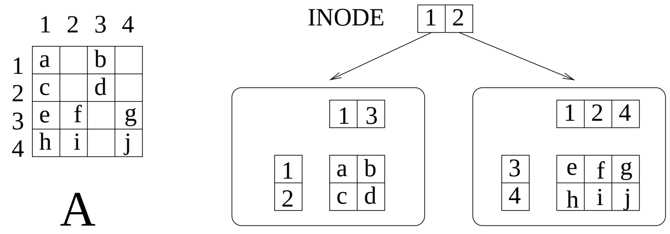
The main algorithm we will consider in this paper is matrix-vector product which is the core computation in iterative solvers for linear systems. Consider the performance (in Mflops) of sparse matrix-vector product on a single processor of an IBM SP-2 for a variety of matrices and storage formats, shown in Table 1 (descriptions of the matrices and the formats can be found in Appendix A). Boxed numbers indicate the highest performance for a given matrix. It is clear from this set of experiments that there is no single format that is appropriate for all kinds of problems. This demonstrates the difficulty of developing a "sparse BLAS" for sparse matrix computations. Even if we limit ourselves to the formats in Table 1, one still has to provide at least $6^2 = 36$ versions of sparse matrix-matrix product



(a) A subgraph generated by 2D linear finite element model



(b) Color/clique reordering in the BlockSolve library



(c) I-node storage

Figure 2: Illustration of the BlockSolve format

(assuming that the result is stored in a single format)!

The lack of extensibility in the sparse BLAS approach has been addressed by object-oriented solver libraries, like the PETSc library from Argonne [4]. Such libraries provide templates for a certain class of solvers (for example, Krylov space iterative solvers) and allow a user to add new formats by providing hooks for the implementations of some algebraic operations (such as matrix-vector product). However, in many cases the implementations of matrix-vector products themselves are quite tedious (as is the case in the BlockSolve library). Also, these libraries are not very useful in developing new algorithms.

A radically different solution is to *generate* sparse matrix programs by using restructuring compiler technology. The compiler is given a *dense* matrix program with declarations about which matrices are actually sparse, and it is responsible for choosing appropriate storage formats and for generating sparse matrix programs. This idea has been explored by Bik and Wijshoff [6, 7], but their approach is limited to simple sparse matrix formats that are not representative of those used in high-performance codes. Intuitively, they trade the ability to handle a variety of formats for the ability to compile arbitrary loop nests.

We have taken a different approach. Previously, we have shown how efficient sparse sequential code can be generated for a variety of storage formats for DOALL loops and loops with reductions [13, 14]. Our approach is based on viewing arrays as *relations*, and the execution of loop nests as evaluation of *relational queries*. We have demonstrated that our method of describing storage formats through access methods is general enough to specify

Name	Diagonal	Coordinate	CRS	ITPACK	JDiag	BS95
`small`	$\boxed{21.972}$	8.595	16.000	7.446	21.818	2.921
`medium`	23.192	7.888	29.874	8.150	$\boxed{32.583}$	19.633
`cfd.1.10`	8.730	4.459	21.678	6.793	11.793	$\boxed{26.070}$
`685_bus`	1.133	5.379	20.421	4.869	$\boxed{31.406}$	2.475
`bcsstm27`	15.130	4.807	23.677	7.714	21.604	$\boxed{28.907}$
`gr_30_30`	$\boxed{29.495}$	4.660	18.136	7.764	22.857	5.374
`memplus`	0.268	4.648	$\boxed{15.299}$	0.250	12.111	4.138
`sherman1`	18.699	5.191	16.756	6.094	$\boxed{28.069}$	2.187

Table 1: Performance of sparse matrix-vector product

a variety of formats yet specific enough to allow important optimizations. Since the class of "DOANY" loops covers not only matrix-vector and matrix-matrix products, but also important kernels within high-performance implementations of direct solvers and incomplete preconditioners, this allows us to address the needs of a number of important applications. One can think of our sparse code generator as providing an extensible set of sparse BLAS codes, which can be used to implement a variety of applications, just like dense BLAS routines.

For parallel execution, one need to specify how data and computation are partitioned. Such information (we call it *distribution relation*) can come in a variety of formats. Just as is the case with sparse matrix formats, the distribution relation formats are also application dependent. In the case of regular block/cyclic distributions the distribution relations can be specified by a closed-form formula. This allows ownership information to be computed at compile-time. However, regular distributions might not provide adequate load-balance in many irregularly structured applications.

The HPF-2 standard [9] provides for two kinds of irregular distributions: *generalized block* and *indirect*. In generalized block distribution, each processor receives a single block of continuous rows. It is suggested in the standard that each processor should hold the block sizes for all processors – that is the distribution relation should be replicated. This permits ownership to be determined without communication. Indirect distributions are the most general: the user provides an array `MAP` such that the element `MAP`(i) gives the processor to which the ith row is assigned. The `MAP` array itself can be distributed a variety of ways. However, this can require communication to determine ownership of non-local data.

The Chaos library [15] allows the user to specify partitioning information by providing the list of row indices assigned to each processor. The list of indices are transferred into *distributed translation table* which is equivalent to having a `MAP` array partitioned blockwise. This scheme is as general as the indirect scheme used in HPF-2 and it also requires communication to determine ownership and to build the translation table.

As we have already discussed, the partitioning scheme used in the BlockSolve library is somewhat different. It is more general than the generalized block distribution provided by HPF-2, yet it has more structure than the indirect distribution. Furthermore, the distribution relation in the BlockSolve library is replicated, since each processor usually receives only a small number of contiguous rows.

Our goal is to provide a parallel code generation strategy with the following properties:

- The strategy should not depend on having a fixed set of sparse matrix formats.
- It should not depend on having a fixed set of distributions.
- The system should be *extensible*. That is it should be possible to add new formats without changing the overall code generation mechanism.
- At the same time, the generality should not come at the expense of performance. The compiler must exploit structure available in sparse matrix and partitioning formats.

To solve this problem we extend our relational approach to the generation of parallel sparse code starting from dense code, a specification of sparse matrix formats and data partitioning information. We view arrays as distributed relations and parallel loop execution as distributed query evaluation. In addition, different ways of representing partitioning information (regular and irregular) are unified by viewing distribution maps themselves as relations.

Here is the outline of the rest of the paper. In Section 2, we outline our relational approach to sequential sparse code generation. In Section 3, we describe our sparse parallel code generation algorithm. In Section 4, we present experimental evidence of the advantages of our approach. Section 5 presents a comparison with previous work. Section 6 presents conclusions and ongoing work.

2 Relational Model of Sparse Code Generation

Consider the matrix-vector product $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$:

```
DO i = 1, N
  DO j = 1, N
    Y(i) = Y(i) + A(i,j) * X(j)
```

Suppose that the matrix $\mathbf{A}$ and the vector $\mathbf{x}$ are sparse, and that the vector $\mathbf{y}$ is dense. To execute this code efficiently, it is necessary to perform only those iterations (i,j) for which A(i,j) and X(j) are not zero. This set of iterations can be described by the following set of constraints:

$$\begin{cases} 1 \leq i \leq N \wedge 1 \leq j \leq N \\ A(i,j,a) \wedge X(j,x) \wedge Y(i,y) \\ a \neq 0 \wedge x \neq 0 \end{cases} \tag{1}$$

The first row represents the loop bounds. The constraints in the second row associate values with array indices: for example, the predicate $A(i,j,a)$ constraints a to be the value of A(i, j). Finally, the constraints in the third row specify which iterations update $\mathbf{Y}$ with non-zero values.

Our problem is to compute an efficient enumeration of the set of iterations specified by the constraints (1). For these iterations, we need efficient access to the corresponding entries in the matrices and vectors. Since the constraints are not linear and the sets being computed are not convex, we cannot use methods based on polyhedral algebra, such as Fourier-Motzkin elimination [2], to enumerate these sets.

Our approach is based on relational algebra, and it models A, X and Y as *relations* (tables) that hold tuples of array indices and values. Conceptually, the relation corresponding to a sparse matrix contains both zero and non-zero values. We view the iteration space of the loop as a relation I of $\langle i, j\rangle$ tuples. Then we can write the first two rows of constraints from (1) as the following *relational query* (relational algebra notation is summarized in Appendix B):

$$Q_{\text{dense}} = I(i,j) \bowtie A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y) \tag{2}$$

To test if elements of sparse arrays A and X are non-zero, we use predicates $NZ(A(i,j))$ and $NZ(X(j))$. Notice that because Y is dense, $NZ(Y(i))$ evaluates to *true* for all array indices $1 \leq i \leq N$. Therefore, the constraints in the third row of (1) can be now rewritten as:

$$\mathcal{P} = NZ(A(i,j)) \wedge NZ(X(j)) \tag{3}$$

The predicate $\mathcal{P}$ is called the *sparsity predicate*. We use the algorithm of Bik and Wijshoff [6, 7] to compute the sparsity predicate in general.

Using the definition of the sparsity predicate, we can finally write down the query which defines the indices and values in the sparse computation:

$$Q_{\text{sparse}} = \sigma_{\mathcal{P}} Q_{\text{dense}} = \sigma_{\mathcal{P}}\Big(I(i,j) \bowtie A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y)\Big) \tag{4}$$

(σ is the relational algebra *selection* operator.)

We have now reduced the problem of efficiently enumerating the iteration points that satisfy the system of constraints (1) to the problem of efficiently computing a relational query involving selections and joins. This problem in turn is solved by determining an efficient order in which the joins in (4) should be performed and determining how each of the joins should be implemented. These decisions depend on the storage formats used for the sparse arrays.

2.1 Describing Storage Formats

Following ideas from relational database literature [16, 20], each sparse storage format is described in terms of its *access methods* and their properties. Unlike database relations, which are usually stored as "flat" collections of tuples, most sparse storage formats have hierarchical structure, which must be exploited for efficiency. For example, the CCS format does not provide a way of enumerating row indices without first accessing a particular column. We use the following notation to describe such hierarchical structure of array indices:

$$J \succ (I, V) \tag{5}$$

which means that for a given column index j we can access a set of $\langle i, v\rangle$ tuples of row indices and values of the matrix. The $\succ$ operator is used to denote the hierarchy of array indices.

For each term in the hierarchy (J and (I, V) in the example), the programmer must provide methods to search and enumerate the indices at that level, and must specify the properties of these methods such as the cost of the search or whether the enumeration produces sorted output. These methods and their properties are used to determine good join orders and join implementations for each relational query extracted from the program, as described in [14].

This way of describing storage formats to the compiler through access methods and properties solves the *extensibility* problem: a variety of storage formats can be described to the compiler, and the compilation strategy does not depend on a fixed set of formats. For the details on how the formats are specified to the compiler, see [13].

2.2 Index Translations

Some sparse formats such as jagged-diagonal storage involve permutations of row and column indices. Permutations and other kinds of index translations can be easily incorporated into our framework. Suppose we have a permutation P which is stored using two integer arrays: `PERM` and `IPERM` – which represent the permutation and its inverse. We can view P as a relation of tuples $\langle i, i'\rangle$, where i is the original index and i' is the permuted index.

Now suppose that rows of the matrix in our example have been permuted using P. Then we can view A as relation of $\langle i', j, a\rangle$ tuples and the query for sparse matrix-vector product is:

$$Q_{\text{sparse}} = \sigma_{\mathcal{P}}\Big(I(i,j) \bowtie X(j,x) \bowtie Y(i,y) \bowtie P(i,i') \bowtie A(i',j,a)\Big) \tag{6}$$

where the sparsity predicate is: $\mathcal{P} = NZ(A'(i',j)) \wedge NZ(X(j))$.

2.3 Summary

Here are the highlights of our approach:

- Arrays (sparse and dense) are relations
- Access methods define the relation as a *view* of the data structures that implement a particular format
- We view loop execution as relational query evaluation
- The query optimization algorithm only needs to know the high-level structure of the relations as provided by the access methods and not the actual implementation (e.g. the role of the `COLP` and `ROWIND` arrays in the CCS storage).
- Permutations also can be handled by our compiler
- The compilation algorithms are independent of any particular set of storage formats and new storage formats can be added to the compiler.

3 Generating parallel code

Ancourt et al. [1] have described how the problem of generating SPMD code for dense HPF programs can be reduced to the computation of expressions in polyhedral algebra. We now describe how the problem of generating *sparse* SPMD code for a loop nest can be reduced to the problem of evaluating relational algebra queries over distributed relations. Section 3.1 describes how distributed arrays are represented. Section 3.2 describes how a distributed query is translated into a sequence of local queries and communication statements. In Section 3.3 we discuss how our code generation algorithm is used in the context of the BlockSolve data structures.

3.1 Representing distributed arrays

In the uniprocessor case, relations are high-level views of the underlying data structures. In the parallel case, each relation is a *view* of the partitions (or *fragments*) stored on each processor. The formats for the fragments are defined using access methods as outlined in Sec. 2.1. The problem we must address is that of describing distributed relations from the fragments.

Let's start with the following simple example:

- The matrix $\mathbf{A}$ is partitioned by row. Each processor p gets a fragment matrix $\mathbf{A}^{(p)}$.
- Let i and j be the row and column indices of an array element in the original matrix, and let i' and j' be the corresponding indices in a fragment $\mathbf{A}^{(p)}$. Because the partition is by row, the the column indices are the same ($j = j'$). However $i \neq i'$. i is the global row index, whereas i' can be thought of the *local row offset*. To translate between i and i', each processor keeps an integer array $\mathtt{IND}^{(p)}$ such that $\mathtt{IND}^{(p)}(i') = i$. That is, each processor keeps the list of global row indices assigned to it.

How do we represent this partition?

Notice that on each processor p the array $\mathtt{IND}^{(p)}$ can be viewed as a relation $IND^{(p)}(i, i')$. The local fragment of the matrix can also be viewed as a relation: $A^{(p)}(i', j, a)$. We can define the global matrix as follows:

$$A(i, j, a) = \bigcup_p \pi_{i,j,a} \left(IND^{(p)}(i, i') \bowtie A^{(p)}(i', j, a) \right) \tag{7}$$

(The *projection* operator π is defined in the Appendix B.)

In this case, each processor p carries the information that translates its own fragment $\mathbf{A}^{(p)}$ into the contribution to the global relation. But there are other situations, when a processors other than p might own the translation information for the fragment stored on p. A good example is the distributed translation table used in the Chaos library [15]. Suppose that the global indices fall into the range $0 \leq i \leq N - 1$ for some N. Also, let P be the number of processors. Let $B = \lceil N/P \rceil$. Then for a given global index i the index of the owner processor p and the the local offset i' are stored on processor

$$q = \lfloor i/B \rfloor \tag{8}$$

Each processor q holds the array of $\langle p, i' \rangle$ tuples indexed by

$$h = i \bmod B \tag{9}$$

We need a general way of representing such index translation schemes. The key is to view the index translation relation itself as a distributed relation. Then, in the first case this global relation is defined as:

$$IND(i, p, i') = \bigcup_p IND^{(p)}(i, i') \times \{p\} \tag{10}$$

In the example from the Chaos library, the relation is defined by:

$$IND(i, p, i') = \bigcup_q \pi_{i,p,i'} \Big(BLOCK(i, q, h) \bowtie IND^{(q)}(h, p, i') \Big) \tag{11}$$

where $IND^{(q)}(h, p, i')$ is the view of the above mentioned array of $\langle p, i' \rangle$ tuples and the relation $BLOCK(i, q, h)$ is the shorthand for the constraints in (8) and (9).

Once we have defined the index translation relation $IND(i, p, i')$, we can rewrite (7) as:

$$A(i, j, a) = \bigcup_p \pi_{i,j,a} \Big(IND(i, p, i') \bowtie A^{(p)}(i', j, a) \Big) \tag{12}$$

where IND can be defined by, for example, (10) or (11).

Similarly, we can define the global relations X and Y for the vectors in the matrix-vector product (assuming they are distributed the same way as the rows of A):

$$X(j, x) = \bigcup_p \pi_{j,x} \Big(IND(j, p, j') \bowtie X^{(p)}(j', x) \Big) \tag{13}$$

$$Y(i, y) = \bigcup_p \pi_{i,y} \Big(IND(i, p, i') \bowtie Y^{(p)}(i', y) \Big) \tag{14}$$

In general, distributed relations are described by:

$$R(\mathbf{a}) = \bigcup_p \pi_{\mathbf{a}} \Big(IND(\mathbf{a}, p, \mathbf{a}') \bowtie R^{(p)}(\mathbf{a}') \Big) \tag{15}$$

where R is the distributed relation, $R^{(p)}$ is the fragment on processor p and IND is the global-to-local index translation relation. The index translation relation can be different for different arrays, but we assume that it always specifies a 1-1 mapping between the global index $\mathbf{a}$ and the pair $\langle p, \mathbf{a}' \rangle$. Notice that our example partitioning of the IND relation in (10) and (11) themselves satisfy definition (15). We call (15) the *fragmentation equation*.

How do we specify the distribution of computation? Recall that the iteration set of the loop is also represented as a relation: $I(i, j)$, in our matrix-vector product example. We

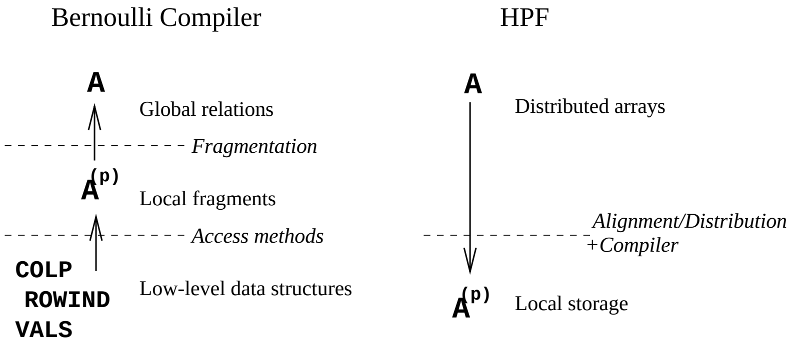


Figure 3: Flow of information in HPF and Bernoulli Compiler

could require the user to supply the full fragmentation equation for I. But this would be too burdensome: the user would have to provide the local iteration set $I^{(p)}$, but this set should really be determined by the compiler using some policy (such as the owner-computes rule). In addition, because the relation I is not stored, there is no need to allow multiple storage formats for it. Our mechanisms are independent of the policy used to determine the distribution relation for iterations; given any distribution relation IND, we can define the local iteration set by:

$$I^{(p)}(i', j) = \pi_{i',j}\Big(IND(i, p, i') \bowtie I(i, j)\Big) \tag{16}$$

This simple definition allows us to treat the iteration set relation I uniformly together with other relations in question.

Notice that the fragmentation equation (15) is more explicit than the alignment-distribution scheme used in HPF. In the Bernoulli compiler global relations are described through a *hierarchy of views*: first local fragments are defined through access methods as the views of the low-level data structures. Then the global relations are defined as views of the local fragments through the fragmentation equation.

In HPF, alignment and distribution provide the mapping from global indices to processors, but not the full global-to-local index translation. Local storage layout (and the full index translation) is derived by the compiler. This removes from the user the responsibility for (and flexibility in) defining local storage formats. The difference in the flow of information between HPF and Bernoulli Compiler is illustrated in Fig. 3.

By mistake, the user may specify inconsistent distribution relations IND. These inconsistencies, in general, can only be detected at runtime. For example, it can only be verified at run-time if a user specified distribution relation IND in fact provides a 1-1 and onto map. This problem is not unique to our framework – HPF with value-based distributions [21] has a similar problem. Basically, if a function is specified by its values at run-time, its properties can only be checked at run-time. It is possible to generate a "debugging" version of the code, that will check the consistency of the distributions, but this is beyond the scope of this paper.

3.2 Translating distributed queries

Let us return to the query for sparse matrix-vector product:

$$Q_{\text{sparse}} = \sigma_{\mathcal{P}} Q_{\text{dense}} = \sigma_{\mathcal{P}} \Big(I(i,j) \bowtie A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y) \Big) \tag{17}$$

The relations A, X and Y are defined by (12), (13) and (14). We translate the distributed query (17) into a sequence of local queries and communication statements by expanding the definitions of the distributed relations and doing some algebraic simplification, as follows.

3.2.1 General strategy

In the distributed query literature the optimization problem is: *find the sites that will evaluate parts of the query (17).* In the context of, say, a banking database spread across branches of the bank, the partitioning of the relations is fixed, and may not be optimal for each query submitted to the system. This is why the choice of sites might be non-trivial in such applications. See [20] for a detailed discussion of the general distributed query optimization problem.

In our case, we expect that the placement of the relations is correlated with the query itself and is given to us by the user. In particular, the placement of the iteration space relation I tells us where the query should be processed. That is the query to be evaluated on each processor p is:

$$Q^{(p)} = \sigma_{\mathcal{P}} Q_{\text{dense}} = \sigma_{\mathcal{P}} \Big(I^{(p)}(i,j) \bowtie A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y) \Big) \tag{18}$$

where $I^{(p)}$ is the set of iterations assigned to processor p. We resolve the references to the global relations A, X and Y by, first, exploiting the fact that the join between some of them (in this case A and Y) do not require any communication at all and can be directly translated into the join between the local fragments. Then, we resolve the remaining references by computing communication sets (and performing the actual communication) for other relations (X in our example).

We now outline the major steps.

3.2.2 Exploiting collocation

In order to expose the fact that the join between A and Y can be done without communication, we expand the join using the definitions of the relations:

$$\begin{aligned} &A(i,j,a) \bowtie_i Y(i,y) = \\ &\left(\bigcup_p \pi_{i,j,a} \Big(IND(i,p,i') \bowtie A^{(p)}(i',j,a) \Big) \right) \bowtie_i \left(\bigcup_q \pi_{i,y} \Big(IND(i,q,i'') \bowtie Y^{(q)}(i'',y) \Big) \right) \end{aligned} \tag{19}$$

Because we have assumed that the index translation relation IND provides a 1-1 mapping between global index and processor numbers, we can deduce that $p = q$. This is nothing more than the statement of the fact that A and Y are aligned [3, 5]. So the join between A and Y can be translated into:

$$A(i,j,a) \bowtie_i Y(i,y) = \bigcup_p \pi_{i,j,a,y}\Big(IND(i,p,i') \bowtie_{i'} A^{(p)}(i',j,a) \bowtie_{i'} Y^{(p)}(i',y)\Big) \tag{20}$$

Notice that the join on the global index i has been translated into the join on the local offsets $i' = i''$. The sparsity predicate $\mathcal{P}$ originally refers to the distributed relations: $\mathcal{P} = NZ(A(i,j)) \wedge NZ(X(j))$. In the translated query, we replace the references to the global relations with the references to the local relations.

3.2.3 Generating communication

The query:

$$Used_X^{(p)}(j) = \pi_j \sigma_{NZ(A^{(p)}(i',j))}\Big(A^{(p)}(i',j,a) \bowtie Y^{(p)}(i',y)\Big) \tag{21}$$

computes the set of global indices j of X that are referenced by each processor. The join of this set with the index translation relation will tell us where to get each element:

$$RecvInd_X^{(p)}(j,q,j') = Used_X^{(p)}(j) \bowtie IND(j,q,j') \tag{22}$$

This tells us which elements of X must be communicated to processor p from processor q. If the IND relation is distributed (as is the case in the Chaos library), then evaluation of the query (22) might itself require communication. This communication can also be expressed and computed in our framework by applying the parallel code generation algorithm recursively.

3.2.4 Summary

Here is the summary of our approach:

- We represent distributed arrays as distributed relations.
- We represent global-to-local index translation relations as distributed relations.
- We represent parallel DOANY loop execution as distributed query evaluation.
- For compiling dense HPF programs, Ancourt et al. [1] describe how the computation sets, communication sets etc. can be described by expressions in polyhedral algebra. We derive similar results for sparse programs, using relational algebra.

3.3 Compiling for the BlockSolve formats.

As was discussed in the introduction, the BlockSolve library splits the matrix into two disjoint data structures: the collection of dense matrices along the diagonal, shown using black triangles in Figure 2(b), and the off-diagonal sparse portions of the matrix stored using i-node format (Figure 2(c)).

In the computation of a matrix-vector product $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ the dense matrices along the diagonal refer only to the local portions of the vector $\mathbf{x}$. Also the off-diagonal sparse blocks are stored in a way that makes it easy to enumerate separately over those elements of the matrix that refer only to the local elements of $\mathbf{x}$ and over those that require communication.

Altogether, we can view a matrix $\mathbf{A}$ stored in the BlockSolve library format as a sum $\mathbf{A}_D + \mathbf{A}_{SL} + \mathbf{A}_{SNL}$, where:

- $\mathbf{A}_D$ represents the dense blocks along the diagonal
- $\mathbf{A}_{SL}$ represents the portions of the sparse blocks that refer to local elements of $\mathbf{x}$
- $\mathbf{A}_{SNL}$ represents the portions of the sparse blocks that refer to non-local elements of $\mathbf{x}$

$\mathbf{A}_D$, $\mathbf{A}_{SL}$ and $\mathbf{A}_{SNL}$ are all partitioned by row. The distribution in the library assigns a small number of continuous rows to each processor. The distribution relation is also replicated, thus reducing the cost of computing the ownership information.

The hand-written library code does not have to compute any communication sets or index translations for the products involving $\mathbf{A}_D$ and $\mathbf{A}_{SL}$ – these portions of the matrix access directly the local elements of $\mathbf{x}$.

How can we use our code generation technology to produce code competitive with the hand-written code?

The straight-forward approach is to start from the sequential dense matrix data-parallel program for matrix-vector product. Since the matrix is represented as three fragments ($\mathbf{A}_D$, $\mathbf{A}_{SL}$ and $\mathbf{A}_{SNL}$), our approach essentially computes three matrix vector products:

$$\begin{aligned} &\mathbf{y} = \mathbf{A}_D \cdot \mathbf{x} \\ &\mathbf{y} = \mathbf{y} + \mathbf{A}_{SL} \cdot \mathbf{x} \\ &\mathbf{y} = \mathbf{y} + \mathbf{A}_{SNL} \cdot \mathbf{x} \end{aligned} \tag{23}$$

The performance of this code is discussed in the next section. Careful comparison of this code with the handwritten code reveals that the performance of our code suffers from the fact that even though the products involving $\mathbf{A}_D$ and $\mathbf{A}_{SL}$ do not require any communication, they still require global-to-local index translation for the elements of $\mathbf{x}$ that are used in the computation. If we view $\mathbf{A}_D$ and $\mathbf{A}_{SL}$ as global relations that stored global row and column indices, then we hide the fact that the local indices of $\mathbf{x}$ can be determined directly from the data structures for $\mathbf{A}_D$ and $\mathbf{A}_{SL}$. This redundant index translation introduces extra level of indirection in the accesses to $\mathbf{x}$ and degrades node program performance. At this point, we have no automatic approach to handling this problem.

We can however circumvent the problem at the cost of increasing the complexity of the input program by specifying the code for the products with $\mathbf{A}_D$ and $\mathbf{A}_{SL}$ at the node program level. The code for the product with $\mathbf{A}_{SNL}$ is still specified at the global (data-parallel) level:

$$\begin{aligned} &\text{local:} && \mathbf{y}^{(p)} = \mathbf{A}_{D^{(p)}} \cdot \mathbf{x}^{(p)} \\ &\text{local:} && \mathbf{y}^{(p)} = \mathbf{y}^{(p)} + \mathbf{A}_{SL}^{(p)} \cdot \mathbf{x}^{(p)} \\ &\text{global:} && \mathbf{y} = \mathbf{y} + \mathbf{A}_{SNL} \cdot \mathbf{x} \end{aligned} \tag{24}$$

where $\mathbf{y}^{(p)}$, etc are the local portions of the arrays and $\mathbf{y}$, $\mathbf{A}_{SNL}$ and $\mathbf{x}$ are the global views. The compiler then generates the necessary communication and index translations for the product with $\mathbf{A}_{SNL}$. This *mixed* specification (both data-parallel and node level programs) is not unique to our approach. For example, HPF allows the programmer to "escape" to the node program level by using extrinsics [9].

In general, sophisticated composite sparse formats, such as the one used in the BlockSolve library, might require algorithm specification at a different level than just a dense loop. We are currently exploring ways of specifying storage formats so that we can get good sequential performance without having to drop down to node level programs for some parts of the application.

4 Experiments

In this section, we present preliminary performance measurements on the IBM SP-2. The algorithm we studied is a parallel Conjugate Gradient [18] solver with diagonal preconditioning (CG), which solves large sparse systems of linear equations iteratively. Following the terminology from Chaos project, the parallel implementation of the algorithm can be divided into the *inspector* phase and the *executor* phase [15]. The inspector determines the the set of values to be communicated and performs some other preprocessing. The executor performs the actual computation and communication. In iterative applications the cost of the inspector can usually be amortized over several iterations of the executor.

In order to verify the quality of the compiler-generated code and to demonstrate the benefit of using the mixed local/global specification (24) of the algorithm in this application we have measured the performance of the inspector and the executor in the following implementations of the CG algorithm:

- `BlockSolve` is the hand-written code from the BlockSolve library.
- `Bernoulli-Mixed` is the code generated by the compiler starting from the mixed local/global specification in (24).
- `Bernoulli` is the "naive" code generated by the compiler starting from fully data-parallel specification (23).

We ran the different implementations of the solver on a set of synthetic three-dimensional grid problems. The connectivity of the resulting sparse matrix corresponds to a 7-point stencil with 5 degrees of freedom at each discretization point. Then, we ran the solver on 2, 4, 8, 16, 32 and 64 processors of the IBM SP-2 at Cornell Theory Center. During each run we kept the problem size per processor constant at $30 \times 30 \times 30$. This places 135×10^3 rows with about 4.5×10^6 non-zeroes total on each processor. We limited the number of solver iterations to 10. Tab. 2 shows the times (in seconds) for the numerical solution phase (the executor). Tab. 3 shows the overhead of the inspector phase as the ratio of the time taken by the inspector to the time taken by a *single* iteration of the executor.

The comparative performance of the `Bernoulli-Mixed` and `BlockSolve` versions verifies the quality of the compiler generated code. The 2-4% difference is due to aggressive

P	BlockSolve	Bernoulli-Mixed		Bernoulli	
	sec	sec	diff.	sec	diff.
2	1.67	1.70	2%	1.81	8%
4	1.73	1.76	2%	1.90	10%
8	1.83	1.83	0%	1.99	9%
16	1.85	1.91	3%	2.04	10%
32	1.96	2.01	3%	2.15	10%
64	2.03	2.11	4%	2.32	14%

Table 2: Numerical computation times (10 iterations)

P	BlockSolve	Bernoulli-Mixed	Bernoulli	Indirect-Mixed	Indirect
2	0.03	0.05	2.21	2.21	4.97
4	0.06	0.13	2.32	3.18	7.21
8	0.08	0.15	2.31	4.37	7.99
16	0.11	0.18	2.35	4.82	9.22
32	0.12	0.22	2.37	5.97	11.16
64	0.13	0.27	2.37	7.96	13.45

Table 3: Inspector overhead

overlapping of communication and computation done in the hand-written code. Currently, the Bernoulli compiler generates simpler code, which first exchanges the non-local values of `x` and then does the computation. While the inspector in `Bernoulli-Mixed` code is about twice as expensive as that in the `BlockSolve` code, its cost is still quite negligible (2.7% of the executor with 10 iterations).

The comparison of the `Bernoulli` and `Bernoulli-Mixed` code illustrates the importance of using the mixed local/global specification (24). The `Bernoulli` code has to perform redundant work in order to discover that most of the references to `x` are in fact local and do not require communication. The amount of this work is proportional to the problem size (the number of unknowns) and is much larger than the number of elements of `x` that are actually communicated. As the result, the inspector in the `Bernoulli` code is an order of magnitude more expensive than the one in the `BlockSolve` or `Bernoulli-Mixed` implementations. The performance of the executor also suffers because of the redundant global-to-local translation, which introduces an extra level of indirection in the final code even for the local references to `x`. As the result, the executor in `Bernoulli` code is about 10% slower than in the `Bernoulli-Mixed` code.

To demonstrate the benefit of exposing structure in distribution relations, we have measured the inspector overhead for using the *indirect* distribution format from the HPF-2 standard [9]. We have implemented two versions of the inspectors using the support for the indirect distribution in the Chaos library [15]:

- `Indirect-Mixed` is the inspector for the mixed local/global specification of (24).

- `Indirect` is the inspector for the fully data parallel specification.

Tab. 3 shows the ratio of the time taken by the `Indirect-*` inspectors to the time taken by the single iteration of the `Bernoulli-*` executors – the executor code is exactly the same in both cases and we have only measured the executors for the `Bernoulli-*` implementations.

The order of magnitude difference between the performance of `Indirect-Mixed` and `Bernoulli-Mixed` inspectors is due to the fact that the `Indirect-Mixed` inspector has to perform asymptotically more work and requires expensive communication. Setting up the distributed translation table in the `Indirect-Mixed` inspector, which is necessary to resolve non-local references, requires the round of all-to-all communication with the volume proportional to the problem size (i.e. the number of unknowns). Additionally, querying the translation table (in order to determine the ownership information) again requires all-to-all communication: for each global index j the processor $q = j/B$ for some block size B is queried for the ownership information – even though the communication pattern for our problems has limited "nearest-neighbor" connectivity.

The difference between `Indirect` and `Bernoulli` inspectors is not as pronounced – the number of references that has to be translated is proportional to the problem size. Still, the `Indirect` inspector has to perform all-to-all communication to determine the ownership of the non-local data.

The relative effect of the inspector performance on the overall solver performance depends, of course, on the number of iterations taken by the solver, which, in turn, depends on the condition number of the input matrix. To get a better idea of the relative performance of the `Bernoulli-Mixed` and `Indirect-Mixed` implementation for a range of problems we have plotted in Fig. 4 the ratios of the time that the `Indirect-Mixed` implementation would take to the time that the `Bernoulli-Mixed` implementation would take on 8 and 64 processors for a range of iteration counts $5 \leq k \leq 100$. The lines in Fig. 4 plot the values of the ratio:

$$\frac{k + r_I}{k + r_B} \tag{25}$$

where r_B is the inspector overhead for the `Bernoulli-Mixed` version, r_I is inspector overhead for the `Indirect-Mixed` version and k is the iteration count. A simple calculation shows that it would take 77 iterations of an `Indirect-Mixed` solver on 64 processors to get within 10% of the performance of the `Bernoulli-Mixed`. On 8 processors the number is 43 iterations. To get within 20% it would take 21 and 39 iteration on 8 and 64 processors, respectively.

These data demonstrate that, while the inspector cost is somewhat amortized in an iterative solver, it is still important to exploit the structure in distribution relations – it can lead to order of magnitude savings in the inspector cost and improves the overall performance of the solver.

It should also be noted that the `Indirect-Mixed` version is not only slower than the two Bernoulli versions but also requires more programming effort. Our compiler starts with the specification at the level of dense loops both in (23) and (24), whereas an HPF compiler needs sequential *sparse* code as input. For our target class of problems – sparse DOANY loops – our approach results in better quality of parallel code while reducing programming effort.

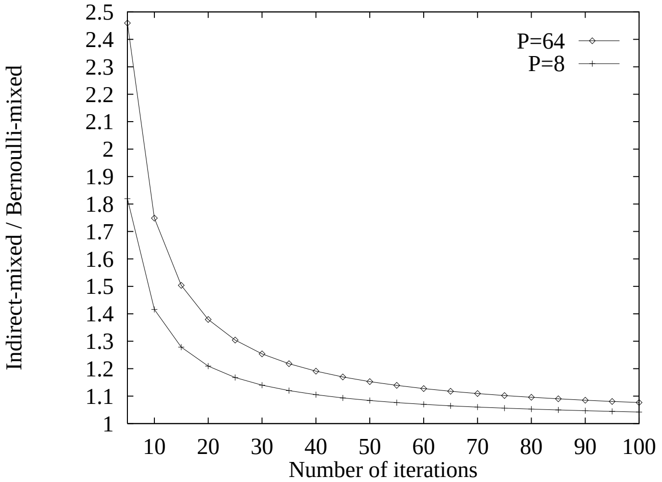


Figure 4: Effect of problem conditioning on the relative performance

5 Previous work

The closest alternative to our work is a combination of Bik's sparse compiler [6, 7] and the work on specifying and compiling sparse codes in HPF Fortran [19, 21, 22]). One could use the sparse compiler to translate dense sequential loops into sparse loops. Then, the Fortran D or Vienna Fortran compiler can be used to compile these sparse loops. However, both Bik's work and the work done by Ujaldon et al. on reducing inspector overheads in sparse codes limits a user to the fixed set of sparse matrix storage and distribution formats, this reducing possibilities for exploiting problem-specific structure.

6 Conclusions

We have presented an approach for compiling parallel sparse codes for user-defined data structures, starting from DOANY loops. Our approach is based on viewing parallel DOANY loop execution as relational query evaluation and sparse matrices and distribution information as distributed relations. This *relational* approach is general enough to represent a variety of storage formats.

However, this generality does not come at the expense of performance. We are able to exploit both the properties of the distribution relation in order to produce inexpensive inspectors, as well as produce quality numerical code for the executors. Our experimental evidence shows that both are important for achieving performance competitive with hand-written library codes.

So far, we have focused our efforts on the versions of iterative solvers, such as the Conjugate Gradient algorithm, which do not use incomplete factorization preconditioners. The core operation in such solvers is the sparse matrix-vector product or the product of a sparse matrix and a skinny dense matrix. We are currently investigating how our techniques can be used in the automatic generation of high-performance codes for such operations as matrix factorizations (full and incomplete) and triangular linear system solution.

References

[1] Corinne Ancourt, Fabien Coelho, Franois Irigoin, and Ronan Keryell. A linear algebra framework for static hpf code distribution. In *CPC'93*, November 1993. http://cri.ensmp.fr/doc/A-250.ps.Z.

[2] Corinne Ancourt and Franois Irigoin. Scanning polyhedra with do loops. In *Principle and Practice of Parallel Programming*, pages 39–50, April 1991.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of PLDI'93*, June 1993. http://suif.stanford.edu/-papers/anderson93/paper.html.

[4] Argonne National Laboratory. *PETSc, the Portable, Extensible Toolkit for Scientific Computation*. http://www.mcs.anl.gov/petsc/petsc.html.

[5] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghil. Solving alignment using elementary linear algebra. In *Proceedings of the 7th LCPC Workshop*, August 1994. Available as Cornell Computer Science Dept. tech report TR95-1478.

[6] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31:14–24, 1995.

[7] Aart J.C. Bik and Harry A.G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109 – 126, 1996.

[8] R.F. Boisvert, R. Pozo, K. Remington, R.F. Barrett, and J.J. Dongarra. *The Quality of Numerical Software: Assessment and Enhancement*, chapter Matrix Market: a web resource for test matrix collections, pages 125–137. Chapman and Hall, London, 1997.

[9] High Performance Fortran Forum. High performance fortran language specification, version 2.0. http://www.crpc.rice.edu/HPFF/home.html.

[10] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Inc., 1981.

[11] Mark T. Jones and Paul E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL–95/48, Argonne National Laboratory, December 1995.

[12] D. Kincaid, J. Respess, D. Young, and R Grimes. Algorithm 586 ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software*, 8(3):302–322, September 1982.

[13] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel sparse code for user-defined data structures. In *Proceedings of Eights SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[14] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to sparse matrix compilation. In *EuroPar*, August 1997. Available as Cornell Computer Science Tech. Report 97–1627.

[15] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93*, pages 361–370, November 1993. ftp://hpsl.cs.umd.edu/pub/papers/comp-mapper.ps.Z.

[16] Raghu Ramakrishnan. *Database Management Systems.* College Custom Series. McGraw-Hill, Inc, beta edition, 1996.

[17] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using ELLPACK.* Springer-Verlag, New York, NY, 1985.

[18] Youcef Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.

[19] Manuel Ujaldon, Emilio Zapata, Barbara M. Chapman, and Hans P. Zima. New data-parallel language features for sparse matrix computations. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna, 1995. http://www.vcpc.univie.ac.at/activities/language.

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

[21] Rinhard v. Hanxleden, Ken Kennedy, and Joel Saltz. Value-based distributions and alignments in Fortran D. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993.

[22] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6), 1995. ftp://hyena.cs.umd.edu/pub/papers/ieee_toc.ps.Z.

Appendix

A Matrix formats

The matrices shown in Table 1 are obtained from the suite of test matrices supplied with the PETSc library [4] (`small`,`medium`,`cfd.1.10`) and from the Matrix Market [8] (`685_bus`, `bcsstm27`, `gr_30_30`, `memplus` and `sherman1`).

The *Diagonal* format is a variant on the banded storage: it stores an arbitrary set of diagonals. Instead of storing an entire diagonal only the entries between the first and last non-zero are stored. This is basically Skyline storage [10] re-oriented along the diagonals. A matrix in *Coordinate* format is stored in three arrays: the array of row indices, of column indices and of the values. Compressed Row Format *(CRS)* stores the transpose of the matrix using the CCS format described in Sec. 1. *ITPACK* format is described in [12, 17]. The Jagged Diagonal (*JDIAG*) format is described in [18]. The BlockSolve format is described in Sec. 1.

B Relational algebra notation

We use the notation $A(i, j, a)$ to name the fields in the relation A. We use bold letters, as in $R(\mathbf{a})$ to denote tuples of fields (like vectors). An equi-join on the common field x between relations $R(x, y)$ and $S(x, z)$ is defined by:

$$R(x, y) \bowtie_x S(x, z) = \{\langle x, y, z\rangle \mid \langle x, y\rangle \in R \wedge \langle x, z\rangle \in S\} \tag{26}$$

The selection operator $\sigma_{\mathcal{P}}$ selects the tuples that satisfy the predicate $\mathcal{P}$:

$$\sigma_{\mathcal{P}} R(\mathbf{a}) = \{\mathbf{a} \mid \mathbf{a} \in R \wedge \mathcal{P}(\mathbf{a})\} \tag{27}$$

The projection operator $\pi_{\mathbf{a}}$ projects a relation on the fields $\mathbf{a}$ (removing any duplicates):

$$\pi_{\mathbf{a}} R(\mathbf{a}, \mathbf{b}) = \{\mathbf{a} \mid \exists \mathbf{b} : \langle \mathbf{a}, \mathbf{b}\rangle \in R\} \tag{28}$$