RELATIONAL ALGEBRAIC TECHNIQUES FOR THE SYNTHESIS OF SPARSE MATRIX PROGRAMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Vladimir Kotlyar

January 1999

ABSTRACT

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 view the problem of supporting effective development of high-performance sparse matrix codes as one of *generic programming*. Generic programming is a discipline of designing and implementing software components which can be used when there is a set of *related data structures* supporting a common semantics described by an API or protocol, and a set of *common algorithms* that can be formulated in terms of this API. When designing a generic programming system one must address the following fundamental questions:

- How do we represent efficient algorithms independently of any particular data-representation scheme?
- How do we provide an interface to a diverse set of data-structures?
- How do we "knit" together the representation of the algorithms and the representation for the data to obtain an efficient implementation?

This dissertation presents a *relational algebraic model* for automatically generating efficient sparse codes starting with dense matrix codes and specification of sparse matrix formats. Our techniques are based on viewing arrays as relations and the execution of DOALL loop nests and loops with reductions as evaluation of queries over these relations. Storage formats are specified to the compiler through search and enumeration access methods and their costs. Code restructuring is then formulated as the search for the most efficient plan for the query. The main step in this process is the identification of simultaneous enumeration of data structures (relational joins) and the determination of the best implementations of this enumeration. This software architecture not only provides for a clean design of the compiler, but it also exposes additional opportunities for code optimization and has led us to more general transformation algorithms than previously reported in the literature.

We present experimental data that demonstrates that the code generated by our compiler achieves performance competitive with that of hand-written codes for important computational kernels.

Contents

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. It if often the case that while a dense matrix version of a numerical algorithm can be described using a dozen lines of Matlab script, the efficient sparse matrix implementation requires hundreds of lines of C or Fortran ([6, 46, 82]).

Libraries of basic linear algebra subroutines (BLAS) have been widely used in dense matrix computations [32, 33, 53]. There are a few ways of storing a dense matrix and, consequently, BLAS libraries have to provide only a few routines. In turn, highly optimized implementations of BLAS are available for each architecture thus facilitating modular development of other high performance dense matrix software, such as the LAPACK library [4].

The situation is radically different in the area of sparse matrix computations due to the presence of a variety of storage formats. There have been attempts at developing a BLAS like library of basic sparse computations, however they suffer from combinatorial explosion in the number of individual routines that library developers have to provide. For example, the NIST Sparse BLAS library [68] supports 13 formats and provides routines for computations where only a single operand is sparse (such as triangular solves). This leaves out important kernels such as the incomplete updates. The library also has no provisions for adding new formats, short of writing all the required routines by hand.

Instead, coarse-grain packages that target specific classes of problems have been developed and are in widespread use today. The layering that these packages use depends on the particular application. For example, the codes for the solution of sparse symmetric positive definite systems and least squares problems developed by at Cornell by Chunguang Sun [82, 83] implement Cholesky and QR factorizations by decomposing them into primitive sparse operations, such as matrix updates, and dense operations, such as partial Cholesky factorization of supernodes. Dense operations are implemented by calling the appropriate LAPACK routines and BLAS. However, the sparse operations are still coded by hand due to the lack of the sparse BLAS support. Another kind of layering is used in the PETSc library [6], which implements various Krylov space iterative solvers (among other things). The library views sparse matrices as algebraic objects that provide two methods: product $\mathbf{Ax}$ with a dense vector and, in the case of pre-conditioned iterative methods, the product $\mathbf{M}^{-1}\mathbf{z}$ of the preconditioner with a dense vector. A user can supply his own sparse matrix data structure by implementing this primitives. There are two problems with this approach. First, some high-performance implementations of sparse matrix-vector products are themselves quite tedious. For example, the *sequential* product in the BlockSolve library takes a few hundred lines of C. The parallel implementation is even worse: additional programming effort goes into setting up and performing communication. Second, these solver packages are limited to a particular class of problems and are not useful in the development of new algorithms.

An alternative approach to providing sparse matrix software has been suggested by Bik and Wijshoff [9, 16, 20]: use a compiler to generate sparse matrix codes automatically starting with dense codes. This approach has one missing ingredient: Bik's sparse compiler only provides the formats which are a variation on the CRS and CCS schemes. No provisions are made for extending the compiler with new storage formats, yet the choice of storage formats greatly affects performance. Our initial experiments have shown that even for such simple computations as matrix-vector product there is no clear choice of a sparse matrix storage format. Depending on the application, the "best" data structure produces a gain of 20-70% in performance over the "second best".

We view the problem of supporting effective development of high-performance sparse matrix codes as one of **generic programming** [62]. Generic programming is a discipline of designing and implementing software components which can be used when there is a set of *related data structures* supporting a common semantics described by an API or protocol, and a set of *common algorithms* that can be formulated in terms of this API. A generic programming system permits library writers to use the API to code generic procedures without any knowledge of what data structures will ultimately be used by the application programmer, and permits applications programmers to invoke generic procedures by passing in an application-specific data structure implementation supporting the necessary API. This gives the applications programmer the effect of having a software component customized for the application. When designing a generic programming system one must address the following fundamental questions [94]:

- How do we represent efficient algorithms independently of any particular data-representation scheme?
- How do we provide an interface to a diverse set of data-structures?
- How do we "knit" together the representation of the algorithms and the representation for the data to obtain an efficient implementation.

The most natural way of expressing sparse matrix computations is in terms of **dense loop nests**, since such specification of matrix algorithms can be readily found in standard texts. Different sparse matrix formats form the set of related data structures. In the rest of the thesis we address the remaining two questions:

- What is the API for specifying sparse matrix data structures?
- What is the "knitting" technology for producing efficient sparse code from dense loop nests and the specification of data structures?

Our approach is to use a **relational algebraic model** for automatically generating efficient sparse codes starting with dense matrix codes and specification of sparse matrix formats. Our techniques are based on viewing arrays as relations and execution of DOALL loop nests and loops with reductions as evaluation of queries over these relations. Storage formats are specified to the compiler through search and enumeration access methods and their costs. Code restructuring is then formulated as the search for the most efficient plan for the query. The main step in this process is the identification of simultaneous enumeration of data structures (relational joins) and the determination of the best implementations of this enumeration. This software architecture not only provides for the clean design of the compiler, but it also exposes additional opportunities for code optimization and has led us to more general transformation algorithms than previously reported in the literature.

We have also applied these relational techniques to parallel sparse code generation. In the case of regular dense codes, most of the compilation steps such as the allocation of local storage and generation of communication sets can be described using the algebra of polyhedra (that is, systems of linear equalities and inequalities). In the case of sparse codes, these steps can be described using relational algebra. Distributed arrays are represented as distributed relations and DOANY loops as distributed queries. To generate SPMD-style node programs, our compiler translates the distributed queries into sequences of local queries which compute communication sets and perform local computation. Experiments show that the performance of the code generated by the compiler for DOANY loops is within 2-4% of the performance of routines in hand-written libraries like the BlockSolve library from Argonne.

We start in Part I of this thesis by describing the core sparse matrix applications and related algorithms and motivating the focus of this thesis on so-called *conjunctive DOANY loops*. This class of loop nests covers many important computational kernels. Being able to automatically generate sparse code for such loop nests in effect provides the user with an extensible set of BLAS routines. Part I is concluded with the introduction of the relational model of arrays (sparse and dense) and loop nests.

Part II deals with techniques for generating sequential sparse codes. We start by introducing a two-level abstraction of sparse matrix storage formats. Low-level accesses to the data structures are described by the means of abstract data types and associated access methods (Chapter 3). High-level structure present in the formats is described by Black Boxes, which are special collections of ADTs (Chapter 4). Chapter 5 describes how sparse code is generated. The code generation process is defined in terms of rewrite rules which transform loops nests that enumerate over the results of relational queries. Part II is concluded with the comparison of the performance of compiler generated code and library codes.

Part III describes the techniques for generating parallel message-passing (SPMD) code. The main difference between our approach and that reported previously in data-parallel language and parallelizing compiler community [38, 86, 88] is that we give the user the control over the formats for local storage of the matrices as well as over the formats for storing the partitioning information. The parallel code generation algorithm is based on translating the queries over global (distributed) views of the data structures into communication statements and queries that run over data structures local to each processor. We report experimental results at the end of Part III.

We conclude in Part IV describing the limitations of our techniques and directions for future research.

Part I

Sparse matrix computations: a compiler perspective

Chapter 1

Sparse matrix computations

1.1 Sparse matrices and their applications

Sparse matrix computations are used in a variety of areas, such as numerical simulations, information retrieval and graph theory. The most common use of matrices is to store the coefficients of a system of linear equations. Many practical problems are either reduced to or are approximated by the solution of systems of linear equations. The relationship between applications is various matrix computations is illustrated in Figure 1.1.

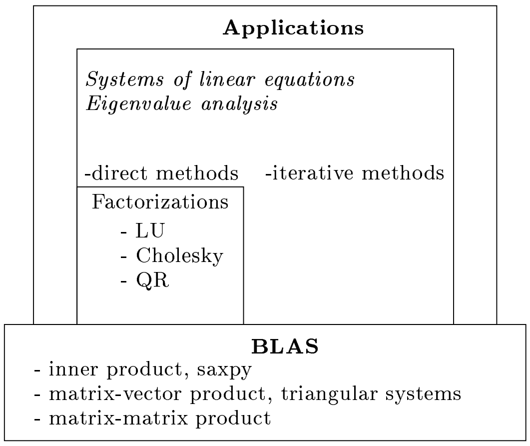


Figure 1.1: Relationship between applications and matrix computations

In many real-world situations the interactions between various entities are sparse. For example, differential equations describe the interaction of physical quantities within an infinitely small neighborhood of each point. These sparse interactions manifest themselves in the resulting systems of linear equations.

1.1.1 Finite Element Method computations

FEM computations are used widely in the solution of partial differential equations (PDEs) over unstructured domains (see [54] for an introduction). The equations come from a variety of physical problems, such as structural analysis and fluid mechanics. To gain some intuition consider a linear PDE over a region $\Omega \subset \mathbb{R}^n$:

$$\mathcal{D}u(x) = f(x), x \in \Omega \tag{1.1}$$

where $\mathcal{D}$ is a linear differential operator, such as the Laplacian operator in $\mathbb{R}^2$: $\mathcal{D}u(x,y) \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. In most cases it is not possible to find a closed-form solution to (1.1) and at the heart of the Finite Element Method is the approximation of the unknown function u as a linear combination of the basis function ϕ_i:

$$u(x) \approx \tilde{u}(x) = \sum_{i=1}^{m} a_i \phi_i(x) \tag{1.2}$$

The basis functions are defined by overlaying the domain Ω with a mesh (Figure 1.2). Then each function is set to a piece-wise polynomial which is zero on most of the domain, except for certain control points. For example, we can associate a "roof" function ϕ_p with each point p in the mesh: its value is 1 at the point p, 0 at all other points in the mesh, and ϕ_p is continuous within each of the triangles (a.k.a. elements) of the mesh. Two such functions for a one-dimensional domain are illustrated in Figure 1.3. The coefficients a_i are

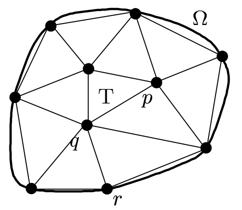


Figure 1.2: A mesh

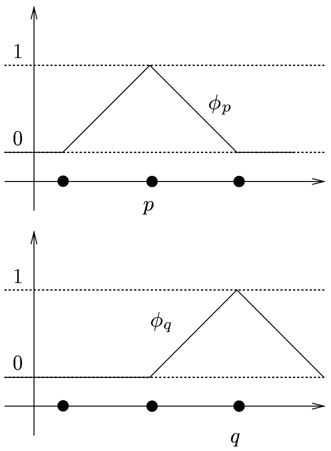


Figure 1.3: "Roof" basis functions in 1D

found by applying the Galerkin's weighted residuals method. The residual $\mathcal{D}\tilde{u}(x) - f(x)$ of the approximate solution is weighted by the basis functions:

$$w_i = \int_{\Omega} \left(\mathcal{D}\tilde{u}(x) - f(x)\right) \phi_i(x) dx, \qquad 1 \le i \le m \tag{1.3}$$

The coefficients are determined by setting each of the integrals w_i to zero. Expanding the definition (1.2) and exploiting the linearity of the operator $\mathcal{D}$, we obtain (for each $1 \leq i \leq m$):

$$\begin{aligned} w_i = 0 &= \int_\Omega \phi_i(x) \left(\mathcal{D}(\sum_j a_j \phi_j(x)) - f(x) \right) dx \\ &= \left(\sum_j a_j \int_\Omega \phi_i(x) \mathcal{D} \phi_j(x) dx \right) - \int_\Omega \phi_i(x) f(x) dx \end{aligned} \tag{1.4}$$

We can rewrite this system of equations in matrix form[1]:

$$\mathbf{Ka} = \mathbf{f} \tag{1.5}$$

where

$$\mathbf{K}_{ij} = \int_\Omega \phi_i(x) \mathcal{D} \phi_j(x) dx \tag{1.6}$$

$$\mathbf{f}_i = \int_\Omega \phi_i(x) f(x) dx \tag{1.7}$$

$$\mathbf{a}_i = a_i \tag{1.8}$$

Turns out that the stiffness matrix $\mathbf{K}$ is sparse: the integrals $\int_\Omega \phi_i(x) \mathcal{D} \phi_j(x) dx$ are non-zero only when the functions ϕ_i and ϕ_j correspond to control points that are "close by" in the mesh. For example, if the linear "roof" functions are used on the domain in Figure 1.2, then ϕ_p is zero when ϕ_r is not and the corresponding integral is zero. However, ϕ_p and ϕ_q are both non-zero within the element T.

In many practical settings the mesh contains millions of elements and there might be tens of basis functions defined per element. The resulting linear system (1.5) can have as many as tens of millions of equations (rows in $\mathbf{K}$) in tens of millions of unknowns. However, the number of non-zero entries in each row or column is bounded by some small constant which depends on the connectivity of the mesh and the choice of the basis functions. It is often on the order of a few tens [45].

1.1.2 Hubs and authorities in hypertext collections

An important problem in navigating collections of hypertext is the identification of good starting points or *hubs* and authoritative sources of information or *authorities*. An example of a hub is the `yahoo.com` Web site. An often cited academic paper is an authority. Hubs and authorities reinforce one another: a good hub points to many authorities, and a good authority is pointed to by many hubs, as illustrated in Figure 1.4. This relationship can be expressed numerically, as follows (see [48]). Given a collection of hypertext, we define a graph $G = (V, E)$ in a natural way: the nodes V are the documents in the collection and there is an edge $(u, v) \in E$ if there is a link from the document u to v. With each node u we associate a non-negative authority weight x_u and a non-negative hub weight y_u. If the document u is pointed to by the document v that has high hub weight y_v, then the authority weight x_u of u should be increased. Conversely, if the document u points to the document v that has high authority weight x_v, then the hub weight y_u of u should be increased. Let us define the following updating operations:

- The operation $\mathcal{I}$ updates the authority weight of each node by the sum of the hub weights of its neighbors along the incoming edges:

$$x_u \leftarrow \sum_{(v,u) \in E} y_v \tag{1.9}$$

[1] We use bold letters to denote vectors and matrices

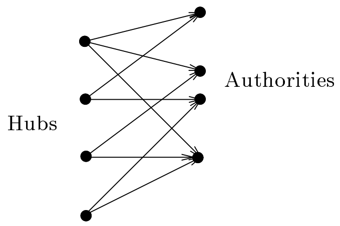


Figure 1.4: Hubs and authorities

- The operation $\mathcal{O}$ updates the hub weight of each node by the sum of the authority weights of its neighbors along the outgoing edges:

$$y_u \leftarrow \sum_{(u,v)\in E} x_v \tag{1.10}$$

Let $\mathbf{x}$ and $\mathbf{y}$ be the vectors of the weights. We can define the adjacency matrix $\mathbf{A}$ of the graph G: the entry $\mathbf{A}_{uv}$ is 1 if there is an edge $(u, v) \in E$ and is zero otherwise. Then the operation $\mathcal{I}$ can be written as $\mathbf{x} \leftarrow \mathbf{A}^T\mathbf{y}$ and the operation $\mathcal{O}$ can be written as $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$.

We can produce a sequence of the weights assignments by executing the following procedure:

```
Let z = (1 1 ...1)^T
x := z; y := z
for i = 1, 2, 3, ...
    x := A^T y
    y := Ax
    normalize x and y
```

It is shown in [48] that the sequence of $(\mathbf{x}, \mathbf{y})$ pairs converges to the limit $(\mathbf{x}_*, \mathbf{y}_*)$ where $\mathbf{x}_*$ is the principal eigenvector[2] of $\mathbf{A}^T\mathbf{A}$ and $\mathbf{y}_*$ is the principal eigenvector of $\mathbf{A}\mathbf{A}^T$. The documents with relatively high weights in the vectors $\mathbf{x}_*$ and $\mathbf{y}_*$ are claimed to be authorities and hubs, respectively. Additionally, non-principal eigenvectors of the above matrices expose dense communities of unrelated hubs and authorities.

It is natural to expect the matrix $\mathbf{A}$ to be sparse: not every document points to every other document. The matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ are also expected to be sparse.

1.2 Sparse matrix algorithms

Given a matrix $\mathbf{A}$ the two most basic problems are solving systems of linear equations and finding the eigenvalues/vectors. In this Section we discuss the related algorithms and summarize the core computational kernels.

1.2.1 Solution of sparse linear systems

Direct methods

Direct methods have for many years been the traditional way of solving linear systems – dense and sparse. They are based on factoring the matrix of equations into products of upper and lower triangular matrices and

[2] A non-zero vector $\mathbf{w}$ is called an eigenvector of the matrix $\mathbf{B}$ if it satisfies the equality: $\mathbf{B}\mathbf{w} = \lambda\mathbf{w}$ – for some constant λ called an eigenvalue. There are usually several eigenvalue/vector pairs. The eigenvector that corresponds to the largest eigenvalue (by absolute value) is called principal. See [40] for more details.

```
1.  for k=1 to n
2.      A(k,k) = √A(k,k)
3.      for i=k+1 to n    ''scale'' loop
4.          A(i,k) = A(i,k) / A(k,k)
5.      for j=k+1,n    ''update'' loop
6.          for i=j,n
7.              A(i,j) = A(i,j) - A(i,k) * A(j,k)
```

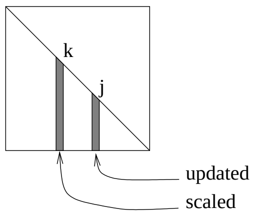


Figure 1.5: Cholesky factorization

solving a sequence of "simpler" equations. For simplicity, assume that the matrix $\mathbf{A}$ is symmetric positive definite (SPD[3]): many of the ideas from the algorithms for the SPD matrices carry over to the general case. A linear system $\mathbf{Ax} = \mathbf{b}$ is solved by, first, computing the lower triangular *Cholesky factor* $\mathbf{L}$ of $\mathbf{A}$ that satisfies $\mathbf{A} = \mathbf{LL}^T$ and, second, solving the sequence of simpler linear systems: $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{L}^T\mathbf{x} = \mathbf{y}$. The program in Figure 1.5 overwrites the lower triangle of $\mathbf{A}$ with the values of $\mathbf{L}$. The computation proceeds column by column. First, the column k is scaled (line 4). Then each column $j > k$ "to the right" is updated by a multiple of column k (line 7). If the matrix is dense, then these factorizations take $O(n^3)$ time (see [40] for more details). Consider the sparse matrix $\mathbf{A}$ and its factor $\mathbf{L}$:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 0 & 6 & 0 & 0 \\ 1 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 16 \end{pmatrix} \quad \mathbf{L} = \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & -1 & 2 & & \\ 1 & -1 & -1 & 3 & \\ 0 & 0 & 0 & 0 & 4 \end{pmatrix} \tag{1.11}$$

We can make a few observations about the matrices:

- The factor $\mathbf{L}$ has more non-zeroes than the matrix $\mathbf{L}$. This phenomenon is referred to as *fill*. Fill occurs because the update loop (line 7) can set the element `A(i,j)` to a non-zero value.
- The first four columns of the factor had to be computed in sequence. However, none of these columns update the fifth column, which can be computed independently. In general, sparse matrix factorizations exhibit additional *parallelism*.
- The sub-matrix $\mathbf{L}(1:4, 1:4)$ formed by the first four rows and columns of $\mathbf{L}$ is completely dense and is the factor of the corresponding sub-matrix $\mathbf{A}(1:4, 1:4)$ of $\mathbf{A}$. In general, fill "leaves behind" the so-called *supernodes* in the factor, which are groups of columns which form a dense sub-matrix.

[3] A symmetric matrix is SPD if for any non-zero vector $\mathbf{x}$ the product $\mathbf{x}^T\mathbf{Ax}$ is strictly positive.

Now lets reverse the order of the rows and column in $\mathbf{A}$. The resulting matrix $\mathbf{B}$ and its factor $\mathbf{M}$ are:

$$\mathbf{B} = \begin{pmatrix} 16 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 & 1 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \mathbf{M} = \begin{pmatrix} 4 & & & & \\ 0 & 3.46 & & & \\ 0 & 0 & 2.44 & & \\ 0 & 0 & 0 & 1.41 & \\ 0 & 0.29 & 0.41 & 0.71 & 0.50 \end{pmatrix} \tag{1.12}$$

Observe that the factor $\mathbf{M}$ has less fill than the factor $\mathbf{L}$. In fact, it has no fill at all! There is also more parallelism in the factorization of $\mathbf{B}$: the first four columns can be scaled independently before the columns 2, 3 and 4 update the fifth column. This example illustrates another important characteristic of sparse factorizations:

- Reordering of rows and columns of $\mathbf{A}$ can reduce the amount of fill in the factor and increase available parallelism.

In general, the order of column scaling is a tree (or a forest of trees), called *elimination tree*. The elimination trees for the matrices $\mathbf{A}$ and $\mathbf{B}$ are illustrated in Figure 1.6.

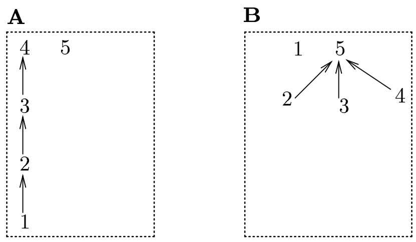


Figure 1.6: Example elimination trees

Given the above considerations, implementations of sparse Cholesky factorization can be split into three phases:

- *Matrix reordering* permutes the rows and columns of the matrix with the goal of reducing fill and (in parallel environment) improving parallelism. The problem of finding reordering that minimizes fill is NP-complete [93] and various heuristics are used. See [39] for an introduction. [72] discusses performance tradeoffs during this step.

- *Symbolic factorization* precomputes the storage for the factor $\mathbf{L}$. There exist optimal algorithms that compute the storage for $\mathbf{L}$ in time proportional to the number of non-zeros in $\mathbf{L}$ (see [39], chapter 5). Matrix reordering and symbolic factorization are usually implemented as computations on the graph reflecting the structure of the input matrix.

- *Numerical factorization* performs the actual computation. The computation is usually reorganized into a sequence of partial dense factorizations and so-called update steps (see [57] for an overview). The update steps compute expressions of the form $\mathbf{U} = \mathbf{U} + \mathbf{V}$, where the set of non-zeros of $\mathbf{V}$ is a subset of the non-zeros of $\mathbf{U}$ (as is pre-allocated by the symbolic step). Numerical factorization usually dominates the overall factorization time because of its higher asymptotic time complexity.

The so-called *multifrontal supernodal* algorithm for Cholesky factorization is outlined in Figure 1.7 (adapted from [57]). To illustrate how the algorithm works consider the following example matrix:

Matrix reordering and symbolic factorization

for each supernode K in some tree order do

  let $S = \{j, j+1, \ldots, j+t\}$ be the columns in the supernode

  let $T = \{j+t, i_1, i_2, \ldots, i_r\}$ be the non-zero rows in the supernode

  collect the supernode into the *frontal* matrix $\mathbf{F}_K$:

$$\begin{pmatrix} a_{j,j} & & & & & & \\ a_{j+1,j} & a_{j+1,j+1} & & & & & \\ \vdots & \vdots & \ddots & & & & \\ a_{j+t,j} & a_{j+t,j+1} & \cdots & a_{j+t,j+t} & & & \\ a_{i_1,j} & a_{i_1,j+1} & \cdots & a_{i_1,j+t} & 0 & & \\ \vdots & \vdots & \ldots & \vdots & \vdots & \ddots & \\ a_{i_r,j} & a_{i_r,j+1} & \cdots & a_{i_r,j+t} & 0 & \ldots & 0 \end{pmatrix}$$

  for all children C of K in the tree do

    $\mathbf{F}_K = \mathbf{F}_K + \mathbf{U}_C$

  enddo

  view $\mathbf{F}_K$ as a $(t+r)$-by-$(t+r)$ dense matrix $\tilde{\mathbf{F}}_K$

  factor the first t columns of $\tilde{\mathbf{F}}_K$

  $\mathbf{F}_K$ is now of the form:

$$\begin{pmatrix} l_{j,j} & & & & & & & \\ l_{j+1,j} & l_{j+1,j+1} & & & & & & \\ \vdots & \vdots & \ddots & & & & & \\ l_{j+t,j} & l_{j+t,j+1} & \cdots & l_{j+t,j+t} & & & & \\ l_{i_1,j} & l_{i_1,j+1} & \cdots & l_{i_1,j+t} & u_{i_1,i_1} & & & \\ l_{i_2,j} & l_{i_2,j+1} & \cdots & l_{i_2,j+t} & u_{i_2,i_1} & u_{i_2,i_2} & & \\ \vdots & \vdots & \ldots & \vdots & \vdots & \vdots & \ddots & \\ l_{i_r,j} & l_{i_r,j+1} & \cdots & l_{i_r,j+t} & ui_r, i_1 & u_{i_r,i_2} & \cdots & u_{i_r,i_r} \end{pmatrix}$$

  set the update matrix $\mathbf{U}_K$ to (u_{i_p,i_q}), $1 \leq p, q, \leq r$.

enddo

Figure 1.7: Supernodal multifrontal Cholesky factorization

$$\mathbf{C} = \begin{pmatrix} 1 & & & & \\ 1 & 2 & & & \\ 0 & 0 & 4 & & \\ 1 & 2 & 0 & 3 & \\ 1 & 2 & 2 & 3 & 5 \end{pmatrix} \tag{1.13}$$

There are three supernodes: columns 1 and 2, column 3 by itself and columns 4 and 5. The first and the second supernodes can be factored independently, while they both update the third supernode. In the first supernode, the set S (line 3, Figure 1.7) is $\{1,2\}$ and the set T (line 4) is $\{4,5\}$. The frontal matrix viewed as a sparse sub-matrix of $\mathbf{C}$ is:

$$\mathbf{F}_1 = \begin{pmatrix} 1 & & & & \\ 1 & 2 & & & \\ 0 & 0 & 0 & & \\ 1 & 2 & 0 & 0 & \\ 1 & 2 & 0 & 0 & 0 \end{pmatrix} \tag{1.14}$$

And when viewed as a dense matrix it is:

$$\tilde{\mathbf{F}}_1 = \begin{pmatrix} 1 & & & \\ 1 & 2 & & \\ 1 & 2 & 0 & \\ 1 & 2 & 0 & 0 \end{pmatrix} \tag{1.15}$$

After factoring the first two columns we get:

$$\mathbf{F}_1 = \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ 1 & 1 & -2 & \\ 1 & 1 & -2 & -2 \end{pmatrix} \tag{1.16}$$

The update matrix $\mathbf{U}_1$ for this supernode is (when viewed as a sparse matrix):

$$\mathbf{U}_1 = \begin{pmatrix} 0 & & & & \\ 0 & 0 & & & \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & -2 & \\ 0 & 0 & 0 & -2 & -2 \end{pmatrix} \tag{1.17}$$

Similarly, for the second supernode:

$$\begin{aligned} \mathbf{F}_2 &= \begin{pmatrix} 0 & & & & \\ 0 & 0 & & & \\ 0 & 0 & 4 & & \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & 2 & 0 & 0 \end{pmatrix} \qquad \tilde{\mathbf{F}}_2 = \begin{pmatrix} 4 & \\ 2 & 0 \end{pmatrix} \overset{factor}{\rightarrow} \begin{pmatrix} 2 & \\ 1 & -1 \end{pmatrix} \\ \mathbf{U}_2 &= \begin{pmatrix} 0 & & & & \\ 0 & 0 & & & \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix} \end{aligned} \tag{1.18}$$

Both update matrices $\mathbf{U}_1$ and $\mathbf{U}_2$ act on the third supernode:

$$\mathbf{F}_3 = \begin{pmatrix} 0 & & & & \\ 0 & 0 & & & \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & 3 & \\ 0 & 0 & 0 & 3 & 5 \end{pmatrix} \quad \mathbf{F}_3 + \mathbf{U}_1 + \mathbf{U}_2 = \begin{pmatrix} 0 & & & & \\ 0 & 0 & & & \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} \tag{1.19}$$

Factoring this last matrix (as a dense matrix) we get the last two column of the factor $\mathbf{N}$ of $\mathbf{C}$:

$$\mathbf{N} = \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 0 & 0 & 2 & & \\ 1 & 1 & 0 & 1 & \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{1.20}$$

Each frontal and update matrix is stored by collecting the non-zeros into a dense matrix and keeping the sets S and T or indices (similar to i-nodes in Figure 1.15), so that the matrices can be interpreted as dense or sparse depending on the context. Storage and computation in other matrix factorizations (QR [83], LU [31]) are organized in similar ways: sparse algorithm is decomposed into a sequence of dense subproblems connected by sparse update steps.

Iterative methods

The time and space complexity of the direct methods preclude their use on large problems. [45] provides the following "rule of thumb" estimates for typical 2- and 3-dimensional FEM computations. Let n be the number of nodes in the mesh. Then the stiffness matrix $\mathbf{K}$ has $O(n)$ rows and columns. The Cholesky factorization of $\mathbf{K}$ takes $O(n^{1.5})$ time for 2D computations and $O(n^2)$ time for 3D. If we start with a 3D mesh containing $\sim 10^6$ nodes, the solution of the linear system will take $\sim 10^{12}$ time!

Iterative methods have been developed as a time and space saving alternative to direct methods. For SPD matrices the Conjugate Gradients (CG) method is used in a variety of application domains. The method is based on finding the minimizers $\mathbf{x}_j$ of the quadratic form $\frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$ over sub-spaces spanned by the vectors $\mathbf{q}, \mathbf{A}\mathbf{q}, \ldots, \mathbf{A}^{k-1}\mathbf{q}$ for $k = 1, 2, \ldots$. The sub-spaces are called *Krylov spaces* and are used in a variety of algorithm for the solution of linear systems and eigenvalue analysis.

The pseudo-code for the CG algorithm is illustrated in Figure 1.8. Observe that the only operations in the algorithm are the product of the sparse matrix $\mathbf{A}$ and a dense vector (line 10) and various operations on dense vectors, such as addition, inner product and scaling. Because the algorithm does not require any extra space beyond the storage for $\mathbf{A}$ and a few vectors, it is very attractive for large sparse problems.

However, time complexity is still an issue. Ignoring round-off errors, the CG algorithm is guaranteed to produce the exact solution in n iterations. Let η_A be the number of non-zeros in $\mathbf{A}$. Sparse matrix-vector product takes $2\eta_A$ floating point operations. Assuming that $\eta_A = \Omega(n)$, running the CG algorithm for n iterations gives us the algorithm with $\Omega(n^2)$ time complexity. This is not acceptable when $n \sim 10^6$. The good news is that if the condition number of $\mathbf{A}$ is close to 1, the CG algorithm converges very fast (see [40]).

If the condition number of $\mathbf{A}$ is large then the *Preconditioned* Conjugate Gradients (PCG) algorithm is used. Instead of solving the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ the algorithm solves an equivalent system:

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} \tag{1.21}$$

where

$$\tilde{\mathbf{A}} = \mathbf{C}^{-1}\mathbf{A}\mathbf{C}^{-1} \quad \tilde{\mathbf{x}} = \mathbf{C}\mathbf{x} \quad \tilde{\mathbf{b}} = \mathbf{C}^{-1}\mathbf{b} \tag{1.22}$$

for some SPD matrix $\mathbf{C}$. Define the *preconditioner* matrix $\mathbf{M} = \mathbf{C}^2$. Using the definition (1.22) of $\tilde{\mathbf{A}}$ we can transform the CG algorithm into the PCG algorithm shown in Figure 1.9 (see [40] for the derivation).

1. k=0; $\mathbf{x}$=0; $\mathbf{r}$=$\mathbf{b}$; $\rho_0 = \|\mathbf{r}\|_2^2$
2. while ($\sqrt{\rho_k} > \epsilon\|\mathbf{b}\|_2 \wedge k < k_{max}$) do
3.   k = k + 1
4.   if k=1 then
5. $\mathbf{p}$ = $\mathbf{r}$
6.   else
7. β_k = ρ_{k-1}/ρ_{k-2}
8. $\mathbf{p}$ = $\mathbf{r}$ + $\beta_k\mathbf{p}$
9.   endif
10. $\mathbf{w}$ = $\mathbf{Ap}$
11. α_k = $\rho_{k-1}/\mathbf{p}^T\mathbf{w}$
12. $\mathbf{x}$ = $\mathbf{x}$ + $\alpha_k\mathbf{p}$
13. $\mathbf{r}$ = $\mathbf{r}$ - $\alpha_k\mathbf{w}$
14. ρ_k = $\|\mathbf{r}\|_2^2$
15. enddo

Figure 1.8: Conjugate Gradients algorithm

Observe that besides simple vector and scalar operations this algorithm has to solve a linear system (line 3) and multiply a vector by the matrix $\mathbf{A}$ (line 12).

For the new algorithm to be an effective technique the system $\mathbf{Mz} = \mathbf{r}$ (line 3) must be easy to solve and the convergence must be fast, i.e. the condition number of $\tilde{\mathbf{A}}$ must be small. The choice of the preconditioner $\mathbf{M}$ that satisfies these requirements depends on the properties of the matrix $\mathbf{A}$. One heuristic is to set $\mathbf{M}$ to the diagonal of $\mathbf{A}$ thus obtaining the so-called *diagonal preconditioner*. Another heuristic, called *Incomplete Cholesky Factorization* (ICC)[4], is to set $\mathbf{M}$ to be the product $\mathbf{HH}^T$, where the matrix $\mathbf{H}$ is somehow "close" to the Cholesky factor of $\mathbf{A}$ but has tractable sparsity structure. A simple way to compute $\mathbf{H}$ is to run the program for Cholesky factorization in Figure 1.5 but only update those elements `A(i,j)` that are non-zero:

```
initialize H to the lower triangle of A
for k=1 to n
    H(k,k) = sqrt(H(k,k))
    for i=k+1 to n
        H(i,k) = H(i,k) / H(k,k)
    for j=k+1,n
        for i=j,n
            if H(i,j) ≠ 0 then
                H(i,j) = H(i,j) - H(i,k) * H(j,k)
```

This is the so-called ICC level 0 or "ICC(0)" preconditioner. Higher-level ICC preconditioners allow controlled amounts of fill. Notice that the update loop performs the computation:

$$\mathbf{B} \approx \mathbf{B} - \mathbf{uu}^T \tag{1.23}$$

[4]ICC stands for "Incomplete **CC** factorization"

```
k=0; x=0; r_0=b;
while not converged do
    Solve Mz = r
    γ_k = r^T z
    k = k + 1
    if k=1 then
        p = z
    else
        β_k = γ_{k-1}/γ_{k-2}
        p = z + β_k p
    endif
    w = Ap_k
    α_k = γ_{k-1}/p^T w
    x = x + α_k p
    r = r - α_k w
enddo
```

Figure 1.9: Preconditioned Conjugate Gradients algorithm

---

where $\mathbf{B}$ is the sub-matrix $\mathbf{H}(k+1:n, k+1:n)$ and $\mathbf{u} = \mathbf{H}(k+1:n, k)$. We call this an *incomplete outer product update*. The "$\approx$" operator indicates that only the elements of $\mathbf{B}$ that are already stored (i.e. non-zero) are updated.

The system $\mathbf{Mz} = \mathbf{r}$ can be solved in $O(\eta_A)$ time by using the Cholesky factorization $\mathbf{M} = \mathbf{HH}^T$. Therefore the main computational kernels in PCG with Incomplete Cholesky Preconditioner are:

- Sparse matrix-vector product $\mathbf{Ap}$
- Solution of triangular systems involving $\mathbf{H}$ and $\mathbf{H}^T$
- Incomplete outer product update (in the computation of the preconditioner)

1.2.2 Eigenvalue analysis

The most basic method for computing eigenvalues and eigenvectors is the *power method*. Let $\mathbf{A}$ be an $n \times n$ matrix. Let $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$ be the eigenvalues of the matrix. Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the associated eigenvectors: $\mathbf{A}\mathbf{x}_k = \lambda_k \mathbf{x}_k$. We assume that they have unit length: $||\mathbf{x}_k|| = 1$. For simplicity, we also assume that the eigenvalues are distinct. See [40] for the treatment of the general case. It can be shown that the vector $\mathbf{q}$ produced by the following iterative process converges to $\mathbf{x}_1$:

Program 1.1

```
g := some non-zero initial vector
for k=1,2,3,...
    z := Aq
    q := z/||z||
```

The *orthogonal iteration* is used to compute other eigenvalues of $\mathbf{A}$:

Program 1.2

```
let U_0 be some unitary matrix
T_0 := U_0^H AU
for k = 1,2,3,...
    compute the QR factorization U_k R_k = T_{k-1}
    T_k := R_k U_k
```

It can be shown (see [40] for details) that the matrices $\mathbf{T}_k$ converge to an upper-triangular matrix with the eigenvalues of $\mathbf{A}$ on the diagonal. QR factorization $\mathbf{U}_k\mathbf{R}_k = \mathbf{T}_{k-1}$ decomposes the matrix $\mathbf{T}_{k-1}$ into the product of a unitary (i.e. orthonormal, if real) matrix $\mathbf{U}_k$ and an upper-triangular matrix $\mathbf{R}_k$.

The power method can be used effectively for sparse matrices, because it only requires sparse matrix-vector products. The orthogonal iteration is not suitable for large sparse matrices, because QR factorization introduces fill, just like Cholesky factorization. In both cases convergence can be problematic if the eigenvalues are close to one another.

For symmetric matrices the *Lanczos method* provides better convergence properties. It is based on the fact that the α_k and β_k coefficients computed is Conjugate Gradient algorithm (Figure 1.8) form a tri-diagonal matrix $\mathbf{T}_k$ whose extreme eigenvalues approximate the extreme eigenvalues of $\mathbf{A}$:

$$\mathbf{T}_k = \begin{pmatrix} \alpha_1 & \beta_1 & & \dots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{k-1} \\ 0 & \dots & & \beta_{k-1} & \alpha_k \end{pmatrix} \tag{1.24}$$

Observe that computing the matrix $\mathbf{T}_k$ only involves sparse matrix-vector products and dense vector operations. In turn, computing the eigenvalues of $\mathbf{T}_k$ only involves dense computations.

Overall, the core computations used in eigenvalue analysis are similar to those used in the solution of linear systems: sparse matrix-vector products, matrix factorizations and various dense operations.

1.2.3 Summary: common features of sparse matrix computations

Sparse matrix computations are ubiquitous in science and engineering. Many practical problems require solution of large sparse linear systems with millions or equations and unknowns. In order to reduce time and space complexity of the computations a variety of compact data structures have been developed for storing sparse matrices. Often, the best data structure depends on an application.

Sparse matrix algorithms discussed in this chapter consist of the following ingredients:

- Graph-theoretic preprocessing
  - Matrix reordering
  - Symbolic factorization
  - Graph partitioning
- Basic computations on sparse matrices and vectors:
  - Sparse matrix-vector products
  - Scaling of sparse vectors (of a column in Cholesky factorization)
  - Incomplete updates $\mathbf{U} \approx \mathbf{U} + \mathbf{V}$ (in Cholesky factorization) and $\mathbf{B} \approx \mathbf{B} + \mathbf{u}\mathbf{u}^T$ (in Incomplete Cholesky)
  - Triangular system solution

- Dense matrix computations
  - Partial factorizations in sparse Cholesky
  - vector operations (scaling, addition, inner products) in iterative methods

1.3 Storing sparse matrices

In order to make computations practical, sparse matrices have to be stored in special compact formats. Matrices with 10^7 or more rows and columns, yet few non-zeroes in each row, are not uncommon in modern applications. Using a dense 10^7-by-10^7 array in this case is rather unwise. Instead we would like to store a sparse matrix in space proportional to the number of non-zeros. Let us consider some examples. Let n be the number of rows in a matrix and η be the number of non-zeros. Assume that the matrix is square.

The simplest example is the coordinate storage format illustrated in Figure 1.11. The matrix stored in shown in Figure 1.10. The format stores the non-zeros in three arrays: the arrays of row and column indices and the array of values. Notice that we had to trade the random access to individual elements for savings in storage.

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & 0 & 0 & 0 \\ c & 0 & 0 & d \\ e & 0 & f & g \end{pmatrix}$$

Figure 1.10: An example matrix

The compressed row storage (CRS) format is illustrated in Figure 1.12. It stores the column indices of the non-zeros and the values of the non-zeros in the respective arrays. Additionally, it keeps an array of "pointer" that indicate where each row starts. The CRS format provides random access to rows, while each row can only be accessed sequentially. We can also compress the matrix by column and obtain the compressed column storage (CCS) illustrated in Figure 1.13.

The above formats are very simple and general. However, they do 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 1.14 (adapted from [46]) 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 Figure 1.15. Storing i-nodes as dense matrices improves the performance of matrix-vector products for the targeted class of applications. In general, the best format depends on the

a	g	f	c	b	e	d	`VALS`
1	4	4	3	1	4	3	`ROWIND`
1	4	3	1	3	1	4	`COLIND`

`N` $=$ `4`   `NNZ` $=$ `7`

Figure 1.11: COORDINATE storage format

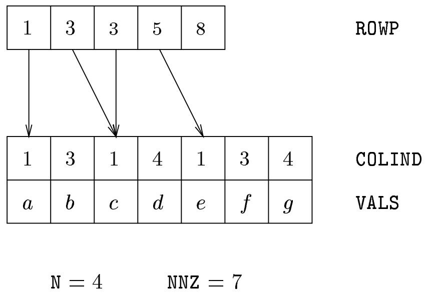


Figure 1.12: Compressed Row Storage

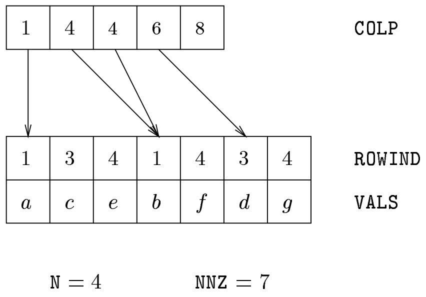


Figure 1.13: Compressed Column Storage

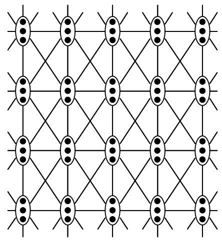


Figure 1.14: A subgraph generated by 2D linear finite element model

$$\begin{pmatrix} a & 0 & b & 0 \\ c & 0 & d & 0 \\ e & f & 0 & g \\ h & i & 0 & j \end{pmatrix}$$

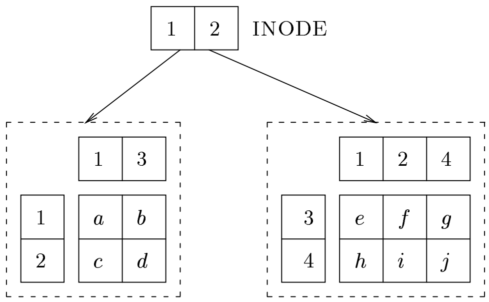


Figure 1.15: I-node storage

application and machine architecture. Consider the comparison in Table 1.1 of the performance of sparse matrix-vector product for a few benchmark matrices from the Matrix Market collection [22] on an IBM RS6000 workstation [5].

The numbers in each column are the Mega-flops rating of sparse matrix-vector product for a given format. The Diagonal storage format is a variation on the CRS format: the matrix is compressed along the diagonals. ITPACK and Jagged Diagonal ("JDiag") formats will be discussed in detail in Section 5.9. The column marked "BS95" shows the performance of the format used in the BlockSolve library. Notice that

Table 1.1: Performance of sparse matrix-vector product

Matrix	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

there is no clear winner! In fact a number of formats are widely used in practice. For example, the NIST Sparse BLAS library (Section 5.9) which is targeted at a variety of applications supports 13 popular formats.

1.4 Sparse matrix software

The following quotation from the second edition of [40] illustrates the *layered approach* to the development of scientific software:

> Great progress has been made in [numerical software] since the mid-1950s. This is borne out by the existence of quality programs for many linear equations, least squares, and eigenvalue problems. Typical are the routines in Eispack and Linpack, whose widespread use has had the effect of elevating the level of algorithmic thought in various areas. By using the programs in these packages as building blocks, scientists and engineers can piece together more complicated software tools that are tailored specifically to their needs. This development encourages the writing of well-structured programs, a welcome trend since more and more scientific research is manifested in software.

This approach has been successful in the domain of *dense* matrix computations. For example, if a numerical code required solution of a system of linear equations it would call the appropriate routines in LAPACK package [4]. In turn, the package uses routines from the dense BLAS library [32, 33, 53]. The main reason for the widespread use of this layering is that there are very few formats for storing dense matrices and these formats can be easily parametrized. For example, a matrix can be stored in a row-major or column-major two-dimensional array. As a result the BLAS libraries have a standard interface which consists of a only few routines. These routines can be hand-tuned for each architecture thus facilitating modular development of other high performance dense matrix software.

The situation is radically different in the area of sparse matrix computations due to the presence of a variety of storage formats. There have been attempts at developing a BLAS like library of basic sparse computations, however they suffer from combinatorial explosion in the number of individual routines that library developers have to provide. For example, the NIST Sparse BLAS library [68] supports 13 formats

[5] IBM RS6000, Model 590 with 66.5MHz POWER2 processor

and provides routines for computations where only a single operand is sparse (such as triangular solves). This leaves out important kernels such as the incomplete updates. The library also has no provisions for adding new formats, short of writing all the required routines by hand.

Instead, coarse-grain packages that target specific classes of problems have been developed and are in widespread use today. The layering that these packages use depends on the particular application. For example, the codes for the solution of sparse symmetric positive definite systems and least squares problems developed by at Cornell by Chunguang Sun [82, 83] implement Cholesky and QR factorizations by decomposing them into primitive sparse operations, such as matrix updates, and dense operations, such as partial Cholesky factorization of supernodes. Dense operations are implemented by calling the appropriate LAPACK routines and BLAS. However, the sparse operations are still coded by hand due to the lack of the sparse BLAS support. Another kind of layering is used in the PETSc library [6], which implements various Krylov space iterative solvers (among other things). The library views sparse matrices as algebraic objects that provide two methods: product $\mathbf{Ax}$ with a dense vector and, in the case of pre-conditioned iterative methods, the product $\mathbf{M}^{-1}\mathbf{z}$ of the preconditioner with a dense vector. A user can supply his own sparse matrix data structure by implementing this primitives. There are two problems with this approach. First, some high-performance implementations of sparse matrix-vector products are themselves quite tedious. For example, the *sequential* product in the BlockSolve library takes a few hundred lines of C. The parallel implementation is even worse: additional programming effort goes into setting up and performing communication. Second, these solver packages are limited to a particular class of problems and are not useful in the development of new algorithms.

An alternative approach to providing sparse matrix software has been suggested by Bik and Wijshoff [9, 16, 20]: use a compiler to generate sparse matrix codes automatically starting with dense codes. This approach has one missing ingredient: Bik's sparse compiler only provides the formats which are a variation on the CRS and CCS schemes. No provisions are made for extending the compiler with new storage formats, yet we have argued that the choice of storage formats greatly affects performance.

1.5 Sparse matrix computations and generic programming

We view the problem of supporting effective development of high-performance sparse matrix codes as one of **generic programming** [62]. Generic programming is a discipline of designing and implementing software components which can be used when there is a set of *related data structures* supporting a common semantics described by an API or protocol, and a set of *common algorithms* that can be formulated in terms of this API. A generic programming system permits library writers to use the API to code generic procedures without any knowledge of what data structures will ultimately be used by the application programmer, and permits applications programmers to invoke generic procedures by passing in an application-specific data structure implementation supporting the necessary API. This gives the applications programmer the effect of having a software component customized for the application.

When designing a generic programming system one must address the following fundamental questions [94]:

- How to represent efficient algorithms independently of any particular data-representation scheme?
- How to provide an interface to a diverse set of data-structures?
- How to "knit" together the representation of the algorithms and the representation for the data to obtain an efficient implementation.

An example of a generic programming system is the C++ Standard Template Library (STL) [61, 78]. STL supports a variety of *container* classes, such as lists, queues and vectors, and generic algorithms that operate on the classes, such as searching or sorting. STL addresses the problem of data representation by introducing the notion of an *iterator*, which is a generalization of a C pointer. An iterator represents a position in a data structure in the same way that a pointer represents a position in memory. Since not all data structures support all of the operations one normally associates with a pointer, such as incrementing/decrementing, random access and dereferencing, STL defines a hierarchy of iterator classes. All data structures provide the most advanced type of iterator they can implement efficiently. In turn, the algorithms in STL are expressed in terms of the iterators. This provides the independence between data representation and algorithm specification. The "knitting" problem is solved by using C++ template mechanism. In particular, the implementor of the library can provide alternate implementations (i.e. template instantiations) of an algorithm for various combinations of the iterator classes.

In our case, the most natural way of expressing sparse matrix computations is in terms of **dense loop nests**. Such specification of matrix algorithms can be readily found in standard texts. Different sparse matrix formats form the set of related data structures. In the rest of the thesis we address the remaining two questions:

- What is the API for specifying sparse matrix data structures?
- What is the "knitting" technology for producing efficient sparse code from dense loop nests and the specification of data structures?

Chapter 2

Compiler perspective

Our goal is automatic generation of sparse matrix codes starting from corresponding dense matrix loop nests and the specification of matrix formats. We start in Section 2.1 by classifying the loop nests that occur in sparse matrix computations and describing the main focus of our compilation effort. Then in Section 2.2 we explore the existing loop transformation technology and discuss its limitations. These limitations lead us to develop a novel *data-centric* approach to code transformations in Section 2.3: instead of finding a loop transformation that best matches preferred access in the data, we generate code by first finding the most efficient enumeration of the data structures. There are two key interrelated problems in implementing this approach in the context of sparse matrix computations: (a) developing an interface for specifying sparse matrix data structures and (b) generating code for efficient enumeration of the data structures.

Our solution is based on re-formulating the code generation problem in the framework of *relational query optimization*: we view vectors and matrices as relations that store tuples of indices and values. Then for a given dense loop nest we can write down a relational query that describes the elements of the data structures that are accessed by the loop nest, as described in Section 2.3.2. This query can be further refined to skip useless computations that involve zeros thus describing the sparse computations. We classify the resulting queries into kinds: *conjunctive* and *disjunctive*. Conjunctive queries only access the matrix and vector elements that are actually stored in the respective data structures. We argue in Section 2.4 that conjunctive queries are most common and focus our attention on them for most of this thesis.

2.1 Classification of matrix codes

Ideally we would like to be able to automatically generate an efficient sparse matrix implementation for any dense loop nest. The complexity of this problem and the feasibility of a solution varies between various kinds of computations. To better understand the issues and focus our attention on the right set of problems, we classify dense matrix computations according to the following two criteria. First, we are interested in *whether the iterations of a loop nest can be arbitrarily reordered.* If so, then we have complete freedom in restructuring the computation for efficient access to data structures. For example, the iterations in matrix-vector product:

```
do i=1, n
    do j=1,n
        y(i) = y(i) + A(i,j) * x(j)
```

can be performed in any order because (a) each element of `y` is computed independently and (b) additions are commutative[1]. We call such loop nests *DOANY*. On the other hand, the computation of the solution of a lower triangular system $\mathbf{Lx} = \mathbf{b}$ imposes sequential order on the outer loop in:

```
do i=1,n
```

[1] Assuming infinite precision arithmetic

Table 2.1: Classification of matrix computations

	No dependences ("DOANY")	Dependences ("DOACROSS")
No fill	Class I $\mathbf{y} = \mathbf{Ax}$ $\mathbf{C} = \mathbf{AB}$ $\mathbf{A} \approx \mathbf{A} + \mathbf{uv}^T$ $\mathbf{z} = a\mathbf{x} + \mathbf{y}$ ($\mathbf{y}$, $\mathbf{z}$ and $\mathbf{C}$ are dense)	Class III Solution of triangular systems
Fill	Class II $\mathbf{y} = \mathbf{Ax}$ $\mathbf{C} = \mathbf{AB}$ $\mathbf{z} = a\mathbf{x} + \mathbf{y}$ ($\mathbf{y}$, $\mathbf{z}$ and $\mathbf{C}$ are sparse)	Class IV Matrix Factorizations

```
    x(i) = b(i)
    do j=1,i-1
        x(i) = x(i) - L(i,j) * x(j)
    x(i) = x(i)/L(i,i)
```

The element of `x(i)` can not be computed before `x(i − 1)`. We call such loop nests *DOACROSS*.

Second, we are interested *whether elements of a vector or a matrix that were originally zero become non-zero during the computation* – i.e. whether *fill* occurs. We have already discussed fill during Cholesky factorization (Section 1.2.1, Figure 1.5). Another example is matrix-matrix product with sparse left-hand side **C**:

```
do i=1,n
    do j=1,n
        do k=1,n
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
```

Fill is important to us for two reasons. First, very specialized techniques are used in sparse matrix software in order to handle fill by judiciously preallocating storage. Second, different data structures allow insertions at different levels. For example, insertion of a non-zero element into a dense matrix is trivial: just set the appropriate location in the array. Whereas, the i-node format used in the BlockSolve library does not allow insertions at all – adding a non-zero to the matrix changes the i-node structure. For these reasons, handling fill requires specialized techniques both in the specification of data structures and in code generation.

Translating the dense codes of Class I into sparse codes involves finding the most efficient access to the data structures. For example, we would like to avoid searching the CRS data structure (Figure 1.12) for a particular column index and would prefer to access this data structure row by row. When translating Class III codes we have to take into account the restriction on the order of the enumeration of the loop indices. Class II requires code that inserts new elements in the data structures, or precomputes non-zero patterns of the left-hand side.

Class IV codes are very special. High-performance sparse implementations of matrix factorization often perform computations which are algebraically different from those done by dense codes. For example, the computations in sparse QR factorization can not be represented by a reordering of the computations in dense QR: the triangular matrix $\mathbf{R}$ is computed though a completely different sequence of orthogonal transformations. The correctness of the sparse algorithm is based on the fact QR factorization is unique up to the sign of the rows of $\mathbf{R}$ [40].

We do not believe it is feasible to *automatically* generate, starting from dense code, high-performance implementations of sparse matrix factorizations, such as supernodal multifrontal algorithm for Cholesky (Figure 1.7 on page 16). However, a closer look at these computations reveals some opportunities. Recall the list of basic kernels used in the algorithm for sparse Cholesky. Besides graph-theoretical manipulation of the matrix, the factorization involves a combination of dense computations and sparse updates of the form $\mathbf{F} = \mathbf{F} + \mathbf{U}$ (line 8, Figure 1.7). Since the fill has already been precomputed by the symbolic factorization, the set of non-zeros in $\mathbf{U}$ is included in $\mathbf{F}$. Therefore this kernel belongs to Class I. In general, sparse factorizations consist of a combination of graph-theoretic manipulation of the data structures, dense matrix computations on sub-problems and Class I sparse computations.

The computations in iterative methods are usually Class I or Class III. Matrix-vector product is in Class I. And the solution of triangular systems is in Class III. ICC(0) is also in Class III. However, sophisticated implementations of these Class III algorithms again require transformations and the knowledge of the algorithm that might be beyond a compiler. The good news is that Class I computations play a key role here as well: inner loops in Class II computations are usually DOANY loops with no fill. An example is the incomplete update in the ICC(0) computation: $\mathbf{B} \approx \mathbf{B} - \mathbf{u}\mathbf{u}^T$.

The only example of Class II computations that we are currently aware are gradient updates of the form $\mathbf{g} = \mathbf{g} + \mathbf{h}$ in codes generated by automatic differentiation packages [21]. The vector $\mathbf{g}$ is usually very sparse and new elements are added to it during the update. We are not aware of occurrences of Class II computations with a matrix on the left-hand side. In general, computations with fill are dealt in special ways by numerical analysts because dynamic insertion into sparse data structures can be rather expensive. In the applications that we have studied (with the notable exception of automatic differentiation) such codes are decomposed "by hand" into a phase that precomputes the fill and the phase that performs the actual computation.

Overall in high-performance sparse codes Class I computations play the role similar to that of dense BLAS codes in dense matrix code – they are the fundamental building block. Unlike dense BLAS code, the Class I codes operate on a variety of data structures thus precluding the development of a sparse BLAS library. Given these considerations, we focus our attention mainly on Class I codes. We will discuss the techniques for handling Class III codes in section 5.9.

2.2 Previous work

Compiler community has paid a great deal of attention to analysis and restructuring of *dense* (a.k.a. regular) loop nests [7, 34–37, 55, 71, 90, 91]. Most of the advances in this area have been facilitated by the *abstraction* of dense loop nests as *polyhedra*: the set of iterations of a loop nest is represented as a polyhedron in an integer space. A coordinate system in this integer space indices the order of enumeration of the points of the polyhedron: points are enumerated in lexicographic order of their coordinates. A linear transformation in the integer space indices a different system of coordinates and, consequently, a different order of enumeration of the polyhedron. The main problem is finding a linear transformation that enhances the performance of the resulting code by improving parallelism or locality of reference. Given the focus of these techniques on directly transforming the code with the goal of indirectly affecting data access, we refer to them as *control-centric transformations*.

We illustrate in section 2.2.1 these ideas on the transformation called *access normalization*. Then in section 2.2.2 we describe how control-centric techniques have been applied in generating sparse codes and point out the limitations of this approach.

2.2.1 Linear loop transformations

Consider dense matrix-vector product:

Program 2.1

```
real y(n), x(n), A(n,n)
do i=1,n
    do j=1,n
        y(i) = y(i) + A(i,j)*x(j)
```

Furthermore, lets assume that the array A is stored in column-major order. Then the successive iterations of the j loop access the elements of A that are n locations apart. For large enough n, on computers with caches this requires bringing in a fresh cache line [42] for every iteration of the j loop. Notice that once the element $\mathtt{A}(\mathtt{i},\mathtt{j})$ is in the cache, then the element $\mathtt{A}(\mathtt{i}+1,\mathtt{j})$ is likely to be in the cache as well, but not the element $\mathtt{A}(\mathtt{i},\mathtt{j}+1)$. We would like to transform the loop nest so as to better exploit this *spatial locality.* This problem is called *access normalization* [55, 56].

Observe that the inner (row) dimension of the A array is accessed in the outer loop. We would like the outer loop, instead, to fix the outer (column) dimension of the array. This suggests a simple solution: interchange the loops, as shown below.

Program 2.2

```
do j=1,n
    do i=1,n
        y(i) = y(i) + A(i,j)*x(j)
```

Now we have *stride-1* access! But what if we have a more complicated array access function, such as in the following fragment?

Program 2.3

```
do i=1,n
    do j=1,n
        ...A(i-j,j+2)...
```

In general we assume that the array access functions are *affine* functions of the loop indices:

$$\begin{aligned} \text{row} &= f_{11}i + f_{12}j + g_1 \\ \text{column} &= f_{21}i + f_{22}j + g_2 \end{aligned} \tag{2.1}$$

where f_{pq} and q_p, $1 \le p, q \le 2$, are integer constants. Such a function can be written in matrix form as:

$$\mathbf{a} = \mathbf{F}\mathbf{i} + \mathbf{g} \tag{2.2}$$

where

$$\mathbf{a} = \begin{pmatrix} \text{row} \\ \text{column} \end{pmatrix} \quad \mathbf{i} = \begin{pmatrix} i \\ j \end{pmatrix} \quad \mathbf{F} = \begin{pmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{pmatrix} \quad \mathbf{g} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} \tag{2.3}$$

In our example $\mathbf{F}$ and $\mathbf{g}$ are:

$$\mathbf{F} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \quad \mathbf{g} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \tag{2.4}$$

The key idea is to model loop iterations as points in integer spaces. In general, we have a loop nest with an array access:

Program 2.4

```
do i1 = ...
    do i2 = ...
```

```
  ...
      do i_d = ...
          ...  A(φ(i)) ...
```

where $\mathbf{i} = (i_1\ i_2\ \ldots\ i_d)^T$ is the vector of loop indices and the array access function $\phi(\mathbf{i})$ is affine. We assume that the loop bounds can be written as linear inequalities on the loop indices. In our example the inequalities are:

$$1 \leq i \leq n \wedge 1 \leq j \leq n \tag{2.5}$$

A conjunction of linear inequalities defines a *polyhedron* $\mathcal{P}$ in the integer space $\mathbb{Z}^d$ [75]. We rewrite out loop nest concisely as:

Program 2.5

```
do i ∈ P
    ...  A(φ(i)) ...
```

Now that we have a mathematical model of loop nests we need a mathematical model of loop transformations. Observe that the loop interchange is a *linear transformation* of the vector of loop indices, which can be modeled by a matrix:

$$\begin{pmatrix} i \\ j \end{pmatrix} \longrightarrow \begin{pmatrix} j \\ i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{2.6}$$

Given a way to represent the transformation we need a way to generate the transformed code. Let $\mathbf{T}$ be the matrix of the transformation. The transformed code enumerates the new loop indices $\mathbf{j} = \mathbf{Ti}$. For now lets assume that $\mathbf{T}$ defines a one-to-one and onto mapping between $\mathbf{j}$ and $\mathbf{i}$ (technically, the matrix $\mathbf{T}$ is *unimodular* [75]). The new code has the same structure as the old:

Program 2.6

```
do j ∈ P'
    ...  A(φ'(j)) ...
```

We have to compute the new loop bounds (that is the polyhedron $\mathcal{P}'$) and the new array access function ϕ'. The set of new iterations $\mathcal{P}'$ is defined by:

$$\mathcal{P}' = \{\mathbf{j} \mid \exists \mathbf{i} \in \mathcal{P} \wedge \mathbf{j} = \mathbf{Ti}\} \tag{2.7}$$

That is each point in $\mathcal{P}'$ obtained by applying the transformation $\mathbf{T}$ to a point in $\mathcal{P}$, which gives us another polyhedron [75]. Projection methods such as Fourier-Motzkin elimination can be used to generate the individual loop bounds [29, 34, 70, 91]. Consider an array element $\mathbf{a}$ accessed in an iteration $\mathbf{i}$ of the original loop. It is also accessed in the iteration $\mathbf{j} = \mathbf{Ti}$ of the new loop nest:

$$\mathbf{a} = \phi(\mathbf{i}) = \phi(\mathbf{T}^{-1}\mathbf{j}) = \phi'(\mathbf{j}) \tag{2.8}$$

In functional notation ϕ' is defined by:

$$\phi' = \phi \circ \mathbf{T}^{-1} \tag{2.9}$$

Let $\phi(\mathbf{i}) = \mathbf{Fi} + \mathbf{g}$ be the definition of the array access function in matrix form. Then ϕ' is defined by:

$$\phi'(\mathbf{j}) = \phi(\mathbf{T}^{-1}\mathbf{j}) = (\mathbf{FT}^{-1})\mathbf{j} + \mathbf{g} = \mathbf{F}'\mathbf{j} + \mathbf{g} \tag{2.10}$$

Now we are ready to reformulate the access normalization problem in terms of the linear loop model. The matrix $\mathbf{F}$ in the original array access function ϕ has the form:

$$\mathbf{F} = \begin{pmatrix} f_{11} & f_{12} & \dots & f_{1d} \\ f_{21} & f_{22} & \dots & f_{2d} \end{pmatrix} \tag{2.11}$$

The last column of this matrix characterizes the accesses in the inner loop. Here are some situations:

1. If $f_{2d} = 0$ and $f_{1d} = 1$ then the inner loop touches successive elements (as stored in memory) of the array.
2. If $f_{2d} = 0$ and $f_{1d} > 1$ then the inner loop touches the array with stride f_{1d}
3. If $f_{2d} \neq 0$ then each iteration of the inner loop touches a different column of the array.

Clearly, case 1 is the most desirable and we would like to avoid case 3. In our initial example (Program 2.1 on page 29) the matrix $\mathbf{F}$ is:

$$\mathbf{F} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.12}$$

Here $f_{2d} = f_{22} = 1$, which is quite bad. In Program 2.2 on page 30:

$$\mathbf{F} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.13}$$

which conforms to case 1. In Program 2.3 on page 30:

$$\mathbf{F} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \tag{2.14}$$

which conforms to case 3. We would like to find a transformation $\mathbf{T}$ such that the new array access matrix $\mathbf{F}'$ has the form:

$$\mathbf{F}' = \mathbf{F}\mathbf{T}^{-1} = \begin{pmatrix} f'_{11} & f'_{12} & \dots & f'_{1,d-1} & 1 \\ f'_{21} & f'_{22} & \dots & f'_{2,d-1} & 0 \end{pmatrix} \tag{2.15}$$

For Program 2.1 on page 29 such transformation is:

$$\begin{aligned} \mathbf{F}' &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \mathbf{F}\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ \mathbf{T} &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned} \tag{2.16}$$

The matrix $\mathbf{T}$ exactly defines loop interchange. The transformation for Program 2.3 on page 30 is:

$$\begin{aligned} \mathbf{F}' &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \mathbf{F}\mathbf{T}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ \mathbf{T} &= \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \end{aligned} \tag{2.17}$$

Let u and v be the new loop indices in this case:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i - j \end{pmatrix} \tag{2.18}$$

The transformed polyhedron is:

$$\begin{aligned} \{u,v \mid \exists i,j : 1 \le i,j \le n \land u = j \land v = i - j\} &= \\ \{u,v \mid 1 \le u + v \le n \land 1 \le u \le n\} \end{aligned} \tag{2.19}$$

The inequalities $1 \le u \le n$ give us the bounds for the outer loop:

```
do u=1,N
```

The inequalities $1 \le u + v \le n$ can be rewritten as bounds $1 - u \le v \le n - u$ on v. The resulting loop nest is:

Program 2.7

```
do u=1,n
    do v=1-u,n-u
        ...  A(v,u+2) ...
```

Details of the computation of the matrix $\mathbf{T}$ in the presence of dependencies between loop iterations can be found in [55, 56].

2.2.2 Control-centric transformations for sparse matrix codes

Bik and Wijshoff have formulated the problem of generating efficient sparse matrix code starting from dense code in terms similar to access normalization. Let us continue with our matrix-vector product example (Program 2.1). Suppose that the matrix $\mathbf{A}$ is sparse and is stored using the CRS format (Figure 1.12). The vectors $\mathbf{y}$ and $\mathbf{x}$ are dense. We would like to convert this program into equivalent sparse matrix code. First, we notice that not all iterations in the set $1 \le i,j \le n$ have to be executed: the iterations for which `A(i,j)` is zero do not update `y(i)` and are redundant. We express the new code as:

Program 2.8

```
do i=1,n
    do j=1,n
        if A(i,j) ≠ 0 then
            y(i) = y(i) + A(i,j) * x(j)
```

The predicate `A(i,j)` $\neq 0$ is called *sparsity guard* and the process of computing it is called *sparsity analysis.* Bik and Wijshoff have formulated the algorithm for computing the guard for general loop bodies [12]. We use the algorithm in our compiler.

Observe that the CRS format provides random access to the rows of the matrix. Each row can viewed as a sparse vector, call it $\mathtt{A}_i$. Now we can rewrite the Program 2.8 as:

Program 2.9

```
do i=1,n
    do j=1,n
        if A_i(j) ≠ 0 then
            y(i) = y(i) + A_i(j) * x(j)
```

Now, instead of numerating all `j` from 1 to `n` we just need to enumerate all non-zero entries in the row:

Program 2.10

```
do i=1,n
    do ⟨j,a⟩ ∈ A_i
```

```
        y(i) = y(i) + a * x(j)
```

Bik and Wijshoff call this technique *guard encapsulation.* Consider the following code fragment with the matrix stored in CRS format:

Program 2.11

```
do i=1,n
    do j=1,n
        if A(p*i+c1,q*i+r*j+c2) ≠ 0 then
            ...  i ...  j ...  A(p*i+c1,q*i+r*j+c2) ...
```

The numbers p, q, r, c_1 and c_2 are all integer constants. Guard encapsulation produces the code:

Program 2.12

```
do i=1,n
    do ⟨a,k⟩ ∈ A_{pi+c1}
        j = (k-q*i-c2)/r
        if j is an integer and 1 ≤ j ≤ n then
            ...  i ...  j ...  a ...
```

Instead of enumerating all j's within the bounds, the inner loop enumerates over all values a and column indices k within the row $\mathtt{A}_{\mathtt{pi+c_1}}$ of A and tests whether this column index could have been accessed by a loop iteration j (and finds such iteration). Notice that the row access function $\mathtt{p} * \mathtt{i} + \mathtt{c}_1$ does not use j. What if did? Then we have to transform the loop nest to bring the access function into the right form – a.la. access normalization described above. Bik and Wijshoff call this *access pattern reshaping* [10]. What if the matrix is stored using CCS format? Then we have to bring access into the form: $\mathtt{A}(\mathtt{q} * \mathtt{i} + \mathtt{r} * \mathtt{j} + \mathtt{c}_2, \mathtt{p} * \mathtt{i} + \mathtt{c}_1)$. Overall, their *sparse compiler* supports data structures which are generalizations of the CRS format in the following sense: the matrix can be compressed along any direction, such as row, column (CRS) or diagonal. The access reshaping method is general enough to handle all these variations. What if we have a more general sparsity predicate P and the loop statement S, each containing references to several sparse arrays, as shown below?

Program 2.13

```
do i=...
    do j=...
        if P then
            S
```

We can perform guard encapsulation if the predicate can be rewritten into the form:

$$P = (A(pi + c_1, qi + rj + c_2) \neq 0) \wedge P' \tag{2.20}$$

for some sparse array A. Then the code can be transformed into:

Program 2.14

```
do i=...
    do ⟨a,k⟩ ∈ A_{pi+c1}
        j = (k-q*i-c2)/r
        if j is an integer and within bounds then
            if P' then
```

```
S'
```

where $\mathtt{S}'$ is the statement `S` with occurrences of the reference to `A` are replaced by `a`. We can then attempt to perform access reshaping and further guard encapsulation. It might also happen that no guard encapsulation is possible. Then (heuristically) some arrays are expanded into dense arrays (which provide random access) and code to search into the data structures for the rest is generated.

These techniques are limited to the variations on CRS format, because it is necessary to have a preferred direction of access (by row, by column, by diagonal) in order to perform linear loop transformations for access reshaping. Suppose the matrix in Program 2.8 on page 33 is stored in the coordinate format (Figure 1.11). This data structure has no preferred direction of access and thus can not be handled by the above techniques. Moreover, the sparse compiler of Bik and Wijshoff has no provisions for adding user-defined formats.

2.3 Our approach

2.3.1 Data-centric transformations

Philosophically, the code transformations described above concentrate on the code – whereas what we really worry about is the efficient access to the data! We would like to drive sparse code generation from the data, which leads us to define *data-centric* code generation methodology: the compiler derives the code that enumerates the data most efficiently and performs the appropriate computation for each accessed data element. Intuitively, we perform guard encapsulation for the whole loop nest at once. Consider the following program fragment with a single sparse matrix access:

Program 2.15

```
do i=...
    do j=...
        if A(φ(i,j)) ≠ 0 then
            ...  A(φ(i,j)) ...
```

The data-centric transformation produces the code:

Program 2.16

```
do ⟨row, column, value⟩ ∈ A
    ⟨i, j⟩ = φ⁻¹(row, column)
    if i and j are integers and within bounds then
        ...  value ...
```

This immediately gives us an idea for specifying data structures to the compiler: for each data structure the user has to provide a macro that implements the loop over the rows, columns and values in the matrix. Unfortunately, this is not the end of the story. Suppose we have a loop nest:

Program 2.17

```
do i=...
    do j=...
        ...  A(φ(i,j)) ...
```

in which the access to the matrix is not guarded by the appropriate conditional. We might not be able to do any better than to search into the data structure. Therefore, our *protocol* for specifying data structures has to include search macros.

In fact, we could build a sparse compiler with such enumeration and search methods. However, when several data structures are involved there are more choices in the process of generating efficient sparse code. As a result, the data structure protocol has to be more elaborate in order to assist the compiler in making the choices.

2.3.2 Relational algebraic model of loop nests

2.3.3 An example

Consider a simple example:

Program 2.18

```
real x(n), y(n)
do i=1,n
    sum = sum + x(i)*y(i)
```

This program computes the dot product $\mathbf{x}^T\mathbf{y}$ of the two vectors. If the vectors are sparse, then we only need to perform the iterations `i` for which both `x(i)` and `y(i)` are non-zero:

Program 2.19

```
do i=1,n
    if x(i) ≠ 0 ∧ y(i) ≠ 0 then
        sum = sum + x(i)*y(i)
```

Now the actual code depends on the data structures that store `x` and `y`. One option is to store each sparse vector in two arrays: the array of non-zero values and the array of the indices of the non-zero values. The storage for `x` is illustrated in Figure 2.1. The vector `x` is stored using the arrays `IX` and `VX`. Similarly, let

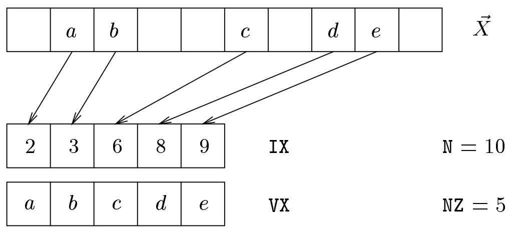


Figure 2.1: Sparse vector storage

the vector `y` be stored using the arrays `IY` (of indices) and `VY` (of values). There are several choices as to how we can translate the rather inefficient Program 2.19 into efficient code. There are three strategies for computing inner product of two sparse vectors [66]:

- ENUMERATE-SEARCH: We can enumerate `x` and for each index and value of `x` search `y` for the matching index and value (or vice versa).
- SCATTER: We can scatter `y` into a dense array, thus reducing the cost of searching.
- MERGE: If the indices of the non-zero values of `x` and `y` can be enumerated in the same sorted order, then we can enumerate the matching indices using the well-known "2-finger" merge algorithm

We now have to answer the following questions:

- How do we choose the best strategy?
- What other information about the data structure has to be provided in order to make the choice?

Consider Program 2.19 from a different angle. We have to generate code to efficiently enumerate the loop indices `i` and the values v_x and v_y of `x` and `y`, respectively, that satisfy the following constraints:

$$L(i, v_x, v_y) = \begin{cases} 1 \le i \le n & \text{dense loop bounds} \\ v_x = \mathtt{x}(i) \wedge v_y = \mathtt{y}(i) & \text{values of the arrays} \\ v_x \ne 0 \wedge v_y \ne 0 & \text{sparse computation} \end{cases} \tag{2.21}$$

Define the following predicates:

$$X(i, v_x) \stackrel{\text{def}}{\equiv} v_x = \mathtt{x}(i) \wedge v_x \ne 0 \tag{2.22}$$

$$Y(i, v_y) \stackrel{\text{def}}{\equiv} v_y = \mathtt{y}(i) \wedge v_y \ne 0 \tag{2.23}$$

$$\mathcal{B}(i) \stackrel{\text{def}}{\equiv} 1 \le i \le n \tag{2.24}$$

Intuitively, the predicates X and Y are true for those indices and values that are actually stored in the data structures and are assumed to be non-zero. Then we can write set of loop indices and values enumerated in Program 2.19 as a *predicate calculus* expression:

$$L(i, v_x, v_y) = \{i, v_x, v_y \mid \mathcal{B}(i) \wedge X(i, v_x) \wedge Y(i, v_y)\} \tag{2.25}$$

Since the arrays `x` and `y` are declared to store elements with indices between 1 and n (see the declaration in Program 2.18), we can simplify this expression by removing the predicate $\mathcal{B}(i)$:

$$L(i, v_x, v_y) = \{i, v_x, v_y \mid X(i, v_x) \wedge Y(i, v_y)\} \tag{2.26}$$

The predicates X and Y already guarantee that i is within loop bounds. This expression can be interpreted in "plain English" as:

> To verify that the loop index i and the values v_x and v_y are enumerated in Program 2.19, verify that both the predicate $X(i, v_x)$ and the predicate $Y(i, v_y)$ are true.

There is another interpretation. We can think of the predicates X and Y as of *sets of tuples* or *relations*. Then verifying whether $X(i, v_x)$ is true is equivalent to verifying whether $\langle i, v_x \rangle \in X$. We can also think of the predicate L as a set – the set of all loop indices and corresponding values enumerated in Program 2.19. From this point of view we can interpret the expression (2.26) as a *recipe* for enumerating the elements of the set L:

> In order to enumerate the the tuples $\langle i, v_x, v_y \rangle$ of L, enumerate the tuples $\langle i, v_x \rangle$ of X and $\langle i, v_x \rangle$ of Y and output those that match on i.

This can be written using the equivalent *relational query*[2] [87]:

$$L\langle i, v_x, v_y \rangle = X\langle i, v_x \rangle \bowtie Y\langle i, v_y \rangle \tag{2.27}$$

X and Y are *relations* (a.k.a. *tables* or *sets of tuples*) that store the indices and values of the respective non-zero elements of the vectors `x` and `y`. The angle bracket notation $X\langle i, v_x \rangle$ is used to name the fields in the tuples and to relate the input tuples to the output tuples. Formally, we can view each tuple μ in X as a mapping from field names (also called *attributes*) to field values. This leads us to the following *set-of-mappings* definition of a relation (see [87], Volume I, Section 2.3):

Definition 2.1 *A* **relation** *R with attributes $\langle a_1, a_2, \ldots, a_n \rangle$ is a set of mappings μ that map attributes to values. The mappings μ are called the* **tuples**. *The set of attributes is called the* **relation schema**.

We will use bold letters to denote a schema $\mathbf{a} = \{a_1, a_2, \ldots, a_n\}$. According to this definition, X has the schema $\langle i, v_x \rangle$ and Y has the schema $\langle i, v_y \rangle$. The *natural join* operator $\bowtie$ outputs those tuples in X and Y that match on the common field i, as illustrated in Figure 2.2. The schema of the output of the natural joins

[2] Relational algebra notation used in this thesis is summarized in Appendix B

$$
\begin{array}{c|l}
\multicolumn{2}{c}{X} \\
i & v_x \\
\hline
1 & 0.5 \\
4 & -7.3 \\
5 & 10.0 \\
8 & -20.456 \\
10 & 3.2
\end{array}
\bowtie
\begin{array}{c|l}
\multicolumn{2}{c}{Y} \\
i & v_x \\
\hline
2 & 12.34 \\
4 & -8.2 \\
8 & 1.3
\end{array}
=
\begin{array}{c|l|l}
i & v_x & v_y \\
\hline
4 & -7.3 & -8.2 \\
8 & -20.456 & 1.3
\end{array}
$$

Figure 2.2: Natural join of X and Y

is the union of the schemata of its inputs.

What is the benefit of writing down the set of values enumerated in a loop nest as a relational algebra expression? First, there exists a large body of research on *query optimization*, that is finding optimal strategies for evaluating relational queries [73, 76, 80, 87]. Second, the relational database community is also faced with the problem of supporting multiple storage formats for the relations. Our goal is to use some of the ideas from the database field in the designing our sparse compiler.

Relational query optimization is based on the following observations:

- Certain identities of relational algebra can be used to rewrite queries (we will see more examples in Part II).
- There are several algorithms for finding the results of applying a relational operator.
- These algorithms can be expressed in terms of generic *access methods* (search, enumeration) of the relations.

In our example, we have only one operator $X \bowtie Y$. There are three basic algorithms for implementing the join ([87], Volume II):

- ENUMERATE-SEARCH join: enumerate the tuples in X and search for matching tuples in Y
- HASH join: insert Y into a hash table H in order to reduce the cost of searching
- SORT-MERGE join: sort X and Y on the common attribute(s) (if not already sorted), then enumerate the matching tuples using the well known "2-finger" merge algorithm.

We now describe how to abstract computations in general loops as relational queries. The rest of the thesis shows how efficient sparse code can be generated from this abstraction.

2.3.4 Relational algebraic model of dense loop computations

Suppose we have a *dense* DOANY loop nest:

Program 2.20

$$
\begin{array}{l}
\texttt{do } \mathtt{i}_1 \texttt{ = ...} \\
\quad \texttt{...} \\
\qquad \texttt{do } \mathtt{i}_n \texttt{ = ...} \\
\qquad\quad \mathtt{S}(\mathbf{s},\mathbf{i},\mathtt{A}_1(\phi_1(\mathbf{s},\mathbf{i})),\mathtt{A}_2(\phi_2(\mathbf{s},\mathbf{i})),\ldots,\mathtt{A}_m(\phi_m(\mathbf{s},\mathbf{i})))
\end{array}
$$

where $\mathbf{i} = (i_1\ i_2\ \ldots\ i_n)^T$ is the vector of loop indices, $\mathbf{s}$ is the vector of *symbolic constants* (i.e. invariants) and $\mathtt{S}$ is some statement that involves array accesses $\mathtt{A}_j(\phi_j(\mathbf{i}))$, $1 \leq j \leq m$. We assume that the loop bounds define a polyhedron $\mathcal{B}$ and the array access functions ϕ_j are affine.

Without loss of generality, assume that the array accesses refer to distinct arrays. Let $\mathbf{a}_j$ denote the vector of the indices of the array in the jth access. Let v_j be a *reference* (in C++ sense [81]) to an array element. We need v_j to be a reference because an array access might occur on the left-hand side of an assignment. For simplicity, we will refer to v_j as the "value".

The loop nest has to enumerate loop indices $\mathbf{i}$, array indices $\mathbf{a}_j$ and array values v_j that satisfy the following constraints:

$$\mathbf{i} \in \mathcal{B} \tag{2.28}$$

$$\mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}), \quad 1 \le j \le m \tag{2.29}$$

$$v_j = \mathtt{A}_j(\mathbf{a}_j), \quad 1 \le j \le m \tag{2.30}$$

(2.28) restricts the loop indices, (2.29) relates array indices to loop indices and invariants, (2.30) relates array values to array indices. Now for each array reference define the predicate $A_j(\mathbf{a}_j, v_j)$ which is true if the value v_j is stored at array location $\mathtt{A}_j(\mathbf{a}_j)$. We can also define the predicate $B(\mathbf{i}) \stackrel{\text{def}}{\equiv} i \in \mathcal{B}$ to represent loop bounds. Then we can rewrite the constraints using predicate calculus:

$$\left\{ \mathbf{i}, \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m \mid B(\mathbf{i}) \wedge \left(\bigwedge_{j=1}^{m} A_j(\mathbf{a}_j, v_j) \wedge \mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}) \right) \right\} \tag{2.31}$$

We can interpret the predicates A_j as relations with the schemata $\langle \mathbf{a}_j, v_j \rangle$. In fact, this gives us the view of arrays $\mathtt{A}_j$ as relations that store indices and values. Before we write down the equivalent relational query, lets define the following predicate:

$$E \stackrel{\text{def}}{\equiv} \bigwedge_{j=1}^{m} \mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}) \tag{2.32}$$

We call E the *data access equation.* The relational query is:

$$\boxed{\sigma_E \Big(B\langle \mathbf{i} \rangle \times A_1\langle \mathbf{a}_1, v_1 \rangle \times \ldots \times A_m\langle \mathbf{a}_m, v_m \rangle \Big)} \tag{2.33}$$

which introduces two new relational operators:

- *Cross product* operator $R\langle \mathbf{a} \rangle \times S\langle \mathbf{b} \rangle$ outputs all possible combinations of tuples in R and in S. The schema of the output is the union $\mathbf{a} \cup \mathbf{b}$ of the schemata of the inputs.
- *Selection* operator $\sigma_{P(\mathbf{a})} R\langle \mathbf{a} \rangle$ outputs those tuples in R that satisfy the predicate $P(\mathbf{a})$ which is a boolean expression in terms of the attributes $\mathbf{a}$.

The meaning of the query (2.33) can be expressed in "plain English" as:

> Take all possible combinations of loop indices that are within loop bounds and the indices and values of the arrays and, then, filter out those that satisfy the data access equation.

We can now rewrite the Program 2.20 as:

Program 2.21
$\mathtt{do}\ \langle \mathbf{i}, \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m \rangle \in \sigma_E \Big(B\langle \mathbf{i} \rangle \times A_1\langle \mathbf{a}_1, v_1 \rangle \times \ldots \times A_m\langle \mathbf{a}_m, v_m \rangle \Big)$
$\quad \mathtt{S}(\mathbf{s}, \mathbf{i}, v_1, \ldots, v_m)$

This loop has the following semantics: first we compute the set L of tuples of indices and values that satisfy the query and then, for each tuple, execute the statement $\mathtt{S}$. Any order of enumeration of the tuples in L is legal, since we have assumed that the loop is DOANY. We call the loop in Program 2.21 the *relational form* of the loop nest in Program 2.20 on the preceding page.

Consider the program that computes the inner product of two dense vectors:

Program 2.22

```
do i=1,n
    sum = sum + x(i)*y(i)
```

The relational form of the program is:

Program 2.23

do $\langle i, i_x, v_x, i_y, v_y \rangle \in \sigma_{i=i_x=i_y}\Big(\{1 \le i \le n\} \times X\langle i_x, v_x\rangle \times Y\langle i_y, v_y\rangle\Big)$

  sum = sum + v_x * v_y

where the relations X and Y store all array indices between 1 and n and the corresponding values. The set $\{1 \le i \le n\}$ of loop indices is treated as a relation with the schema $\langle i \rangle$.

The query (2.33) defines the relational model of executing a *dense* loop nest. We now augment this model to account for sparsity both in the arrays and in the iteration set.

2.3.5 Relational algebraic model for sparse loop computations

Preliminaries

We need some terminology before we move on to define the relational model of loop nests that involve sparse matrices and vectors. First of all, we will use the term *tensor* to refer to either a matrix or a vector or a higher dimensional object. We need to differentiate between tensors and arrays. We will use the term "array" to refer to an array of scalars as provided by the underlying programming language. On the other, the term "tensor" refers to an object that can be indexed by an integer tuple $\mathbf{a}$. An array can store a whole tensor as is the case with dense matrices and vectors. However a tensor might require several array to be stored in as is the case with most sparse matrix data structures. From now on we will use bold letters to denote tensors, as in $\mathbf{A}$, and teletype font to denote "low-level" arrays, as in `A`.

With each tensor we associate a *bounding box* which defines the ranges of the tensor indices $\mathbf{a}$:

$$\mathrm{Bbox}(\mathbf{A}) = \{\mathbf{a} \mid L_1 \le a_1 \le U_1 \wedge \ldots \wedge L_d \le a_d \le U_d\} \tag{2.34}$$

where L_k and U_k are affine functions of symbolic constants. This, of course, includes the special case of the bounds being integer constants. In Program 2.18, the vector `x` has the bounding box:

$$\mathrm{Bbox}(\mathbf{A}) = \{i | 1 \le i \le n\} \tag{2.35}$$

Let $\mathrm{Stored}(\mathbf{A})$ be the set of tensor indices $\mathbf{a}$ that are physically stored in the data structure for vA. It follows from the definition of the bounding box that $\mathrm{Stored}(\mathbf{A}) \subseteq \mathrm{Bbox}(\mathbf{A})$. We say that a tensor $\mathbf{A}$ is *sparse* if $\mathrm{Stored}(\mathbf{A}) \ne \mathrm{Bbox}(\mathbf{A})$ and say that it is *dense* when $\mathrm{Stored}(\mathbf{A}) = \mathrm{Bbox}(\mathbf{A})$.

We assume that the values that are not stored are all zero:

$$\mathbf{a} \notin \mathrm{Stored}(\mathbf{A}) \Rightarrow \mathbf{A}(\mathbf{a}) = 0 \tag{2.36}$$

In general there might be some other "default" value. However, we have seen no applications with it being anything other than zero. Notice that the converse of (2.36) is not true:

$$\mathbf{a} \in \mathrm{Stored}(\mathbf{A}) \not\Rightarrow \mathbf{A}(\mathbf{a}) \ne 0 \tag{2.37}$$

because *accidental cancellation* might set a value to a zero.

What is the value of the expression $\mathbf{A}(\mathbf{a})$ for some sparse tensor $\mathbf{A}$ and index $\mathbf{a}$? If $\mathbf{a}$ is stored in $\mathbf{A}$ than this expression returns a reference v to (equivalently, a "location" of) the value of the element of the tensor. If $\mathbf{a}$ is not stored in $\mathbf{A}$ then the expression $\mathbf{A}(\mathbf{a})$ returns the special value ω. When used on the right-hand side of an assignment ω is equivalent to a zero. When used on the left-hand side ω causes a run-time error.

Relational form of sparse loop nests

Consider a loop nest:

Program 2.24
```
do i_1 = ...
    ...
        do i_n = ...
            S(s, i, A_1(φ_1(s, i)), A_2(φ_2(s, i)), ..., A_m(φ_m(s, i)))
```

where `S` is an assignment statement. Given that some of the tensors are sparse, there are situations when the statement `S` is a no-op for certain combinations of the values of the tensors being zero. Moreover, if the statement has an access to a sparse tensor on the left-hand side, the program might fail because it would try to assign to the location ω. For example, in Program 2.18 the statement is

```
sum = sum + X(i)*Yx(i)
```

If either $\mathbf{X}(i)$ or $\mathbf{Y}(i)$ are zero then the statement reduces to:

```
sum = sum + 0
```

and can be skipped. Ideally we would like to execute the statement only for such iterations `i` that the predicate $P \stackrel{\text{def}}{\equiv} \mathbf{X}(i) \neq 0 \wedge \mathbf{Y}(i) \neq 0$ is true. P can be rather expensive to test, since even sparse tensors can store zeros. However we can use a *weaker* predicate:

$$P' = (i \in \text{Stored}(\mathbf{X}) \wedge i \in \text{Stored}(\mathbf{Y})) \tag{2.38}$$

It is safe to use this predicate because $\neg P' \Rightarrow \neg P$ – the iterations that are skipped by P' are also skipped by P, although under P' more iterations might eventually be executed. We are interested in P' because, as we will see later, there are efficient ways of enumerating solutions to it.

Consider the loop nest:

Program 2.25
```
do i=1,n
    A(i) = B(i)
```

If both $\mathbf{A}$ and $\mathbf{B}$ are sparse, there are four combinations of the values of the predicates $i \in \text{Stored}(\mathbf{A})$ and $i \in \text{Stored}(\mathbf{B})$:

1. When $i \in \text{Stored}(\mathbf{A}) \wedge i \in \text{Stored}(\mathbf{B})$ we just assign the value
2. When $i \notin \text{Stored}(\mathbf{A}) \wedge i \notin \text{Stored}(\mathbf{B})$, the statement is equivalent to `0` $=$ `0` and we can skip the iteration.
3. When $i \notin \text{Stored}(\mathbf{A}) \wedge i \in \text{Stored}(\mathbf{B})$ we need to insert the element into $\mathbf{A}$. This case is called *fill*.
4. When $i \in \text{Stored}(\mathbf{A}) \wedge i \notin \text{Stored}(\mathbf{B})$ then we might want to delete the element of $\mathbf{A}$, because it is being set to zero. This case is called *annihilation*.

In order to produce minimally correct code we need to account for fill:

Program 2.26
```
do i=1,n
    if i ∉ Stored(A) then insert(A,i) endif
    A(i) = B(i)
```

Or we might want to account for all the non-trivial cases:

Program 2.27

```
do i=1,n
    if i ∈ Stored(A) ∨ i ∈ Stored(B) then
        if i ∉ Stored(A) then    ``fill''
            insert(A,i)
            A(i) = B(i)
        else if i ∉ Stored(B) then    ``annihilation''
            delete(A,i)
        else
            A(i) = B(i)
        endif
    endif
enddo
```

In general we need to perform the following tasks, collectively referred to as *sparsity analysis*:

- Compute the *sparsity guard* SP which filters out useless iterations.
- Modify the loop body to account for fill and annihilation.

Bik and Wijshoff describe rules for computing the sparsity guard for an assignment statement:

```
var = rhs
```

The statement must be executed whenever `rhs` evaluates to a non-zero *OR* `var` is physically allocated (it is a scalar or a stored element of a tensor). In the opposite case we have $\texttt{rhs} = \texttt{0}$ and `var` refers to an element of a sparse tensor that is not stored and is assumed to be zero, so the statement is $\texttt{0} = \texttt{0}$.

Using the notation from [79], we can write the guard for the statement as

$$ALC(\texttt{var}) \vee NZ(\texttt{rhs}) \tag{2.39}$$

The NZ predicate is defined by recursion on the structure of the `rhs` expression[3]:

$$\begin{array}{rcl} NZ(e1+e2) & = & NZ(e1) \vee NZ(e2) \\ NZ(e2*e2) & = & NZ(e1) \wedge NZ(e2) \\ NZ(e1/e2) & = & NZ(e1) \\ NZ(e1^{e2}) & = & NZ(e1) \\ NZ(\texttt{var}) & = & (\texttt{var} \neq 0) \\ NZ(0) & = & \texttt{false} \\ NZ(const) & = & \texttt{true} \end{array} \tag{2.40}$$

$ALC(\texttt{var})$ is defined as:

$$ALC(\texttt{var}) = \begin{cases} \texttt{true} & \text{if } \texttt{var} \text{ is a scalar variable} \\ \mathbf{a} \in \text{Stored}(\mathbf{A}) & \text{if } \texttt{var} \text{ is a tensor access } \mathbf{A}(\mathbf{a}) \end{cases} \tag{2.41}$$

Observe that $ALC(\mathbf{A}(\mathbf{a}))$ is always true for a dense tensor $\mathbf{A}$, unless the input program accesses out of bounds elements of $\mathbf{A}$ and is thus illegal.

A very important special case of the statement `S` is when it is of the form:

```
var = var + update
```

[3]This grammar is variation on the one presented in [9, 14]

In this case the sparsity guard is $NZ(\texttt{updates})$.

The grammar (2.40) does not introduce "negative" terms of the form $\texttt{var} = 0$ into the predicate. It follows that we can replace the $\texttt{var} \neq 0$ terms by $ALC(\texttt{var})$ terms. These terms are either `true` or have the value $\mathbf{a} \in \text{Stored}(\mathbf{A})$ for some tensor $\mathbf{A}$. Let us consider some examples.

Program 2.28

```
do i=1,n
    A(i) = B(i)
```

The predicate is:

$$\mathbf{i} \in \text{Stored}(\mathbf{A}) \vee \mathbf{i} \in \text{Stored}(\mathbf{B}) \tag{2.42}$$

Program 2.29

```
do i=1,n
    A(i) = A(i) + B(i)*C(i)
```

The predicate is:

$$\mathbf{i} \in \text{Stored}(\mathbf{B}) \wedge \mathbf{i} \in \text{Stored}(\mathbf{C}) \tag{2.43}$$

Program 2.30

```
real x
x = ...
do i=1,n
    A(i) = x
```

The predicate is:

$$(ALC(\mathbf{A}(i)) \vee ALC(\texttt{x})) = (\mathbf{i} \in \text{Stored}(\mathbf{A}) \vee \texttt{true}) = \texttt{true} \tag{2.44}$$

Suppose we have computed the sparsity guard P for the statement in Program 2.24 on page 41. By construction, P is a boolean expression that combines terms of the form:

$$\phi_k(\mathbf{s}, \mathbf{i}) \in \text{Stored}(\mathbf{A}_k) \tag{2.45}$$

using logical OR and logical AND operators. Let $\mathcal{B}$ be the set of dense loop iterations given by the loop bounds. The the set of loop iterations, tensor indices and tensor values enumerated by the loop nest is given by the following constraints (compare to (2.28)):

$$\mathbf{i} \in \mathcal{B} \tag{2.46}$$

$$\mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}), \quad 1 \leq j \leq m \tag{2.47}$$

$$v_j = \mathbf{A}_j(\mathbf{a}_j), \quad 1 \leq j \leq m \tag{2.48}$$

$$P(\mathbf{a}_1 \in \text{Stored}((A)_1), \ldots, \mathbf{a}_m \in \text{Stored}((A)_m)) \tag{2.49}$$

We can rewrite this in relational calculus form by defining the predicates $\mathbf{A}_j(\mathbf{a}_j, v_j, b_j)$ as follows.

- $\mathbf{A}_j(\mathbf{a}_j, v_j, true)$ is true if $\mathbf{a}_j \in \text{Stored}(\mathbf{A}_j)$ and v_j is the value of tensor element with index $\mathbf{a}_j$.
- $\mathbf{A}_j(\mathbf{a}_j, \omega, false)$ is true if $\mathbf{a}_j \notin \text{Stored}(\mathbf{A}_j)$.

That is, the flag b_j is just an indicator of whether the element of the tensor is stored or not. The enumeration of the loop nest is then defined by the following expression:

$$\left\{ \mathbf{i}, \mathbf{a}_1, v_1, b_1, \ldots, \mathbf{a}_m, v_m, b_m \;\middle|\; B(\mathbf{i}) \wedge P(b_1, \ldots, b_m) \wedge \Big(\bigwedge_{j=1}^{m} \mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}) \wedge \mathbf{A}_j(\mathbf{a}_j, v_j, b_j) \Big) \right\} \tag{2.50}$$

In "plain English" the expression says:

> The tuple $\langle \mathbf{i}, \mathbf{a}_1, v_1, b_1, \ldots, \mathbf{a}_m, v_m, b_m \rangle$ of loop indices, tensor indices, values and indicators is enumerated in the sparse version of the Program 2.24 if and only if the loop indices are within loop bounds, and the indicators satisfy the sparsity predicate P, and the loop indices and tensor indices satisfy tensor access functions and the tensor indices, values and indicators agree.

Just as we did in Section 2.3.4 we can interpret the predicates B and $\mathbf{A}_j$ as relations. The equivalent relational query is then:

$$\boxed{\sigma_P \sigma_E \Big(B\langle \mathbf{i} \rangle \times \mathbf{A}_1 \langle \mathbf{a}_1, v_1, b_1 \rangle \times \ldots \times \mathbf{A}_m \langle \mathbf{a}_m, v_m, b_m \rangle \Big)} \tag{2.51}$$

where E is the data access equation as defined by (2.32) and P is the sparsity predicate written in terms of the indicators b_j. Observe that the sub-query:

$$\sigma_E \Big(B\langle \mathbf{i} \rangle \times \mathbf{A}_1 \langle \mathbf{a}_1, v_1, b_1 \rangle \times \ldots \times \mathbf{A}_m \langle \mathbf{a}_m, v_m, b_m \rangle \Big) \tag{2.52}$$

describes the computation in the dense loop nest. In the light of this, the query in (2.51) provides the following recipe for the enumerating the sparse loop nest:

> Take the tuples that are enumerated in the dense loop nest and filter them through the sparsity predicate.

Now in order to rewrite the original dense loop in sparse relational form we need to modify the body to account for fill. Assume that the statement in `S` in Program 2.24 is of the form `var` = `rhs`. The relational form of the sparse loop nest is:

Program 2.31

```
do ⟨i, a1, v1, b1, ..., am, vm, bm⟩ ∈ ...
        ... σP σE (B⟨i⟩ × A1⟨a1, v1, b1⟩ × ... × Am⟨am, vm, bm⟩)
    if NZ(rhs) then
        if ¬ALC(var) then insert into var endif
        S1
    else if ALC(var) then
        delete from var
    endif
enddo
```

The statement `S1` is obtained by replacing array references in `S` by their respective value fields v_j. The predicates $NZ(\texttt{rhs})$ and $ALC(\texttt{var})$ can be written in terms of the indicators b_j.

The relational form of Program 2.27 is:

Program 2.32

do $\langle i, i_a, v_a, b_a, i_b, v_b, b_b \rangle \in \ldots$

$\ldots \sigma_{b_a \vee b_b} \sigma_{i=i_a=i_b} \Big(\{1 \leq i \leq n\} \times \mathbf{A}\langle i_a, v_a, b_a \rangle \times \mathbf{B}\langle i_b, v_b, b_b \rangle \Big)$

if b_b then

if $\neg b_a$ then v_a = insert($\mathbf{A}$,i) endif

v_a = v_b

else if b_a then

delete($\mathbf{A}$,i)

endif

enddo

2.3.6 Strengthening the sparsity predicate

Sometimes the dense loop nest does not convey all of the information in the code. Take, for example, the incomplete outer-product update $\mathbf{B} \approx \mathbf{B} + \mathbf{u}\mathbf{v}^T$. The dense loop nest:

Program 2.33

```
do i=1,n
    do j=1,n
        B(i,j) = B(i,j) + u(i)*v(j)
```

does not convey the fact that we only want to update the elements in $\mathbf{B}$ that are already stored. We should instead write:

Program 2.34

do i=1,n

do j=1,n

if $\langle i, j \rangle \in \mathrm{Stored}(\mathbf{B})$ then

$\mathbf{B}$(i,j) = $\mathbf{B}$(i,j) + $\mathbf{u}$(i)*$\mathbf{v}$(j)

Then it is quite easy to strengthen the sparsity predicate from

$$i \in \mathrm{Stored}(\mathbf{u}) \wedge j \in \mathrm{Stored}(\mathbf{v}) \tag{2.53}$$

to

$$i \in \mathrm{Stored}(\mathbf{u}) \wedge j \in \mathrm{Stored}(\mathbf{v}) \wedge \langle i, j \rangle \in \mathrm{Stored}(\mathbf{B}) \tag{2.54}$$

There are several benefits to strengthening the predicate:

- We might avoid handling fill and/or annihilation
- We obtain the loop nest which only accesses the elements of the tensors that are stored. We call such loop a *conjunctive* loop.

Another way to strengthen the predicate is though an assertion. Consider the update loop in Cholesky factorization:

Program 2.35

```
do i=1,n
    do j=1,n
```

```
        F(i,j) = F(i,j) + U(i,j)
```

In this form the loop nest does not convey the fact that **F** is preallocated. In particular the user can assert that the following predicate holds:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{U}) \Rightarrow \langle i,j\rangle \in \text{Stored}(\mathbf{F}) \tag{2.55}$$

We can use this assertion to strengthen the sparsity predicate from:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{U}) \tag{2.56}$$

to:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{U}) \wedge \langle i,j\rangle \in \text{Stored}(\mathbf{F}) \tag{2.57}$$

We will write the update loop as:

Program 2.36

```
do i=1,n
    do j=1,n
        if (⟨i,j⟩ ∈ Stored(U) ∧ ⟨i,j⟩ ∈ Stored(F)) then
            F(i,j) = F(i,j) + U(i,j)
```

Assertions are implicit for accesses to dense arrays. Consider sparse matrix-vector product:

Program 2.37

```
do i=1,n
    do j=1,n
        Y(i) = Y(i) + A(i,j)*X(j)
```

In general case, the sparsity predicate is:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{A}) \wedge j \in \text{Stored}(\mathbf{X}) \tag{2.58}$$

However, since vY is dense we know that $i \in \text{Stored}(\mathbf{Y})$ is true (as long as i is within dense array bounds). Therefore we can strengthen the predicate to:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{A}) \wedge j \in \text{Stored}(\mathbf{X}) \wedge i \in \text{Stored}(\mathbf{Y}) \tag{2.59}$$

Asserting that **Y** is dense is equivalent to rewriting Program 2.37 as:

Program 2.38

```
do i=1,n
    do j=1,n
        if(i ∈ Stored(Y) ∧ ⟨i,j⟩ ∈ Stored(A) ∧ j ∈ Stored(X)) then
            Y(i) = Y(i) + A(i,j)*X(j)
```

In general, we start with a dense loop nest with the conditional and assertions:

Program 2.39

```
do i1 = ...
    ...
```

```
do i_n = ...
    assert(T)
    if C then
        S(s, i, A_1(φ_1(s, i)), A_2(φ_2(s, i)), ..., A_m(φ_m(s, i)))
```

After computing the sparsity predicate P we can strengthen it to:

$$P' = P \wedge T \wedge C \tag{2.60}$$

2.4 Focus of the thesis

Observe that all but one of the sparse DOANY kernels discussed in Sections 1.2.3 and 2.1 correspond to conjunctive loop nests:

- Sparse matrix-vector product $\mathbf{Y} = \mathbf{AX}$ with dense $\mathbf{Y}$ when $\mathbf{Y}$ is dense
- Incomplete outer-product update
- Update in Cholesky factorization

These codes access only the elements of the vectors and matrices that are physically stored. Our current implementation handles conjunctive loop nests only and we for most of this thesis we focus our attention on generating parallel and sequential code for this category.

We can simplify the relational model of conjunctive loop nests by removing the indicator fields. Consider a general sparse loop nest:

Program 2.40

```
do ⟨i, a_1, v_1, b_1, ..., a_m, v_m, b_m⟩ ∈ ...
        ... σ_P σ_E (B⟨i⟩ × A_1⟨a_1, v_1, b_1⟩ × ... × A_m⟨a_m, v_m, b_m⟩)
    if NZ(rhs) then
        if ¬ALC(var) then insert into var endif
        S1
    else if ALC(var) then
        delete from var
    endif
enddo
```

The loop in conjunctive if the predicate P is:

$$P = b_1 \wedge b_2 \wedge \ldots \wedge b_m \tag{2.61}$$

Clearly, no fill or annihilation happens in this case. Moreover we can rewrite the loop nest without the indicator fields and the sparsity predicate:

Program 2.41

```
do ⟨i, a_1, v_1, ..., a_m, v_m⟩ ∈ ...
        ... σ_E (B⟨i⟩ × A_1⟨a_1, v_1⟩ × ... × A_m⟨a_m, v_m⟩)
        S1
```

where the relations $\mathbf{A}_j$ hold only the tuples that are actually stored in the corresponding tensors. Not surprisingly this loop nest is identical to the dense loop nest 2.21 on page 39, because loop nests over dense tensors also access only those elements that are stored.

Part II

Sequential code generation

Preview

The key idea of our work is that tensors can viewed as relations and loop nests as relational queries. The data structures for storing tensors provide access methods for searching and enumerating individual elements. Different data structures provide different access methods which, in turn, can have varying costs. As a result, program transformations based on relational query rewriting are necessary to achieve the best performance.

Since our compiler does not have a fixed set of data structures "hard-wired" into it, it is necessary to present an abstraction of storage formats to the compiler for use in code generation. Such an abstraction must be sufficiently general to describe a variety of storage formats and, at the same time, it must not prevent the compiler from generating efficient code. We employ a two-level abstraction: basic program objects, such as scalars and arrays, are combined together to form instances of abstract data types. ADTs, in turn, are used to specify storage formats in a structured manner.

Chapter 3 deals with mechanisms for manipulating abstract data types. ADTs provide a view of low-level *concrete* variables (integers, arrays, strings) in terms of high-level *methods*. The methods are essentially macros and are translated at compile-time into operations on concrete variables. We call the programs that operate on ADTs *abstract programs* and the programs that operate only on concrete variables *concrete programs*.

It is convenient to represent each format as a collection of ADTs. For example, the CCCS format (Figure 4.2) can be decomposed into the ADT for enumerating over the columns and the ADT representing each column. We call such nesting *index hierarchy*. Chapter 4 describes the *Black Box Protocol* which ties together several ADTs into a *black box* – the representation of a storage format.

Query optimization consists of three steps: query scheduling (Section 5.2), join scheduling (Section 5.5) and instantiation (Section 5.6). Query scheduling uses *rewrite rules* to transform a query into nested loops that enumerate over the results of *one-dimensional affine joins*. These joins generalize equi-joins: the predicate can be of the form $a \cdot i = b \cdot j + c$, where i and j are field of the relations being joined and a, b, c are integers. The key operation in query rewriting is discovery of 1D affine joins. We use the echelon form of the *data access equation* to this end. We call the result of query scheduling a *high-level plan*. Join scheduling translates high-level plans into low-level plans by assigning an implementation to each affine join. The choice is based on the properties of storage formats as provided by the Black Box Protocol. A low-level plan is an abstract program in terms of ADTs that implement the matrices. One option is to mechanically translate it into a concrete program by expanding method invocations in some standard order. However, it turns out that selectively expanding some methods before others enables important optimizations, as described in Section 5.8.

The overall architecture of the sequential code generator is shown in Figure 2.3. We illustrate the main ideas on the following example. Consider the code fragment that computes the matrix-vector product $\mathbf{Y} = \mathbf{AX}$:

Program 2.42
```
do i=1,n
    do j=1,n
        Y(i) := Y(i) + A(i,j)*X(j)
```

This is a DOANY loop – the iterations can be performed in any order. Assume that the matrix and the

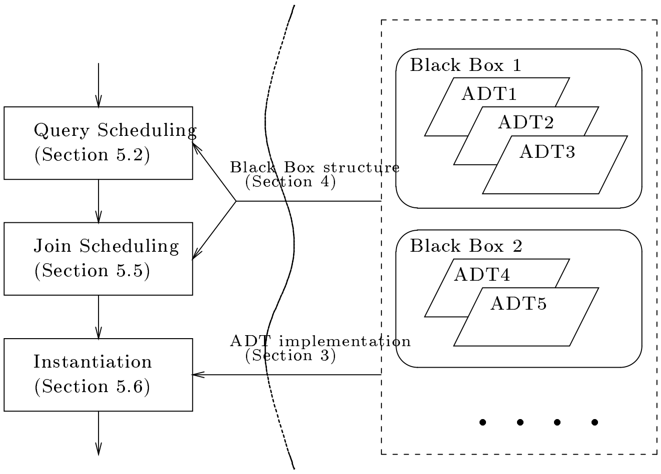


Figure 2.3: Architecture of sequential code generator

vector **X** are sparse and the vector **Y** is dense. The sparsity predicate for the loop nest is

$$\langle i,j\rangle \in \text{Stored}(\mathbf{A}) \wedge \langle j\rangle \in \text{Stored}(\mathbf{X}) \tag{2.62}$$

If either the element of **A** or the element of **X** is zero, then the update to $\mathbf{Y}(i)$ is redundant. Since **Y** is dense, we can strengthen the predicate to:

$$\langle i,j\rangle \in \text{Stored}(\mathbf{A}) \wedge \langle j\rangle \in \text{Stored}(\mathbf{X}) \wedge \langle i\rangle \in \text{Stored}(\mathbf{Y}) \tag{2.63}$$

Because we have to enumerate over only those elements of the vectors and the matrix that are physically stored, this is a conjunctive DOANY loop. We can rewrite it in relational form as:

Program 2.43

`do` $\langle i,j,r,c,a,i_y,y,j_x,x\rangle \in \ldots$

$\quad\quad \ldots\sigma_{r=i_y=i \wedge c=j_x=j}\left(\ \{1 \le i,j \le n\} \times \mathbf{A}\langle r,c,a\rangle \times \mathbf{X}\langle j_x,x\rangle \times \mathbf{Y}\langle i_y,y\rangle\ \right)$

```
    y := y + a * x
```

In the query **A** is the relation that stores the tuples of row and column indices, and the non-zero values. **X** is the relation that stores the tuples of vector indices and non-zero values. **Y** is the relation that stores the tuples of vector indices and values, all of which are assumed to be non-zero. Our goal now is to translate the loop nest in Program 2.43 into efficient code.

The translation depends on the properties of the data structures. Suppose the matrix is stored in the CCS format (Figure 1.13). Then the columns c of the matrix can be enumerated in the outer loop and the rows – in the inner loop. Since $j_x = c = j$ we can enumerate in the outer loop the *join* between the set of columns of **A** and the set of indices of **X**. In the inner loop the enumeration is over the join between the rows of **A** and the indices of **Y**:

Program 2.44

`do` $\langle j_x,c\rangle \in \mathbf{A} \bowtie \mathbf{X}$

$\quad$ `do` $\langle i_y,r\rangle \in \mathbf{A}_c \bowtie \mathbf{Y}$

If the matrix were stored in CRS format (Figure 1.12), then the order of the joins could be reversed:

Program 2.45

```
do ⟨i_y, r⟩ ∈ A ⋈ Y
    do ⟨j_x, c⟩ ∈ A_r ⋈ X
```

In general, the loops over the results of relational queries, such as the one in Program 2.43, are translated into a nesting of joins by repeatedly applying certain transformation rules, as described in Section 5.4. The rules are based on linear algebraic manipulation of the data access equation in order to discover equalities between fields (indices). The choice of which rules to apply depends on the nesting of the indices within each relation (matrix). Black Box protocol provides the interface through which the nesting is described to the compiler. In turn, enumeration of the indices at each level of the nesting is described by an ADT. The rules for specifying matrix formats through ADTs are described in Chapter 4. Basic notation for specifying ADTs is discussed in Chapter 3.

The loop nests in Programs 2.44 and 2.45 are called *high-level plans*. They specify the nesting of the joins, but do not yet specify how the joins are to be implemented. The step of join implementation, described in Section 5.5, selects the best implementation algorithm for each of the joins, based on the properties of the storage. For example, in Program 2.45 the set of row indices of $\mathbf{A}$ is the interval $[1 \ldots n]$, as is the set of vector indices of $\mathbf{Y}$. To join the two we simply computes the intersection of the intervals. The code for the outer loop in the program is:

```
do i_y=r=1,n
```

The inner loop in the program joins a row of $\mathbf{A}$ and the vector $\mathbf{X}$. Both are sparse. One possibility is to scatter $\mathbf{X}$ into a dense vector $\mathbf{H}$ and then enumerate over the row $\mathbf{A}$ and the matching element of $\mathbf{X}$ in the dense vector. This is basically the *hash* implementation of the join:

```
allocate H
do ⟨j_x, x⟩ ∈ X
    insert ⟨j_x, x⟩ into H
enddo
do c ∈ A_r
    lookup j_x = c in H
enddo
```

When we put the two loops together, we can lift the allocation of $\mathbf{H}$ and scattering of $\mathbf{X}$ outside of the outer loop. This results in the following *low-level plan*:

Program 2.46

```
allocate H
do ⟨j_x, x⟩ ∈ X
    insert ⟨j_x, x⟩ into H
enddo
do i_y=r=1,n
    do c ∈ A_r
        lookup j_x = c in H
```

The low-level plan has joins translated into primitive operations on the data structures, such as the enumeration over the elements of $\mathbf{X}$ and $\mathbf{A}_r$. These operations are defined by the access methods present in the ADT for the relations. To obtain executable code the low-level plan is instantiated by macro-expanding the definitions of the access methods, as described in Section 5.6. Various optimizations are possible, just before instantiation, as described in Section 5.8.

Chapter 3

Programming with ADTs

3.1 Notation

We start by describing the intermediate language used by the Bernoulli compiler. Then we describe the notation for specifying and manipulating ADTs. The intermediate language is basically Fortran with some minor changes for clarity and expressiveness:

- We use square brackets in array declarations and to denote array accesses, as in `X[i]`.
- On paper we use mathematical symbols $\leq$, $\geq$, etc. instead of `.le.`, `.ge`, etc.
- We use := for assignment and = for comparison.
- We allow declarations in the middle of a block. The scoping rules are the same as in C++.
- We allow array bounds to be any valid expressions, as in

```
integer n
n := ...
real A[n+1]
```

- On paper we use *italics* for program comments.

3.2 Abstract Data Types

3.2.1 Method invocation and expansion

Let's start with an example. Consider a data structure for sparse vectors illustrated in Figure 3.1. The array `IX` holds the indices of the non-zero values and the array `VX` holds the corresponding values. Additionally, the integer `N` holds the range of vector indices and `NZ` holds the number of entries actually stored. Assume for now that the entries in the `IX` array are sorted in ascending order. We wish to manipulate the variables `N`, `NZ`, `IX` and `VX` as an instance of a "sparse vector" ADT with the following methods:

- `search(i,h)` searches the data structure for a particular index `i` and sets the *iterator* `h` to point to the entry in the data structure. If the index is not found, then `h` is set to some default value.
- `valid(h)` tests whether the iterator `h` points to a valid entry in the sparse vector.

Of course, the question now is: how is `h` represented? In our example an entry in the sparse vector is identified by its offset within `IX` and `VX` arrays. So `h` is an instance of a "sparse vector iterator" ADT which views an integer variable, call it `ii`, as a pointer to the individual elements in the sparse vector. We can use the value `ii` = `0` as an invalid iterator.

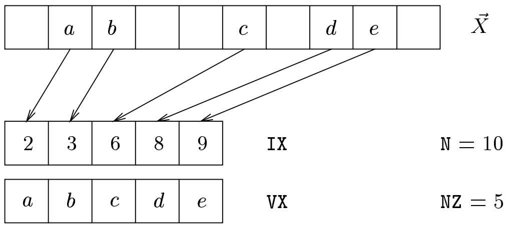


Figure 3.1: An example sparse vector storage

```
integer N, NZ, i, ii
integer IX[NZ]
real VX[NZ]
abstract X(SPVO n=N nz=NZ ind=IX val=VX)
abstract h(SPVO_ITER offset=ii)
i := ...
X.search(i,h)
if (X.valid(h)) then
          ....
endif
```

Figure 3.2: Programming with ADTs

```
1.  SPVO.search(i,h) ≡ Stmt⟨
2.      begin
3.          integer $mid, $low, $high
4.          $low := 1; $high := nz
5.          h.offset() := 0
6.          while ($low ≤ $high) do
7.              mid := ⌈($low+$high)/2⌉;
8.              if (i = ind[$mid]) then
9.                  h.offset() := $mid
10.                 break
11.             else if (i < ind[$mid]) then $high := $mid
12.             else $low := $mid
13.             endif
14.         enddo
15.     end ⟩
16.
17. SPVO.valid(h) ≡ Expr⟨
18.     h.offset() ≠ 0 ⟩
```

Figure 3.3: Methods for `SPVO` ADT

The code fragment in Figure 3.2 illustrates our notation for manipulating ADTs. Lines 1 through 3 declare the concrete variables. We use `teletype` font for concrete variables and **bold** font for abstract variables. Line 4 declares **X** as an instance of the sparse vector ADT. "`SPVO`" stands for "`SP`arse `V`ector `O`rdered". The variables with lower case names: `n`, `nz`, `ind` and `val` – are the *formal arguments* to the ADT. The assignments that follow the ADT name (e.g. `ind` = `IX`) associate formal arguments with their values which can be any valid expression. Line 5 defines an instance of an iterator.

The constructs **X**`.search` and **X**`.valid` (lines 7 and 8) denote invocations of methods. The invocations are expanded by the compiler into either expressions, statements or variable declarations. Inside of the compiler ADTs are represented as objects that generate code fragments for method invocations: given the name of method (e.g. "search") and the representation of the arguments as abstract syntax trees (ASTs), an ADT returns the code fragment for the method invocation. By default, ADTs provide methods that return the values of the formal arguments.

The representation of ADTs as compiler objects is quite flexible. One can devise ADTs that expand method invocations into code fragments with structure dependent on the form of the trees for the input arguments. However, often the methods are just glorified macros. The notation for defining methods in these cases is illustrated in Figure 3.3.

Lines 1 and 17 start method definitions. The notation: "`Stmt`⟨" and "`Expr`⟨" – tell us that the method returns a statement or an expression, respectively. We also use "`Decl`⟨" to denote a method that returns a list of variable declarations. Unique names are generated for the variables whose names start with the dollar sign "$". This is done in order to avoid shadowing.

The semantics of ADT definitions is that the compiler replaces method invocations with code inside of method definitions modulo replacement of formal arguments to the ADT (e.g. `nz` on line 4 and `ix` on line 8) as well as formal argument to the method (e.g. `i` and **h**) with their values. Notice that several "rounds" of such macro expansion is necessary to obtain the *concrete program* – the program in terms of concrete variables only. In our example, expansion of **X**.valid on line 8 in Figure 3.2 produces:

`if (`**h**`.offset()` $\neq$ `0) then`

which, in turn, expands into:

`if (ii` $\neq$ `0) then`

ADTs can also provide *metrics* which are the methods that do not expand into any code, but provide compiler with some information about the data type. For example, the sparse vector ADT can provide a `sorted_p` metric to tell the compiler whether the entries can be enumerated in sorted order. In the case of `SPVO` format the definition of this metric is:

`SPVO.sorted_p` $\equiv \langle$ `true` $\rangle$

3.2.2 ADT expressions

Sometimes it is convenient to manipulate instances of ADTs without explicitly naming them. We introduce *ADT expressions* of the form:

```
(SPVO n=N nz=NZ ind=IX val=VX)
```

which can be used instead of instance names. For example, the statement:

`if ( (SPVO n=N nz=NZ ind=IX val=VX).valid(`**h**`) ) then`

will be expanded into:

`if (`**h**`.offset()` $\neq$ `0) then`

The introduction of ADT expression allows us to construct methods that return instances of ADTs. This idiom is used heavily in the definitions of ADTs for tensors.

3.2.3 Discussion

For now, we leave the order of method expansion undefined. We assume that the compiler non-deterministically picks the next method to expand and repeats the expansion until we are left with a concrete program. Notice that, in general, the concrete program depends on the order of method expansion and there are no guarantees that the expansion process terminates. However, the ADTs for tensors are very structured and there is less room for an error.

A legitimate question is: why use this ADT mechanism and not the class abstraction mechanism found in object-oriented languages, such as C++ or Java? The primary reason is efficiency. As we will see in the next chapter, ADTs are ultimately used to describe how the individual elements of sparse matrices are accessed. In the context of high-performance numerical software we simply can not afford to pay the overhead of method dispatch and invocation for each matrix element. The method expansion mechanism that we use is a *light-weight* alternative to object-oriented languages.

Our representation of ADTs as essentially collections of macros is closest to the mechanism described by Gordon Novak [63, 64] for the creation and manipulation of *views*. For example, a point in two dimensions is a view of a pair of numbers x and y as its Cartesian coordinates. Methods, such as translation or scaling can be defined in terms of the appropriate operations on the numbers x and y. For us,a point is an abstract variable which is stored in terms of two concrete variables: x and y.

Chapter 4

Black Box Protocol

Our goal is to be able to generate efficient sparse matrix code from dense conjunctive DOANY loops and specification of sparse matrix formats. As we will discuss in Chapter 5, there are usually several implementations for a given DOANY loop nest (and the corresponding relational query). The choice of the best implementation depends on the access methods that are available for the data structures in the input code, as well as on the properties of the methods.

In this chapter we discuss the Black Box Protocol for specifying storage formats. The main idea is to represent data formats as collections of ADTs based on the nesting of indices within each format. We call this nesting *index hierarchy*. For example, the Compressed Row Storage format (Figure 1.12 on page 23) allows random access to rows, but sequential access within each row. We represent this format by using one ADT to describe the access to rows and another to describe the access within each row.

We start this chapter by discussing the concept of index hierarchies. Then we describe how Black Box Protocol decomposes a data structure into a collection of ADTs based on the index hierarchy. Some examples are presented at the end of the chapter.

4.1 Index hierarchy

Consider the Compressed Compressed Column Storage (CCCS) format illustrated in Figure 4.2. The matrix for this example is shown in Figure 4.1. This format is similar to the CCS format (Figure 1.13), although the list of columns is compressed, as well. For a given offset `jj` between 1 and `NZCOL`: `COLIND[jj]` is the column index j and `COLP[jj]` points to the sections of the `ROWIND` and `VALS` arrays that store the column j. A loop nest that enumerates over the row and column indices and values of the matrix is illustrated in Figure 4.3. Notice that the column indices are enumerated in the outer loop and the row indices – in the inner loop. It turns out that many data structures for sparse matrices exhibit this kind of nesting which we call *index hierarchy*.

More formally, the CCCS data structure stores the set $\mathbf{A}$ of $\langle i, j, value\rangle$ tuples. Let $\mathbf{A}.\mathtt{enum}()$ be the set of $\langle jj, j\rangle$ pairs enumerated in lines 1 and 2 (Figure 4.3). Let $\mathbf{A}_{jj}$ be the set of $\langle i, value\rangle$ tuples that belong to the column pointed to by jj. Then, the jj loop on line 1 decomposes $\mathbf{A}$ into the disjoint union:

$$\mathbf{A}\langle i, j, value\rangle = \biguplus_{\langle jj,j\rangle \in \mathbf{A}.\mathtt{enum}()} \{j\} \times \mathbf{A}_{jj}\langle i, value\rangle \tag{4.1}$$

Similarly, the loop on line 3 decomposes $\mathbf{A}_{jj}$ into the disjoint union:

$$\mathbf{A}_{jj}\langle i, value\rangle = \biguplus_{\langle ii,i\rangle \in \mathbf{A}_{\mathtt{jj}}.\mathtt{enum}()} \{i\} \times \mathbf{A}_{jj,ii}\langle value\rangle \tag{4.2}$$

$$\begin{pmatrix} a & 0 & b & 0 \\ 0 & 0 & 0 & 0 \\ c & 0 & 0 & d \\ e & 0 & f & g \end{pmatrix}$$

Figure 4.1: An example matrix

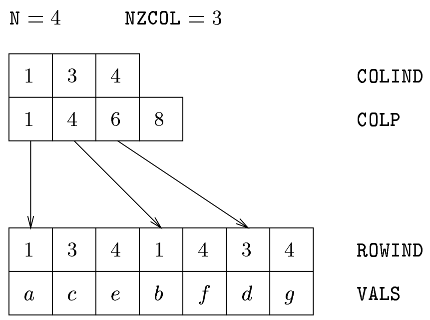


Figure 4.2: CCCS format

```
do jj=1, NZCOL
    j := COLIND[jj]
    do ii=COLP[jj],COLP[jj+1]-1
        i := ROWIND[ii]
        value := VALS[ii]
    enddo
enddo
```

Figure 4.3: Enumeration over CCCS format

Putting it all together, the data structure represents $\mathbf{A}$ as the double (disjoint) union:

$$\mathbf{A}\langle i, j, value\rangle = \left(\biguplus_{\substack{\langle jj,j\rangle\in\\ \mathbf{A}.\mathtt{enum}()}} \{j\} \times \left(\biguplus_{\substack{\langle ii,i\rangle\in\\ \mathbf{A}_{\mathtt{jj}}.\mathtt{enum}()}} \{i\} \times \mathbf{A}_{jj,ii}\langle value\rangle \right)\right) \tag{4.3}$$

Notice that the unions can not be interchanged because the set $\mathbf{A}_{\mathtt{jj}}.\mathtt{enum}()$ of $\langle ii, i\rangle$ pairs depends on the value of jj. We denote such hierarchy by the expression:

$$j \prec i \prec value \tag{4.4}$$

Now consider the ELEMENT storage format illustrated in Figure 4.4. The loop nest that enumerates over the data structure is shown in Figure 4.5. Observe that the enumeration of the row indices i (lines 1 and 2) can be interchanged with the enumeration of the column indices j (lines 3 and 4). Let $\mathbf{A}.\mathtt{enum}_{\mathtt{i}}()$ be the set of $\langle ii, i\rangle$ tuples and $\mathbf{A}.\mathtt{enum}_{\mathtt{j}}()$ be the set of $\langle jj, j\rangle$ tuples enumerated in the loop nest. By analogy with (4.3), we can write the decomposition of $\mathbf{A}$ into the disjoin union:

$$\mathbf{A}\langle i, j, value\rangle = \biguplus_{\substack{\langle ii,i\rangle\in\\ \mathbf{A}.\mathtt{enum}_{\mathtt{i}}()}} \biguplus_{\substack{\langle jj,j\rangle\in\\ \mathbf{A}.\mathtt{enum}_{\mathtt{j}}()}} \{i\} \times \{j\} \times \mathbf{A}_{ii,jj}\langle value\rangle \tag{4.5}$$

However, in this case the unions are independent can be interchanged. We denote such hierarchy by $(i \times j) \prec value$. Notice that dense storage has the same hierarchy.

Consider the COORDINATE storage format illustrated in Figure 4.6. The loop nest that enumerates over the data structure is shown in Figure 4.7. In this case the data structure provides a "flat" view of the set $\mathbf{A}$:

$$\mathbf{A} = \biguplus_{\substack{\langle kk,i,j\rangle\in\\ \mathbf{A}.\mathtt{enum}()}} \{i\} \times \{j\} \times \mathbf{A}_{kk}\langle value\rangle \tag{4.6}$$

We denote this hierarchy as $\langle i, j\rangle \prec value$.

One more example. Consider a permutation $\mathbf{P}$. We can think of it as a relation that stores $\langle i, i'\rangle$ tuples of related indices. A possible data structure for a permutation is to store it using two integer arrays: `PERM` and `IPERM` so that:

$$i' = \mathtt{PERM}[i] \quad i' = \mathtt{IPERM}[i]$$

This data structure has two kinds of nesting. For a given value of i we can obtain i'. Also for a given value of i' we can obtain i. We say that this data structure provides two hierarchies:

$$(i \prec i') \oplus (i' \prec i) \tag{4.7}$$

(For lack of a better symbol, we use "$\oplus$" to denote a disjunction of hierarchies.) Overall, here is the grammar

2	5	12	14	COLIND

ROWIND

ROWIND	VALS			
3	a	b	c	d
8	e	f	g	h
11	i	j	k	l

N = 20 NROW = 3 NCOL = 4

Figure 4.4: ELEMENT storage

```
do ii = 1, NROW
    i := ROWIND[ii]
    do jj = 1, NCOL
        j := COLIND[jj]
        value := VALS[ii,jj]
    enddo
enddo
```

Figure 4.5: The enumeration over ELEMENT storage

a	g	f	c	b	e	d	VALS
1	4	4	3	1	4	3	ROWIND
1	4	3	1	3	1	4	COLIND

N = 4 NNZ = 7

Figure 4.6: COORDINATE storage format

```
do kk = 1, NNZ
    i := ROWIND[kk]
    j := COLIND[kk]
    value := VALS[kk]
enddo
```

Figure 4.7: Enumeration over COORDINATE storage

Table 4.1: Examples of index hierarchy kinds

Data structure	Hierarchy kind
set of integers	i^*
a real scalar	v:Real
vector	$i^* \prec v$:Real
compressed column storage	$j^* \prec i^* \prec v$:Real
dense matrix	$(i^* \times j^*) \prec v$:Real
coordinate storage	$\langle i, j \rangle^* \prec v$:Real
a tuple	$\langle i, j \rangle^1$
permutation	$(i^* \prec j^1) \oplus (j^* \prec i^1)$

for building index hierarchy specifications:

$$
\begin{array}{lcl}
\text{Hierarchy-Kind} & : & \text{Hierarchy}\ (\oplus\text{Hierarchy})^* \\
\text{Hierarchy} & : & \text{Term}\ (\prec \text{Term})^*\ [\prec \text{ValueField}] \\
 & & \mid \text{ValueField} \\
\text{Term} & : & \text{Product-Term} \mid \text{Tuple-Term} \\
\text{Product-Term} & : & \text{Field}\ (\times\text{Field})^+ \\
\text{Tuple-Term} & : & \langle \text{Identifier}\ (,\text{Identifier})^* \rangle^{\text{Arity}} \mid \text{Identifier}^{\text{Arity}} \\
\text{Field} & : & \text{Identifier}^{\text{Arity}} \\
\text{ValueField} & : & \text{Identifier : Type} \\
\text{Type} & : & \text{“Real”} \mid \text{“Integer”} \\
\text{Arity} & : & \text{“1”} \mid \text{“}*\text{”}
\end{array}
\tag{4.8}
$$

Square brackets denote an optional element in the grammar. The arity indicators "^{1}" and "*" are used to distinguish between data structures that hold exactly one element and those that hold zero or more. Some examples are shown in Table 4.1. For brevity we will omit the arity indicators whenever possible and assume "*" for integer index fields. We also omit type indicators for value field and assume the type "Real", unless specified otherwise. For example, "$i \prec j \prec v$" means "$i^* \prec j^* \prec v$:Real".

4.2 Decomposition into ADTs

Based on the index hierarchy we represent each data structure as the collection of ADTs. Suppose we have a data structure for a relation $\mathbf{A}$ with fields $f_1, f_2, \ldots, f_d$. According to the grammar in (4.8), the hierarchy kind is a disjunction of one or more hierarchies:

$$\mathcal{K} = \bigoplus_k \mathcal{H}_k \tag{4.9}$$

And, in turn, each of the hierarchies has the form:

$$\mathcal{H} = \mathcal{T}_1 \prec \mathcal{T}_2 \prec \ldots \prec \mathcal{T}_m \tag{4.10}$$

where each $\mathcal{T}_j$ denotes a product, a tuple or a value term. A black box has a *root ADT* that corresponds to the disjunction in (4.9). The root ADT takes as arguments all concrete variables and expressions that constitute the data structure. By convention, the root ADT has the same name as the data structure. The root provides the following methods and metrics:

- The metric `schema()` describes the schema of the stored relations. It returns the list of field names. The list is used by other ADTs in the black box in order to identify the position of each field in the hierarchy.

- The metric hierarchy provides the hierarchy kind (i.e. the disjunction (4.9)).
- bounds() returns the expression for the bounds of the integer fields (i.e. indices of a tensor)
- $\mathtt{h}_k()$ returns an ADT expression for each hierarchy in the disjunction. For brevity we will use "**A**" instead of "$\mathbf{A}.\mathtt{h}_1()$" if the union (4.9) contains only a single hierarchy.

Other ADTs correspond to the terms in the hierarchy (4.10). The access methods that each ADT provides depend on the structure of the term and on the position of the term in the hierarchy. We distinguish two kinds of terms. *Bottom* terms occur at the bottom of the hierarchy (e.g $\mathcal{T}_m$ in (4.10)). All other terms are referred to as *middle* terms.

Each term can be of the form "$\langle f_1, \ldots, f_q\rangle^1$", "$\langle f_1, \ldots, f_q\rangle^*$", "$f_1 \times \ldots \times f_q$" or "$v$:Type". In each of the cases the ADT has to provide the fields() metric that returns the list of field names. Other methods depend on the structure of the term:

4.2.1 Singleton tuple term

A singleton tuple term is of the form

$$\mathcal{T} = \langle f_1, \ldots, f_q\rangle^1 \tag{4.11}$$

The ADT represents a single tuple and has to provide the following methods and metrics:

- $\mathtt{deref}_{f_j}()$, $1 \leq j \leq q$, method provides the expression for the value of the field f_j.
- If this is a middle term then the hderef() method returns an ADT expression for the next level down in the hierarchy.

These methods allows us terms to express the relation **A** as:

$$\mathbf{A}\langle\mathbf{f}\rangle = \{\langle f_1, \ldots, f_q\rangle\} \times \mathbf{A}.\mathtt{hderef}()\langle\mathbf{f}'\rangle \tag{4.12}$$

where $f_j = \mathbf{A}.\mathtt{deref}_{f_j}()$ and $\mathbf{f}'$ is the vector of remaining fields. Interpreting the vectors of fields as sets of fields we can write $\mathbf{f}' = \mathbf{f} \setminus \{f_1, \ldots, f_q\}$.

4.2.2 Tuple stream term

A tuple stream term is of the form:

$$\mathcal{T} = \langle f_1, \ldots, f_q\rangle^* \tag{4.13}$$

The ADT provides methods to enumerate and search a set of tuples:

- enum() is used in high-level plans (Section 5). It returns the set of $\langle\ \mathbf{h}, f_1, \ldots, f_q\ \rangle$ tuples, where **h** is the iterator over this level in the hierarchy. The enum() method does not expand into any code and is later replaced by some of the methods described below.
- declare_iter(**h**) returns the declarations for the iterator **h**. For example, in the case of the SPVO storage this method can be defined as:

```
SPVO.declare_iter(h) ≡ Decl⟨
    integer $ii
    abstract h(SPVO_ITER offset=$ii)
⟩
```

(Recall that the dollar sign indicates the variables for which the compiler pick a unique name.)

- $\texttt{open}(\mathbf{h}, \texttt{pos})$ returns the statement that initializes the iterator. If the tuples can be enumerated both in ascending and descending order, then $\texttt{open}(\mathbf{h}, 0)$ positions the iterator at the "smallest" tuple and $\texttt{open}(\mathbf{h}, 1)$ positions the iterator at the "largest" tuple in the lexicographic ordering. If the tuples are not stored in a sorted order, then either invocation positions the iterator at some arbitrary "first" tuple.

- $\texttt{valid}(\mathbf{h})$ returns the expression that tests whether the iterator $\mathbf{h}$ is valid.

- $\texttt{deref}_{f_j}(\mathbf{h})$ is the expression for the field f_j in the tuple.

- $\texttt{next}(\mathbf{h})$ is the statement that advances the iterator to the next "larger" tuple in the lexicographic ordering.

- $\texttt{prev}(\mathbf{h})$ is the statement that advances the iterator to the next "smaller" tuple in the lexicographic ordering.

- `sorted_p()` metric returns `true` if the tuples can be enumerated in lexicographic order. The presence of `next` or `prev` methods indicates whether the tuples can enumerated in ascending or descending order. If $\texttt{sorted}_\texttt{p}$ is `false`, then only the method `next` should be provided.

- $\texttt{close}(\mathbf{h})$ is the statement that "cleans up" the iterator. In most cases, this is a no-op.

- If $\mathcal{T}$ is a middle term, then $\texttt{hderef}(\mathbf{h})$ is the ADT expression that represents the next level down in the hierarchy. $\texttt{hderef}(\mathbf{h})$ is not used if $\mathcal{T}$ is the bottom term. In relational expressions we use the shorthand notation "$\mathbf{A_h}$" instead of "$\mathbf{A}.\texttt{hderef}(\mathbf{h})$".

- $\texttt{search}(\mathbf{h}, \texttt{f}_1, \ldots, \texttt{f}_\texttt{q})$ returns the statement that positions the iterator at the tuple with the given values of fields. The resulting iterator is valid if the tuple is found and is invalid otherwise.

- `search_cost()` metric informs the compiler of the cost of the search. The return value one of the symbols: `O1`, `Olog`, `On` – which indicate whether the search can be done in constant time, logarithmic time or linear time. The metric is optional with `On` being the default value.

- `search_dense_p()` metric returns `true` if any combination of the values of the fields f_1 through f_q that satisfies the bounds given by the root ADT is guaranteed to be stored.

The above methods express the relation $\mathbf{A}$ as:

$$\mathbf{A} = \biguplus_{\substack{\langle \mathbf{h}, f_1, \ldots, f_q \rangle \in \\ \mathbf{A}.\texttt{enum}()}} \{\langle f_1, \ldots, f_q \rangle\} \times \mathbf{A_h}\langle \mathbf{f}' \rangle \tag{4.14}$$

where $\mathbf{f}' = \mathbf{f} \setminus \{f_1, \ldots, f_q\}$.

These methods are sufficient to generate code. However, there are a two special cases that, when recognized, can lead to better code:

- **DENSE**: The term has the form $\mathcal{T} = i^*$, i takes on integer values and all the values within a range are stored. Additionally, i serves as the iterator. For example, the row index in the CRS data structure satisfies this properties. In this case three special methods have to be provided: `lb()`, `ub()` – they return the expression for the lower bound and upper bound of the enumeration, respectively. The methods `open`, `valid`, `deref`, `next`, `prev`, `close`, `search` and `search_cost` are ignored as they can be generated from the specification of the enumeration range. `hderef` still has to be provided.

- **INDIRECT**: The iterator is an integer offset, say `ii`, and the values of the fields are obtained via an indirection array: $\texttt{f}_j = \texttt{some\_array}_\texttt{j}[\texttt{ii}]$. In this case the ADT has to provide the bounds for the offset, as well as the special methods $\texttt{ideref}_j$. $\texttt{ideref}_j$ returns the name of the indirection array for the j-th field. For example, we can define the methods for the `SPVO` data structure as follows:

  $\texttt{SPVO.lb()} \equiv \texttt{Expr}\langle\ 1\ \rangle$

$\texttt{SPVO.ub()} \equiv \texttt{Expr}\langle\ \texttt{nz}\ \rangle$
$\texttt{SPVO.ideref()} \equiv \texttt{Expr}\langle\ \texttt{ind}\ \rangle$

As before, the `open`, `valid`, `next`, `prev`, `close` methods are redundant and are ignored. $\texttt{deref}_j$ methods can replace $\texttt{ideref}_j$ in situations when the iterator is just an integer, but the dereference expression is more complicated than a simple array lookup. `search` method is optional.

Although each of the special cases can be deduced from the methods present, we require the ADTs to provide a metric that distinguishes the cases:

- `special_kind()` metric returns one of the symbols: `Stream`, `Dense`, `Indirect`. `Stream` marks the ADTs that provide general stream interface.

4.2.3 Product term

A product term has the form:

$$\mathcal{T} = f_1 \times \ldots \times f_q \tag{4.15}$$

The protocol for product terms is motivated by the `ELEMENT` storage format [1] illustrated in Figure 4.4. For a given row offset `ii` and column offset `jj`, `VALS[ii,jj]` is the value of a non-zero and `ROWIND[ii]` and `COLIND[jj]` are its row and column indices, respectively. The hierarchy kind for this data structure is:

$$i \times j \prec v \tag{4.16}$$

since the row and column indices can be enumerated independently. Notice that `ii` can be seen as the iterator over the row indices and `jj` can be seen as the iterator over the column indices. So we need to specify two sets of enumeration and search methods – one for each iterator. In general, the following methods should be provided:

- $\texttt{enum}_{f_j}()$, $1 \leq j \leq q$ are used in high-level plans. They return the set of $\langle \mathbf{h}_j, f_j \rangle$ tuples.
- $\texttt{declare\_iter}_{f_j}(\mathbf{h}_{f_j})$ provide the declarations of the iterators for each dimension of the product.
- The following subscripted methods have the same meaning as their counterparts in the protocol for tuple terms:

$\texttt{open}_{f_j}(\mathbf{h}_{f_j}, \texttt{pos})$	$\texttt{valid}_{f_j}(\mathbf{h}_{f_j})$	$\texttt{deref}_{f_j}(\mathbf{h}_{f_j})$
$\texttt{sorted\_p}_{f_j}()$	$\texttt{next}_{f_j}(\mathbf{h}_{f_j})$	$\texttt{prev}_{f_j}(\mathbf{h}_{f_j})$
$\texttt{close}_{f_j}(\mathbf{h}_{f_j})$	$\texttt{search}_{f_j}(\mathbf{h}_{f_j}, \texttt{f}_\texttt{j})$	$\texttt{search\_cost}_{f_j}()$
$\texttt{search\_dense\_p}_{f_j}()$	$\texttt{special\_kind}_{f_j}()$	

- $\texttt{hderef}(\mathbf{h}_1, \ldots, \mathbf{h}_q)$ returns the ADT expression that represents the next level down in the hierarchy. In relational expressions we will use the shorthand "$\mathbf{A}_{\mathbf{h}_1,\ldots,\mathbf{h}_q}$" instead of "$\mathbf{A}.\texttt{hderef}(\mathbf{h}_1, \ldots, \mathbf{h}_\mathbf{q})$".

Just as is the case with the tuple terms, the methods $\texttt{lb}_{\mathtt{f_j}}()$ and $\texttt{ub}_{\mathtt{f_j}}()$ can be specified for **DENSE** range enumeration and $\texttt{lb}_{\mathtt{f_j}}()$, $\texttt{ub}_{\mathtt{f_j}}()$ and $\texttt{ideref}_{\mathtt{f_j}}()$ can be specified for **INDIRECT** enumeration for each dimension of the product.

The product term expresses the relation as:

$$\mathbf{A}\langle\mathbf{f}\rangle = \biguplus_{\substack{\langle \mathbf{h}_1, f_1\rangle \in \\ \mathbf{A}.\texttt{enum}_{\mathtt{f}_1}()}} \cdots \biguplus_{\substack{\langle \mathbf{h}_q, f_q\rangle \in \\ \mathbf{A}.\texttt{enum}_{\mathtt{f}_q}()}} \{\langle f_1, \ldots, f_q \rangle\} \times \mathbf{A}_{\mathbf{h}_1,\ldots\mathbf{h}_q}\langle \mathbf{f}' \rangle \tag{4.17}$$

where $\mathbf{f}' = \mathbf{f} \setminus \{f_1, \ldots, f_q\}$.

[1] The name "`ELEMENT`" comes from the application of this data structure in finite-element computations [54].

4.2.4 Value term

A value term simply has the form:

$$v : \text{Type} \tag{4.18}$$

The ADT for the term has to provide the following methods:

- `deref()` returns an expression for the value of the field
- `lvalue_p()` is the metric that tells the compiler whether the above expression can be used as an l-value (i.e. on the left-hand side of an assignment)

The value term expresses the relation $\mathbf{A}\langle v\rangle$ as singleton set:

$$\mathbf{A}\langle v\rangle = \{v\} \tag{4.19}$$

4.3 Requirements for iterators

In Chapter 3 the iterators were used rather informally. We now state the requirements for the iterators used in Black Boxes. The ADTs for iterators are required to provide the following methods and metrics:

- The method `marshal(b,k)` marshals the state of the iterator into the array `b` of integers starting with position `k`.
- The method `unmarshal(b,k)` recovers the state of the iterator from the array `b` of integers starting with position `k`.
- The metric `size()` returns the size (in integers) of the buffer necessary to hold the state of the iterator.

The (un)`marshal()` methods are used in to maintain hash-tables of iterators in hash-join algorithms. Both methods return a statement. Usually an iterator is represented by few integer variables or pointer. Therefore, we have chosen to count the size in integers.

We assume that if the region of the buffer `b` from which the iterator is unmarshalled is all zero, then the iterator sets itself to an invalid state.

In the special cases of **DENSE** or **INDIRECT** enumerations the marshaling of the iterator is implicit.

4.4 Assumptions

The following assumptions are necessary for the correctness of the code generation algorithms that we describe in Chapter 5:

- The access methods do not change the state of a black box instance. (They can, of course, change the state of an iterator.)
- The `open()`, `next()`, `deref()` methods return a stream of unique indices.

4.5 Examples

4.5.1 Sparse vector storage.

The hierarchy kind for the `SPVO` storage is: $i^* \prec v$:Real. The root ADT has the name "`SPVO`", the ADT for the i^* – `SPVO1` and the ADT for the value term v:Real is `REAL_VAL`:

Formal arguments to the root: `n, nz, ind, val`

`SPVO.schema()` $\equiv \langle$ `i, j, v` $\rangle$

```
SPVO.hierarchy() ≡ ⟨ i* ≺ v:Real ⟩
SPVO.bounds() ≡ Expr⟨ 1 ≤ i ≤ n ⟩
SPVO.h() ≡ Expr⟨ (SPVO1 nz=nz ind=ind val=val) ⟩

SPVO1.fields() ≡ ⟨ i ⟩
SPVO1.special_kind() ≡ ⟨ Indirect ⟩
SPVO1.lb() ≡ Expr⟨ 1 ⟩
SPVO1.ub() ≡ Expr⟨ nz ⟩
SPVO1.sorted_p() ≡ ⟨ true ⟩
SPVO1.ideref() ≡ Expr⟨ ind ⟩
SPVO1.hderef(ii) ≡ Expr⟨ (REAL_VAL name=v e=val[ii]) ⟩

REAL_VAL.fields() ≡ ⟨ name ⟩
REAL_VAL.deref() ≡ Expr⟨ e ⟩
REAL_VAL.lvalue_p() ≡ ⟨ true ⟩
```

Suppose a vector **X** has the declaration:

```
abstract X(SPVO n=N nz=NZ ind=IX val=VX)
```

then the pseudo-code:

```
do ⟨i,v⟩ ∈ X
    print(i, v)
enddo
```

can be rewritten in terms of access methods as:

```
integer ii
do ii = X.h().lb(), X.h().ub(), X.h().step()
    print( (X.h().ideref())[ii], (X.h().hderef(ii)).deref() )
enddo
```

which, in turn, expands into the following concrete program:

```
integer ii
do ii = 1, NZ, 1
    print( IX[ii], VX[ii] )
enddo
```

4.5.2 CCCS

The hierarchy kind is $j^* \prec i^* \prec v$:Real. The ADT for the j^* term is `CCCS_COL`, the ADT for the i^* term is `CCCS_ROW` and, again, we use `REAL_VAL` for the value term:

Formal arguments: `n, nzcol, colind, colp, rowind, vals`

```
CCCS.schema() ≡ ⟨ i, j, v ⟩
CCCS.hierarchy() ≡ ⟨ j* ≺ i* ≺ v:Real ⟩
CCCS.bounds() ≡ ⟨ 1 ≤ i,j ≤ n ⟩
CCCS.h() ≡ ⟨ (CCCS_COL n=n ....)  ⟩

CCCS_COL.fields ≡ ⟨ j ⟩
CCCS_COL.special_kind ≡ ⟨ Indirect ⟩
CCCS_COL.lb() ≡ Expr⟨ 1 ⟩
CCCS_COL.ub() ≡ Expr⟨ nzcol ⟩
CCCS_COL.ideref() ≡ Expr⟨ colind ⟩
CCCS_COL.sorted_p() ≡ Expr⟨ true ⟩
CCCS_COL.hredef(jj) ≡ Expr⟨
    (CCCS_ROW
        start=colp[jj] end=(colp[jj+1]-1)
        ind=rowind vals=vals) ⟩
CCCS_COL.fields ≡ ⟨ i ⟩
CCCS_COL.special_kind ≡ ⟨ Indirect ⟩
CCCS_ROW.lb() ≡ Expr⟨ start ⟩
CCCS_ROW.ub() ≡ Expr⟨ end ⟩
CCCS_ROW.step() ≡ Expr⟨ 1 ⟩
CCCS_ROW.ideref() ≡ Expr⟨ rowind ⟩
CCCS_COL.sorted_p() ≡ Expr⟨ true ⟩
CCCS_ROW.hderef(ii) ≡ Expr⟨ (REAL_VAL name=v e=val[ii]) ⟩
```

The pseudo-code:

```
abstract A(CCCS n=N nzcol=NZCOL colind=COLIND
        colp=COLP rowind=ROWIND rowp=ROWP)
do ⟨i,j,v⟩ ∈ A
    print(i, j, v)
enddo
```

can be translated into the following program in terms of the ADTs:

```
integer jj
do jj = A.h().lb(), A.h().ub(), A.h().step()
    integer ii
    do ii = A.h().hderef(jj).lb(),A.h().hderef(jj).ub(),
                                 A.h().hderef(jj).step()
```

```
        print ( (A.h().hderef(jj).ideref())[ii],
                (A.h().ideref())[jj],
                A.h().hderef(jj).hderef(ii).deref())
    enddo
enddo
```

The concrete program is:

```
integer jj
do jj = 1, NZCOL, 1
    integer ii
    do ii = COLP[jj], COLP[jj+1]-1, 1
        print ( ROWIND[ii], COLIND[jj], VALS[ii] )
    enddo
enddo
```

4.5.3 Dense matrix

In many applications we have to view a "flat" buffer of real values as a dense matrix. BLAS library interface [33] specifies the following parameters that describe the layout of a $\mathbf{A}$ matrix in memory:

- buf is the address of the start of the matrix. In Fortran terms $\texttt{buf}[1]$ holds the value of $\mathbf{A}_{11}$.
- m is the number of rows.
- n is the number of columns.
- lda is the *leading dimension* of the matrix, whose role will become apparent shortly. lda must not be smaller than m.
- The boolean flag T tells whether the matrix should be transposed or not. Equivalently, $T = false$ says that the matrix is stored in column-major order and $T = true$ says that the matrix is stored in row-major order (the transpose of column-major, which is the default).

If $T = false$, then the matrix $\mathbf{A}$ is laid out in *column-major* order as follows:

$$\begin{array}{lcl}
\mathbf{A}_{11} & \longrightarrow & \texttt{buf}[1] \\
\mathbf{A}_{21} & \longrightarrow & \texttt{buf}[2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{m1} & \longrightarrow & \texttt{buf}[\texttt{m}] \\
\mathbf{A}_{12} & \longrightarrow & \texttt{buf}[\texttt{lda}+1] \\
\mathbf{A}_{12} & \longrightarrow & \texttt{buf}[\texttt{lda}+2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{m2} & \longrightarrow & \texttt{buf}[\texttt{lda}+\texttt{m}] \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{1n} & \longrightarrow & \texttt{buf}[(\texttt{n}-1)*\texttt{lda}+1] \\
\mathbf{A}_{2n} & \longrightarrow & \texttt{buf}[(\texttt{n}-1)*\texttt{lda}+2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{mn} & \longrightarrow & \texttt{buf}[(\texttt{n}-1)*\texttt{lda}+\texttt{m}]
\end{array} \tag{4.20}$$

Intuitively, it $\mathbf{A}$ is the first m rows of the matrix $lda \times n$ matrix $\mathbf{B}$, which is laid out in column-major order. If $T = true$ then$\mathbf{A}$ is laid out as:

$$\begin{array}{lll}
\mathbf{A}_{11} & \longrightarrow & \mathtt{buf}[1] \\
\mathbf{A}_{12} & \longrightarrow & \mathtt{buf}[2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{1n} & \longrightarrow & \mathtt{buf}[\mathtt{n}] \\
\mathbf{A}_{21} & \longrightarrow & \mathtt{buf}[\mathtt{lda}+1] \\
\mathbf{A}_{22} & \longrightarrow & \mathtt{buf}[\mathtt{lda}+2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{2n} & \longrightarrow & \mathtt{buf}[\mathtt{lda}+\mathtt{n}] \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{m1} & \longrightarrow & \mathtt{buf}[(\mathtt{m}-1)*\mathtt{lda}+1] \\
\mathbf{A}_{m2} & \longrightarrow & \mathtt{buf}[(\mathtt{m}-1)*\mathtt{lda}+2] \\
\vdots & \vdots & \vdots \\
\mathbf{A}_{mn} & \longrightarrow & \mathtt{buf}[(\mathtt{m}-1)*\mathtt{lda}+\mathtt{n}]
\end{array} \tag{4.21}$$

The DENSE_MATRIX black box described this format. It has the following formal arguments:

- start = buf[address] declares that the matrix is stored in the array buf starting with the element address.
- colmajor = true declares that the matrix is stored in column-major order. colmajor = false declares that the matrix is stored in row-major order.
- leading = lda is the leading dimension of the layout.
- nrow = m and ncol = n are the dimensions of the matrix.
- istart and jstart give the indices of the first element of $\mathbf{A}$ stored. Equivalently, the arguments declare that the sub-matrix $\mathbf{A}(istart : m, jstart : n)$ is stored.

Here is the Black Box:

Program 4.1

DENSE_MATRIX.schema() ≡ ⟨i,j,v⟩

DENSE_MATRIX.hierarchy() ≡ $\langle (i \times j) \prec v$:Real ⟩

DENSE_MATRIX.bounds() ≡ $\langle\ istart \le i \le nrow \wedge jstart \le j \le ncol\ \rangle$

DENSE_MATRIX.h() ≡ ⟨

```
    (DENSE_MATRIX_SUB start=start colmajor=colmajor
        leading=leading nrow=nrow ncol=ncol
        istart=istart jstart=jstart
        rowname=i colname=j valname=v) ⟩
```

DENSE_MATRIX_SUB.fields() ≡ ⟨ rowname, colname ⟩

DENSE_MATRIX_SUB.special_kind$_{rowname}$() ≡ ⟨ Dense ⟩

DENSE_MATRIX_SUB.lb$_{rowname}$() ≡ Expr⟨ istart ⟩

DENSE_MATRIX_SUB.ub$_{rowname}$() ≡ Expr⟨ nrow ⟩

DENSE_MATRIX_SUB.special_kind$_{colname}$() ≡ ⟨ Dense ⟩

```
DENSE_MATRIX_SUB.lb_colname() ≡ Expr⟨ jstart ⟩
DENSE_MATRIX_SUB.ub_colname() ≡ Expr⟨ ncol ⟩
DENSE_MATRIX_SUB.hderef(i,j) ≡ Expr⟨
    (DENSE_MATRIX_ELT start=start colmajor=colmajor
        leading=leading
        istart=istart jstart=jstart
        i=i j=j name=valname) ⟩

DENSE_MATRIX_ELT.fields() ≡ ⟨ name ⟩
DENSE_MATRIX_ELT.deref() ≡Expr⟨
    if colmajor=true then this method returns:
        start[1+(j-jstart)*leading + (i-istart)]
    else:
        start[1+(i-istart)*leading + (j-jstart)]
⟩
DENSE_MATRIX_ELT.lvalue_p() ≡ ⟨ true ⟩
```

The DENSE_MATRIX_SUB ADT provides the method for the product term $i \times j$. Since this ADT is used in other Black Boxes (see Section 4.5.4 and Chapter 6) we pass it the names for its fields. The DENSE_MATRIX_ELT is the ADT for the element of a dense matrix.

The main motivation behind this Black Box is that the code fragment:

Program 4.2

```
real Y[...], X[...]
real buf[...]
i0 = ...; i1 = ...;
j0 = ...; j1 = ...;
abstract A(DENSE_MATRIX start=buf[address] leading=lda
        nrow=m ncol=n colmajor=true
        istart=i0 jstart=j0)
doany i=i0,i1
    doany j=j0,j1
        Y[i] := Y[i] + A[i0,j0]*X[j]
```

can be translated into the following call to the DGEMV routine in BLAS:

Program 4.3

```
call DGEMV('N',i1-i0+1,j1-j0+1,1.0,buf[address],
        lda,X[istart],1,1.0,Y[jstart],1)
```

(We use Fortran semantics for passing an address as an argument: A[x] is equivalent to &A[x] in C. The transposition flag 'N' is replaced by 'T' if colmajor = false.)

DGEMV computes the gaxpy operation:

$$\mathbf{Y} := \alpha\mathbf{Y} + \beta \cdot op(\mathbf{A}) \cdot \mathbf{X} \tag{4.22}$$

where $op()$ is either identity of transposition. If the layout of $\mathbf{A}$ is given by the parameters buf, m, n, lda and T (see above), then the calling sequence for DGEMV is:

```
DGEMV(T,m,n,α,buf[1],lda,X,strideX,β,Y,strideY)
```

`strideX` and `strideY` are the strides with which the elements of `X` and `Y` should be accessed (1 in our case).

4.5.4 ELEMENT

The hierarchy kind is $(i \times j) \prec v$:Real. The ADT for the product term is `ELEMENT1` and we use `DENSE_MATRIX_ELT` for the v^1 term:

Formal arguments: `n, nrow, ncol, colind, rowind, vals`

```
ELEMENT.schema() ≡ ⟨ i, j, v ⟩
ELEMENT.bounds() ≡ ⟨ (i × j) ≺ v:Real ⟩

ELEMENT1.h() ≡ Expr⟨ (ELEMENT1 n=n ....)  ⟩
ELEMENT1.fields() ≡ ⟨ i, j ⟩
ELEMENT1.special_kind_i() ≡ ⟨ Indirect ⟩
ELEMENT1.lb_i() ≡ Expr⟨ 1 ⟩
ELEMENT1.ub_i() ≡ Expr⟨ nrow ⟩
ELEMENT1.ideref_i() ≡ Expr⟨ rowind ⟩
ELEMENT1.sorted_p_i() ≡ ⟨ true ⟩
ELEMENT1.special_kind_j() ≡ ⟨ Indirect ⟩
ELEMENT1.lb_j() ≡ Expr⟨ 1 ⟩
ELEMENT1.ub_j() ≡ Expr⟨ ncol ⟩
ELEMENT1.ideref_j() ≡ Expr⟨ colind ⟩
ELEMENT1.sorted_p_j() ≡ ⟨ true ⟩
ELEMENT1.hderef(ii,jj) ≡ Expr⟨
    (DENSE_MATRIX_ELT start=vals[1] colmajor=colmajor
        leading=nrow
        istart=1 jstart=1
        i=ii j=jj name=v) ⟩
```

The pseudo-code:

```
abstract A(ELEMENT n=N ncol=NCOL nrow=NROW
        colind=COLIND rowind=ROWIND vals=VALS)
do ⟨i,j,v⟩ ∈ A
    print(i, j, v)
enddo
```

can be translated into the following program:

```
integer ii
integer jj
do ii = A.lb_1(), A.ub_1()
    do jj = A.lb_2(), A.ub_2()
        print ( (A.ideref_1())[ii],
```

```
                (A.ideref2())[jj],
                A.hderef(ii,jj).deref() )
    enddo
enddo
```

The concrete program is:

```
integer ii
integer jj
do ii = 1, NROW, 1
    do jj = 1, NCOL, 1
        print ( ROWIND[ii], COLIND[ii], VALS[ii + NROW*(jj-1) ] )
    enddo
enddo
```

4.5.5 Storage via run-length compression

In some cases it might be profitable to store the vector using run-length encoding, as illustrated in Figure 4.8. Each maximal sub-sequence of length l of repeated values w is represented as pair $\langle l, w \rangle$. N is the size of the

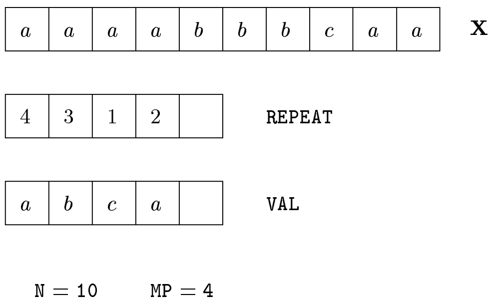


Figure 4.8: Run-length encoding of a vector

encoded vector (i.e. the range of vector indices). The array REPEAT stores the length l for each sub-sequence. The array VAL stores the values. The integer MP holds the number of the pairs. For simplicity, we assume that the arrays always have size MP + 1. The index hierarchy for the RL black box is $i^* \prec v$:Real – same as for the SPVO black box. However, this example illustrates a general iterator which is described by more than a single integer offset. We start by describing the iterator ADT. We call this ADT: RL_ITER. A position within the vector can be described by the following variables:

- The current index i
- The current offset ii. VAL[ii] is the value of the i-th element of the vector.
- The count m of how many times the current value has to be repeated.

Consequently, an instance of the iterator is described by the three formal arguments in addition to the variables that describe the storage itself. Overall , the ADT is:

Formal arguments: `i, ii, m, n, repeat, val, mp`

RL_ITER.deref$_i$() $\equiv$ Expr$\langle$ i $\rangle$

RL_ITER.deref_v() ≡ Expr⟨ val[ii] ⟩

```
RL_ITER.open() ≡ Stmt⟨
    i := 1; ii := 1;
    m := repeat[1]; ⟩
RL_ITER.valid() ≡ Expr⟨ 1 ≤ i ≤ n ⟩
RL_ITER.next() ≡ Stmt⟨
    i := i + 1
    if (m > 0) then m := m - 1
    else
        ii := ii + 1
        m := repeat[ii]
    endif ⟩
RL_ITER.size() ≡ ⟨ 3 ⟩
RL_ITER.marshal(b,k) ≡ Stmt⟨
    b[k] := i; b[k+1] := ii; b[k+2] := m; ⟩
RL_ITER.unmarshal(b,k) ≡ Stmt⟨
    i := b[k]; ii := b[k+1]; m := b[k+2]; ⟩
```

In a program the iterator can be used as:

```
integer MP, N
....
integer REPEAT[MP+1]
real VAL[MP+1]
integer j, jj, m
abstract h(RL_ITER i=j ii=jj m=m n=N repeat=REPEAT val=VAL mp=MP)
h.open()
while (h.valid()) do
    ...h.deref_i()...
    ...h.deref_v()...
enddo
```

Of course, the iterator is not intended to be used by itself, but rather as part of the RL Black Box:

Formal arguments: n, mp, repeat, val

RL.schema() ≡ ⟨ i, v ⟩

RL.hierarchy() ≡ ⟨ $i^* \prec v$:Real ⟩

RL.bounds() ≡ ⟨ $1 \le i \le n$ ⟩

RL.h() ≡ Expr⟨ (RL1 n=n mp=mp repeat=repeat val=val) ⟩

RL1.fields() ≡ ⟨ i ⟩

RL1.special_kind() ≡ ⟨ Stream ⟩

RL1.declare_iter(**h**) ≡ Decls⟨

```
    integer $i, $ii, $m    -- unique names are created --
    abstract h(RL_ITER i=$i ii=$ii m=$m
                n=N repeat=repeat val=val mp=mp) ⟩
RL1.open(h) ≡ Stmt⟨ h.open() ⟩
RL1.next(h) ≡ Stmt⟨ h.next() ⟩
RL1.valid(h) ≡ Expr⟨ h.valid() ⟩
RL1.deref(h) ≡ Expr⟨ h.deref_i() ⟩
RL1.hderef(h) ≡ Expr⟨
    (REAL_VAL name=v e=(h.deref_v())) ⟩
```

Chapter 5

Query optimization

As we have discussed in Section 2.3.2, execution of a loop nest can be modeled as enumeration of the solution to the corresponding relational query. Having described how data structures for tensors are represented in the compiler, we now introduce the query optimization algorithm.

Our query optimization algorithm is split into three phases: *query scheduling*, *join scheduling* and *instantiation*. We start by representing a conjunctive loop nest in *affine* Θ*-join form* (Section 5.1). The query scheduling process transforms the input loop nest in this form into a nesting of conditionals and loops that enumerate over *one-dimensional* affine joins. The transformation process is based distributive property of affine joins with respect to disjoint unions. We discuss the prerequisite algebraic manipulation of queries and join predicates in Sections 5.2 and 5.3. Query scheduling is then described in Section 5.4.

Join scheduling (Section 5.5) selects the best algorithms for performing the one-dimensional affine joins. The choice is based on the properties of the relations as expressed through the metrics in the Black Box Protocol. The result of join scheduling is an abstract program in terms of the ADTs that constitute the individual Black Boxes. Section 5.6 discusses the process of translating the abstract program into a concrete program. various optimizations are discussed in Section 5.8.

5.1 Affine Θ-join

Consider the loop nest in Figure 5.1. It computes the sum of the products of individual elements of the matrices $\mathbf{A}$ and $\mathbf{B}$. The term "$\mathbf{A}(\mathtt{i},\mathtt{j})$" predicate on line 2 is the shortcut for "$\langle i,j\rangle \in \text{Stored}(\mathbf{A})$". In this computation we are only interested in the elements of the matrices that are physically stored, since we assume all others to be zero. The product of zero elements do not contribute to the sum.

We view matrices as relations: $\mathbf{A}$ is the set of $\langle r_a, c_a, v_a\rangle$ tuples of row indices, column indices and values and $\mathbf{B}$ is the set of $\langle r_b, c_b, v_b\rangle$ tuples. Then the loop nest enumerates the indices of the loops and the vectors and the values of the vectors that satisfy the following relational query:

$$Q\left\langle \begin{array}{l} i,j,r_a,c_a,v_a, \\ r_b,c_b,v_b \end{array} \right\rangle = \sigma_{i=r_a=r_b \wedge j=c_a=c_b} \left(\begin{array}{l} \{\langle i,j\rangle \mid 1 \le i,j \le N\}\times \\ \times\mathbf{A}\langle r_a,c_a,v_a\rangle \times \mathbf{B}\langle r_b,c_b,v_b\rangle \end{array} \right) \tag{5.1}$$

In "plain English", this query selects those loop indices within loop bounds and matrix indices and values that match according to the matrix accesses in Figure 5.1: row indices r_a and r_b are set to i, column indices are set to j.

We can rewrite (5.1) more compactly by using the Θ-join operator. By definition (see [87]):

$$\bigominus_P (\ \mathbf{R},\mathbf{S}\ ) \stackrel{\text{def}}{\equiv} \sigma_P (\ \mathbf{R}\times\mathbf{S}\ ) \tag{5.2}$$

```
doany i = 1, N; j = 1, N
    if A(i,j) ∧ B(i,j) then
        sum := sum + A[i,j] * B[i,j]
```

Figure 5.1: An example loop nest

The query (5.1) becomes:

$$Q\left\langle \begin{array}{l} i, j, r_a, c_a, v_a, \\ r_b, c_b, v_b \end{array} \right\rangle = \bigominus_{i=r_a=r_b \wedge j=c_a=c_b} \left(\begin{array}{l} \{\langle i,j \rangle \mid 1 \le i,j \le N\}, \\ \mathbf{A}\langle r_a, c_a, v_a \rangle, \mathbf{B}\langle r_b, c_b, v_b \rangle \end{array} \right) \tag{5.3}$$

The join in (5.3) has the special form:

- One of the input relations is an integer set defined by affine inequalities: $\{\langle i,j \rangle \mid 1 \le i,j \le N\}$
- The join predicate expresses the integer fields: r_a, c_a, r_b, c_b – as an affine function of the values of the fields of the integer set.

We call such join an *affine* Θ*-join* or, simply, an *affine join.* In general an affine join has the form:

$$\bigominus_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\ \mathcal{B}\langle \mathbf{i} \rangle, \mathbf{R}_1(\mathbf{f}_1, \mathbf{v}_1), \ldots, \mathbf{R}_m(\mathbf{f}_m, \mathbf{v}_m) \ \right) \tag{5.4}$$

where $\mathbf{f}_k$, $1 \le k \le m$, is the vector of integer fields of the relation $\mathbf{R}_k$. The vector $\mathbf{f}$ is formed by concatenating all integer fields $\mathbf{f}_k$. $\mathbf{s}$ is the vector of *symbolic constants*, which are the integer parameters used to constraint the integer fields in the query. The matrices $\mathbf{F}$ and $\mathbf{G}$ and the vector $\mathbf{f}^{(0)}$ provide the affine function that maps $\mathbf{i}$ and $\mathbf{s}$ to $\mathbf{f}$. $\mathcal{B}$ represents affine inequalities in terms of $\mathbf{i}$. $\mathbf{v}_k$, $1 \le k \le m$, is the vector of non-integer fields of the relation $\mathbf{R}_k$.

By analogy with [55], we call the predicate $\mathbf{f} = \mathbf{F}\mathbf{i} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)}$ the *data access equation* and the matrix $\mathbf{F}$ the *data access matrix.* In our example the equation is:

$$\begin{pmatrix} r_a \\ c_a \\ r_b \\ c_b \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.5}$$

In general, an affine join describes the set of loop and tensor indices and tensor values enumerated in a conjunctive "DOANY" loop nest with affine tensor access functions and affine loop bounds. More precisely, consider the loop nest:

```
doany i ∈ B
    if (A1(φ1(i,s)) ∧ ... ∧ Am(φm(i,s))) then
        S(s,i,A1[φ1(i,s)], ...,Am[φm(i,s)])
```

where S is a statement that uses loop indices $\mathbf{i}$, symbolic constants $\mathbf{s}$ and tensor elements $\mathbf{A}_k[\phi_k(\mathbf{i},\mathbf{s})]$, $1 \le k \le m$. We assume that the tensor access functions ϕ_k are affine functions of loop indices and symbolic constants. $\mathcal{B}$ represents loop bounds and is assumed to be a conjunction of affine inequalities in terms of $\mathbf{i}$ and $\mathbf{s}$. The term $\mathbf{A}_k(\phi_k(\mathbf{i},\mathbf{s}))$, $1 \le k \le m$, is the shortcut for $\langle \phi_k(\mathbf{i},\mathbf{s}) \rangle \in \text{Stored}(\mathbf{A})$. We call the loop nest *conjunctive* because the predicate in the conditional is the conjunction of such terms – one for each tensor access.

```
doany ⟨ i, f_1, v_1, ..., f_m, v_m ⟩ ∈ Θ_{f=Fi+Gs+f^(0)} ( B⟨i⟩, A_1⟨f_1, v_1⟩, ...
                                                        ..., A_m⟨f_m, v_m⟩ )
    S(s,i,v_1,...,v_m)
```

Figure 5.2: Affine Θ-join form of a loop nest

Each array access function ϕ_k is of the form:

$$\mathbf{f}_k = \mathbf{F}_k\mathbf{i} + \mathbf{G}_k\mathbf{s} + \mathbf{f}_k{}^{(0)} \tag{5.6}$$

We obtain the data access equation by concatenating the subscripted matrices and vectors in (5.6):

$$\mathbf{f} = \mathbf{F}\mathbf{i} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)} \tag{5.7}$$

The corresponding affine join for the loop nest is:

$$\boxed{\bigominus_{\mathbf{f}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{f}^{(0)}} \left(\mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}_1\langle\mathbf{f}_1, v_1\rangle, \ldots, \mathbf{A}_m\langle\mathbf{f}_m, v_m\rangle \right)} \tag{5.8}$$

where v_k is the value field in the relation $\mathbf{A}_k$. We can rewrite the loop nest in the *affine Θ-join form* as shown in Figure 5.2. Notice that the values fields v_k replace the array references $\mathbf{A}[\phi_k(\mathbf{i},\mathbf{s})]$ in the statement S. Our goal is to transform such loop nest into efficient executable code.

5.2 Algebraic rules for affine joins and unions

The loop nest in Figure 5.2 is a *declarative* specification of the computation: it specifies *what* set of indices and values have to be enumerated. We now have to translate it into a form that specifies *how* to enumerate the set. Traditional database systems solve this problem by (a) assigning the default executable meaning to each relational operator and (b) transforming the original query into an equivalent but more efficient form. The transformation is based on various identities of relational algebra (see [87], vol. 2, chapter 11).

The structure of the input relations and the presence affine joins force us to develop new algebraic transformations. We need to account for two factors:

- Black-box protocol expresses the relations as nested disjoint unions. We would like to exploit this structure in the optimization process.
- Queries in traditional relational database environments usually involve equi-joins. As a consequence, we are not aware of any work in database literature that deals with more general, affine, joins.

In our case the input queries are always affine joins. The transformed queries use combinations of affine joins and disjoint union. We start by defining the meaning for the affine join and disjoint union operators in the context of execution of loop nests. Suppose we have a loop nest:

```
doany ⟨f, v⟩ ∈ Q
    S(f,v)
```

If the query Q is an affine join (5.3), then the loop can always be executed by literally using the definition (5.2) of the affine join operator:

```
doany ⟨f_1, v_1⟩ ∈ A_1
```

```
...
    doany ⟨f_m, v_m⟩ ∈ A_m
        doany ⟨i ∈ B'(f)⟩
            S
```

where $\mathcal{B}'$ is the set of loop iterations $\mathbf{i}$ that are within the bounds $\mathcal{B}$ and make the current values of the fields $\mathbf{f}_j$ satisfy the data access equation:

$$\mathcal{B}'(\mathbf{f}) = \{\mathbf{i} \mid \mathbf{i} \in \mathcal{B} \wedge \mathbf{f} = \mathbf{Fi} + \mathbf{Gs} + \mathbf{f}^{(0)}\} \tag{5.9}$$

The enumeration of $\mathbf{i}$ over $\mathcal{B}'$ can be generated using well known linear algebraic methods (see Appendix A). The enumerations over the individual relations can be generated from the black-box protocol as discussed in Section 5.5. Assuming that the enumerations do not output duplicate fields, this loop nest performs the same computation as the original loop.

An important case of the affine join operator is *one-dimensional* affine join. It has the same form as (5.3) with the exception that the tuples of integer fields: $\mathbf{i}$, $\mathbf{f}_1$, ..., $\mathbf{f}_m$ – have only one component each. For example:

$$\bigominus_{i_y=2i+5 \wedge i_x=i-10} \left(\ \{1 \le i \le N\}, \mathbf{X}\langle i_x, v_x\rangle, \mathbf{Y}\langle i_y, v_y\rangle \ \right) \tag{5.10}$$

One-dimensional affine join is a generalization of equi-join: instead of simple equalities we are dealing with pairwise affine equalities between the fields. In (5.10) we can eliminate i to deduce the affine equality between i_x and i_y: $2i_x = -i_y - 15$. Similarly, the algorithms for computing equi-joins can be generalized to the case of 1D affine joins. Therefore, one of the goals of the optimization process is to decompose the multi-dimensional affine joins into several one-dimensional joins.

The decomposition is aided by the representation of the relations in the black-box protocol as nested disjoint unions. Operationally, the unions correspond to loop nests. Suppose the query Q has the form:

$$Q\langle \mathbf{f}, \mathbf{v}\rangle = \biguplus_{\langle \mathbf{f}', \mathbf{h}\rangle \in R} \{\mathbf{f}'\} \times Q_{\mathbf{h}}\langle \mathbf{f}'', \mathbf{v}\rangle \tag{5.11}$$

where $\mathbf{f}'$ and $\mathbf{f}''$ partition the set of fields $\mathbf{f}$ and R is just another query. Then we can safely execute the original loop as:

```
doany ⟨f', h⟩ ∈ R
    doany ⟨f'', v⟩ ∈ Q_h
        S
```

Given the above semantics of the multi-dimensional affine join operator, in most cases the direct execution of the loop in Figure 5.2 would be rather inefficient. However, we can use the distributive property of the affine join operator over the union operator and transform the original query into a nesting of unions. First, a couple of basic facts from relational algebra (see [87] for proofs).

Theorem 5.1 *Selection can be distributed over set union:*
$\sigma_P(\mathbf{R} \cup \mathbf{S}) = \sigma_P\mathbf{R} \cup \sigma_P\mathbf{S}$

Theorem 5.2 *Cross product can be distributed over set union:*
$(\mathbf{R} \cup \mathbf{S}) \times \mathbf{T} = (\mathbf{R} \times \mathbf{T}) \cup (\mathbf{S} \times \mathbf{T})$

Corollary 5.3 Θ*-join can be distributed over set union:*
$\bigominus_P(\mathbf{R} \cup \mathbf{S}, \mathbf{T}) = \bigominus_P(\mathbf{R}, \mathbf{T}) \cup \bigominus_P(\mathbf{S}, \mathbf{T})$

Proof: Follows from Theorems 5.1 and 5.2:

$$\begin{aligned}
\ominus_P(\mathbf{R}\cup\mathbf{S},\mathbf{T}) &= \sigma_P((\mathbf{R}\cup\mathbf{S})\times\mathbf{T}) \\
&= \sigma_P((\mathbf{R}\times\mathbf{T})\cup(\mathbf{S}\times\mathbf{T})) \\
&= \sigma_P(\mathbf{R}\times\mathbf{T})\cup\sigma_P(\mathbf{S}\times\mathbf{T}) \\
&= \ominus_P(\mathbf{R},\mathbf{T})\cup\ominus_P(\mathbf{S},\mathbf{T})
\end{aligned} \tag{5.12}$$

■

Lets return to our example in Figure 5.1. Suppose that the matrices **A** and **B** are stored using CCCS format. The CCCS black box represents the matrices as disjoint unions:

$$\begin{aligned}
\mathbf{A}\langle r_a, c_a, v_a\rangle &= \biguplus_{\substack{\langle c_a,\mathbf{h}\rangle\in\\ \mathbf{A}.\mathtt{enum}()}} \left(\ \{c_a\}\times\mathbf{A}.\mathtt{hderef}(\mathbf{h})\langle r_a, v_a\rangle\ \right) \\
\mathbf{B}\langle r_b, c_b, v_b\rangle &= \biguplus_{\substack{\langle c_b,\mathbf{g}\rangle\in\\ \mathbf{B}.\mathtt{enum}()}} \left(\ \{c_b\}\times\mathbf{B}.\mathtt{hderef}(\mathbf{g})\langle r_b, v_b\rangle\ \right)
\end{aligned} \tag{5.13}$$

We can also split the iteration space as:

$$\{\langle i,j\rangle \mid 1\le i,j\le N\} = \{\langle i\rangle \mid 1\le i\le N\}\times\{\langle j\rangle \mid 1\le j\le N\} \tag{5.14}$$

Using (5.13) and (5.14) we rewrite the query in (5.3) as:

$$Q\langle i,j,r_a,c_a,v_a,r_b,c_b,v_b\rangle =$$

$$\begin{aligned}
&= \ominus_{\substack{i=r_a=r_b\wedge\\ j=c_a=c_b}} \left(\begin{array}{l} \{\langle i,j\rangle \mid 1\le i,j\le N\}, \\ \mathbf{A}\langle r_a,c_a,v_a\rangle, \mathbf{B}\langle r_b,c_b,v_b\rangle \end{array}\right) \\
&= \biguplus_{\substack{\langle c_a,\mathbf{h}\rangle\in\\ \mathbf{A}.\mathtt{enum}()}} \biguplus_{\substack{\langle c_b,\mathbf{g}\rangle\in\\ \mathbf{B}.\mathtt{enum}()}} \biguplus_{\langle j\rangle\in 1..N} \ominus_{\substack{i=r_a=r_b\wedge\\ j=c_a=c_b}} \left(\begin{array}{l} \{\langle c_a,c_b,j\rangle\}, \{\langle i\rangle \mid 1\le i\le N\}, \\ \mathbf{A}.\mathtt{hderef}(\mathbf{h})\langle r_a,v_a\rangle, \\ \mathbf{B}.\mathtt{hderef}(\mathbf{g})\langle r_b,v_b\rangle, \end{array}\right)
\end{aligned} \tag{5.15}$$

The column indices c_a, c_b and the loop index j are related by the equalities $c_a = c_b = j$. Instead of taking all possible combinations of the indices in the nested unions, we are interested in a much more restricted set, which is a one dimensional affine-join:

$$R\langle c_a,\mathbf{h},c_b,\mathbf{g},j\rangle = \ominus_{j=c_a=c_b} \left(\begin{array}{l} \{j \mid 1\le j\le N\}, \mathbf{A}.\mathtt{enum}()\langle c_a,\mathbf{h}\rangle, \\ \mathbf{B}.\mathtt{enum}()\langle c_b,\mathbf{g}\rangle \end{array}\right) \tag{5.16}$$

We can now rewrite the query in (5.15) as:

$$\biguplus_{\langle c_a,\mathbf{h},c_b,\mathbf{g},j\rangle\in R} \ominus_{i=r_a=r_b} \left(\begin{array}{l} \{\langle c_a,c_b,j\rangle\}, \{\langle i\rangle \mid 1\le i\le N\}, \\ \mathbf{A}.\mathtt{hderef}(\mathbf{h})\langle r_a,v_a\rangle, \\ \mathbf{B}.\mathtt{hderef}(\mathbf{g})\langle r_b,v_b\rangle \end{array}\right) \tag{5.17}$$

Observe that this query has the same form as (5.11). The statement:

`doany` $\langle r_a, c_a, v_a, r_b, c_b, v_b, i, j\rangle \in Q$

    sum := sum + v_a * v_b

can be transformed into:

$$
\begin{array}{l}
\texttt{doany}\ \langle c_a, \mathbf{h}, c_b, \mathbf{g}, j\rangle \in R \\
\qquad \texttt{doany}\ \langle r_a, v_a, r_b, v_b, i\rangle \in \bigominus_{i=r_a=r_b} \left(\begin{array}{l} \{\langle c_a, c_b, j\rangle\}, \{\langle i\rangle \mid 1 \le i \le N\}, \\ \mathbf{A}.\texttt{hderef}(\mathbf{h})\langle r_a, v_a\rangle, \\ \mathbf{B}.\texttt{hderef}(\mathbf{g})\langle r_b, v_b\rangle \end{array} \right) \\
\qquad\qquad \texttt{sum := sum + v}_a \texttt{ * v}_b
\end{array}
$$

This example illustrates the main algebraic transformation used in our query optimization algorithm. We start with a query:

$$Q = \bigominus_{P(\mathbf{r},\mathbf{s})} \left(\ \mathbf{R}\langle\mathbf{r}\rangle, \mathbf{S}\langle\mathbf{s}\rangle \ \right) \tag{5.18}$$

where $\mathbf{R}$ and $\mathbf{S}$ are relation with the schemata $\mathbf{r}$ and $\mathbf{s}$, respectively. $\mathbf{P}$ is a predicate in terms of $\mathbf{r}$ and $\mathbf{s}$. Suppose that the relations can be decomposed into the following disjoint unions:

$$
\begin{array}{rcl}
\mathbf{R} & = & \biguplus_{\langle r,\mathbf{h}\rangle \in \mathbf{R}'} \{r\} \times \mathbf{R_h}\langle\mathbf{r}'\rangle \\
\mathbf{S} & = & \biguplus_{\langle s,\mathbf{g}\rangle \in \mathbf{S}'} \{s\} \times \mathbf{S_g}\langle\mathbf{s}'\rangle
\end{array} \tag{5.19}
$$

where r and s are some fields of the relations, $\mathbf{r}' = \mathbf{r} \setminus \{r\}$ and $\mathbf{s}' = \mathbf{s} \setminus \{s\}$. $\mathbf{h}$ and $\mathbf{g}$ are some iterator objects that allows us to dereference the "sub-relations" $\mathbf{R_h}$ and $\mathbf{S_g}$. Furthermore, suppose that we can rewrite the predicate P in the *echelon form* as:

$$P(\mathbf{r}, \mathbf{s}) = P_1(r, s) \wedge P_2(r, s, \mathbf{r}', \mathbf{s}') \tag{5.20}$$

Intuitively, P_1 is a "projection" of the information in P onto the fields r and s, and P_2 represents the constraints on $\mathbf{r}'$ and $\mathbf{s}'$ for fixed values of r and s. We will discuss in Section 5.3 how P_1 and P_2 are computed.

We are now ready to prove the following theorem:

Theorem 5.4 *Given the conditions* (5.19) *and* (5.20)*, the query* (5.18) *computes the same result as the query:*

$$Q' = \biguplus_{\substack{\langle r,s,\mathbf{h},\mathbf{g}\rangle \in \\ \Theta_{P_1(r,s)}\left(\ \mathbf{R}', \mathbf{S}' \ \right)}} \bigominus_{P_2(r,s,\mathbf{r}',\mathbf{s}')} \left(\begin{array}{l} \{\langle r, s\rangle\}, \mathbf{R_h}\langle\mathbf{r}'\rangle, \\ \mathbf{S_g}\langle\mathbf{s}'\rangle \end{array} \right) \tag{5.21}$$

Proof: Using the definition of the Θ-join, the decomposition (5.19) of $\mathbf{R}$ and $\mathbf{S}$ and the splitting of P:

$$
\begin{array}{c}
\langle \mathbf{r}, \mathbf{s}\rangle \in Q \\
\Updownarrow \\
\mathbf{r} \in \mathbf{R} \wedge \mathbf{s} \in \mathbf{S} \wedge P(\mathbf{r}, \mathbf{s}) \\
\Updownarrow \\
\exists \mathbf{h}, \mathbf{g} : \left\{ \begin{array}{l} \langle r, \mathbf{h}\rangle \in \mathbf{R}' \wedge \mathbf{r}' \in \mathbf{R_h} \wedge \\ \langle s, \mathbf{g}\rangle \in \mathbf{S}' \wedge \mathbf{s}' \in \mathbf{S_g} \wedge \\ P_1(r, s) \wedge P_2(r, s, \mathbf{r}', \mathbf{s}') \end{array} \right. \\
\Updownarrow \\
\langle r, s, \mathbf{r}', \mathbf{s}'\rangle \in Q'
\end{array}
$$

which proves that $Q = Q'$. ∎

Theorem 5.4 can be trivially generalized to joins of more than two relations.

5.3 Manipulating the data access equation

We have yet to explain not how the predicate P is split into $P_1 \wedge P_2$. In our example $P \stackrel{\text{def}}{\equiv} i = r_a = r_b \wedge j = c_a = c_b$, which can be split by defining $P_1 \stackrel{\text{def}}{\equiv} j = c_a = c_b$ and $P_2 \stackrel{\text{def}}{\equiv} i = r_a = r_b$. In general it might not be possible to syntactically separate the fields constrained in P_1 and P_2. However, in our application the predicates P and P_1 are of special form. Recall the affine Θ-join form of a loop nest in Figure 5.2. The predicate P is the data access equation $\mathbf{f} = \mathbf{F}\mathbf{i} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)}$, where $\mathbf{i}$ is the vector of loop indices and $\mathbf{s}$ is the vector of invariants.

We can manipulate the data access equation in order to discover the fields that are related by affine equalities. First, lets prove the following fact, that will allows us to discover affine equalities between the fields of the relations:

Theorem 5.5 *Consider two affine functions $x : \mathbb{Z}^n \mapsto \mathbb{Z}$ and $y : \mathbb{Z}^n \mapsto \mathbb{Z}$ defined as: $x(\mathbf{i}) = \mathbf{a}^T\mathbf{i} + x_0$ and $y(\mathbf{i}) = \mathbf{b}^T\mathbf{i} + y_0$. Then the following holds:*

$$\exists \alpha, \beta, \gamma \in \mathbb{Z} : \forall \mathbf{i} \in \mathbb{Z}^n : \alpha x(\mathbf{i}) + \beta y(\mathbf{i}) = \gamma \quad \Leftrightarrow \quad \exists \alpha, \beta \in \mathbb{Z} : \alpha\mathbf{a} + \beta\mathbf{b} = \mathbf{0} \tag{5.22}$$

In "plain English", Theorem 5.5 states that two affine functions are related by an affine equality if and only if their linear coefficients are multiples of one another.

Proof: The "$\Leftarrow$" implication is trivial. The value of γ can be computed as $\alpha x_0 + \beta y_0$. To prove the "$\Rightarrow$" implication, we exploit the universal quantification of $\mathbf{i}$. Let's set $\mathbf{i}$ to zero. By the definition of x and y, the constraint $\alpha x(\mathbf{i}) + \beta y(\mathbf{i}) = \gamma$ implies that $\alpha x_0 + \beta y_0 = \gamma$. Now for all $\mathbf{i} \in \mathbb{Z}^n$:

$$\gamma \quad = \quad \alpha x(\mathbf{i}) + \beta y(\mathbf{i}) \quad = \quad (\alpha\mathbf{a}^T\mathbf{i} + \beta\mathbf{b}^T\mathbf{i}) + (\alpha x_0 + \beta y_0) \quad = \quad (\alpha\mathbf{a}^T\mathbf{i} + \beta\mathbf{b}^T\mathbf{i}) + \gamma$$

which implies that $\alpha\mathbf{a}^T\mathbf{i} + \beta\mathbf{b}^T\mathbf{i} = \mathbf{0}$, for all $\mathbf{i} \in \mathbb{Z}^n$. This is true if and only if $\alpha\mathbf{a} + \beta\mathbf{b} = \mathbf{0}$. ■

We can use Theorem 5.5 to discover affine equalities between the fields of the relations in the query. The fields are related to the loop indices $\mathbf{i}$ and invariants $\mathbf{s}$ by the data access equation $\mathbf{f} = \mathbf{F}\mathbf{i} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)}$. We are interested in situations where for any fixed value of $\mathbf{s}$ there exists coefficients α, β and γ that for all values of loop indices $\mathbf{i}$ relate a pair of elements of f_1 and f_2 of $\mathbf{f}$ via an affine equality $\alpha f_1 + \beta f_2 = \gamma$. More formally, we are looking for a necessary and sufficient condition for the following to hold:

$$\forall \mathbf{s} : \exists \alpha, \beta, \gamma : \forall \mathbf{i} : \alpha f_1(\mathbf{s}, \mathbf{i}) + \beta f_2(\mathbf{s}, \mathbf{i}) = \gamma \tag{5.23}$$

The order of the quantifiers implies that, in general, the values of α, β and γ depend on $\mathbf{s}$. However, it turns out that only γ has to depend on $\mathbf{s}$.

Without loss of generality assume that f_1 and f_2 are the first two elements in $\mathbf{f}$. Let $\mathbf{F}_1$, $\mathbf{F}_2$, $\mathbf{G}_1$ and $\mathbf{G}_2$ be the first two rows of $\mathbf{F}$ and $\mathbf{G}$. Let $f_1{}^{(0)}$ and $f_2{}^{(0)}$ be the first two elements of $\mathbf{f}^{(0)}$. Then we can express f_1 and f_2 as:

$$\begin{array}{rcl} f_1 & = & \mathbf{F}_1\mathbf{i} + \underline{\mathbf{G}_1\mathbf{s} + f_1{}^{(0)}} \\ f_2 & = & \mathbf{F}_2\mathbf{i} + \underline{\mathbf{G}_2\mathbf{s} + f_2{}^{(0)}} \end{array} \tag{5.24}$$

For a fixed value of $\mathbf{s}$ the underlined expressions are constant. It follows from Theorem 5.5 that (5.23) holds if and only if $\mathbf{F}_1$ and $\mathbf{F}_2$ are multiples of one another:

$$\alpha\mathbf{F}_1 + \beta\mathbf{F}_2 = \mathbf{0} \tag{5.25}$$

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigodot_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i}\rangle, \mathbf{A}_1\langle \mathbf{f}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$$\quad \texttt{S}(\mathbf{s}, \mathbf{i}, \texttt{v}_1, \ldots, \texttt{v}_m)$$

Figure 5.3: Source loop nest

This result can be trivially extended to the case of more than two fields being related by an pairwise affine equalities:

Corollary 5.6 *Let $\mathbf{f} = \mathbf{F}\mathbf{i} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)}$ be a data access equation. Let $\mathbf{f}'$ be some subset of the elements of $\mathbf{f}$. For any fixed value of $\mathbf{s}$ the elements of $\mathbf{f}'$ are related by pairwise affine equalities if and only if the corresponding rows of $\mathbf{F}$ are all multiples of one another.*

Now suppose that we have discovered such subset of the rows of $\mathbf{F}$. Lets permute the data access equation so that these rows form a block at the top of $\mathbf{F}$:

$$\begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \end{pmatrix} \mathbf{i} + \begin{pmatrix} \mathbf{G}_1 \\ \mathbf{G}_2 \end{pmatrix} \mathbf{s} + \begin{pmatrix} \mathbf{f}_1{}^{(0)} \\ \mathbf{f}_2{}^{(0)} \end{pmatrix} \tag{5.26}$$

Now $\mathbf{f}_1$ are the fields that are related pairwise by affine equalities. Because all the rows in the block $\mathbf{F}_1$ are multiples of one another, we can apply unimodular column operations and transform it into:

$$\mathbf{F}_1' = \mathbf{F}_1\mathbf{U} = \begin{pmatrix} \mathbf{z} & \mathbf{0} \end{pmatrix} \tag{5.27}$$

where $\mathbf{z}$ is the first column of $\mathbf{F}_1'$ and $\mathbf{U}$ is the matrix of the column transformations. Let $\mathbf{u} = \mathbf{U}^{-1}\mathbf{i}$. Let u_1 be the first element of $\mathbf{u}$ and $\mathbf{u}'$ be the rest of $\mathbf{u}$. Let $\mathbf{w}$ be the first column of $\mathbf{F}_2$ and $\mathbf{F}'$ be the rest of the columns of $\mathbf{F}_2$. Then we can rewrite (5.26) as:

$$\begin{array}{lclll} \mathbf{f}_1 & = & \mathbf{z}u_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}_1{}^{(0)} \\ \mathbf{f}_2 & = & \mathbf{w}u_1 & +\mathbf{F}'\mathbf{u}' & +\mathbf{G}_2\mathbf{s} + \mathbf{f}_2{}^{(0)} \end{array} \tag{5.28}$$

We can use (5.28) to split the data access equation. The original predicate P is given by (5.26). The split predicate is given by:

$$\begin{array}{lcl} P_1(\mathbf{f}_1, u_1) & \stackrel{\text{def}}{\equiv} & \mathbf{f}_1 = \mathbf{z}u_1 + \mathbf{G}_1\mathbf{s} + \mathbf{f}_1{}^{(0)} \\ P_2(u_1, \mathbf{u}', \mathbf{f}_2) & \stackrel{\text{def}}{\equiv} & \mathbf{f}_2 = \mathbf{w}u_1 + \mathbf{F}'\mathbf{u}' + \mathbf{G}_2\mathbf{s} + \mathbf{f}_2{}^{(0)} \end{array} \tag{5.29}$$

We can interpret the vector $\mathbf{u}$ are the new iteration space and the matrix $\mathbf{U}^{-1}$ as the transformation from the old iteration space $\mathbf{i}$. Then P_1 gives us an affine join between the fields $\mathbf{f}_1$ and the first index u_1 of the new iteration space. For a fixed value of u_1, P_2 relates all the other fields $\mathbf{f}_2$ and the rest of the new loop indices $\mathbf{u}'$.

5.4 Query scheduling

We are now ready to put together the results of Sections 5.2 and 5.3 and describe the rules used to transform the queries. We start with the affine Θ-join form of a loop nest as shown in Figure 5.3. This loop nest is translated into a nesting of affine joins and conditionals by repeatedly applying one of the following rules: PICK-HIERARCHY, JOIN, SEARCH, TUPLE-ENUM, SINGLETON, VALUE, VALUE-UPDATE. It turns out that more than one rule is applicable to a loop nest and heuristics have to be used to determine the order in which the rules are applied.

After a few definitions in Section 5.4.1, we describe the transformation rules in Section 5.4.2. The heuristics are described in Section 5.4.4.

5.4.1 Preliminaries

Definition 5.1 *An integer field $f \in \mathbf{f}$ is* **determined** *if the data access equation binds it to a single value for any combination of the values of the invariants* $\mathbf{s}$.

Observe that a field f is determined if and only if the corresponding row of the data access matrix $\mathbf{F}$ is all zero.

Definition 5.2 *Let g be an integer field of the relation $\mathbf{R}$. Let $\mathcal{K} = \oplus_j \mathcal{H}_j$ be the hierarchy kind for $\mathbf{R}$. Let $\mathcal{T}_j$ be the first (top) term in the hierarchy $\mathcal{H}_j$. We say that the field g is* **exposed** *if it is contained in one of the top terms $\mathcal{T}_j$ and that term is either a tuple stream term containing the field g only or a product term containing g.*

Observe that only an integer field can be determined or exposed. For example, suppose we have a code fragment:

N := ...

doany $\langle i, j, v, k, l, w\rangle \in \bigominus_{k=N \wedge j=k} \left(\mathbf{A}(i,j,v), \mathbf{B}(k,l,w) \right)$

...

enddo

where the relation $\mathbf{A}$ has index hierarchy kind $(i \prec j \prec v) \oplus ((i \times j) \prec v)$. The field j is determined, because it is bound to the invariant N by the data access equation $k = N \wedge j = k$. Both fields i and j are exposed. The field v is not exposed. Suppose that $\mathbf{B}$ has the hierarchy kind $\langle k, l\rangle^* \prec w$. Then, according to the Definition 5.2, neither k nor l is exposed.

5.4.2 The rules

Notation

For the clarity in presenting the rewrite rules we need a bit of notation that will unify the treatment of product terms and tuple stream terms in index hierarchies. Take a relation $\mathbf{A}$ with the fields $\mathbf{f}$. Let $\mathcal{T}$ be the top term in the index hierarchy of $\mathbf{A}$, There are two cases that we are most concerned with:

- $\mathcal{T}$ is a stream tuple term with a single field: g^*. Let $\mathbf{h}$ be a valid iterator for the term. We will use "$\mathbf{A}_{g:\mathbf{h}}$" instead of "$\mathbf{A}.\mathtt{hderef}(\mathbf{h})$". Moreover, suppose that the first two terms in the hierarchy are $g \prec j$. Let $\mathbf{h}$ be the iterator, as before. And let $\mathbf{k}$ be a valid iterator for the j term. Then we will replace "$\mathbf{A}.\mathtt{hderef}(\mathbf{h}).\mathtt{hderef}(\mathbf{k})$" with "$\mathbf{A}_{g:\mathbf{h},j:\mathbf{k}}$". And so on.

  Additionally, we will subscript the enumeration and search methods for g (e.g. `enum`, `search`, `next`) with "g".

- $\mathcal{T}$ is a product tuple term: $g_1 \times g_2 \times \dots g_q$. In this case $\mathbf{A}$ can be decomposed into the nested union:

$$\mathbf{A} = \biguplus_{\substack{\langle g_1, \mathbf{h}_1\rangle \in \\ \mathbf{A}.\mathtt{enum}_{g_1}()}} \cdots \biguplus_{\substack{\langle g_q, \mathbf{h}_q\rangle \in \\ \mathbf{A}.\mathtt{enum}_{g_q}()}} \{\langle g_1, \dots, g_q\rangle\} \times \mathbf{A}.\mathtt{hderef}(\mathbf{h}_1, \dots, \mathbf{h}_q) \tag{5.30}$$

The operators in this expression can be freely permuted. Let's divide the fields g_i into two sets: $g'_1, \ldots, g'_s$ and $g''_1, \ldots, g''_t$ $(q = s + t)$. Let $\mathbf{h}'_1, \ldots, \mathbf{h}'_s$ and $\mathbf{h}''_1, \ldots, \mathbf{h}''_t$ be the associated iterators. Then we can rearrange the unions in (5.30) as:

$$\mathbf{A} = \biguplus_{\substack{\langle g'_1, \mathbf{h}'_1 \rangle \in \\ \mathbf{A}.\mathtt{enum}_{g'_1}()}} \cdots \biguplus_{\substack{\langle g'_s, \mathbf{h}'_s \rangle \in \\ \mathbf{A}.\mathtt{enum}_{g'_s}()}} \{\langle g'_1, \ldots, g'_s \rangle\} \times \mathbf{A}'(\mathbf{h}'_1, \ldots, \mathbf{h}'_s) \tag{5.31}$$

where the relation **A**' is parametrized by the first set $\mathbf{h}'_1, \ldots, \mathbf{h}'_s$ of the iterators:

$$\mathbf{A}' = \biguplus_{\substack{\langle g''_1, \mathbf{h}''_1 \rangle \in \\ \mathbf{A}.\mathtt{enum}_{g''_1}()}} \cdots \biguplus_{\substack{\langle g''_t, \mathbf{h}''_t \rangle \in \\ \mathbf{A}.\mathtt{enum}_{g''_t}()}} \{\langle g''_1, \ldots, g''_t \rangle\} \times \quad \mathbf{A}.\mathtt{hderef}(\mathbf{h}_1, \ldots, \mathbf{h}_q) \tag{5.32}$$

(Notice that the iterators $\mathbf{h}_1, \ldots, \mathbf{h}_q$ are just some permutation of the $\mathbf{h}'_1, \ldots, \mathbf{h}'_s$ and $\mathbf{h}''_1, \ldots, \mathbf{h}''_t$ iterators.) We will use the notation:

$$\mathbf{A}_{g'_1:\mathbf{h}'_1, g'_2:\mathbf{h}'_2, \ldots, g'_s:\mathbf{h}'_s}$$

to denote the relation $\mathbf{A}'$ in (5.32). This relation has the schema $\mathbf{f}' = \mathbf{f} \setminus \{g'_1, \ldots, g'_s\}$.

The above notation allows us to "peel off" union operators in a uniform manner for both tuple and product terms:

$$\mathbf{A}\langle \mathbf{f} \rangle = \biguplus_{\langle g, \mathbf{h} \rangle \in \mathbf{A}.\mathtt{enum}_g()} \{g\} \times \mathbf{A}_{g:\mathbf{h}}\langle \mathbf{f}' \rangle \tag{5.33}$$

PICK-HIERARCHY

Suppose that the hierarchy kind for the relation $\mathbf{A}_j$ in the loop nest is the union: $\oplus_k \mathcal{H}_k$. Subsequently, the ADT for the relation provides the access methods $\mathtt{h}_k()$ that return an ADT expression for each hierarchy $\mathcal{H}_k$. At this point we can replace the occurrence of $\mathbf{A}_j$ in the query with the method invocation $\mathbf{A}_j.\mathtt{h}_\mathtt{k}()$ for some k. We apply the PICK-HIERARCHY first in order to select a definite hierarchy for each relation.

JOIN

The JOIN rule converts the query into a nesting of a loop nest that enumerates over the result of a one-dimensional affine joins and a loop nest enumerating over another query. This rule is applicable if there exist the fields $a_1, a_2, \ldots, a_k$ $(1 \le k \le m)$ such that:

1. These fields are all exposed
2. They are all related pairwise by affine equalities

The first condition is established by inspecting the hierarchy kinds of the relations in the query. The second condition is established by determining in the data access equation the equivalence classes of rows of **F** that are multiples of one another. We say that the fields that satisfy these conditions are *joinable*.

Let $\mathbf{A}_q$, $1 \le q \le k$, be the relation which stores the field a_q. Notice that these relations are not necessarily distinct: a product term exposes several fields at once. By definition of the black box protocol, the relation $\mathbf{A}_q$ provides the method $\mathbf{A}_q.\mathtt{enum}_{a_q}()$. It returns a set of $\langle a_q, \mathbf{h}_q \rangle$ pairs, where $\mathbf{h}_q$ is the value of the iterator used to dereference the "sub-relation" $(\mathbf{A}_q)_{a_q:\mathbf{h}_q}$.

Let $\mathbf{f}' = (a_1\ a_2\ \ldots\ a_k)^T$ and $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. Since the fields a_q are joinable, we can transform the loop indices and express the fields using (5.28):

$$
\begin{array}{rcllll}
\mathbf{f}' & = & \mathbf{z}u_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}_1{}^{(0)} \\
\mathbf{f}'' & = & \mathbf{w}u_1 & +\mathbf{F}'\mathbf{u}' & +\mathbf{G}_2\mathbf{s} + \mathbf{f}_2{}^{(0)}
\end{array}
\tag{5.34}
$$

We are at first tempted to use Theorem 5.4 and simply rewrite the *query* in Figure 5.3 as:

$$
Q' = \biguplus_{\substack{\langle u_1, a_1, \mathbf{h}_1, \ldots,\\ a_k, \mathbf{h}_k\rangle \in R}} \mathop{\Large\ominus}_{\substack{\mathbf{f}''=\mathbf{w}u_1+\\ \mathbf{F}'\mathbf{u}'+\mathbf{G}_2\mathbf{s}+\mathbf{f}_2{}^{(0)}}} \left(\begin{array}{l} \{\langle u_1, a_1, \ldots, a_k\rangle\}, \mathcal{B}'(u_1, \mathbf{u}'), \\ (\mathbf{A}_1)_{a_1:\mathbf{h}_1}\langle \mathbf{f}_1'\rangle, \ldots, (\mathbf{A}_k)_{a_k:\mathbf{h}_k}\langle \mathbf{f}_k'\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{f}_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{f}_m\rangle \end{array} \right)
\tag{5.35}
$$

where $\mathbf{f}_q'$, $1 \le q \le k$ denote the new schemata of the relations $\mathbf{A}_q'$: $\mathbf{f}_q' = \mathbf{f}_q \setminus \{a_q\}$. The query R is defined as:

$$
R = \mathop{\Large\ominus}_{\mathbf{f}_1=\mathbf{z}u_1+\mathbf{G}_1\mathbf{s}+\mathbf{f}_1{}^{(0)}} \left(\begin{array}{l} \mathcal{B}_1\langle u_1\rangle, \mathbf{A}_1.\mathtt{enum}_{a_1}()\langle a_1, \mathbf{h}_1\rangle, \ldots, \\ \ldots, \mathbf{A}_k.\mathtt{enum}_{a_k}()\langle a_k, \mathbf{h}_k\rangle \end{array} \right)
\tag{5.36}
$$

$\mathcal{B}'$ are the bounds on $\mathbf{u}$, which can be computed by transforming the bounds $\mathcal{B}$ on the original loop indices $\mathbf{i}$:

$$
\mathcal{B}' = \{\mathbf{u} \mid \mathbf{U}\mathbf{u} \in \mathcal{B}\}
\tag{5.37}
$$

where $\mathbf{U}$ is the matrix of column transformations in (5.27). $\mathcal{B}_1$ is the projection of $\mathcal{B}'$ onto u_1.

However, the the result of the query Q' is different from that of the query in Figure 5.3: Q' enumerates over different loop indices $\mathbf{u} = \mathbf{U}^{-1}\mathbf{i}$. But we can still rewrite the *loop nest* as:

1. $\mathtt{doany}\ \langle u_1, a_1, \mathbf{h}_1, \ldots, a_k, \mathbf{h}_k\rangle \in R$
2. $\qquad \mathtt{doany}\ \langle \mathbf{u}', \mathbf{f}_1', \ldots, \mathbf{f}_k', \mathbf{f}_{k+1}, \ldots, \mathbf{f}_m\rangle \in Q'$
3. $\qquad\qquad \mathtt{S}(\mathbf{s}, \mathbf{U}\mathbf{u}, \mathbf{f}_1, \ldots, \mathbf{f}_m)$

Notice that the inner loop on line 2 together with the statement on line 3 satisfy the affine Θ-join form of a loop nest:

- $\mathbf{u}'$ is the new iteration space
- The set of invariants now is: $\mathbf{s} \cup \{u_1\}$.
- $\mathcal{B}'$ gives the bounds on $\mathbf{u}'$ in terms of the new invariants.

Now we can recursively apply the transformations to the new query!

The JOIN rule is summarized in Figure 5.4. Observe that the rule allows a join of zero tensor fields with a loop index. Such join is, of course, just a dense loop enumerating a "slice" of the iteration space. These degenerate joins are necessary when the matrix $\mathbf{F}$ in the data access equation $\mathbf{f} = \mathbf{F}\mathbf{i} + \ldots$ does not have full column rank and we need dense loop in order to enumerate over the loop iterations which are not restricted by the tensor fields. Here is a simple example:

Program 5.1
$\mathtt{doany}\ \langle i, j, i_x, v_x\rangle \in \mathop{\Large\ominus}_{i_x=i} \left(\ \{1 \le i, j \le n\}, \mathbf{X}\langle i_x, v_x\rangle\ \right)$
$\qquad \mathtt{...v}_x\mathtt{...}$

Suppose that the vector $\mathbf{X}$ is sparse and is stored in the `SPVO` data structure (Section 5.4.3). Then it's index hierarchy is $i_x \prec v_x$. At the start of query scheduling the field i_x is joinable. The result of applying the JOIN rule is:

Program 5.2

$$\texttt{doany}\ \langle i, i_x, \mathbf{h}\rangle \in \bigominus_{i=i_x} \left(\ \{1 \le i \le n\}, \mathbf{X}.\mathtt{enum}()_{i_x}\langle i_x, \mathbf{h}\rangle\ \right)$$

$$\quad \texttt{doany}\ \langle j, v_x\rangle \in \bigominus \left(\ \{1 \le j \le n\}, \mathbf{X}_{i_x:\mathbf{h}}\langle v_x\rangle\ \right)$$

$$\qquad \ldots \mathtt{v}_x \ldots$$

The loop index j is not restricted by the data access equation of the join in the inner. Moreover, there are no tensor indices left to be enumerated. We can apply the JOIN rules again, with the empty of set of joinable fields:

Program 5.3

$$\texttt{doany}\ \langle i, i_x, \mathbf{h}\rangle \in \bigominus_{i=i_x} \left(\ \{1 \le i \le n\}, \mathbf{X}.\mathtt{enum}()_{i_x}\langle i_x, \mathbf{h}\rangle\ \right)$$

$$\quad \texttt{doany}\ \langle j\rangle \in \{1 \le j \le n\}$$

$$\qquad \texttt{doany}\ \langle v_x\rangle \in \mathbf{X}_{i_x:\mathbf{h}}\langle v_x\rangle$$

$$\qquad\quad \ldots \mathtt{v}_x \ldots$$

The second loop is just a dense loop over the range of j's. The third loop is in fact just a dereference of the element of $\mathbf{X}$. SINGLETON rule (and variations) below governs how this pattern is transformed.

SEARCH

The SEARCH rule "surrounds" the query by a conditional that tests whether the value of a field is present in the corresponding relation. Without loss of generality, assume that the top term in the hierarchy of relation $\mathbf{A}_1$ is a stream tuple term: $\langle a_1, \ldots, a_k\rangle^*$. Furthermore, all the fields a_q, $1 \le q \le k$, are determined.

Let $\mathbf{a} = (a_1\ \ldots a_k)^T$. The data access equation gives us the value of $\mathbf{a}$ as a function of the invariants $\mathbf{s}$

$$\mathbf{a} = \mathbf{H}\mathbf{s} + \mathbf{a}^{(0)} \tag{5.38}$$

where $\mathbf{H}$ represents the rows of $\mathbf{G}$ that correspond to the fields in $\mathbf{a}$. By definition of the black-box protocol, the fields in $\mathbf{a}$ have an associated `search()` method. We can use this method to convert the loop nest in Figure 5.3 into:

$$\mathbf{a}\ \texttt{:=}\ \mathbf{H}\mathbf{s} + \mathbf{a}^{(0)}$$

$$\mathbf{A}\texttt{.declare\_iter(}\mathbf{h}\texttt{)}$$

$$\mathbf{A}_1\texttt{.search}_a\texttt{(}\mathbf{h}\texttt{,}\mathbf{a}\texttt{)}$$

$$\texttt{if}\ \mathbf{A}_1\texttt{.valid}_a\texttt{(}\mathbf{h}\texttt{) then}$$

$$\quad \texttt{doany}\ \left\langle\ \mathbf{i}, \mathbf{f}_1', v_1, \ldots, \mathbf{f}_m, v_m\ \right\rangle \in \bigominus_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i}\rangle, \\ (\mathbf{A}_1)_{\mathbf{a}:\mathbf{h}}\langle \mathbf{f}_1', v_1\rangle, \\ \mathbf{A}_2\langle \mathbf{f}_2, v_2\rangle, \ldots, \\ \mathbf{A}_m\langle \mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$$\qquad \texttt{S(}\mathbf{s}\texttt{,}\mathbf{i}\texttt{,}\mathtt{v}_1\texttt{,}\ldots\texttt{,}\mathtt{v}_m\texttt{)}$$

$$\quad \texttt{enddo}$$

$$\texttt{endif}$$

where $\mathbf{f}_1' = \mathbf{f}_1 \setminus \mathbf{a}$.

The SEARCH rule is summarized is Figure 5.5.

TUPLE-ENUM

Without loss of generality, suppose that the top term in the hierarchy kind for the relation $\mathbf{A}_1$ is a stream tuple term that involves more than one field: $\langle a_1, \ldots, a_k \rangle^*$, $1 < k$. Let $\mathbf{f}' = \{a_1, \ldots, a_k\}$ and $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. By analogy with (5.34), we can compute the unimodular matrix $\mathbf{U}$ such that:

$$\begin{array}{lcllll} \mathbf{f}' & = & \mathbf{L}\mathbf{u}_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)} \\ \mathbf{f}'' & = & \mathbf{W}\mathbf{u}_1 & +\mathbf{F}'\mathbf{u}_2 & +\mathbf{G}_2\mathbf{s} + \mathbf{f}''^{(0)} \end{array} \tag{5.39}$$

where:

$$\mathbf{F}\mathbf{U} = \begin{pmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{W} & \mathbf{F}' \end{pmatrix} \quad \mathbf{u} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i} \tag{5.40}$$

Moreover $\mathbf{L}$ has full column rank and is in column echelon form (see Appendix A). We can use standard methods to compute the bounds $\mathcal{B}_1$ on $\mathbf{u}_1$ and $\mathcal{B}_2$ on $\mathbf{u}_2$ in terms of $\mathbf{u}_1$. Using the above definitions, the TUPLE-ENUM rule transforms the loop nest in Figure 5.3 into:

```
doany ⟨a_1, ..., a_k, h⟩ ∈ A_1.enum()
    find u_1 such that f' = L u_1 + G_1 s + f'^(0)
    if (found ∧ u_1 ∈ B_1) then
        doany ⟨u_2, f'_1, v_1, ..., f_m, v_m⟩ ∈ Q'
            S(s,Uu,v_1,...,v_m)
        enddo
    endif
enddo
```

where $\mathbf{f}'_1 = \mathbf{f}_1 \setminus \{a_1, \ldots, a_k\}$ and the new query is:

$$Q' = \bigominus_{\mathbf{f}''=\mathbf{W}\mathbf{u}_1+\mathbf{F}'\mathbf{u}_2+\mathbf{G}_2\mathbf{s}+\mathbf{f}''^{(0)}} \left(\begin{array}{c} \mathcal{B}_2\langle \mathbf{u}_2 \rangle, (\mathbf{A}_1)_{\mathbf{h}}\langle \mathbf{f}'_1, v_1 \rangle, \\ \mathbf{A}_2\langle \mathbf{f}_2, v_2 \rangle, \ldots, \mathbf{A}_m\langle \mathbf{f}_m, v_m \rangle \end{array} \right) \tag{5.41}$$

The code to check the existence of the solution $\mathbf{u}_1$ to

$$\mathbf{f}' = \mathbf{L}\mathbf{u}_1 + \mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)} \tag{5.42}$$

and the code to find the solution itself can be easily generated because $\mathbf{L}$ is in column echelon form. The TUPLE-ENUM rule is summarized in Figure 5.6.

SINGLETON

The SINGLETON rule is designed to "peel-off" singleton tuple terms in Black Boxes. Suppose the top term in the hierarchy for a relation $\mathbf{A}$ is a singleton tuple term: $\langle a_1, \ldots, a_k \rangle^1$. Let $\mathbf{f}' = \{a_1, \ldots, a_k\}$ and $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. Using (5.39), (5.40) and the definitions of the bounds $\mathcal{B}_1$ and $\mathcal{B}_2$ above, the SINGLETON rule transforms the loop nest into:

```
a_1 := A_1.deref_1()
...
a_k := A_1.deref_k()
find u_1 such that f' = L u_1 + G_1 s + f'^(0)
```

```
if (found ∧ u₁ ∈ B₁) then
    doany ⟨u₂, f'₁, v₁, ..., f_m, v_m⟩ ∈ Q'
        S(s,Uu,v₁,...,v_m)
    enddo
endif
```

where the new query is:

$$Q' = \bigominus_{\mathbf{f}''=\mathbf{W}\mathbf{u}_1+\mathbf{F}'\mathbf{u}_2+\mathbf{G}_2\mathbf{s}+\mathbf{f}''^{(0)}} \left(\begin{array}{l} \mathcal{B}_2\langle \mathbf{u}_2 \rangle, \mathbf{A}_1.\mathtt{hderef}()\langle \mathbf{f}'_1, v_1 \rangle, \\ \mathbf{A}_2\langle \mathbf{f}_2, v_2 \rangle, \ldots \mathbf{A}_m\langle \mathbf{f}_m, v_m \rangle \end{array} \right) \tag{5.43}$$

The SINGLETON rule is summarized in Figure 5.7.

VALUE

The VALUE rule replaces a relation that represents a scalar value (usually, left after all the integer fields have been "peeled-off") with the expression for the value. Suppose that we have the following loop nest:

Program 5.4

```
doany ⟨..., v, ...⟩ ∈ ⊖... ( ..., R⟨v⟩, ... )
    ...v...
```

where the index hierarchy for $\mathbf{R}$ is just v:Type. The transformed code is:

Program 5.5

```
doany ⟨...⟩ ∈ ⊖... ( ... )
    ...  R.deref() ...
```

Since v might appear of the left-hand side of an assignment, we should check the value of $\mathbf{R}$.`lvalue_p`() metric: if the metric returns `false` then the value can not appear on the left-hand side. If v is not being written into, then we can generate more efficient code:

Program 5.6

```
real z
z := R.deref()
doany ⟨...⟩ ∈ ⊖... ( ... )
    ...  z ...
```

The transformation is summarized in Figure 5.8.

VALUE-UPDATE

Suppose that we have a loop nest with a commutative and associative update in the body:

Program 5.7

```
doany ⟨..., v, ...⟩ ∈ ⊖... ( ..., R⟨v⟩, ... )
    v := v op update
```

where `op` is a commutative and associative operation, such as addition or multiplication. Moreover, the term in the index hierarchy of the relation $\mathbf{R}$ is a value term: v:Type. Then, instead of applying the VALUE rule, we generate the following code:

QUERY:

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigominus_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i}\rangle, \mathbf{A}_1\langle \mathbf{f}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$\texttt{S(}\mathbf{s}\texttt{,}\mathbf{i}\texttt{,v}_1\texttt{,}\ldots\texttt{,v}_m\texttt{)}$

PREPROCESSING:

- Fields $\mathbf{f}' = (a_1, \ldots, a_k)^T$ are joinable (exposed and not determined). The field a_q belongs to the relation $\mathbf{A}_q$, $1 \le q \le k$.
- Let $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. Using unimodular column transformations $\mathbf{U}$ on $\mathbf{F}$ the data access equation is transformed into:

$$\begin{array}{rcllll} \mathbf{f}' & = & \mathbf{z}u_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}_1{}^{(0)} \\ \mathbf{f}'' & = & \mathbf{w}u_1 & +\mathbf{F}'\mathbf{u}' & +\mathbf{G}_2\mathbf{s} + \mathbf{f}_2{}^{(0)} \end{array}$$

where

$$\mathbf{F}\mathbf{U} = \begin{pmatrix} \mathbf{z} & \mathbf{0} \\ \mathbf{w} & \mathbf{F}' \end{pmatrix} \quad \mathbf{u} = \begin{pmatrix} u_1 \\ \mathbf{u}' \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i}$$

TRANSFORMATION:

$\texttt{doany}\ \langle u_1, a_1, \mathbf{h}_1, \ldots, a_k, \mathbf{h}_k\rangle \in R$

$\quad\texttt{doany}\ \langle \mathbf{u}', \mathbf{f}_1', \ldots, \mathbf{f}_k', \mathbf{f}_{k+1}, \ldots, \mathbf{f}_m\rangle \in Q'$

$\quad\quad\texttt{S(}\mathbf{s}, \mathbf{U}\mathbf{u}, \mathbf{f}_1, \ldots, \mathbf{f}_m\texttt{)}$

WHERE:

$$R = \bigominus_{\mathbf{f}_1=\mathbf{z}u_1+\mathbf{G}_1\mathbf{s}+\mathbf{f}_1{}^{(0)}} \begin{pmatrix} \mathcal{B}_1\langle u_1\rangle, \mathbf{A}_1.\mathtt{enum}_{a_1}()\langle a_1, \mathbf{h}_1\rangle, \ldots, \\ \ldots, \mathbf{A}_k.\mathtt{enum}_{a_k}()\langle a_k, \mathbf{h}_k\rangle \end{pmatrix}$$

$$Q' = \biguplus_{\substack{\langle u_1, a_1, \mathbf{h}_1, \ldots, \\ a_k, \mathbf{h}_k\rangle \in R}} \bigominus_{\substack{\mathbf{f}''=\mathbf{w}u_1+ \\ \mathbf{F}'\mathbf{u}'+\mathbf{G}_2\mathbf{s}+\mathbf{f}_2{}^{(0)}}} \begin{pmatrix} \{\langle u_1, a_1, \ldots, a_k\rangle\}, \mathcal{B}'(u_1, \mathbf{u}'), \\ (\mathbf{A}_1)_{a_1:\mathbf{h}_1}\langle \mathbf{f}_1'\rangle, \ldots, (\mathbf{A}_k)_{a_k:\mathbf{h}_k}\langle \mathbf{f}_k'\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{f}_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{f}_m\rangle \end{pmatrix}$$

Figure 5.4: The JOIN rule

QUERY:

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigominus_{\mathbf{f}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i} \rangle, \mathbf{A}_1\langle \mathbf{f}_1, v_1 \rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{f}_m, v_m \rangle \end{pmatrix}$$

$\quad\texttt{S}(\mathbf{s},\mathbf{i},\mathrm{v}_1,\ldots,\mathrm{v}_m)$

PREPROCESSING:

- The top term in the hierarchy for $\mathbf{A}_1$ is a stream tuple term: $\langle a_1, \ldots, a_k \rangle^*$. Let $\mathbf{a} = (a_1 \ \ldots a_k)^T$ be the vector of the fields in the term.
- All the fields a_q, $1 \leq q \leq k$, are determined. Their values are given in the data access equation by

  $$\mathbf{a} = \mathbf{Hs} + \mathbf{a}^{(0)}$$

  where $\mathbf{H}$ is the matrix of the appropriate rows of $\mathbf{G}$.
- Let $\mathbf{f}_1' = \mathbf{f}_1 \setminus \mathbf{a}$ be all the other fields of $\mathbf{A}_1$.

TRANSFORMATION:

$\mathbf{a}$:= $\mathbf{Hs} + \mathbf{a}^{(0)}$

$\mathbf{A}_1$.declare_iter($\mathbf{h}$)

$\mathbf{A}_1$.search($\mathbf{h}$,$\mathbf{a}$)

if $\mathbf{A}_1$.valid($\mathbf{h}$) then

$$\quad\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1', v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigominus_{\mathbf{f}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i} \rangle, \\ (\mathbf{A}_1)_{\mathbf{a}:\mathbf{h}}\langle \mathbf{f}_1', v_1 \rangle, \\ \mathbf{A}_2\langle \mathbf{f}_2, v_2 \rangle, \ldots, \\ \mathbf{A}_m\langle \mathbf{f}_m, v_m \rangle \end{pmatrix}$$

$\qquad\texttt{S}(\mathbf{s},\mathbf{i},\mathrm{v}_1,\ldots,\mathrm{v}_m)$

$\quad$enddo

endif

Figure 5.5: The SEARCH rule

QUERY:

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigominus_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}_1\langle\mathbf{f}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle\mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$\quad\texttt{S(}\mathbf{s}\texttt{,}\mathbf{i}\texttt{,v}_1\texttt{,}\ldots\texttt{,v}_m\texttt{)}$

PREPROCESSING:

- The top term in the hierarchy for $\mathbf{A}_1$ is a stream tuple term with more than one field: $\langle a_1, a_2, \ldots, a_k\rangle^*$, $1 < k$. Let $\mathbf{f}'$ be the set the fields in the term.
- Let $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. Using unimodular column transformations $\mathbf{U}$ on $\mathbf{F}$ the data access equation is transformed into:

$$\begin{array}{rcllll} \mathbf{f}' & = & \mathbf{L}\mathbf{u}_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)} \\ \mathbf{f}'' & = & \mathbf{W}\mathbf{u}_1 & +\mathbf{F}'\mathbf{u}_2 & +\mathbf{G}_2\mathbf{s} + \mathbf{f}''^{(0)} \end{array}$$

$$\text{where}\quad \mathbf{F}\mathbf{U} = \begin{pmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{W} & \mathbf{F}' \end{pmatrix} \quad \mathbf{u} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i}$$

- Compute the bounds $\mathcal{B}_1$ on $\mathbf{u}_1$ and $\mathcal{B}_2$ on $\mathbf{u}_2$ in terms of $\mathbf{u}_1$.

TRANSFORMATION:

$\texttt{doany}\ \langle a_1, \ldots, a_k, \mathbf{h}\rangle \in \mathbf{A}_1.\mathtt{enum}()$

$\quad\texttt{find}\ \mathbf{u}_1\ \texttt{such that}\ \mathbf{f}' = \mathbf{L}\mathbf{u}_1 + \mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)}$

$\quad\texttt{if (found}\ \wedge\ \mathbf{u}_1 \in \mathcal{B}_1\texttt{) then}$

$\qquad\texttt{doany}\ \langle \mathbf{u}_2, \mathbf{f}_1', v_1, \ldots, \mathbf{f}_m, v_m\rangle \in Q'$

$\qquad\quad\texttt{S(}\mathbf{s}\texttt{,}\mathbf{U}\mathbf{u}\texttt{,v}_1\texttt{,}\ldots\texttt{,v}_m\texttt{)}$

WHERE:

$$Q' = \bigominus_{\mathbf{f}''=\mathbf{W}\mathbf{u}_1+\mathbf{F}'\mathbf{u}_2+\mathbf{G}_2\mathbf{s}+\mathbf{f}''^{(0)}} \begin{pmatrix} \mathcal{B}_2\langle\mathbf{u}_2\rangle, (\mathbf{A}_1)_{\mathbf{h}}\langle\mathbf{f}_1', v_1\rangle, \\ \mathbf{A}_2\langle\mathbf{f}_2, v_2\rangle, \ldots, \mathbf{A}_m\langle\mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$$\mathbf{f}_1' = \mathbf{f}_1 \setminus \{a_1, \ldots, a_k\}$$

Figure 5.6: The TUPLE-ENUM rule

QUERY:

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigominus_{\mathbf{f}=\mathbf{F}\mathbf{i}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}_1\langle\mathbf{f}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle\mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$\quad \texttt{S}(\mathbf{s}, \mathbf{i}, \texttt{v}_1, \ldots, \texttt{v}_m)$

PREPROCESSING:

- The top term in the hierarchy for $\mathbf{A}_1$ is a singleton tuple term: $\langle a_1, a_2, \ldots, a_k\rangle^1$. Let $\mathbf{f}'$ be the set of the fields in the term.
- Let $\mathbf{f}'' = \mathbf{f} \setminus \mathbf{f}'$. Using unimodular column transformations $\mathbf{U}$ on $\mathbf{F}$ the data access equation is transformed into:

$$\begin{array}{rclll} \mathbf{f}' & = & \mathbf{L}\mathbf{u}_1 & & +\mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)} \\ \mathbf{f}'' & = & \mathbf{W}\mathbf{u}_1 & +\mathbf{F}'\mathbf{u}_2 & +\mathbf{G}_2\mathbf{s} + \mathbf{f}''^{(0)} \end{array}$$

$$\text{where}\quad \mathbf{F}\mathbf{U} = \begin{pmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{W} & \mathbf{F}' \end{pmatrix} \quad \mathbf{u} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i}$$

- Compute the bounds $\mathcal{B}_1$ on $\mathbf{u}_1$ and $\mathcal{B}_2$ on $\mathbf{u}_2$ in terms of $\mathbf{u}_1$.

TRANSFORMATION:

$\texttt{a}_1 \ \texttt{:=}\ \mathbf{A}_1.\texttt{deref}_1\texttt{();}\ \texttt{...;}\ \texttt{a}_k \ \texttt{:=}\ \mathbf{A}_1.\texttt{deref}_k\texttt{()}$

$\texttt{find}\ \mathbf{u}_1\ \texttt{such that}\ \mathbf{f}' = \mathbf{L}\mathbf{u}_1 + \mathbf{G}_1\mathbf{s} + \mathbf{f}'^{(0)}$

$\texttt{if (found} \wedge \mathbf{u}_1 \in \mathcal{B}_1\texttt{) then}$

$\quad \texttt{doany}\ \langle\mathbf{u}_2, \mathbf{f}_1', v_1, \ldots, \mathbf{f}_m, v_m\rangle \in Q'$

$\quad\quad \texttt{S}(\mathbf{s}, \mathbf{U}\mathbf{u}, \texttt{v}_1, \ldots, \texttt{v}_m)$

WHERE:

$$Q' = \bigominus_{\mathbf{f}''=\mathbf{W}\mathbf{u}_1+\mathbf{F}'\mathbf{u}_2+\mathbf{G}_2\mathbf{s}+\mathbf{f}''^{(0)}} \begin{pmatrix} \mathcal{B}_2\langle\mathbf{u}_2\rangle, \mathbf{A}_1.\texttt{hderef()}\langle\mathbf{f}_1', v_1\rangle, \\ \mathbf{A}_2\langle\mathbf{f}_2, v_2\rangle, \ldots \mathbf{A}_m\langle\mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$$\mathbf{f}_1' = \mathbf{f}_1 \setminus \{a_1, \ldots, a_k\}$$

Figure 5.7: The SINGLETON rule

QUERY:

doany $\langle \ldots, v, \ldots \rangle \in \bigominus_{\ldots} (\; \ldots, \mathbf{R}\langle v \rangle, \ldots \;)$

```
    ...v...
```

PREPROCESSING:

- The top term in the index hierarchy of $\mathbf{R}$ is v:Type.
- If v appears of the left-hand side of an assignment, the value of the $\mathbf{R}$.`lvalue_p`() metric must be `true`. Otherwise, signal an error.

TRANSFORMATION (if v is being written into):

doany $\langle \ldots \rangle \in \bigominus_{\ldots} (\; \ldots \;)$

    ... R.deref_v() ...

TRANSFORMATION (if v is read-only):

```
real z
z := R.deref()
```

doany $\langle \ldots \rangle \in \bigominus_{\ldots} (\; \ldots \;)$

```
    ...  z ...
```

Figure 5.8: The VALUE rule

Program 5.8

```
real z
z := identity
```

`doany` $\langle \ldots \rangle \in \Theta_{\ldots}(\ \ldots\ )$

```
    z := z op update
enddo
```

$\mathbf{R}$`.deref() :=` $\mathbf{R}$`.deref() op z`

where `identity` is the identity value of the operator `op`: 0 for addition, 1 for multiplication. Of course, the $\mathbf{R}$.`lvalue_p`() metric must return `true`. The rule is summarized in Figure 5.9. The VALUE-UPDATE rule

QUERY:

`doany` $\langle \ldots, \mathbf{a}, v, \ldots \rangle \in \Theta_{\ldots}(\ \ldots, \mathbf{R}\langle v \rangle, \ldots\ )$

```
    v := v op update
```

PREPROCESSING:

- `op` is commutative and associative.
- `update` is an expression
- The top term in the index hierarchy of $\mathbf{R}$ is v:Type.
- $\mathbf{R}$.`lvalue_p`() returns `true`

TRANSFORMATION:

```
real z
z := identity
```

`doany` $\langle \ldots \rangle \in \Theta_{\ldots}(\ \ldots\ )$

```
    z := z op update
enddo
```

$\mathbf{R}$`.deref() :=` $\mathbf{R}$`.deref() op z`

WHERE:

- `identity` is the identity value for the `op` operator: 0 for addition and 1 for multiplication.

Figure 5.9: The VALUE-UPDATE rule

is quite effective in improving performance in codes such as matrix-vector product. Usually the $\mathbf{R}$.`deref`() method expands into an array reference. For example, if we are dealing with a sparse vector $\mathbf{X}$ stored using the `SPV0` format, then $\mathbf{R}$ is given by the ADT expression $\mathbf{X}$.`hderef`(ii) where `ii` is the offset into the data structure. $\mathbf{X}$.`hderef`(ii).`deref`() expands into `VX[ii]` where `VX` is the array that stores the non-zero values of $\mathbf{X}$. The loop becomes:

```
doany ...
    VX[ii] := VX[ii] op update
enddo
```

When translated into machine code, this operation requires, during each iteration of the loop, loading the value of `VX[ii]` into a register, updating the value in the register and storing the result back in memory. Sometimes C and Fortran compilers can optimize this operation along the lines of the VALUE-UPDATE rule and use only one load/store operation for the whole loop. Currently, our compiler produces C code and a C compiler can not always perform this optimization due to the complexity of resulting index expressions and possible aliasing between arrays (when using C semantics). In our experience, delegating this optimization to the sparse compiler has improved performance by 10-20% for matrix-vector products during our experiments with the Sparse BLAS library (see Section 6.1). Generating Fortran code can potentially remove the need to the VALUE-UPDATE rule.

5.4.3 Examples

After all the mind-numbing algebra it is time for some examples. The first example illustrates the choices available during query scheduling. The second example illustrates the use of the echelon form of the data access equation.

The search problem

Consider the loop nest that performs an incomplete outer-product update $\mathbf{A} \approx \mathbf{A} + (\mathbf{PY})\mathbf{X}^T$:

```
doany i=1..N; i'=1..N; j=1..N
    if A(i,j) ∧ X(j) ∧ P(i,i') ∧ Y(i') then
        A[i,j] := A[i,j] + X[j] * Y[i']
```

We call the product "incomplete" because only the elements that are already stored in $\mathbf{A}$ are actually updated (this is the meaning of the $\approx$ sign in this example). $\mathbf{P}$ is a permutation applied to the elements of $\mathbf{Y}$. We can view permutation as a special kind of a sparse matrix: it stores the subset of all possible $\langle i, i' \rangle$ pairs of indices and it does not store any values. In our relational model $\mathbf{P}$ is yet another relation.

Let the schema for the relations be:

$$\mathbf{A}\langle r, c, a\rangle \quad \mathbf{X}\langle j_x, x\rangle \quad \mathbf{Y}\langle i_y, y\rangle \quad \mathbf{P}\langle p, q\rangle \tag{5.44}$$

Then we can rewrite the loop nest in the affine join form:

$$\texttt{doany } \langle i, j, r, c, a, j_x, x, i_y, y, p, q\rangle \in \ldots$$

$$\ldots \bigominus_{\substack{i=r=p\wedge \\ j=c=j_x\wedge \\ i'=q=i_y}} \left(\begin{array}{l} \{\langle i, i', j\rangle \mid 1 \le i, i', j \le N\}, \\ \mathbf{A}\langle r, c, a\rangle, \mathbf{X}\langle j_x, x\rangle, \mathbf{Y}\langle i_y, y\rangle, \mathbf{P}\langle p, q\rangle \end{array} \right)$$

```
    a := a + x * y
```

Notice that the field a of $\mathbf{A}$ is used as an l-value. The vector of integer fields is $\mathbf{f} = (r\ c\ j_x\ i_y\ p\ q)^T$, the vector of invariants is $\mathbf{s} = (N)$ and the iteration vector is $\mathbf{i} = (i\ i'\ j)^T$. The data access equation is:

$$\begin{pmatrix} r \\ c \\ j_x \\ i_y \\ p \\ q \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ i' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.45}$$

Suppose that the matrix $\mathbf{A}$ is stored using the CCCS format and the vectors $\mathbf{X}$ and $\mathbf{Y}$ are stored using SPVO format (Figure 3.1). The storage format for the permutation $\mathbf{P}$ is described in Section 4.1. The hierarchy kinds are:

$$\begin{array}{lcl} \mathbf{A} & : & c \prec r \prec a \\ \mathbf{X} & : & j_x \prec x \\ \mathbf{Y} & : & i_y \prec y \\ \mathbf{P} & : & (p^* \prec q^1) \oplus (q^* \prec p^1) \end{array} \tag{5.46}$$

First, we have to apply the PICK-HIERARCHY rule. There isn't much choice for the tensors. For the permutation $\mathbf{P}$, let's pick the first hierarchy $p^* \prec q^1$. The resulting code is:

$$\texttt{doany}\ \langle i,j,r,c,a,j_x,x,i_y,y,p,q\rangle \in \ldots$$

$$\ldots \bigominus_{\substack{i=r=p\wedge\\ j=c=j_x\wedge\\ i'=q=i_y}} \left(\begin{array}{l} \{\langle i,i',j\rangle \mid 1 \le i,i',j \le N\}, \\ \mathbf{A}.\mathtt{h}_1()\langle r,c,a\rangle, \mathbf{X}.\mathtt{h}_1()\langle j_x,x\rangle, \\ \mathbf{Y}.\mathtt{h}_1()\langle i_y,y\rangle, \mathbf{P}.\mathtt{h}_1()\langle p,q\rangle \end{array} \right)$$

```
    a := a + x * y
```

At this point, the fields c, j_x, i_y and p are exposed. No fields have been determined, yet. The matrix $\mathbf{F}$ in the data access equation (5.45) is:

$$\mathbf{F} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{5.47}$$

Rows 2 and 3 are trivially multiples of one another. They correspond to c and j_x. Therefore c and j_x are joinable. i_y is joinable, by itself. So is p. Let's apply the JOIN rule to the fields c and j_x. First, we permute the data access equation so that these fields appear on the top:

$$\begin{pmatrix} c \\ j_x \\ r \\ i_y \\ p \\ q \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ i' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.48}$$

To get the equation in the echelon form (5.34) we need to swap columns 1 and 3 of the new $\mathbf{F}$ matrix:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{5.49}$$

The new data access equation in the echelon form (5.39) is:

$$\begin{aligned} \begin{pmatrix} c \\ j_x \end{pmatrix} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} (u_1) + \begin{pmatrix} 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} r \\ i_y \\ p \\ q \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (u_1) + \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \end{aligned} \tag{5.50}$$

The new loop indices $\mathbf{u}$ are related to the old $\mathbf{i}$ by:

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} i \\ i' \\ j \end{pmatrix} \tag{5.51}$$

$u_1 = j$, therefore the bound $\mathcal{B}_1$ on u_1 is $1 \leq u_1 \leq N$. The bound $\mathcal{B}_2$ on the rest of $\mathbf{u}$ is $1 \leq u_2, u_3 \leq N$. The loop nest becomes:

doany $\langle c, \mathbf{h}, j_x, \mathbf{g}, u_1 \rangle \in \ldots$

$$\ldots \bigominus_{u_1=c=j_x} \left(\ \{1 \leq u_1 \leq N\}, \mathbf{A}.\mathtt{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle \ \right)$$

  doany $\langle r, a, x, i_y, y, p, q \rangle \in \ldots$

$$\ldots \bigominus_{\substack{u_3=r=p\wedge \\ u_2=q=i_y}} \left(\begin{array}{l} \{1 \leq u_2, u_3 \leq N\}, \\ (\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}}\langle r, a\rangle, (\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}}\langle x\rangle, \\ \mathbf{Y}.\mathrm{h}_1()\langle i_y, y\rangle, \mathbf{P}.\mathrm{h}_1()\langle p, q\rangle \end{array} \right)$$

```
    a := a + x * y
```

The inner loop is again in the affine join form. The iteration vector is $(u_2\ u_3)^T$ and the vector of invariants is $(u_1\ N)^T$. The data access equation is:

$$\begin{pmatrix} r \\ i_y \\ p \\ q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} u_1 \\ N \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.52}$$

The hierarchies for the relations in the query are:

$$\begin{array}{lcl} (\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}} & : & r^* \prec a \\ (\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}} & : & x \\ \mathbf{Y}.\mathrm{h}_1() & : & i_y^* \prec y \\ \mathbf{P}.\mathrm{h}_1() & : & p^* \prec q^1 \end{array} \tag{5.53}$$

The fields r, i_y and p are exposed. The fields r and p are joinable because $r = p = u_3$. The field i_y is joinable with itself. We can also apply the VALUE rule to x. Let's start with x. According to Figure 5.7, the transformed code is:

doany $\langle c, \mathbf{h}, j_x, \mathbf{g}, u_1 \rangle \in \ldots$

$$\ldots \bigominus_{u_1=c=j_x} \left(\ \{1 \leq u_1 \leq N\}, \mathbf{A}.\mathtt{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle \ \right)$$

  REAL x

  x := $(\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}}$.deref()

  doany $\langle r, a, i_y, y, p, q \rangle \in \ldots$

$$\ldots \bigominus_{\substack{u_3=r=p\wedge \\ u_2=q=i_y}} \left(\begin{array}{l} \{1 \leq u_2, u_3 \leq N\}, (\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}}\langle r, a\rangle, \\ \mathbf{Y}.\mathrm{h}_1()\langle i_y, y\rangle, \mathbf{P}.\mathrm{h}_1()\langle p, q\rangle \end{array} \right)$$

```
    a := a + x * y
```

The data access equation (5.52) applies to the query in the innermost loop, as well. At this point we can either apply the JOIN rule to r and p, or apply the JOIN rule to i_y. Let's try the latter. First, we permute the data access equation (5.52) to put i_y on the top:

$$\begin{pmatrix} i_y \\ r \\ p \\ q \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} u_1 \\ N \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.54}$$

The equation is already in echelon form:

$$\begin{aligned}
(i_y) &= (1)\,(u_2) + (0)\begin{pmatrix} u_1 \\ N \end{pmatrix} + (0) \\
\begin{pmatrix} r \\ p \\ q \end{pmatrix} &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}(u_2) + \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}(u_3) + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\begin{pmatrix} u_1 \\ N \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
\end{aligned} \tag{5.55}$$

The transformed code is:

$\texttt{doany}\ \langle c, \mathbf{h}, j_x, \mathbf{g}, u_1\rangle \in \ldots$

$\qquad \ldots \bigodot_{u_1=c=j_x} \left(\ \{1 \le u_1 \le N\}, \mathbf{A}.\mathtt{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle\ \right)$

$\quad \texttt{REAL x}$

$\quad \texttt{x := } (\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}}.\mathtt{deref}()$

$\quad \texttt{doany}\ \langle i_y, \mathbf{k}, u_2\rangle \in \bigodot_{i_y=u_2} \left(\ \{1 \le u_2 \le N\}, \mathbf{Y}.\mathrm{h}_1().\mathtt{enum}_{i_y}()\ \right)$

$\qquad \texttt{doany}\ \langle r, a, y, p, q\rangle \in \ldots$

$\qquad\quad \ldots \bigodot_{\substack{u_3=r=p\wedge \\ u_2=q=i_y}} \left(\begin{array}{l} \{1 \le u_2, u_3 \le N\}, (\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}}\langle r, a\rangle, \\ (\mathbf{Y}.\mathrm{h}_1())_{i_y:\mathbf{k}}\langle y\rangle, \mathbf{P}.\mathrm{h}_1()\langle p, q\rangle \end{array}\right)$

$\qquad\quad \texttt{a := a + x * y}$

Now we can join r and p:

$\texttt{doany}\ \langle c, \mathbf{h}, j_x, \mathbf{g}, u_1\rangle \in \ldots$

$\qquad \ldots \bigodot_{u_1=c=j_x} \left(\ \{1 \le u_1 \le N\}, \mathbf{A}.\mathtt{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle\ \right)$

$\quad \texttt{REAL x}$

$\quad \texttt{x := } (\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}}.\mathtt{deref}()$

$\quad \texttt{doany}\ \langle i_y, \mathbf{k}, u_2\rangle \in \bigodot_{i_y=u_2} \left(\ \{1 \le u_2 \le N\}, \mathbf{Y}.\mathrm{h}_1().\mathtt{enum}_{i_y}()\ \right)$

$\qquad \texttt{doany}\ \langle r, \mathbf{d}, p, \mathbf{e}, u_3\rangle \in \ldots$

$\qquad\quad \ldots \bigodot_{u_3=r=p} \left(\begin{array}{l} \{1 \le u_3 \le N\}, ((\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}}).\mathtt{enum}_r()\langle r, \mathbf{d}\rangle, \\ \mathbf{P}.\mathrm{h}_1().\mathtt{enum}_p()\langle p, \mathbf{e}\rangle \end{array}\right)$

$\qquad\quad \texttt{doany}\ \langle a, y, q\rangle \in \ldots$

$\qquad\qquad \ldots \bigodot_{u_2=q} \left(\begin{array}{l} (\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}, r:\mathbf{d}}\langle a\rangle, \\ (\mathbf{Y}.\mathrm{h}_1())_{i_y:\mathbf{k}}\langle y\rangle, (\mathbf{P}.\mathrm{h}_1())_{p:\mathbf{e}}\langle q\rangle \end{array}\right)$

$\qquad\qquad\quad \texttt{a := a + x * y}$

At this point we can apply the VALUE-UPDATE rule to $\mathbf{A}$:

$\texttt{doany}\ \langle c, \mathbf{h}, j_x, \mathbf{g}, u_1\rangle \in \ldots$

$\qquad \ldots \bigodot_{u_1=c=j_x} \left(\ \{1 \le u_1 \le N\}, \mathbf{A}.\mathtt{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle\ \right)$

$\quad \texttt{REAL x}$

$\quad \texttt{x := } (\mathbf{X}.\mathrm{h}_1())_{j_x:\mathbf{g}}.\mathtt{deref}()$

$\quad \texttt{doany}\ \langle i_y, \mathbf{k}, u_2\rangle \in \bigodot_{i_y=u_2} \left(\ \{1 \le u_2 \le N\}, \mathbf{Y}.\mathrm{h}_1().\mathtt{enum}_{i_y}()\ \right)$

$\qquad \texttt{doany}\ \langle r, \mathbf{d}, p, \mathbf{e}, u_3\rangle \in \ldots$

$\qquad\quad \ldots \bigodot_{u_3=r=p} \left(\begin{array}{l} \{1 \le u_3 \le N\}, ((\mathbf{A}.\mathrm{h}_1())_{c:\mathbf{h}}).\mathtt{enum}_r()\langle r, \mathbf{d}\rangle, \\ \mathbf{P}.\mathrm{h}_1().\mathtt{enum}_p()\langle p, \mathbf{e}\rangle \end{array}\right)$

```
REAL z = 0
doany ⟨y, q⟩ ∈ ...
        ... ⊖_{u2=q} ( (Y.h1())_{iy:k}⟨y⟩, (P.h1())_{p:e}⟨q⟩ )
                z := z + x * y
enddo
A.h1())_{c:h,r:d}.deref() := A.h1())_{c:h,r:d}.deref() + z
```

Observe that at this point q is determined because u_2 is invariant with respect to the innermost loop. Also we seem to have "lost" the iteration space in this loop. In fact, it is empty! There is nothing in the definition of the JOIN rule that would prevent the vector **u**' in Figure 5.4 from being empty. Only the SINGLETON rule is available at this point. After applying it to q we get:

```
doany ⟨c, h, jx, g, u1⟩ ∈ ...
        ... ⊖_{u1=c=jx} ( {1 ≤ u1 ≤ N}, A.enum_c()⟨c, h⟩, X.enum_{jx}()⟨jx, g⟩ )
    REAL x
    x := (X.h1())_{jx:g}.deref()
    doany ⟨iy, k, u2⟩ ∈ ⊖_{iy=u2} ( {1 ≤ u2 ≤ N}, Y.h1().enum_{iy}() )
        doany ⟨r, d, p, e, u3⟩ ∈ ...
            ... ⊖_{u3=r=p} ( {1 ≤ u3 ≤ N}, ((A.h1())_{c:h}).enum_r()⟨r, d⟩,
                             P.h1().enum_p()⟨p, e⟩ )
            REAL z = 0
            q := ((P.h1())_{p:e}).deref_q()
            if (q = u2) then
                doany ⟨y⟩ ∈ ⊖_true ( (Y.h1())_{iy:k}⟨y⟩, )
                        z := z + x * y
                enddo
            endif
            A.h1())_{c:h,r:d}.deref() := A.h1())_{c:h,r:d}.deref() + z
```

The innermost loop is still in the affine join form! It is a rather degenerate case: there are no integer fields left and the data access equation is a tautology. After applying the VALUE rule to y we get:

```
doany ⟨c, h, jx, g, u1⟩ ∈ ...
        ... ⊖_{u1=c=jx} ( {1 ≤ u1 ≤ N}, A.enum_c()⟨c, h⟩, X.enum_{jx}()⟨jx, g⟩ )
    REAL x
    x := (X.h1())_{jx:g}.deref()
    doany ⟨iy, k, u2⟩ ∈ ⊖_{iy=u2} ( {1 ≤ u2 ≤ N}, Y.h1().enum_{iy}() )
        doany ⟨r, d, p, e, u3⟩ ∈ ...
            ... ⊖_{u3=r=p} ( {1 ≤ u3 ≤ N}, ((A.h1())_{c:h}).enum_r()⟨r, d⟩,
                             P.h1().enum_p()⟨p, e⟩ )
            REAL z = 0
            q := ((P.h1())_{p:e}).deref_q()
            if (q = u2) then
```

REAL y

y := $\mathbf{Y}.\mathtt{h}_1())_{i_y:\mathbf{k}}$.deref()

z := z + x * y

endif

$\mathbf{A}.\mathtt{h}_1())_{c:\mathbf{h},r:\mathbf{d}}$.deref() := $\mathbf{A}.\mathtt{h}_1())_{c:\mathbf{h},r:\mathbf{d}}$.deref() + z

This is just one plan for performing the computation. Informally, it performs the following steps:

1. doany $j \in \mathbf{A} \bowtie \mathbf{X}$
2.   doany $i' \in \mathbf{Y}$
3.   doany $i \in \mathbf{A}_j \bowtie \mathbf{P}$
4.   if $\mathbf{P}_i = i'$ then
5. $\mathbf{A}_{j,i}$:= $\mathbf{A}_{j,i}$ + $\mathbf{X}_j$ * $\mathbf{Y}_{i'}$

What is the asymptotic complexity of this code? Let's define the following quantities:

- η_c is be the number of columns stored $\mathbf{A}$
- η_r is the largest number of elements in a column of $\mathbf{A}$
- η_x is the number of elements stored in $\mathbf{X}$
- η_y is the number of elements stored in $\mathbf{Y}$

Then the loop on line 1 takes $O(\eta_c + \eta_x)$ time. The loop on line 2 takes $O(\eta_y)$ time. And the loop on line 3 can be done in $O(\eta_r)$ time. Altogether the time complexity of the final code is:

$$O((\eta_c + \eta_x)\eta_y\eta_r) = O(\eta_c\eta_r\eta_y + \eta_x\eta_y\eta_r) \tag{5.56}$$

What other plans can we generate? After we join c and u_1, we can choose to join the p and r fields before enumerating i_y. The code at this stage becomes:

doany $\langle c, \mathbf{h}, j_x, \mathbf{g}, u_1 \rangle \in \ldots$

$\ldots \bigominus_{u_1=c=j_x} \left(\{1 \le u_1 \le N\}, \mathbf{A}.\mathsf{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathsf{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle \right)$

REAL x

x := $\mathbf{X}.\mathtt{h}_1())_{j_x:\mathbf{g}}$.deref()

doany $\langle r, \mathbf{e}, p, \mathbf{d} \rangle \in \ldots$

$\ldots \bigominus_{r=p=u_3} \left(\{1 \le u_3 \le N\}, \mathbf{A}_{c:\mathbf{h}}.\mathsf{enum}_r()\langle r, \mathbf{e}\rangle, \mathbf{P}.\mathsf{enum}_p()\langle p, \mathbf{d}\rangle \right)$

doany $\langle a, i_y, y, q \rangle \in \ldots$

$\ldots \bigominus_{u_2=q=i_y} \left(\{1 \le u_2 \le N\}, \mathbf{A}_{c:\mathbf{h},r:\mathbf{e}}\langle a\rangle, \mathbf{P}_{p:\mathbf{d}}\langle q\rangle, \mathbf{Y}\langle i_y, y\rangle \right)$

a := a + x * y

There are two choices now: either enumerate i_y by using the JOIN rule or set the value for q using the SINGLETON rule. Let's choose the latter. According to Figure 5.7, the vector $\mathbf{u}_1$ is (u_2) and the vector $\mathbf{u}_2$ is empty:

doany $\langle c, \mathbf{h}, j_x, \mathbf{g}, u_1 \rangle \in \ldots$

$\ldots \bigominus_{u_1=c=j_x} \left(\{1 \le u_1 \le N\}, \mathbf{A}.\mathsf{enum}_c()\langle c, \mathbf{h}\rangle, \mathbf{X}.\mathsf{enum}_{j_x}()\langle j_x, \mathbf{g}\rangle \right)$

```
x := (X.h_1())_{j_x:g}.deref_x()
doany ⟨r, e, p, d⟩ ∈ ...⟩
        ... ⊖_{r=p=u_3} ( {1 ≤ u_3 ≤ N}, A_{c:h}.enum_r()⟨r, e⟩, P.enum_p()⟨p, d⟩ )
    q := (P_{p:d}).deref_q()
    u_2 := q
    if 1 ≤ u_2 ≤ N then
        doany ⟨a, i_y, y⟩ ∈ ⊖_{u_2=i_y} ( A_{c:h,r:e}⟨a⟩, Y⟨i_y, y⟩ )
            a := a + x * y
```

Now in the innermost loop the value of i_y is determined, because at u_2 is an invariant. We can apply the SEARCH rule:

```
doany ⟨c, h, j_x, g, u_1⟩ ∈ ...
        ... ⊖_{u_1=c=j_x} ( {1 ≤ u_1 ≤ N}, A.enum_c()⟨c, h⟩, X.enum_{j_x}()⟨j_x, g⟩ )
    REAL x
    x := (X.h_1())_{j_x:g}.deref_x()
    doany ⟨r, e, p, d⟩ ∈ ...⟩
            ... ⊖_{r=p=u_3} ( {1 ≤ u_3 ≤ N}, A_{c:h}.enum_r()⟨r, e⟩, P.enum_p()⟨p, d⟩ )
        q := (P_{p:d}).deref_q()
        u_2 := q
        if 1 ≤ u_2 ≤ N then
            i_y := u_2
            Y.declare_iter_{i_y}(k)
            Y.search_{i_y}(i_y,k)
            if Y.valid_{i_y}(k)
                doany ⟨a, y⟩ ∈ ⊖_true ( A_{c:h,r:e}⟨a⟩, Y_{i_y:k}⟨y⟩ )
                        a := a + x * y
```

To obtain the final code we apply the VALUE rule to a and y:

```
doany ⟨c, h, j_x, g, u_1⟩ ∈ ...
        ... ⊖_{u_1=c=j_x} ( {1 ≤ u_1 ≤ N}, A.enum_c()⟨c, h⟩, X.enum_{j_x}()⟨j_x, g⟩ )
    REAL x
    x := (X.h_1())_{j_x:g}.deref_x()
    doany ⟨r, e, p, d⟩ ∈ ...⟩
            ... ⊖_{r=p=u_3} ( {1 ≤ u_3 ≤ N}, A_{c:h}.enum_r()⟨r, e⟩, P.enum_p()⟨p, d⟩ )
        q := (P_{p:d}).deref_q()
        u_2 := q
        if 1 ≤ u_2 ≤ N then
            i_y := u_2
            Y.declare_iter_{i_y}(k)
```

```
Y.search_{i_y}(i_y,k)
if Y.valid_{i_y}(k)
    REAL y
    y := Y_{i_y:k}.deref_y()
    ((A.h_1())_{c:h,r:d}).deref_a() := ...
            ... ((A.h_1())_{c:h,r:d}).deref_a() + x * y
```

Informally, this code performs the following steps:

```
doany j ∈ A ⋈ X
    doany i ∈ A_j ⋈ P
        i' := P_i
        if i' ∈ Y then
            A_{j,i} := A_{j,i} + Y_i' * X_j
```

How does this code compare with the previous? If the vector $\mathbf{Y}$ is stored using the SPVO format then the inclusion $i' \in \mathbf{Y}$ can be tested in $O(\log \eta_y)$ time. The overall complexity becomes:

$$O((\eta_c + \eta_x)\eta_r \log \eta_y) = O(\eta_c \eta_r \log \eta_y + \eta_x \eta_r \log \eta_y) \tag{5.57}$$

This is asymptotically faster than (5.56), since we have replaced the η_y multiplier by $\log \eta_y$.

We discuss the heuristics for choosing the order of the application of rewrite rules in Section 5.4.4.

Echelon form of the data access equation

Consider the following variation of matrix-vector product:

```
doany i=1..N; j=1..N
    if Y(i) ∧ X(j) ∧ A(i − j, j) then
        Y[i] := Y[i] + A[i-j,j] * X[j]
```

Assume that $\mathbf{Y}$ is dense, $\mathbf{A}$ and $\mathbf{X}$ are sparse. Moreover, $\mathbf{A}$ is stored using the CRS format. We can think of $\mathbf{A}$ as actually storing a matrix $\mathbf{A}$' that is compressed along the diagonals: the element (i,j) of $\mathbf{A}$' is stored as the element $(i-j,j)$ of $\mathbf{A}$. This is why we get this peculiar array access function "$(i-j,j)$".

Using the same schemata as in (5.44), the affine join form of the loop is:

```
doany ⟨i, j, r, c, a, i_y, y, j_x, x⟩ ∈ ...
        ... ⊖_{r=i−j ∧ c=j_x=j ∧ i_y=r} ( {1 ≤ i, j ≤ N}, A⟨r, c, a⟩, Y⟨i_y, y⟩, X⟨j_x, x⟩ )
    y := y + a * x
```

And the data access equation is:

$$\begin{pmatrix} r \\ c \\ j_x \\ i_y \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.58}$$

The hierarchy kinds for the relations are:

$$\begin{array}{lcl} \mathbf{A}\langle r,c,a\rangle & : & r \prec c \prec a \\ \mathbf{X}\langle j_x,x\rangle & : & j_x \prec x \\ \mathbf{Y}\langle i_y,y\rangle & : & i_y \prec y \end{array} \tag{5.59}$$

The fields r, j_x and i_y are exposed and no fields have been determined, yet. We can not join any of these fields together. Let's apply the JOIN rule to r. In order to bring the data access equation into the echelon form we have to add the first column of the matrix $\mathbf{F}$ to the second:

$$\begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{5.60}$$

The matrix $\mathbf{U}$ of the transformation and its inverse are:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad \mathbf{U}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \tag{5.61}$$

The new iteration vector $\mathbf{u} = (u_1 \ u_2)^T$ is:

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \mathbf{U}^{-1}\mathbf{i} = \begin{pmatrix} i-j \\ j \end{pmatrix} \tag{5.62}$$

which is not surprising, since we want the outer loop to enumerate over $r = i - j$. In compact form, the new data access equation is:

$$\begin{array}{lcl} r & = & u_1 \\ c & = & u_2 \\ j_x & = & u_2 \\ i_y & = & u_1 + u_2 \end{array} \tag{5.63}$$

The bounds $\mathcal{B}_1$ on u_1 are:

$$1 - N \le u_1 \le N - 1 \tag{5.64}$$

The bounds $\mathcal{B}_2$ on u_2 in terms of u_1 are:

$$\max(1, 1-u_1) \le u_2 \le \min(N, N-u_1) \tag{5.65}$$

The code after applying the JOIN rule becomes:

$$\begin{array}{l} \texttt{doany}\ \langle r, \mathbf{h}, u_1\rangle \in \ldots \\ \qquad \ldots \bigominus_{r=u_1} \left(\ \{1-N \le u_1 \le N-1\}, \mathbf{A}.\mathrm{enum}_r()\langle r,\mathbf{h}\rangle\ \right) \\ \quad \texttt{doany}\ \langle c, a, i_y, y, j_x, x, u_2\rangle \in \ldots \\ \qquad \ldots \bigominus_{\substack{c=j_x=u_2\wedge \\ i_y=u_1+u_2}} \left(\begin{array}{l} \{\max(1,1-u_1) \le u_2 \le \min(N, N-u_1)\}, \\ \mathbf{A}_{r:\mathbf{h}}\langle c,a\rangle, \mathbf{Y}\langle i_y,y\rangle, \mathbf{X}\langle j_x,x\rangle \end{array} \right) \\ \qquad \texttt{y := y + a * x} \end{array}$$

The index u_1 is invariant with respect to the inner loop. For this loop the data access equation is:

$$\begin{pmatrix} c \\ j_x \\ i_y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (u_2) + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ N \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{5.66}$$

Now all the remaining integer fields: c, j_x and i_y – have become joinable because all the rows of the new data access matrix are the same. The transformed code is:

$$
\begin{array}{l}
\texttt{doany}\ \langle r, \mathbf{h}, u_1 \rangle \in \ldots \\
\qquad \ldots \bigominus_{r=u_1} \left(\ \{1 - N \le u_1 \le N - 1\}, \mathbf{A}.\mathtt{enum}_r()\langle r, \mathbf{h}\rangle\ \right) \\
\quad \texttt{doany}\ \langle c, \mathbf{g}, j_x, \mathbf{d}, i_y, \mathbf{k}, u_2 \rangle \in \ldots \\
\qquad \ldots \bigominus_{\substack{c=j_x=u_2 \wedge \\ i_y = u_1 + u_2}} \left(\begin{array}{l} \{\max(1, 1 - u_1) \le u_2 \le \min(N, N - u_1)\}, \\ \mathbf{A}_{r:\mathbf{h}}.\mathtt{enum}_c()\langle c, \mathbf{g}\rangle, \\ \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{d}\rangle, \mathbf{Y}.\mathtt{enum}_{i_y}()\langle i_y, \mathbf{k}\rangle \end{array} \right) \\
\qquad \texttt{doany}\ \langle a, y, x \rangle \in \bigominus_{\mathtt{true}} \left(\ \mathbf{A}_{r:\mathbf{h}, c:\mathbf{g}}\langle a\rangle, \mathbf{Y}_{i_y:\mathbf{k}}\langle y\rangle, \mathbf{X}_{j_x:\mathbf{d}}\langle x\rangle\ \right) \\
\qquad\quad \texttt{y := y + a * x}
\end{array}
$$

After applying VALUE rule to the remaining fields, we get:

$$
\begin{array}{l}
\texttt{doany}\ \langle r, \mathbf{h}, u_1 \rangle \in \ldots \\
\qquad \ldots \bigominus_{r=u_1} \left(\ \{1 - N \le u_1 \le N - 1\}, \mathbf{A}.\mathtt{enum}_r()\langle r, \mathbf{h}\rangle\ \right) \\
\quad \texttt{doany}\ \langle c, \mathbf{g}, j_x, \mathbf{d}, i_y, \mathbf{k}, u_2 \rangle \in \ldots \\
\qquad \ldots \bigominus_{\substack{c=j_x=u_2 \wedge \\ i_y = u_1 + u_2}} \left(\begin{array}{l} \{\max(1, 1 - u_1) \le u_2 \le \min(N, N - u_1)\}, \\ \mathbf{A}_{r:\mathbf{h}}.\mathtt{enum}_c()\langle c, \mathbf{g}\rangle, \\ \mathbf{X}.\mathtt{enum}_{j_x}()\langle j_x, \mathbf{d}\rangle, \mathbf{Y}.\mathtt{enum}_{i_y}()\langle i_y, \mathbf{k}\rangle \end{array} \right) \\
\qquad \texttt{REAL a, x} \\
\qquad \texttt{a := } (\mathbf{A}_{r:\mathbf{h}, c:\mathbf{g}})\texttt{.deref()} \\
\qquad \texttt{x := } (\mathbf{X}_{j_x:\mathbf{d}})\texttt{.deref()} \\
\qquad (\mathbf{Y}_{i_y:\mathbf{k}})\texttt{.deref() := } (\mathbf{Y}_{i_y:\mathbf{k}})\texttt{.deref() + a * x}
\end{array}
$$

This example illustrates an important point: the more indices become invariant the more fields become joinable. The echelon form of the data access equation simply lets us keep track of the relationships between the invariants and the exposed fields.

5.4.4 Heuristics

Traditional database systems incorporate sophisticated methods of estimating the cost of query plans based on statistical properties of the relations and their storage. The System R [25] would enumerate all possible plans and then compare them according to their estimated cost. In our case the precise information about the sizes of the relations (i.e. tensors) is simply not available. We have to exploit the index hierarchy and the information available in the metrics provided by the Black Box Protocol.

Currently, our compiler prunes the space of possible plans by making heuristic choices described below. Usually, it leads to a single plan. If several plans are generated, then the heuristics described at the of this section are used to choose the best one. The pruning heuristics are base on the following considerations. First, consider the application of JOIN, TUPLE-ENUM, SEARCH, SINGLETON, VALUE and VALUE-UPDATE rules. These rules operate on the fields that appear in the top terms in index hierarchies of the relations. These fields can be classified as follows:

1. Exposed fields

(a) Joinable fields: exposed and not determined

(b) Searchable fields: exposed and determined

2. Fields in singleton tuple terms

3. Fields in stream tuple terms, other than exposed (more than one field in a tuple)

   (a) All fields in the tuple term are determined

   (b) Some fields in the tuple term are not determined

4. Fields in value terms

Each of these cases presents an opportunity for applying a rule. To make a deterministic decision we prioritize these situations as follows:

- 4 will result in the application of the VALUE or VALUE-UPDATE method. These should be applied as soon as they become available.
- 1b and 3a will result in the application of a search method. This should be done as early as possible: searches are costly and should not be performed deep in the loop nest.
- 2 is exploited next: "peeling off" a singleton tuple term does not cost anything, yet it potentially exposes more fields.
- 3b can only be resolved by enumerating over the relation. Invoking another rule might determine one or more fields of the tuple and make the enumeration more expensive or require a search.
- 1a is the last one in the order. The joinable fields can be partitioned into equivalence classes: each class corresponds to a one-dimensional affine join. It does not make sense to apply the JOIN rule only to a subset of a class, because the rest of the class becomes determined and will require search. So we always apply the JOIN rule one class at a time. When there are several classes, we pick the one with most fields. The idea is that "peeling off" these fields might expose more. When all classes have the same size, we break the ties arbitrarily.

Additional considerations are:

- Dense loops running over loop indices are used only after all of the fields have been determined. It is more efficient to place these loops innermost and to place joins and searches outermost.
- When applying the PICK-HIERARCHY rule, we try to pick those hierarchies that will enable most joins.

The above rules let us reduce the number of possible plans. Sometimes, as in the first example from Section 5.4.3, we can still arrive at more than a single plan and have to choose the best plan. Since we do not have a precise cost model of the plans, we have to make a heuristic choice. The following rule can pick the plan with lower asymptotic complexity:

- Pick the plan with fewer one-dimensional joins. Intuitively, in such plan a join is replaced by a search, just as in the example from Section 5.4.3. In many cases, the search takes constant time or logarithmic time.

These heuristics are by no means complete and we can still arrive at several plans that the compiler currently can not compare and has to make an arbitrary choice.

5.5 Join scheduling

5.5.1 Preliminaries

The query rewrite rules SEARCH, TUPLE-ENUM, SINGLETON, VALUE and VALUE-UPDATE generate code fragments that can be directly instantiated into concrete code. The results of the JOIN rule are a bit more complicated. This rule "peels off" loops that enumerate over the results of one-dimensional affine joins of the general form:

Program 5.9

doany $\langle u, f_1, \mathbf{h}_1, \ldots, f_m, \mathbf{h}_m \rangle \in \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}(\mathbf{s})\langle \mathbf{u} \rangle, \\ \mathbf{R}_1\langle f_1, \mathbf{h}_1 \rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m \rangle \end{pmatrix}$

    S(u,$\mathbf{h}_1$,...,$\mathbf{h}_m$)

enddo

where $\mathbf{f} = (f_1, \ldots, f_m)^T$. $\mathbf{z} = (z_1, \ldots, z_m)^T$ is the vector of coefficients that relate the loop index u and the fields in $\mathbf{f}$. The loop bounds $\mathcal{B}$ are parametrized by the invariants $\mathbf{s}$. $\mathbf{R}_k$ are the invocations of the `enum()` methods. $\mathbf{h}_k$ are the iterator values used to access relations in other parts of the code. The `enum()` methods do not expand into any concrete code. However, they can be implemented using either the corresponding stream methods `open()`, `next()`, etc. or the methods `lb()`, `ub()`, etc. that describe the special situations: DENSE and INDIRECT (Section 4.2).

We proceed as follows. First, we consider all the relations $\mathbf{R}_k$ in Program 5.9 that enumerate a dense region. These relations indirectly provide bounds for the index u. After eliminating these relations from the join we are left with a loop nest over another affine join, just like in Program 5.9. Second, we decide on the implementation strategy for this join. There are three basic strategies: HASH, MERGE and ENUMERATE-SEARCH. ENUMERATE-SEARCH is the simplest one: we pick a relation and search it for the value produced by other relations in the join. HASH join is an optimization on ENUMERATE-SEARCH: the relation being searched is inserted into a hash table. We discuss both of these strategies in Section 5.5.4. Optimizations that further improve the performance of the HASH join algorithm are described in Section 5.5.5. MERGE join algorithm is applicable when the relations can be enumerated in sorted order. We discuss the algorithm in Section 5.5.7.

5.5.2 Folding DENSE enumerations

Derivation

Without loss of generality, assume that the first k relations in Program 5.9 provide dense enumerations. In particular, the ADT for each of the relations $\mathbf{R}_j$, $1 \le j \le k$, provides the `lb()` and `ub()` methods that express the interval for the field f_j. Let L_j and U_j be the shorthand for the values returned by the methods:

$$L_j \le f_j \le U_j, \quad 1 \le j \le k \tag{5.67}$$

Let $\mathbf{g}_j^T$ be the j-th row of the matrix $\mathbf{G}$ in the data access equation of the query. Then each field f_j is expressed through the loop index u and the invariants $\mathbf{s}$:

$$f_j = z_j u + \mathbf{g}_j^T \mathbf{s} + f_j{}^{(0)}, \quad 1 \le j \le k \tag{5.68}$$

These equations together with the bounds in (5.67) refine the loop bounds:

$$\mathcal{B}_1(\mathbf{s}) = \left\{ u \;\middle|\; \begin{array}{l} u \in \mathcal{B}(\mathbf{s}) \wedge \\ L_1 \le z_1 u + \mathbf{g}_1^T \mathbf{s} + f_1{}^{(0)} \le U_1 \wedge \\ \ldots \wedge \\ L_k \le z_k u + \mathbf{g}_k^T \mathbf{s} + f_k{}^{(0)} \le U_k \wedge \end{array} \right\} \tag{5.69}$$

Using the new bounds $\mathcal{B}_1$ we can rewrite the loop as shown in Figure 5.10.

INPUT:

doany $\langle u, f_1, \mathbf{h}_1, \ldots, f_m, \mathbf{h}_m \rangle \in \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{R}_1\langle f_1, \mathbf{h}_1 \rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$

    S(u,$\mathbf{h}_1$,...,$\mathbf{h}_m$)

enddo

PREPROCESSING:

- $\mathbf{R}_1$ through $\mathbf{R}_k$ are dense enumerations.
- Let $f_1, \ldots, f_k$ be the dense fields.
- Let $\mathbf{f}' = \mathbf{f} \setminus \{f_1, \ldots, f_k\}$ be all the other fields.
- Let $\mathbf{f}' = u\mathbf{z}' + \mathbf{G}'\mathbf{s} + \mathbf{f}'^{(0)}$ be the data access equation for the fields $\mathbf{f}'$.
- Let $\mathcal{B}_1$ be the refined bounds given by (5.69).

TRANSFORMATION:

doany $\langle u, f_{k+1}, \mathbf{h}_{k+1}, \ldots, f_m, \mathbf{h}_m \rangle \in \ldots$

$\ldots \bigominus_{\mathbf{f}'=u\mathbf{z}'+\mathbf{G}'\mathbf{s}+\mathbf{f}'^{(0)}} \left(\begin{array}{l} \mathcal{B}_1(\mathbf{s})\langle u \rangle, \\ \mathbf{R}_{k+1}\langle f_{k+1}, \mathbf{h}_{k+1} \rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$

    f_1 := $z_1 u + \mathbf{g}_1^T \mathbf{s} + {f_1}^{(0)}$ *(see equation (5.68))*

    ...

    f_k := $z_k u + \mathbf{g}_k^T \mathbf{s} + {f_k}^{(0)}$

    S(u,f_1,...,f_k,$\mathbf{h}_{k+1}$,...,$\mathbf{h}_m$)

enddo

Figure 5.10: Folding DENSE enumerations

An example

We illustrate the above transformation on an example. The following loop nest computes the product $\mathbf{Y}$ of a sparse matrix $\mathbf{A}$ stored in the CRS format (Figure 1.12) and a vector $\mathbf{X}$ stored in some sparse vector format:

Program 5.10

abstract $\mathbf{A}$(CRS n=N ...)

real $\mathbf{Y}$[N]

abstract $\mathbf{X}$(....)

doany $\langle i, j, r, c, a, i_y, y, j_x, x \rangle \in \ldots$

$\ldots \bigominus_{\substack{i=r=i_y \wedge \\ j=c=j_x}} \left(\begin{array}{l} \{1 \le i, j \le N\}, \\ \mathbf{A}\langle r, c, a \rangle, \mathbf{Y}\langle i_y, y \rangle, \mathbf{X}\langle j_x, x \rangle \end{array} \right)$

    y := y + a * x

```
enddo
```

Query scheduling would produce the following code:

Program 5.11

doany $\langle i, r, \mathbf{h}, i_y, \mathbf{g}\rangle \in \bigominus_{i=r=i_y} \begin{pmatrix} \{1 \le i \le n\}, \\ \mathbf{A}.\mathtt{enum}_r()\langle r, \mathbf{h}\rangle, \\ \mathbf{Y}.\mathtt{enum}_{i_y}\langle i_y, \mathbf{g}\rangle \end{pmatrix}$

  doany $\langle j, c, \mathbf{d}, j_x, \mathbf{e}\rangle \in \bigominus_{j=c=j_x} \begin{pmatrix} \{1 \le j \le N\}, \\ \mathbf{A}_{r:\mathbf{h}}.\mathtt{enum}_c()\langle c, \mathbf{d}\rangle, \\ \mathbf{X}.\mathtt{enum}_{j_x}\langle j_x, \mathbf{e}\rangle \end{pmatrix}$

    $\mathbf{Y}_{i_y:\mathbf{g}}$.deref() := $\mathbf{Y}_{i_y:\mathbf{g}}$.deref() +

      + $\mathbf{A}_{r:\mathbf{h},c:\mathbf{d}}$.deref() * $\mathbf{X}_{j_x:\mathbf{e}}$.deref()

Notice that the join in the outer loop involves dense enumerations on $\mathbf{A}$ and $\mathbf{Y}$. r is constrained to be in the range $1 \le r \le N$ by the CRS black box. Similarly, i_y is constrained to be in the range $1 \le i_y \le N$. Altogether, we have the following constraints on the fields and the loop index i:

$$\begin{array}{l} 1 \le i \le N \\ 1 \le r = i \le N \\ 1 \le i_y = i \le N \end{array} \tag{5.70}$$

The resulting loop bounds are (just as before):

$$1 \le i \le N \tag{5.71}$$

and the transformed code is:

Program 5.12

doany $\langle i\rangle \in 1..N$

  r := i

  $\mathtt{i}_y$:= i

  doany $\langle j, c, \mathbf{d}, j_x, \mathbf{e}\rangle \in \bigominus_{c=j_x=j} \begin{pmatrix} \{1 \le j \le N\}, \\ \mathbf{A}_{r:r}.\mathtt{enum}_c()\langle c, \mathbf{d}\rangle, \\ \mathbf{X}.\mathtt{enum}_{j_x}\langle j_x, \mathbf{e}\rangle \end{pmatrix}$

    $\mathbf{Y}_{i_y:i_y}$.deref() := $\mathbf{Y}_{i_y:i_y}$.deref() +

      + $\mathbf{A}_{r:r,c:\mathbf{d}}$.deref() * $\mathbf{X}_{j_x:\mathbf{e}}$.deref()

  enddo

enddo

5.5.3 Simplifying loop bounds

Consider the inner loop in Program 5.12. Given the declaration of the matrix $\mathbf{A}$ in Program 5.10, the bound on the row and column indices that are stored in $\mathbf{A}$ are:

$$1 \le r, c \le N \tag{5.72}$$

Since the data access equation for the loop constraints the loop index j to be equal to the column index c, we know that we can never enumerate the loop index j that is out of bounds. The following loop enumerates over the same set of indices:

Program 5.13

...

doany $\langle j, c, \mathbf{d}, j_x, \mathbf{e}\rangle \in \bigominus_{c=j_x=j} \left(\underline{\mathbb{Z}\langle j\rangle}, \mathbf{A}_{r:r}.\mathtt{enum}_c()\langle c, \mathbf{d}\rangle, \mathbf{X}.\mathtt{enum}_{j_x}\langle j_x, \mathbf{e}\rangle \right)$

...

enddo

The new (underlined) bounds on j simply restrict it to be an integer. The benefit of removing the bounds is that we do not need to generate bounds checks in join implementations (see Sections 5.5.4 through 5.5.7, below).

In general, suppose we have a loop nest over 1D affine join:

Program 5.14

doany $\langle u, f_1, \mathbf{h}_1, \ldots, f_m, \mathbf{h}_m\rangle \in \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{Gs}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u\rangle, \\ \mathbf{R}_1\langle f_1, \mathbf{h}_1\rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m\rangle \end{array} \right)$

...

The loop enumerates those iterations u that belong to the intersection of two sets (one is the loop bounds):

$$\begin{aligned} & u \in \mathcal{B}(\mathbf{s}) \cap \mathcal{D}(\mathbf{s}) \\ & \mathcal{D}(\mathbf{s}) = \left\{ u | \exists \mathbf{f} \in \mathcal{C}(\mathbf{s}) : \mathbf{f} = u\mathbf{z} + \mathbf{Gs} + \mathbf{f}^{(0)} \right\} \end{aligned} \tag{5.73}$$

where $\mathcal{C}(\mathbf{s})$ are the bounds on the fields $\mathbf{f}$ as given by the `bounds()` metric of the relevant Black Boxes. The set $\mathcal{D}(\mathbf{s})$ contains those iterations that access tensors within bounds. We would like to simplify the constraints that define the loop bounds $\mathcal{B}(\mathbf{s})$ as much as possible, given that we are assured that the loop index u always falls within the set $\mathcal{D}(\mathbf{s})$. Formally, we have to find another set $\mathcal{B}'(\mathbf{s})$ such that:

$$\mathcal{B}'(\mathbf{s}) \cap \mathcal{D}(\mathbf{s}) = \mathcal{B}(\mathbf{s}) \cap \mathcal{D}(\mathbf{s}) \tag{5.74}$$

and the constraints that define $\mathcal{B}'$ are in some sense "simpler" than the constraints for the set $\mathcal{B}$. The set $\mathcal{B}'$ is called the *gist of $\mathcal{B}$ given $\mathcal{D}$* [69]. Intuitively, $\mathcal{B}'$ contains the same information given that $\mathcal{D}$ is satisfied. The algorithm for computing the gist of set is described in [69]. In our example, the set $\mathcal{B}$ is given by:

$$\mathcal{B}(N) = \{u | 1 \le u \le N\} \tag{5.75}$$

And the set $\mathcal{D}$ is:

$$\mathcal{D}(N) = \{u | \exists r, i_y : u = r = i_y \wedge 1 \le r, i_y \le N\} = \{u | 1 \le u \le N\} \tag{5.76}$$

The gist is just a tautology:

$$\mathcal{B}' = \{u | True\} \tag{5.77}$$

since:

$$\{u | True\} \cap \{u | 1 \le u \le N\} = \{u | 1 \le u \le N\} \cap \{u | 1 \le u \le N\} = \{u | 1 \le u \le N\} \tag{5.78}$$

INPUT:

$$\texttt{doany}\ \langle u, f_1, \mathbf{h}_1, \ldots, f_m, \mathbf{h}_m \rangle \in \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{R}_1\langle f_1, \mathbf{h}_1 \rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

...

PREPROCESSING:

- Let $calD(\mathbf{s}) = \left\{ u | \exists \mathbf{f} \in \mathcal{C}(\mathbf{s}) : \mathbf{f} = u\mathbf{z} + \mathbf{G}\mathbf{s} + \mathbf{f}^{(0)} \right\}$
- Let $\mathcal{B}'(\mathbf{s})$ be the gist of $\mathcal{B}$ given $\mathcal{D}$.

TRANSFORMATION:

$$\texttt{doany}\ \langle u, f_1, \mathbf{h}_1, \ldots, f_m, \mathbf{h}_m \rangle \in \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}'(\mathbf{s})\langle u \rangle, \\ \mathbf{R}_1\langle f_1, \mathbf{h}_1 \rangle, \ldots, \mathbf{R}_m\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

...

Figure 5.11: Simplifying loop bounds

5.5.4 Rewrite rules for ENUMERATE-SEARCH and HASH join algorithms

Code generation of the ENUMERATE-SEARCH and HASH join algorithms can be described through simple rewrite rules. We start with ENUMERATE-SEARCH algorithm. Consider the loop enumerating over the results of a one-dimensional affine join:

$$\begin{array}{l} \texttt{doany}\ \langle \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m \rangle \in \ldots \\ \qquad \ldots \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\texttt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_m.\texttt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{array} \right) \\ \quad \texttt{S}(\mathbf{s}, u, \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m) \end{array}$$

where $\mathbf{s}$ is the vector of invariants, u is the loop index associated with the join, $\mathbf{f}$ is the vector of all integer fields f_k, $1 \leq k \leq m$, enumerated in the join and $\mathbf{h}_k$, $1 \leq k \leq m$, are the iterators produced by the `enum` methods. $\mathcal{B}(\mathbf{s})$ are the loop bounds (parametrized by the invariants).

Without loss of generality, we can pick the relation $\mathbf{A}_1$ to be searched into and transform the code as shown in Figure 5.12 (page 112): we join the rest of the relations and then search into $\mathbf{A}_1$. To lower the cost of the search we can store the iterators $\mathbf{h}_1$ in a hash table $\mathbf{T}$ indexed by the value of the field f_1. The corresponding transformation is illustrated in Figure 5.13. If the matrix $\mathbf{A}_1$.`search_dense_p` returns `true` then we can omit the conditional on line 6 in Figure 5.12 and on line 10 in Figure 5.13: we know that the search always succeeds as long as the array access is in bounds.

Notice that the hash table is represented by the abstract variable $\mathbf{T}$. The formal arguments to the `HASH_TABLE` ADT are:

- `lb` is the lower bound of the key
- `ub` is the upper bound of the key
- `width` is the size of the iterator to be stored in the table (see Section 4.3)

The `HASH_TABLE` ADT has to provide the following methods:

- $\mathbf{T}$.`decls` generates the declarations of the concrete variables that implement the table
- $\mathbf{T}$.`init` initializes the table

- $\mathbf{T}.\mathtt{put}(i, \mathbf{h})$ inserts the iterator $\mathbf{h}$ into the has table.
- $\mathbf{T}.\mathtt{get}(i, \mathbf{g})$ find the iterator corresponding to the key (loop index) i. If the index is not stored in the table, then the iterator is set to an invalid value.
- $\mathbf{T}.\mathtt{clear}$ clears the table.

The `init` method takes time proportional to the size of the data structure for the hash table, which can be as large as the the size of the range of the keys (i.e. $O(\beta - \alpha)$). However, the `init` method needs to be called only once for the duration of the program. The `clear` method can be used in order to reset the table between uses. We assume that the `clear` method takes $O(\eta)$ time, where η is the number of entries in the table.

In sparse matrix computations a dense array usually plays the role of the hash table. The iterators are stored in an array of integers of size $\mathtt{ub}() - \mathtt{lb}() + 1) * \mathtt{width}()$. The definition of the HASH_TABLE_DENSE ADT is shown in Figure 5.14. The *link* array and the *last* variable keep the linked list of entries in the table. This way we can perform the `clear` method in $O(\eta)$ time. The implementation assumes that for each index i the `put` method is called only once – otherwise the linked list will be in an inconsistent state.

Let's this all together on an example. Consider the loop nest that computes the dot product of two vectors $\mathbf{X}$ and $\mathbf{Y}$:

$$\mathtt{doany}\ \langle i, i_x, v_x, i_y, v_y \rangle \in \mathop{\Theta}_{i=i_x=i_y} \left(\ \{1 \le i \le N\}, \mathbf{X}\langle i_x, v_x\rangle, \mathbf{Y}\langle i_y, v_y\rangle \ \right)$$
$$\quad \mathtt{sum\ :=\ sum\ +\ v}_x \ \mathtt{*}\ \mathtt{v}_y$$

Assume that the vectors are stored using the `SPV0` format (Figure 3.1). The result of applying query scheduling to this program is:

$$\mathtt{doany}\ \langle i, i_x, \mathbf{h}, i_y, \mathbf{g} \rangle \in \mathop{\Theta}_{i=i_x=i_y} \left(\ \{1 \le i \le N\}, \mathbf{X}\langle i_x, \mathbf{h}\rangle, \mathbf{Y}\langle i_y, \mathbf{g}\rangle \ \right)$$
$$\quad \mathtt{sum\ :=\ sum\ +\ } \mathbf{X}_{i_x:\mathbf{h}}.\mathtt{deref}()\ \mathtt{*}\ \mathbf{Y}_{i_y:\mathbf{g}}.\mathtt{deref}()$$

If we use HASH join implementation, then the concrete program that we obtain is:

```
integer link[0:N]
integer last
integer buf[N]
integer iix iterator into X
integer iiy iterator into Y

do iix = 1, NY
    i := IX[iix]
    buf[i] := iix
    link[last] := i
    last := i
enddo

do iiy = 1, NY
    i := IY[iiy]
    iix := buf[i]
    if (iix ≠ 0) then
```

```
18.           sum := sum + VY[iiy] * VX[iix]
19.       endif
20.   enddo
21.
22.   i := link[0]
23.   while (link[i] ≠ 0) do
24.       j := link[i]
25.       link[i] := 0
26.       buf[i] := 0
27.       i := j
28.       enddo
29.   last := 0
30.   link[0] := 0
```

The loop starting on line 7 fills the hash table with the entries from $\mathbf{X}$. The loop starting on line 14 performs the actual computation. The code starting with line 22 cleans up the hash table.

5.5.5 SCATTER and GATHER

The experiments on IBM SP-2 have shown that the above code under performs the *scatter* implementation of sparse dot product found in textbooks and libraries (see [66]). This implementation stores the values of the elements of $\mathbf{X}$ in the dense array, as opposed to storing the offsets (iterators) into the data structure for $\mathbf{X}$. Instead of the code

```
buf[i] := iix
```

on line 9, the code in the scatter version is:

```
buf[i] := VX[iix]
```

(assuming `buf` is re-declared as array of reals). And instead of the loop on line 14, the scatter version has the loop:

```
do iiy = 1, NY
    i := IY[iiy]
    sum := sum + VY[iiy] * buf[i]
enddo
```

Notice the absence of the conditional and of the extra level of indirection into $\mathbf{X}$:

```
iix := buf[i]
if ....
    ...VX[iix]...
```

The difference in performance is quite substantial. The loop on line 14 is two times slower than the corresponding loop in the scatter version. Overall our implementation is about 20% slower.

We can remedy this situation by recognizing and optimizing for such situations. What are the characteristics of the code that we should look for? Without loss of generality, we start with a join involving a relation $\mathbf{A}_1$ with the hierarchy kind $f_1^* \prec v_1^1$:

BEFORE:

$$\texttt{doany}\ \langle \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m \rangle \in \ldots$$

$$\ldots \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\texttt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_m.\texttt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{pmatrix}$$

$$\qquad \texttt{S}(\mathbf{s}, u, \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m)$$

AFTER:

$$\mathbf{A}_1.\texttt{declare\_iter}_{f_1}(\mathbf{h})$$

$$\texttt{doany}\ \langle \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m \rangle \in \ldots$$

$$\ldots \bigominus_{\mathbf{f}'=u\mathbf{z}'+\mathbf{G}'\mathbf{s}+\mathbf{f}'^{(0)}} \begin{pmatrix} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\texttt{enum}_{f_2}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_1.\texttt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{pmatrix}$$

$$\qquad \texttt{f}_1\ \texttt{:=}\ z_1 u + \mathbf{G}_1^T \mathbf{s} + {f_1}^{(0)}$$

$$\qquad \mathbf{A}_1\texttt{.search}_{f_1}\texttt{(}\ \texttt{f}_1\texttt{,}\ \mathbf{h}_1\ \texttt{)}$$

$$\qquad \texttt{if (}\mathbf{A}\texttt{.valid}_{f_1}\texttt{(}\mathbf{h}_1\texttt{) then}$$

$$\qquad\qquad \texttt{S}(\mathbf{s}, u, \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m)$$

$$\qquad \texttt{endif}$$

WHERE:

- $\mathbf{f}' = \mathbf{f} \setminus \{f_1\}$
- $\mathbf{z}' = \mathbf{z} \setminus \{z_1\}$
- $\mathbf{G}_1^T$ is the first row of $\mathbf{G}$. $\mathbf{G}'$ is the rest of $\mathbf{G}$.

Figure 5.12: Transformation for ENUMERATE-SEARCH algorithm

BEFORE:

doany $\langle \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\mathtt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_m.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{pmatrix}$$

    S($\mathbf{s}, u, \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m$)

AFTER:

$\mathbf{A}_1$.declare_iter$_{f_1}$($\mathbf{h}$)

abstract $\mathbf{T}$(HASH_TABLE lb=α ub=β width=($\mathbf{h}$.size()))

$\mathbf{T}$.decls(); $\mathbf{T}$.init(); $\mathbf{A}_1$.open($\mathbf{h}$)

while ($\mathbf{A}_1$.valid($\mathbf{h}$)) do

    f$_1$:= $\mathbf{A}$.deref$_{f_1}$($\mathbf{h}_1$); $\mathbf{T}$.put(f$_1$, $\mathbf{h}_1$); $\mathbf{A}_1$.next($\mathbf{h}_1$)

enddo

doany $\langle \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{f}'=u\mathbf{z}'+\mathbf{G}'\mathbf{s}+\mathbf{f}'^{(0)}} \begin{pmatrix} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\mathtt{enum}_{f_2}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_1.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{pmatrix}$$

    f$_1$:= $z_1 u + \mathbf{G}_1^T \mathbf{s} + {f_1}^{(0)}$; $\mathbf{T}$.get(f$_1$, $\mathbf{h}_1$)

    if ($\mathbf{A}_1$.valid$_{f_1}$($\mathbf{h}_1$) then

        S($\mathbf{s}, u, \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m$)

    endif

WHERE:

- α and β are the lower and upper bounds on the field f_1 as determined from the `bounds` metric (Section 4.2).
- $\mathbf{f}' = \mathbf{f} \setminus \{f_1\}$; $\mathbf{z}' = \mathbf{z} \setminus \{z_1\}$; $\mathbf{G}_1^T$ is the first row of $\mathbf{G}$. $\mathbf{G}'$ is the rest of $\mathbf{G}$.

Figure 5.13: Transformation for HASH algorithm

```
n() ≡= Expr ⟨ ub() - lb() + 1 ⟩
decls() ≡= Decls ⟨
    integer $buf[n() * width()]
    integer $link[0:n()]
    integer $last ⟩
init() ≡= Stmt ⟨
    $last := 0
    do i = 0, n()
        $link[i] := 0
    enddo ⟩
put(i,h) ≡= Stmt ⟨
    h.marshal($buf, (i-lb())*width()+1)
    $link[$last] := i; $last=i ⟩
get(i,h) ≡= Stmt ⟨
    h.unmarshal($buf, (i-lb())*width()+1) ⟩
clear() ≡= Stmt ⟨
    integer $i, $j
    $i := $link[0]
    while ($link[$i] ≠ 0) do
        $j := $link[$i]; $link[$i] := 0; $i := $j
    enddo
    $last := $link[0] := 0 ⟩
```

Figure 5.14: HASH_TABLE_DENSE ADT

`doany` $\ldots \in \bigominus_{\ldots} \left(\ldots \mathbf{A}_1.\texttt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle \ldots \right)$

  `S(`$\ldots \mathbf{A}_{1,f_1:\mathbf{h}_1}.\texttt{deref}() \ldots$`)`

There are several cases, depending on the position of the reference $\mathbf{A}_{1,f_1:\mathbf{h}_1}.\texttt{deref}()$ within the statement `S`:

- The reference does not appear on the left-hand side in an assignment. Moreover, partial evaluation of the statement `S` with $\mathbf{A}_{1,f_1:\mathbf{h}_1}.\texttt{deref}()$ set to 0.0 produces an no-op. In our dot product example the statement becomes:

  `sum := sum + 0.0 *` $\mathbf{Y}_{i_y:\mathbf{g}}.\texttt{deref}()$

  which is equivalent to:

  `sum := sum`

  In this case we can store the values of $\mathbf{A}_1$ in an array of real values and eliminate the test on line 17 as shown in Figure 5.15. We call this implementation SCATTER join. The `SCATTERV` ADT is defined in Figure 5.16. In the dot product example the resulting code is equivalent to that found in textbooks and libraries:

```
 1.  integer link[0:N]
 2.  integer last
 3.  real buf[N]
 4.  integer iix iterator into X
 5.  integer iiy iterator into Y
 6.
 7.  do iix = 1, NY
 8.      i := IX[iix]
 9.      buf[i] := VX[iix]
10.      link[last] := i
11.      last := i
12.  enddo
13.
14.  do iiy = 1, NY
15.      i := IY[iiy]
16.      sum := sum + VY[iiy] * buf[i]
17.  enddo
18.
19.  i := link[0]
20.  while (link[i] ≠ 0) do
21.      j := link[i]
22.      link[i] := 0
23.      buf[i] := 0.0
24.      i := j
25.      enddo
26.  last := 0
27.  link[0] := 0
```

- The statement `S` *updates* the value of $\mathbf{A}_{1,f_1:\mathbf{h}_1}.\texttt{deref}()$:

BEFORE:

doany $\langle \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\mathtt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_m.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

$\quad$ S$(\mathbf{s}, u, \mathbf{f}, \mathbf{A}_{1,f_1:\mathbf{h}_1}.\mathtt{deref}(), \mathbf{h}_2, \ldots, \mathbf{h}_m)$

AFTER:

$\mathbf{A}_1$.declare_iter$_{f_1}$($\mathbf{h}$)

abstract $\mathbf{T}$(SCATTERV lb=α ub=β)

$\mathbf{T}$.decls(); $\mathbf{T}$.init(); $\mathbf{A}_1$.open($\mathbf{h}$)

while ($\mathbf{A}_1$.valid($\mathbf{h}$)) do

$\quad$ v$_1$:= $\mathbf{A}_1$.deref$_{f_1}$($\mathbf{h}_1$); $\mathbf{T}$.put(f$_1$, $\mathbf{h}_1$); $\mathbf{A}_1$.next($\mathbf{h}_1$)

enddo

doany $\langle \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{f}'=u\mathbf{z}'+\mathbf{G}'\mathbf{s}+\mathbf{f}'^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\mathtt{enum}_{f_2}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \\ \ldots, \mathbf{A}_1.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

$\quad$ f$_1$:= $z_1 u + \mathbf{G}_1^T \mathbf{s} + {f_1}^{(0)}$

$\quad$ S$(\mathbf{s}, u, \mathbf{f}', \mathbf{T}.\mathtt{get}(f_1), \mathbf{h}_2, \ldots, \mathbf{h}_m)$

WHERE: (see Figure 5.13 for definitions)

Figure 5.15: Transformation for SCATTER algorithm

$\mathbf{A}_{1,f_1:\mathbf{h}_1}.\mathtt{deref}()$:= $\mathbf{A}_{1,f_1:\mathbf{h}_1}.\mathtt{deref}()$ + Delta

and this reference does not appear in the expression Delta. We can then accumulate the updates Delta in an array of reals and then store the cumulative result back into $\mathbf{A}$. We call this GATHER join. The code is shown in Figure 5.17. The loop starting on line 8 accumulates the updates in the array and the loop on line 14 adds the updates back to $\mathbf{A}_1$.

5.5.6 Single-relation join

The rewrite rules for the ENUMERATE-SEARCH and HASH algorithm remove the relations one-by-one from the original join. Consequently, there is a point when we are left with a one-dimensional affine-join that enumerates over a single relation:

Program 5.15
doany $\langle u, f, \mathbf{h} \rangle \in \bigominus_{f=zu+\mathbf{g}^T\mathbf{s}+f^{(0)}} \left(\mathcal{B}(\mathbf{s})\langle u \rangle, \mathbf{A}.\mathtt{enum}_f\langle f, \mathbf{h} \rangle \right)$
$\quad$ S(s,u,$\mathbf{h}$)

This single-relation join is translated into the following code:

```
n() ≡= Expr ⟨ ub() - lb() + 1 ⟩
decls() ≡= Decls ⟨
    real $buf[n()]
    integer $link[0:n()]
    integer $last ⟩
init() ≡= Stmt ⟨
    $last := 0
    do i = 0, n()
        $link[i] = 0
        $buf[i] = 0.0
    enddo ⟩
put(i,v) ≡= Stmt ⟨ $buf[i] := v ⟩
get(i) ≡= Expr ⟨ $buf[i] ⟩
clear() ≡= Stmt ⟨
    integer $i, $j
    $i = link[0]
    while ($link[$i] ≠ 0) do
        $buf[i] := 0.0
        $j = $link[$i]; $link[$i] = 0; $i = $j
    enddo
    $last = $link[0] = 0 ⟩
```

Figure 5.16: SCATTERV ADT

BEFORE:

doany $\langle \mathbf{f}, \mathbf{h}_1, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{f}=u\mathbf{z}+\mathbf{G}\mathbf{s}+\mathbf{f}^{(0)}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_1.\mathtt{enum}_{f_1}()\langle f_1, \mathbf{h}_1 \rangle, \ldots, \mathbf{A}_m.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

    S$(\mathbf{s}, u, \mathbf{f}, \mathbf{A}_{1,f_1:\mathbf{h}_1}$.deref$(), \mathbf{h}_2, \ldots, \mathbf{h}_m)$

AFTER:

$\mathbf{A}_1$.declare_iter$_{f_1}$($\mathbf{h}$)

abstract $\mathbf{T}$(SCATTERV lb=α ub=β)

$\mathbf{T}$.decls(); $\mathbf{T}$.init(); $\mathbf{A}_1$.open($\mathbf{h}$)

while ($\mathbf{A}_1$.valid($\mathbf{h}$)) do

    f$_1$:= $\mathbf{A}_1$.deref$_{f_1}$($\mathbf{h}_1$); v$_1$:= $\mathbf{A}_{1,f_1:\mathbf{h}_1}$.deref();

    $\mathbf{T}$.put(f$_1$, $\mathbf{h}_1$); $\mathbf{A}_1$.next($\mathbf{h}_1$)

endd

doany $\langle \mathbf{f}', \mathbf{h}_2, \ldots, \mathbf{h}_m \rangle \in \ldots$

$$\ldots \bigominus_{\substack{\mathbf{f}'=u\mathbf{z}'+ \\ +\mathbf{G}'\mathbf{s}+\mathbf{f}'^{(0)}}} \left(\begin{array}{l} \mathcal{B}(\mathbf{s})\langle u \rangle, \\ \mathbf{A}_2.\mathtt{enum}_{f_2}()\langle f_2, \mathbf{h}_2 \rangle, \ldots, \mathbf{A}_m.\mathtt{enum}_{f_m}()\langle f_m, \mathbf{h}_m \rangle \end{array} \right)$$

    f$_1$:= $z_1 u + \mathbf{G}_1^T \mathbf{s} + {f_1}^{(0)}$

    $\mathbf{T}$.get(f$_1$) := $\mathbf{T}$.get(f$_1$) + Delta

enddo

$\mathbf{A}_1$.open($\mathbf{h}$)

while ($\mathbf{A}_1$.valid($\mathbf{h}$)) do

    f$_1$:= $\mathbf{A}_1$

    $\mathbf{A}_{1,f_1:\mathbf{h}_1}$.deref() := $\mathbf{A}_{1,f_1:\mathbf{h}_1}$.deref() + $\mathbf{T}$.get(f$_1$)

enddo

$\mathbf{T}$.clear()

WHERE: (see Figure 5.13 for definitions)

Figure 5.17: Transformation for GATHER algorithm

Program 5.16

```
A.declare_iter()_f(h)
A.open_f(h)
while (A.valid_f(h)) do
    f := A.deref_f(h)
    if (f is divisible by z) then
        u := (f − g^T s + f^(0))/z
        if (u ∈ B(s)) then
            S(s,u,h)
        endif
    endif
    A.next_f(h)
enddo
A.close_f(h)
```

5.5.7 MERGE join

Consider an affine join of the enumerations on the relation **A** and **B**:

$$\texttt{doany}\ \langle i, a, \mathbf{h}, b, \mathbf{g}\rangle \in \bigominus_{a=\alpha(i,\mathbf{s}) \wedge b=\beta(i,\mathbf{s})} \left(\begin{array}{l} \{L(\mathbf{s}) \le i \le U(\mathbf{s})\}, \\ \mathbf{A}.\mathtt{enum}_a()\langle a, \mathbf{h}\rangle, \mathbf{B}.\mathtt{enum}_b()\langle b, \mathbf{g}\rangle \end{array} \right)$$

```
    S(i,a,b,h,g)
```

where L and U are expressions in terms of the invariants $\mathbf{s}$. α and β are affine functions of the loop iteration i and the invariants. The enumerations $\mathbf{A}.\mathtt{enum}_a$ and $\mathbf{B}.\mathtt{enum}_b$ produce streams of values of the fields a and b. We can also think of the enumerations as producing streams of the values of the loop index i. If the streams of the loop index are sorted, then we can merge them using the well known "two-finger" merge algorithm.

Let the functions α and β be defined as:

$$\begin{array}{l} a = \alpha(i, \mathbf{s}) = zi + \mathbf{w}^T\mathbf{s} + a_0 \\ b = \beta(i, \mathbf{s}) = xi + \mathbf{v}^T\mathbf{s} + b_0 \\ x, z, \neq 0 \end{array} \tag{5.79}$$

The enumerations for a and b can be three kinds:

- Unordered. In this case MERGE join in not applicable.
- Uni-directional. Only the enumeration in one direction – ascending or descending – is possible for either field. If the coefficients z and x in (5.79) have the same sign, then the streams of a's and b's can be converted into the two streams of i's that are ordered in the same way. Then the MERGE join is possible.
- Bi-directional. MERGE join is always possible.

Consider the case when both x and z are positive and the fields a and b can be enumerated in ascending order. The code for MERGE join is shown in Figure 5.18. For other combinations of signs of x and y the calls to `next` methods on lines 11 or 14 have to be replaced with the calls to `prev` methods. On line 219 the position indicator (zero in this case) has to be set accordingly (see Section 4.2.2). This algorithm can be easily extended to multiple (more than 2) relations. We leave the details to the reader.

```
A.declare_iter_a(h); B.declare_iter_b(g)
A.open_a(h,0); B.open_b(g,0)
while (A.valid(h) ∧ B.valid(g)) do
    find i_a such that a = α(i_a, s)
    find i_b such that b = α(i_b, s)
    if both found then
        if (i_a = i_b) then
            S(i_a,a,b,h,g)
        endif
        if (i_a ≤ i_b) then
            A.next_a(h)
        endif
        if (i_b ≤ i_a) then
            B.next_b(g)
        endif
    endif
enddo
```

Figure 5.18: MERGE join of two relations

5.5.8 Heuristics

So far, we have discussed the join implementation strategies available without discussing their relative merits. There are two sources of non-determinism in join scheduling: the choice of strategy and the choice of the relation to base the strategy on (in the case of HASH and ENUMERATE-SEARCH joins). Here are some considerations that we have used when implementing the heuristic for resolving the non-determinism:

- HASH join[1] and MERGE join have the same asymptotic complexity (ignoring the initial cost of initializing hash tables). Their actual comparative performance depends on the architecture. However, HASH join is orders of magnitude faster when the cost of filling in the table and clearing the table can be amortized over several executions of an outer loop. In this case the performance of the loop that uses the hash table (line 7 in Figure 5.13, page 113) dominates the cost and this is loop is much faster than the merge loop (Figure 5.18).

  On the other hand, there are situations when we do not have enough memory to allocate the table which is just a dense vector in sparse matrix computations. This is especially true in parallel code, when we avoid allocating data structures with size proportional to the size of the overall (distributed) problem.

  Currently, we follow the following rule: for sequential code HASH joins are used and MERGE joins are used in parallel code.

- ENUMERATE-SEARCH join is applicable when one of the relations provides an $O(1)$-time search method (as returned by the `search_cost` metric (Section 4.2.2). In fact, when a relation has a $O(1)$-time search method, we use it as an "invitation" to use the ENUMERATE-SEARCH algorithm based on this relation, even if MERGE join is possible.

- When ENUMERATE-SEARCH is unavoidable we pick the relation with the lowest cost of search (base on `search_cost` metric). Ties are arbitrarily broken.

Unlike commercial relational database systems, our compiler does not have the statistics about the actual sizes of the relations. Therefore, only "rule of thumb" choices are possible.

5.6 Instantiation

After query scheduling and join scheduling we are left with the abstract program that can be *instantiated* into concrete code by expanding the definitions of the ADT methods. The basic instantiation procedure is quite simple. The compiler walks the abstract syntax tree (AST) of the program in depth-first order and expands the methods as it encounters them. The result of an expansion is processed again. When the user (through complier flags) requests BLAS pattern-matching (Section 5.8.1 below) the compiler does not expand the instances of the `DENSE_MATRIX_ELT` ADT.

5.7 Projection views

There are cases when we want the schema of the relation stored in the data structure to be something other than the usual $\langle i, j, value\rangle$. Consider the block formats described in more detail in Section 6.1. The Black Boxes for these formats expose the $\langle bi, bj, i, j, value\rangle$ schema, where bi and bj are block row and column indices. First of all, this schema is used in specifying more complex DOACROSS algorithms. Second, specifying these formats with the $\langle i, j, value\rangle$ schema would require rather complicated iterators and address computations.

The good news is that we can declare a *projection view* with the more natural schema as:

$$\mathbf{A}\langle i, j, v\rangle = \pi_{i,j,v}\mathbf{B}\langle bi, bj, i, j, v\rangle \tag{5.80}$$

[1] We include SCATTER and GATHER join in this category, as well

and then use this view in the program code:

Program 5.17

```
abstract B(BSR ...  )
projection A⟨i,j,v⟩ of B⟨bi,bj,i,j,v⟩
...
doany i=1,n; j=1,n
    ...A[i,j]...
```

The projection in (5.80), in general, has to eliminate duplicates. More precisely, the loop:

Program 5.18
doany $\langle i, j, v\rangle \in \pi_{i,j,v}\mathbf{B}\langle bi, bj, i, j, v\rangle$

```
    S(i,j,v)
```

has to execute the statement `S` only for unique values of the tuple $\langle i, j, v\rangle$. Otherwise, we would execute the statement twice and compute a different output. However, in the examples of the block formats we can safely drop the projection:

Program 5.19
doany $\langle bi, bj, i, j, v\rangle \in \mathbf{B}\langle bi, bj, i, j, v\rangle$

```
    S(i,j,v)
```

The statement `S` is guaranteed to be executed only for unique value of its inputs because there is a *functional dependence* from the tuple $\langle bi, bj\rangle$ to $\langle i, j, v\rangle$ [87]:

Definition 5.3 *Let* $\mathbf{R}$ *be a relation. Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be some fields (or tuples of fields) of the relation. Moreover, for any tuples* $\mu_1, \mu_2 \in \mathbf{R}$ *the values of the* $\mathbf{y}$ *field(s) agree if the values of the* $\mathbf{x}$ *field(s) agree:*

$$\forall \mu_1, \mu_2 \in \mathbf{R} : \mu_1(\mathbf{x}) = \mu_2(\mathbf{x}) \Rightarrow \mu_1(\mathbf{y}) = \mu_2(\mathbf{y}) \tag{5.81}$$

Then we say that there is a **functional dependence** *from the field(s)* $\mathbf{x}$ *to field(s)* $\mathbf{y}$*.*

Intuitively, there is some function $f(\mathbf{x})$ that determines the values of the field(s) $\mathbf{y}$.

Theorem 5.7 *Let* $\mathbf{R}\langle\mathbf{r}\rangle$ *be a relation with the schema* $\mathbf{r}$*. Let* $\langle\mathbf{x}, \mathbf{y}\rangle$ *be a partition of the schema into disjoint sets of fields* $\mathbf{x}$ *and* $\mathbf{y}$ *such that there is a functional dependence from* $\mathbf{x}$ *to* $\mathbf{y}$ *(i.e.* $\mathbf{y} = f(\mathbf{x})$*). Then the following two programs compute the same result:*

Program 5.20
doany $\langle\mathbf{x}\rangle \in \pi_{\mathbf{x}}\mathbf{R}$

```
    S(x)
```

and

Program 5.21
doany $\langle\mathbf{x}, \mathbf{y}\rangle \in \mathbf{R}$

```
    S(x)
```

Proof: The above programs execute the statement `S` with the same *sets* of inputs, but potentially with different *sequences* of inputs: Program 5.21 can execute the statement for duplicate values of $\mathbf{x}$ which are removed by the projection in Program 5.20. We prove by contradiction that this is not possible, given the functional dependence from $\mathbf{x}$ to $\mathbf{y}$. Suppose Program 5.21 executes `S` twice for the same values $\mathbf{x}_0$ of the fields $\mathbf{x}$. This means that the first time around the `doany` loop enumerated the value $\mathbf{y}_1$ of the field $\mathbf{y}$ and the second time around – another value $\mathbf{y}_2 \neq \mathbf{y}_1$. But this contradicts the definition functional dependence. ■

In "plain English" the Theorem tells us that we can always introduce dependent fields in the computation or, equivalently, omit the projection onto the sources of the functional dependence. Now consider a loop nest:

Program 5.22
doany $\langle \mathbf{i}, \mathbf{a}, v, \mathbf{f} \rangle \in \bigominus_P (\ \ldots, \mathbf{A}\langle \mathbf{a}, v \rangle, \ldots \ )$
    S($\mathbf{i}$,v,$\mathbf{f}$)

where $\mathbf{f}$ are the fields of the other relations that participate in the query. Suppose that $\mathbf{A}$ is defined by a projection:

$$\mathbf{A} = \pi_{\mathbf{a},v} \mathbf{B}\langle \mathbf{a}, \mathbf{b}, v \rangle \tag{5.82}$$

and there is a functional dependence from $\mathbf{a}$ to $\mathbf{b}$. This is the case in the blocked formats: the block indices are dependent on the point indices. Without the knowledge of the dependence, the equivalent program is:

Program 5.23
doany $\langle \mathbf{i}, \mathbf{a}, v, \mathbf{f} \rangle \in \pi_{\mathbf{i},\mathbf{a},v,\mathbf{f}} \bigominus_P (\ \ldots, \mathbf{B}\langle \mathbf{a}, \mathbf{b}, v \rangle, \ldots \ )$
    S($\mathbf{i}$,v,$\mathbf{f}$)

However, since there is a functional dependence from $\mathbf{a}$ to $\mathbf{b}$, there is also a dependence from $\langle$**i**,**a**,v,**f**$\rangle$ to $\mathbf{b}$: we can add fields to the source of the dependence. Now we have partitioned the fields of the query in Program 5.23 (before the projection) into two disjoint sets: $\langle$**i**,**a**,v,**f**$\rangle$ and $\mathbf{b}$ – with a functional dependence from the former to the latter. Immediately, Theorem 5.7 tells us that Program 5.23 is equivalent to:

Program 5.24
doany $\langle \mathbf{i}, \mathbf{a}, \mathbf{b}, v, \mathbf{f} \rangle \in \bigominus_P (\ \ldots, \mathbf{B}\langle \mathbf{a}, \mathbf{b}, v \rangle, \ldots \ )$
    S($\mathbf{i}$,v,$\mathbf{f}$)

Based on the above considerations we have introduced *projection views* into the source language. The directive:

projection $\mathbf{A}\langle$**f**,v$\rangle$ of $\mathbf{B}\langle$**g**,v$\rangle$

with $\mathbf{f} \subseteq \mathbf{g}$ declares two facts:

1. $\mathbf{A}\langle \mathbf{f}, v \rangle = \pi_{\mathbf{a},v} \mathbf{B}\langle \mathbf{g}, v \rangle$
2. There is a functional dependence from the fields $\mathbf{f}$ to the fields $\mathbf{g} \setminus \mathbf{f}$ (i.e. the fields that are projected away).

So that given a loop nest (Program 5.22) that uses $\mathbf{A}$ we can always use $\mathbf{B}$ instead as in Program 5.24. Chapter 6 has examples of exploiting projection views.

5.8 Optimizations

5.8.1 Exploiting dense BLAS

Often a sparse matrix is stored as a collection of small dense sub-matrices. This decomposes the sparse matrix-vector product into a sequence of small dense products which can be performed by calling hand-tuned implementations, such as found in vendor-supplied BLAS libraries or hand-tuned macros [46, 47]. In some cases it is possible after join scheduling to pattern match the code in order to discover opportunities for using these optimized implementations. We have currently implemented pattern-matching for calls to general dense matrix-vector product subroutine DGEMV and for calls to triangular dense matrix-vector product subroutine DTRMV. In the future, we plan to implement similar techniques to pattern-match other computations, such as matrix-matrix products and outer product updates.

An example

We start with an example. Consider the following code fragment:

Program 5.25

$$
\begin{array}{l}
\texttt{doany ii = }\mathbf{A}\texttt{.lb() .. }\mathbf{A}\texttt{.ub()}\\
\quad \texttt{i := }\mathbf{A}\texttt{.ideref()[ii]}\\
\quad \texttt{doany jj = }\mathbf{A}_{ii}\texttt{.lb() .. }\mathbf{A}_{ii}\texttt{.ub()}\\
\quad\quad \texttt{j := }\mathbf{A}_{ii}\texttt{.ideref()[jj]}\\
\quad\quad \mathbf{Y}_i\texttt{.deref() := }\mathbf{Y}_i\texttt{.deref() + }\mathbf{A}_{ii,jj}\texttt{.deref() * }\mathbf{X}_j\texttt{.deref()}
\end{array}
$$

It computes a matrix vector product $\mathbf{Y} = \mathbf{AX}$, where $\mathbf{Y}$ and $\mathbf{X}$ are both dense and the matrix is stored using `ELEMENT` format (Section 4.5.4). The result of query scheduling for this computation is:

Program 5.26

$$
\begin{array}{l}
\texttt{doany } \langle i, r, \mathbf{h}, i_y, \mathbf{g}\rangle \in \bigominus_{i=r=i_y} \begin{pmatrix} \{1 \le i \le n\}, \\ \mathbf{A}.\mathtt{enum}_r()\langle r, \mathbf{h}\rangle, \\ \mathbf{Y}.\mathtt{enum}_{i_y}\langle i_y, \mathbf{g}\rangle \end{pmatrix}\\
\quad \texttt{doany } \langle j, c, \mathbf{d}, j_x, \mathbf{e}\rangle \in \bigominus_{j=c=j_x} \begin{pmatrix} \{1 \le j \le N\}, \\ \mathbf{A}_{r:\mathbf{h}}.\mathtt{enum}_c()\langle c, \mathbf{d}\rangle, \\ \mathbf{X}.\mathtt{enum}_{j_x}\langle j_x, \mathbf{e}\rangle \end{pmatrix}\\
\quad\quad \mathbf{Y}_{i_y:\mathbf{g}}.\mathtt{deref()} \texttt{ := } \mathbf{Y}_{i_y:\mathbf{g}}.\mathtt{deref()} \texttt{ +}\\
\quad\quad\quad\quad \texttt{+ } \mathbf{A}_{r:\mathbf{h},c:\mathbf{d}}.\mathtt{deref()} \texttt{ * } \mathbf{X}_{j_x:\mathbf{e}}.\mathtt{deref()}
\end{array}
$$

Both joins are scheduled by, first, removing the dense enumerations on $\mathbf{Y}$ and $\mathbf{X}$:

Program 5.27

$$
\begin{array}{l}
\texttt{doany } \langle i, r, \mathbf{h}\rangle \in \bigominus_{i=r=i_y} \begin{pmatrix} \{1 \le i \le N\}, \\ \mathbf{A}.\mathtt{enum}_r()\langle r, \mathbf{h}\rangle \end{pmatrix}\\
\mathtt{i}_y \texttt{ := i}\\
\quad \texttt{doany } \langle j, c, \mathbf{d}, j_x, \mathbf{e}\rangle \in \bigominus_{j=c=j_x} \left(\{1 \le j \le N\}, \mathbf{A}_{r:\mathbf{h}}.\mathtt{enum}_c()\langle c, \mathbf{d}\rangle \right)\\
\quad \mathtt{j}_x \texttt{ := j}\\
\quad\quad \mathbf{Y}_{i_y}.\mathtt{deref()} \texttt{ := } \mathbf{Y}_{i_y}.\mathtt{deref()} \texttt{ +}\\
\quad\quad\quad\quad \texttt{+ } \mathbf{A}_{r:\mathbf{h},c:\mathbf{d}}.\mathtt{deref()} \texttt{ * } \mathbf{X}_{j_x}.\mathtt{deref()}
\end{array}
$$

Notice that the values of the integer fields i_y and j_x also serve as iterators into $\mathbf{X}$ and $\mathbf{Y}$. $\mathbf{Y}_{i_y}$ is the ADT expression for the i_y-th element of $\mathbf{Y}$. $\mathbf{X}_{j_x}$ is the ADT expression for the j_x-th element of $\mathbf{X}$. The remaining joins simply enumerate over the rows and columns of $\mathbf{A}$. These enumerations are of the special kind – INDIRECT (Section 4.2.2). The methods `lb()` and `ub()` provide the loop parameters for the iterator integer `ii` (intuitively, the offset into the data structure). The `ideref()` method returns the name of the indirection array for the field: `ideref()[ii]` is the value of the field. By implementing the remaining one-relation joins via these methods we obtain the code Program 5.25. Expansion of the method invocations, except for $\mathbf{A}_{r:\mathbf{h},c:\mathbf{d}}$.`deref()`, in Program 5.25 produces the following code:

Program 5.28
```
doany ii = 1, NROW
    i := ROWIND[ii]
    doany jj = 1, NCOL
        j := COLIND[jj]
        Y[i] := Y[i] +
            (DENSE_MATRIX_ELT start=VALS[1] colmajor=colmajor
                leading=nrow
                istart=1 jstart=1
                i=ii j=jj name=v) * X[j]
```

This can be translated into a dense BLAS call as:

Program 5.29
```
real YY[NROW], XX[NCOL]
-- collect the required elements of X into XX --
doany jj = 1, NCOL
    XX[jj] := X[ROWIND[jj]]
enddo
-- compute YY = matrix * XX --
call DGEMV('N',NROW,NCOL,1.0,VALS[1],NROW,
        XX[1],1,0.0,YY[1],1)
-- add the result back to Y --
doany ii = 1, NROW
    Y[ROWIND[ii]] := Y[ROWIND[ii]] + YY[ii]
```

Access normalization

The patterns that we describe below depend on the input code being transformed into a normalized form, as follows. We start with a general loop nest:

Program 5.30
```
doany i=L,U
    doany j=M,V
        Y[f(i)] := Y[f(i)] +
            -- dense matrix access A[i+K,j+H] --
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=i0 jstart=j0
                i=(i+K) j=(j+H) name=v) * X[g(j)]
```

We make the following assumptions about it:

Assumption 5.1 *The bounds* `L`, `U`, `M` *and* `V` *are loop invariant.*

Assumption 5.2 *The offsets* `i0`, `j0`, `K`, `H`, `addr` *and the leading dimension* `lda` *are loop invariant.*

Assumption 5.3 *The array access functions* `f(i)` *and* `g(i)` *have no side effects.*

Since the "upper-left" corner of the matrix has the index $(i0, j0)$, we effectively have access $(i + K - i0 + 1, j + H - j0 + 1)$ to the matrix that starts with $(1, 1)$. Let

$$u = i + (K - i0 + 1) \tag{5.83}$$

$$w = j + (H - j0 + 1) \tag{5.84}$$

be a loop transformation. The normalized loop nest is:

Program 5.31

```
doany u=L',U'
    doany w=M',V'
        Y[f'(u)] := Y[f'(u)] +
            -- dense matrix access A[u,v] --
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=u j=w name=v) * X[g'(w)]
```

where:

$$L' = L + (K - i0 + 1) \tag{5.85}$$

$$U' = U + (K - i0 + 1) \tag{5.86}$$

$$M' = M + (H - j0 + 1) \tag{5.87}$$

$$V' = V + (H - j0 + 1) \tag{5.88}$$

$$f'(u) = f(u - (K - i0 + 1)) \tag{5.89}$$

$$g'(w) = g(w - (H - j0 + 1)) \tag{5.90}$$

Observe that the new loop nest satisfies the Assumptions 5.1, 5.2 and 5.3. We say that Program 5.31 is in *normal form #1*. By shifting the loop indices and array accesses we can obtain the *normal form #2*:

Program 5.32

```
doany u=1,U
    doany w=1,V
        Y[f'(u)] := Y[f'(u)] +
            -- dense matrix access A[u+K,w+H] --
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=(u+K) j=(w+H) name=v) * X[g'(w)]
```

General matrix-vector product (DGEMV)

We start with a loop nest in normal form #1:

Program 5.33

```
doany u=L,U
    doany w=M,V
        Y[f(u)] := Y[f(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=v j=w name=v) * X[g(w)]
```

This loop nest can be rewritten using the DGEMV routine as:

Program 5.34

```
real YY[U-L+1], XX[V-M+1]
doany j=M,V
    XX[j-M+1] := X[g(j)]
enddo
call DGEMV(transa,U,V,0.0,buf[addr'],lda,
        XX,1,1.0,YY,1)
doany i=L,U
    Y[f(i)] := Y[f(i)] + YY[i-L+1]
enddo
```

where `transa` flag is set to 'N' if the `colmajor` argument to the `DENSE_MATRIX_ELT` ADT is `true` and is set to 'T', otherwise. The new starting address $\texttt{addr}'$ is calculated depending on the `colmajor` flag:

$$\texttt{addr}' = \begin{cases} \texttt{addr} + \texttt{lda} * (\texttt{M} - 1) + (\texttt{L} - 1) & \text{if} \quad \texttt{colmajor} = \texttt{true} \\ \texttt{addr} + \texttt{lda} * (\texttt{L} - 1) + (\texttt{M} - 1) & \text{if} \quad \texttt{colmajor} = \texttt{false} \end{cases} \tag{5.91}$$

The transformation is summarized in Figure 5.19. The following simplifications are possible:

PATTERN(normalized):

```
doany u=L,U
    doany w=M,V
        Y[f(u)] := Y[f(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=v j=w name=v) * X[g(w)]
```

CONDITIONS:

- Assumptions 5.1, 5.2 and 5.3.

OUTPUT:

```
real YY[U-L+1], XX[V-M+1]
doany j=M,V
    XX[j-M+1] := X[g(j)]
enddo
call DGEMV(transa,U,V,0.0,buf[addr'],lda, XX,1,1.0,YY,1)
doany i=L,U
    Y[f(i)] := Y[f(i)] + YY[i-L+1]
enddo
```

WHERE

- $\texttt{trans} =' \texttt{N}'$ if $\texttt{flag} = \texttt{true}$ and $\texttt{trans} =' \texttt{T}'$ if $\texttt{flag} = \texttt{false}$.
- $\texttt{addr}'$ is set according to (5.91).

Figure 5.19: DGEMV pattern-matching

- If the access to `X` is $\texttt{X}[\texttt{j} + \texttt{C}]$ for some loop invariant `C`, the we do not need to gather the values of `X` into a separate array `XX`:

  Program 5.35
  ```
  real YY[U]
  call DGEMV(trans,U,V,0.0,buf[addr'],lda,XX[1+C],1,1.0,YY,1)
  doany i=1,U
      Y[f(i)] := Y[f(i)] + YY[i]
  ```

```
enddo
```

- If the access to Y is Y[i + M] for some loop invariant M, then we do not need to keep the array YY:

  Program 5.36
  ```
  real XX[V]
  doany j=1,V
      XX[j] := X[g(j)]
  enddo
  call DGEMV(trans,U,V,1.0,buf[addr'],lda,XX,1,1.0,Y[M+1],1)
  ```

- If both access are as above, then we just need a call to DGEMV:

  Program 5.37
  ```
  call DGEMV(trans,U,V,1.0,buf[addr'],lda,X[1+C],1,1.0,Y[M+1],1)
  ```

Triangular matrix-vector product (DTRMV)

Another pattern we look for is an opportunity to call the DTRMV routine which multiplies a dense vector by an upper or lower triangular matrix. We start with a loop nest in normal form #2 (Program 5.32):

Program 5.38
```
doany u=1,U
    doany w=1,V
        Y[f'(u)] := Y[f'(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=(u+K) j=(w+H) name=v) * X[g'(w)]
```

And make another assumption:

Assumption 5.4 *The upper bound* V *is of the form* u − D *for a* **positive integer constant** D*.*

The loop bounds now are:

$$1 \leq u \leq U \wedge 1 \leq w \leq u - D \tag{5.92}$$

The bounds for w are not empty only when $u \geq D + 1$. So equivalent loop bounds are:

$$D + 1 \leq u \leq U \wedge 1 \leq w \leq u - D \tag{5.93}$$

We make a substitution (and equivalent loop transformation):

$$u' = u - D \tag{5.94}$$

The new loop bounds are:

$$1 \leq u' \leq U - D \wedge 1 \leq w \leq u' \tag{5.95}$$

The offset K is replaced by:

$$K' = K + D \tag{5.96}$$

Overall, we get the loop nest of the form:

Program 5.39

```
doany u=1,U
    doany w=1,u
        Y[f'(u)] := Y[f'(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=colmajor
                leading=lda
                istart=1 jstart=1
                i=(u+K) j=(w+H) name=v) * X[g'(w)]
```

Effectively this computes the product with a $U \times U$ triangular matrix that starts at the position $(K+1, H+1)$ in the original matrix. This can be done using the DTRMV routine.

The DTRMV routine performs the multiplication of a vector $\mathbf{X}$ by a triangular matrix $\mathbf{A}$ as:

$$\mathbf{X} \leftarrow op(\mathbf{A})\mathbf{X} \tag{5.97}$$

The calling sequence is:

```
call DTRMV(uplo,trans,diag,n,a,lda,X,strideX)
```

where the flag `uplo` is either 'U' for an upper triangular matrix or 'L' for a lower triangular matrix. `a` is the address of the start of storage for $\mathbf{A}$ and `lda` is the leading dimension of the storage. `trans` is the usual transposition flag. `diag` is either 'U' for a unit-triangular matrix (diagonal is all 1) or 'N' for the general triangular matrix. `X` and `strideX` have the same meaning as before. Given this calling sequence, Program 5.39 is translated into:

Program 5.40

```
real XX[U]
doany u=1,U
    XX[u] := X[g(u)]
enddo
call DTRMV('L',trans,'N',U,buf[addr'],lda,XX,1)
doany u=1,U
    Y[f(u)] := Y[f(u)] + XX[u]
enddo
```

The `trans` flag is set in accordance with the `colmajor` flag, as before, and `addr`$'$ is computed from the offsets `K` and `H` a.la. equation (5.91):

$$\texttt{addr}' = \begin{cases} \texttt{addr} + \texttt{lda} * (\texttt{H} - 1) + (\texttt{K} - 1) & \text{if} \quad \texttt{colmajor} = \texttt{true} \\ \texttt{addr} + \texttt{lda} * (\texttt{K} - 1) + (\texttt{H} - 1) & \text{if} \quad \texttt{colmajor} = \texttt{false} \end{cases} \tag{5.98}$$

The transformation is summarized in Figure 5.20. Similar transformation is possible in the upper-triangular case (Figure 5.21).

PATTERN: (normalized)

```
doany u=1,U
    doany w=1,i
        Y[f(u)] := Y[f(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=flag
                leading=lda
                istart=1 jstart=1
                i=(u+K) j=(w+H) name=v) * X[g(w)]
```

CONDITIONS:

- Assumptions 5.1, 5.2 and 5.3.

OUTPUT

```
real XX[U]
doany u=1,U
    XX[u] := X[g(u)]
enddo
call DTRMV('L',trans,'N',U,buf[addr'],lda,XX,1)
doany u=1,U
    Y[f(u)] := Y[f(u)] + XX[u]
enddo
```

WHERE

- `trans` $='$ `N`$'$ if `flag` $=$ `true` and `trans` $='$ `T`$'$ if `flag` $=$ `false`.
- `addr`$'$ is set according to (5.98).

Figure 5.20: DTRMV (lower triangular) pattern matching

PATTERN: (normalized)

```
doany w=1,U
    doany u=1,w
        Y[f(u)] := Y[f(u)] +
            (DENSE_MATRIX_ELT start=buf[addr] colmajor=flag
                leading=lda
                istart=1 jstart=1
                i=(u+K) j=(w+H) name=v) * X[g(w)]
```

CONDITIONS:

- Assumptions 5.1, 5.2 and 5.3.

OUTPUT

```
real XX[U]
doany u=1,U
    XX[u] := X[g(u)]
enddo
call DTRMV('U',trans,'N',U,buf[addr],lda,XX,1)
doany u=1,U
    Y[f(u)] := Y[f(u)] + XX[u]
enddo
```

WHERE

- trans $='$N$'$ if flag = true and trans $='$T$'$ if flag = false.
- addr$'$ is set according to (5.98).

Figure 5.21: DTRMV (upper triangular) pattern matching

Discussion

The above transformations are beneficial when the matrix sizes are above a few tens. For smaller matrices the overhead of the function call as well as of the argument dispatch within BLAS routines actually reduce the overall performance. BlockSolve library [46, 47] uses C macros in place of the BLAS calls for small matrices. The macros have the loops unrolled by hand and perform better than either a BLAS call or unmodified code. The library decides at run-time which version (macro or a function) to call (compare to Figure 5.19):

Program 5.41

```
if (U and V are large enough) then
    call DGEMV(trans,U,V,0.0,buf[addr+H*lda+K],lda,
         XX,1,1.0,YY,1)
else
    MACRO_DGEMV(trans,U,V,0.0,buf[addr+H*lda+K],lda,
         XX,1,1.0,YY,1)
endif
```

This handles the choice between large matrices and smaller matrices, but in the cases when the size of the matrix is very small (close to 1), the overhead of the conditional itself starts degrading the performance. At some point the compiler or the user has to make the decision whether it is beneficial to apply the BLAS pattern-matching. Since the sizes of the dense matrices depend on the input to the program it is unreasonable to expect the compiler to be able to make the right decision, so we leave the decision to the user. The compiler can be invoked with special flags that instruct it to perform the pattern-matching.

We discuss the impact of the pattern-matching on the performance of the output code in Section 6.3.

5.8.2 Precomputing searches

Suppose we have a code fragment that involves a `search` method:

Program 5.42

```
 1.  while (....)  do
 2.       ...
 3.       doany i ∈ 𝓑
 4.                 A.declare_iter(h)
 5.                 a := f(i)
 6.                 A.search(h,a)
 7.                 if A.valid(h) then
 8.                     S
 9.                 endif
10.       enddo
11.       ...
12.  enddo
```

where the `doany` loop on line 3 can be a nest of loops. Such code fragment can appear as the result of applying the ENUMERATE-SEARCH join implementation (see Figure 5.12). If the loop bounds $\mathcal{B}$ are invariant with respect to the outer `while` loop, then we can transform the code into:

Program 5.43

```
integer k, nn, sz
```

```
 2.  A.declare_iter(h)
 3.  sz := h.size()
 4.  k := 0
 5.  -- compute the trip-count of the loop --
 6.  doany i ∈ B
 7.      k := k + 1
 8.  enddo
 9.  nn := k
10.  -- allocate the storage for all iterators --
11.  integer H[sz * nn]
12.  k := 0;
13.  -- pre-compute all iterators --
14.  doany i ∈ B
15.      a := f(i)
16.      A.search(h,a)
17.      h.marshal(H, sz * k)
18.      k := k + 1
19.  enddo
20.  while (....)  do
21.      ...
22.      k := 0
23.      doany i ∈ B
24.          -- use the pre-computed iterators --
25.          h.unmarshal(H, sz * k)
26.          if A.valid(h) then
27.              S
28.          endif
29.          k := k + 1
30.      enddo
31.      ...
32.  enddo
```

The loop on line 6 computes the trip-count of the `doany` loop nest. Then the buffer that stores all the iterators accessed in the loop nest is allocated on line 11. The iterators are computed on line 14 and used on line 25.

This optimization is used heavily on node programs produced by the SPMD code generator (Part III). In this situation the array **A** is usually a communication buffer. The above optimization is the generalization of *array slicing* [30].

5.9 Handling dependencies

So far, we have been discussing DOANY loops, that give us complete freedom in reordering the computations to obtain efficient enumerations over data structures. However, if we start with a dense DOACROSS loop nest, there are restrictions on the possible reordering of computation: the final code has to respect *dependencies* present in the original code. In this Section we describe how the query scheduling algorithm can be modified in order to account for dependencies in the source (dense) loop nest.

We start in Section 5.9.1 with an example and the overview of dependence analysis of loop nests and representations of dependencies. In Section 5.9.3 we show that there is a 1-1 correspondence between query schedules, loop transformations and echelon forms of the data access equation. If the loop nest is DOACROSS, then not every transformation is legal. We show in Section 5.9.4 how and under which conditions an illegal transformation can be "corrected". As the result, we arrive at a very simple modification to the query rewrite rules that guarantees that a legal query schedule will be found, if one exists.

We make two restrictions on the data structures:

- The index hierarchies must be of the form:

$$i_1^* \prec i_2^* \prec \ldots \prec i_n^* \prec v^1 \tag{5.99}$$

  where $i_1, \ldots, i_n$ is some permutation of the indices of the tensor. This is basically a generalization of CRS format. We also do not allow projection views. Current Black Box protocol provides no way of relating the order of the enumeration of "new" fields to the order of the fields in the projection.

- The Black Boxes must provide access methods to enumerate the indices in both ascending and descending order. As we discuss shortly, the result of scheduling a DOACROSS loop is a nesting of one-dimensional joins with the restriction that the loop indices be enumerated in ascending order. In order to implement these joins we need to be able to enumerate tensor indices in both ascending and descending order. (We can lift this restriction by introducing join implementations that include sorting of the operand streams.)

5.9.1 Preliminaries

Consider the following program that computes the solution of a dense unit lower-triangular system of linear equations $\mathbf{LX} = \mathbf{B}$:

Program 5.44

```
real B[n], L[n,n], X[n]
for i = 1, n
    for j = 1, i-1
        B[i] := B[i] - L[i,j]*B[j]
```

The code overwrites the array `B` with the values of the solution $\mathbf{X} = \mathbf{L}^{-1}\mathbf{B}$. The dense loop has two dependencies [2]:

- A *flow dependence* from the write into `B[i]` to the read from `B[j]`.
- An *output dependence* between successive updates to `B[i]`.

We will ignore the output dependence, assuming that the updates are commutative and associative, as is standard. Let us focus on the flow dependence. There is a flow dependence from the iteration (i_w, j_w) that

[2] For a detailed discussion of dependence analysis see Wolfe's book on optimizing compilers [91]

writes the value of `B[i`$_w$`]` to the iteration (i_r, j_r) that reads the value `B[j`$_r$`]` if the following is true:

$$\begin{cases} i_w = j_r & \text{write/read the same array element} \\ 1 \leq j_w < i_w \leq n & \text{loop bounds for the write} \\ 1 \leq j_r < i_r \leq n & \text{loop bounds for the read} \\ (i_r, j_r) \succ (i_w, j_w) & \text{write before read} \end{cases} \tag{5.100}$$

where the operator $\succ$ denotes *lexicographic* comparison.In this case it is easy to see there is a dependence from the iteration (i_w, j_w) to $(i_r > i_w, j_r = i_w)$. There are various ways of representing this information. In the most general case, one can compute a decision tree (also known as a *quast* [34] or a last-write-tree [60]) which represents the destination of the dependence as an affine function of the source of the dependence, with the coefficients of the function being determined by certain conditions on the source of the dependence. The literature from the Omega project uses the ubiquitous term "relation" to denote the mapping between the source and the destination of a dependence [70, 71].

The most general representations are sometimes harder to manipulate and people have devised the *approximate* representation of dependencies via *distance/direction* vectors. This representation is motivated by the fact that often the source and destination of the dependence are separated by a constant number of iterations. We can define the distance vector (for a flow dependence) by:

$$\mathbf{d} = \begin{pmatrix} i_r \\ j_r \end{pmatrix} - \begin{pmatrix} i_w \\ j_w \end{pmatrix} \tag{5.101}$$

In general the value of the vector depends on the value of the source of the dependence. In some cases, the dependence is *uniform* – that is the vector is constant. Another useful approximation is to use the *direction* vector. It is defined in the same way as the distance vector. However, for a non-constant entry it uses one of the direction signs "+", "−", "∗" to indicate whether the entry is positive, negative or of indeterminate sign, respectively. In our example the distance vector is:

$$\begin{pmatrix} i_r - i_w \\ j_r - j_w \end{pmatrix} = \begin{pmatrix} + \\ + \end{pmatrix} \tag{5.102}$$

Because the destination of a dependence always occurs after the source, the distance vector must always be lexicographically positive. This gives us a very simple characterization of legal linear loop transformations. Consider a d-dimensional loop nest with indices $\mathbf{i} = (i_1\ i_2\ \ldots\ i_d)^T$. Let $\mathbf{T}$ be a linear loop transformation. Let $\mathbf{i}_w$ be the source of a (flow) dependence and $\mathbf{i}_r$ be the destination. By definition the difference between the two (i.e. the distance vector) is lexicographically positive:

$$\mathbf{d} = \mathbf{i}_r - \mathbf{i}_w \succ 0 \tag{5.103}$$

A vector is lexicographically positive if it start with (possibly empty) prefix consisting of all zeros followed immediately by a positive element:

$$\mathbf{d} \succ 0 \Leftrightarrow \exists k : \mathbf{d}(1 : k-1) = \mathbf{0} \wedge \mathbf{d}(k) > 0 \tag{5.104}$$

In the transformed loop nest the distance vector must be lexicographically positive, as well:

$$\mathbf{Ti}_r - \mathbf{Ti}_w = \mathbf{d}_{transformed} = \mathbf{Td} \succ 0 \tag{5.105}$$

In other words, the transformation must respect the partial order of the iterations, as given by the dependence vectors.

Observe that the entries of the dependence vectors $\mathbf{d}$ are not just integers, but come from an extended set:

$$\mathbb{D} = \mathbb{Z} \cup \{\text{"+"}, \text{"−"}, \text{"∗"}\} \tag{5.106}$$

We can easily extend integer arithmetic to this set. For example:

$$\begin{aligned}
&a > 0 \Rightarrow a \cdot \text{“+”} = \text{“+”} \\
&a < 0 \Rightarrow a \cdot \text{“+”} = \text{“-”} \\
&\text{“+”} + \text{“+”} = \text{“+”} \\
&\text{“+”} + \text{“-”} = \text{“*”}
\end{aligned} \tag{5.107}$$

There are techniques in compiler literature for computing a legal transformation $\mathbf{T}$ given the set $\mathcal{D}$ of dependence vectors. Of particular interest to us is the so-called *completion procedure* developed by Li and Pingali [55, 56]. The input to the procedure is the set $\mathcal{D}$ of dependence vectors and the first k rows of the transformation matrix $\mathbf{T}$ that do not violate the dependencies in the following sense:

$$\forall \mathbf{d} \in \mathcal{D} : \mathbf{T}_1\mathbf{d} \succ 0 \wedge \mathbf{T}_1\mathbf{d} = 0 \tag{5.108}$$

where $\mathbf{T}_1$ denote the given rows of $\mathbf{T}$:

$$\mathbf{T}_1 = \begin{pmatrix} \mathbf{t}_1^T \\ \mathbf{t}_2^T \\ \vdots \\ \mathbf{t}_k^T \end{pmatrix} \tag{5.109}$$

The completion procedure computes (is possible) the remaining rows of $\mathbf{T}$ such that the transformation is legal:

$$\forall \mathbf{d} \in \mathcal{D} : \mathbf{T}\mathbf{d} \succ 0 \tag{5.110}$$

We will use the completion procedure in order to correct an illegal query schedule to a legal one.

Recall that query scheduling implicitly uses loop transformations. In particular, the vector $\mathbf{i}$ of loop iterations is transformed to obtain the block-echelon form of the data access equation when applying the JOIN and SEARCH rules (cf. Figures 5.4, 5.6 and 5.7). Each query schedule can be described by a sequence of rules that have been applied in order to obtain the schedule. The loop transformations used for each of the rules can be composed to obtain the loop transformation that characterizes the whole schedule. When we are dealing with DOANY loops there are no restrictions on this transformation. When we are dealing with DOACROSS loops this transformation must respect the dependencies.

Let us return to our example. Observe that the loop nest is conjunctive: we only need to use the values and locations already stored in the arrays. However, we can not write the loop nest in affine-join form, because this is not a DOANY loop. To be able to specify the affine-join form of DOACROSS loop nest we introduce the `dolex` statement as in:

Program 5.45

`dolex` $\langle i, j, r, c, v_L, i_B, v_{B1}, j_B, v_{B2} \rangle \in \ldots$

$$\ldots \bigominus_{r=i_B=i \wedge c=j_B=j} \left(\begin{array}{l} \{1 \le j < i \le n\}, \mathbf{L}\langle r, c, v_L\rangle, \\ \mathbf{B}\langle i_B, v_{B1}\rangle, \mathbf{B}\langle j_B, v_{B2}\rangle \end{array} \right)$$

$\mathtt{v}_{B1}$ `:=` $\mathtt{v}_{B1}$ `-` $\mathtt{v}_L$ `*` $\mathtt{v}_{B2}$

Like the `doany` loop nest it enumerates over the solutions to the query, but it is required to enumerate the loop indices (i and j in this case) in lexicographic order – just as the ordinary dense loop in Program 5.44 would. Query scheduling for DOANY loops translates a `doany` loop nest over a multi-dimensional affine join into a nesting of `doany` loops over one-dimensional affine joins. For DOACROSS loops, query scheduling translates multi-dimensional `dolex` loops into a nesting of one-dimensional `dolex` loops.

Suppose that the matrix $\mathbf{L}$ is stored in CCS format. The query for the iterations of the loop nest is:

$$\bigominus_{\substack{r=i_B=i \wedge \\ c=j_B=j}} \left(\; \{1 \le j < i \le n\}, \mathbf{L}\langle r, c, v_L\rangle, \mathbf{B}\langle i_B, v_{B1}\rangle, \mathbf{B}\langle j_B, v_{B2}\rangle \; \right) \tag{5.111}$$

The data access equation in matrix form is:

$$\begin{pmatrix} r \\ c \\ i_B \\ j_B \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{5.112}$$

The index hierarchy of $\mathbf{L}$ is: $c \prec r \prec v_L$. Therefore, there are two groups of joinable fields: $c = j_B$ and i_B. Heuristically, we can decide to use the larger set. Now, according to the JOIN rule, we need to transform the data access equation into the block echelon form:

$$\begin{pmatrix} c \\ j_B \\ r \\ i_B \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \quad \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{5.113}$$

The order of the loops for the corresponding query schedule is:

Program 5.46

dolex $\langle u_1, c, \mathbf{h}, j_B, \mathbf{g} \rangle \in \ldots$

$\ldots \bigominus_{u_1 = c = j_B} \left(\ \{1 \le u_1 \le n-1\}, \mathbf{L}.\mathtt{enum}_\mathtt{c}\langle c, \mathbf{h} \rangle, \mathbf{B}.\mathtt{enum}_{\mathtt{j_B}}\langle j_B, \mathbf{g} \rangle \ \right)$

dolex $\langle u_2, r, i_B \rangle \in \ldots$

$\ldots \bigominus_{u_2 = r = i_B} \left(\ \{u_1 < u_2 \le n\}, \mathbf{L}_{c:\mathbf{h}}.\mathtt{enum}()\langle r, \mathbf{k} \rangle, \mathbf{B}.\mathtt{enum}_{\mathtt{i_B}}\langle i_B, \mathbf{f} \rangle \ \right)$

$\mathbf{B}_{i_B:\mathbf{f}}$:= $\mathbf{B}_{i_B:\mathbf{f}}$ - $\mathbf{L}_{c:\mathbf{h},r:\mathbf{k}}$ * $\mathbf{B}_{j_B:\mathbf{g}}$

The loop transformation:

$$\mathbf{T} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{5.114}$$

is legal:

$$\mathbf{Td} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} + \\ + \end{pmatrix} = \begin{pmatrix} + \\ + \end{pmatrix} \tag{5.115}$$

Therefore, the resulting code is legal. However, the echelon form (5.113) is not unique. For example, we can change the sign of the first column and obtain:

$$\begin{pmatrix} c \\ j_B \\ r \\ i_B \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \quad \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{5.116}$$

From the point of view of scheduling a DOANY loop, there is no difference between (5.113) and (5.116). The pseudo-code for the second query schedule is:

Program 5.47

dolex $\langle u_1, c, \mathbf{h}, j_B, \mathbf{g} \rangle \in \ldots$

$\ldots \bigominus_{-u_1 = c = j_B} \left(\ \{1-n \le u_1 \le -1\}, \mathbf{L}.\mathtt{enum}_\mathtt{c}\langle c, \mathbf{h} \rangle, \mathbf{B}.\mathtt{enum}_{\mathtt{j_B}}\langle j_B, \mathbf{g} \rangle \ \right)$

dolex $\langle u_2, r, i_B \rangle \in \ldots$

$\ldots \bigominus_{u_2 = r = i_B} \left(\ \{u_1 < u_2 \le n\}, \mathbf{L}_{c:\mathbf{h}}.\mathtt{enum}()\langle r, \mathbf{k} \rangle, \mathbf{B}.\mathtt{enum}_{\mathtt{i_B}}\langle i_B, \mathbf{f} \rangle \ \right)$

```
B_{i_B:f} := B_{i_B:f} - L_{c:h,r:k} * B_{j_B:g}
```

This code does not respect the dependencies, because the corresponding loop transformation is not legal:

$$\mathbf{Td} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} + \\ + \end{pmatrix} = \begin{pmatrix} - \\ + \end{pmatrix} \not\succ 0 \tag{5.117}$$

Observe that we can "correct" this illegal transformation and obtain a legal one by negating the first row of $\mathbf{T}$ or, equivalently, reversing the order of the u_1 loop in Program 5.47. We now describe a general technique for correcting a loop transformation in order to obtain a legal schedule.

5.9.2 Outline of the technique

The result of query scheduling of a DOANY loop nest is code of the form:

Program 5.48

```
search h_0
    join u(1), j_1
        search h_1
            ...
                join u(r), j_r
                    search h_r
                        dense loops for u(r+1:n)
```

where n is the dimension of the original loop nest and r is the rank of the data access matrix $\mathbf{F}$ in the data access equation:

$$\mathbf{f} = \mathbf{Fi} + \mathbf{Gs} + \mathbf{f}^{(0)} \tag{5.118}$$

The vectors $\mathbf{h}_i$, $0 \leq i \leq r+1$ contain the fields that are searched in the schedule. In particular, $\mathbf{h}_0$ is the vector of the fields that are initially determined by the symbolic constants (invariants) $\mathbf{s}$. The vectors $\mathbf{j}_i$, $1 \leq i \leq r$, denote those fields that are being joined. The vector $\mathbf{u}$ denotes the new loop indices. In particular, the first f elements of $\mathbf{u}$: $\mathbf{u}(1)$ through $\mathbf{u}(r)$ are the loop indices of the corresponding one-dimensional affine joins. The rest of the loop indices: $\mathbf{u}(r+1:n)$ – do not participate in any joins. The new loop indices $\mathbf{u}$ are related to the loop indices $\mathbf{i}$ in the original loop nest via a unimodular transformation $\mathbf{T}$:

$$\mathbf{u} = \mathbf{Ti} \tag{5.119}$$

This transformation is the composition of individual transformations applied to the loop indices during the application of the JOIN rule (cf. Figure 5.4).

We assume that each join enumerates over at least one relation – that is the vectors $\mathbf{j}_i$ have length of at least 1. Otherwise, the corresponding joins would degenerate into dense loops over the loop indices u_i and should be placed innermost. We should note that the transformation rules of query scheduling allow for such dense loops to be placed anywhere in the schedule. However, it is more efficient to place dense loops innermost and joins and searches outermost.

If we start with a DOANY loop nest, the joins in Program 5.48 are also DOANY loops and there are no restrictions on the order of the enumeration of the joins. If we start with a DOACROSS loop nest, then a particular schedule can violate dependencies. There are two reasons why this might happen:

- The transformation $\mathbf{T}$ is legal with respect to the dependencies in the original loops nest (i.e. $\mathbf{Td} \succ 0$, for all dependence vectors $\mathbf{d}$), however, the schedule does not enumerate loop indices $\mathbf{u}$ in a lexicographic order. This problem is easy to fix: since we assume that the data structures provide enumerations of the fields in both ascending and descending order, we can adjust join implementations to enumerate the loop indices in ascending order. If every join in Program 5.48 enumerates its loop index in ascending order, then the whole loop nest enumerates the loop indices in lexicographic order.

- The transformation $\mathbf{T}$ is not legal (i.e. $\mathbf{Td} \not\succ 0$, for some dependence vector(s) $\mathbf{d}$). The idea is to adjust the schedule so that the new schedule has the same order of joins and searches (on the same fields), but has a slightly different transformation $\mathbf{T}'$ which is legal. This is in fact what happened in the example of the previous Section. The schedule in Program 5.47 is illegal because the corresponding transformation is illegal, but it can be adjusted to the schedule in Program 5.46 by reversing one of the loops.

We focus on the problem of adjusting the schedule. As we demonstrate shortly, if there is a correction to the schedule, then it can be performed by (a) reversing some of the loops over one-dimensional joins and (b) applying a linear loop transformation to the inner-most dense loops (over $\mathbf{u}(r+1:n)$).

The correction is based on the relationship between query schedules and column echelon forms of the data access matrix $\mathbf{F}$. The order of the fields in the joins and searches is given by the step structure of column echelon form. We discuss this relationship in Section 5.9.3. Correction of a schedule is based on correcting the echelon form so that it has the same structure but with different values. The correction of the echelon form is equivalent to pre-multiplying the transformation matrix $\mathbf{T}$ by a matrix $\mathbf{M}$ that is in *partially block-diagonal (PBD) form*:

$$\mathbf{M} = \begin{pmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \tag{5.120}$$

where $\mathbf{J}$ is an $r \times r$ diagonal matrix with ± 1's on the diagonal. Pre-multiplying $\mathbf{T}$ by $\mathbf{M}$ is equivalent to reversing some of the loops over the joins and transforming the inner-most dense loops in Program 5.48. Section 5.9.4 is devoted to proving that such correction is *complete*: if there is a correction to the schedule, then it has the form (5.120).

Of course, not every schedule can be corrected to an equivalent legal one. In fact, we might have to explore the space of possible schedules and try to correct each one. We have to verify that this space is complete: it should include at least one schedule that can be corrected. A reasonable fall-back position is to ensure that the schedule which simply enumerates the original dense loop nest and the searches into the relations is included in the space. This default schedule has the form:

Program 5.49
```
dolex i ∈ B
    search f
```

where the searches are generated by repeatedly applying the SEARCH rule to the fields as they become exposed. Unfortunately, this schedule does not fit the pattern in Program 5.48: it has searches in the inner-most position. To fix this problem, we look for the schedules of the form:

Program 5.50
```
search h_0
    join u(1), j_1
        search h_1
            ...
                join u(r), j_r
                    search h_r
                        dense loops for u(r+1:n)
                            search g
```

```
1.  Choose a subset g of the fields position
2.  Compute the query schedule that searches g at the innermost position
3.  Let T be the corresponding loop transformation
4.  Find a PDB matrix M such that MT is a legal transformation
5.  if (found M) then
6.      Adjust the schedule in accordance with M
7.  else
8.      fail
9.  endif
```

Figure 5.22: Non-deterministic algorithm for scheduling DOACROSS loops

If we let the set **g** of the fields that are searched inside of the dense loops to contain all of the tensor fields, then we get the schedule shown in Program 5.49. The non-deterministic algorithm for scheduling DOACROSS loops is outlined in Figure 5.22. The non-determinism comes from the choice of the fields **g** on line 1. Given that the total number of fields involved in the input program is small (less that 10), one solution is to try all possible choice of **g**. At this point we have not explored any heuristics.

In the following sections we describe the mathematical details of finding the correction matrix **M** and its relationship to the query schedule.

5.9.3 Echelon form of the data access equation

Let us define a formal notion of column echelon form of a matrix. Let **A** be a matrix with m rows and n columns.

Definition 5.4 *The sequence of integers* $(r_0, r_1, \ldots, r_n)$ *is called the* **profile** *of the matrix* **A** *if:*

- $r_0 = 0$
- r_k *is the column index of the right-most non-zero entry in the* k*-th row of* **A**. *If the row is all zero, then* $r_k = 0$.

For example, the matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 3 & 0 \\ 4 & 0 & 0 \end{pmatrix} \tag{5.121}$$

has the profile: $(0, 3, 0, 2, 1)$. We use the notion of profile in order to define column-echelon form:

Definition 5.5 *A matrix* **A** *is said to be in* **column-echelon form** *if its profile* $(r_0, r_1, \ldots, r_n)$ *satisfies the following condition:*

$$\forall k \in 1..n : r_k = r_{k-1} + 1 \wedge r_k = r_{k-1} \tag{5.122}$$

In other words, the profile of a matrix in column-echelon form is (a) non-decreasing and (b) each entry of the profile is larger than the previous one at most by 1. For brevity, we will use the term "echelon form" in place of "column-echelon form". Observe that the rank [40] of the matrix in echelon form is given by the last entry in the profile:

$$\text{rank}(\mathbf{A}) = r_n \tag{5.123}$$

Clearly, the matrix in (5.121) is not in echelon form. However, the following matrix is:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 4 & 0 \end{pmatrix} \tag{5.124}$$

Its profile is $(0, 0, 1, 1, 2)$ and its rank is 2. Given a matrix $\mathbf{A}$, one can compute its echelon form $\mathbf{H}$ by a combination of row permutations $\mathbf{P}$ and unimodular column operations $\mathbf{U}$ (see [27]):

$$\mathbf{PAU} = \mathbf{H} \tag{5.125}$$

Now we are ready to formally state the relationship between query scheduling and echelon forms. We start with a loop nest in affine join form:

Program 5.51

$$\texttt{doany}\ \langle\ \mathbf{i}, \mathbf{f}_1, v_1, \ldots, \mathbf{f}_m, v_m\ \rangle \in \bigodot_{\mathbf{f}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{f}^{(0)}} \begin{pmatrix} \mathcal{B}\langle \mathbf{i}\rangle, \mathbf{A}_1\langle \mathbf{f}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{f}_m, v_m\rangle \end{pmatrix}$$

$$\quad \texttt{S}(\mathbf{s}, \mathbf{i}, \texttt{v}_1, \ldots, \texttt{v}_m)$$

We also assume that the index hierarchies of the relations in the loop nest satisfy the restriction (5.99). One consequence of this restriction is that TUPLE-ENUM and SINGLETON rewrite rules of query scheduling are not applicable.

Observe that the loop indices $\mathbf{u}$ in the final schedule and related to the original loop indices $\mathbf{i}$ by a unimodular linear transformation:

$$\mathbf{u} = \mathbf{Ti} \tag{5.126}$$

This transformation is the composition of the transformations used to expose joinable fields in the data access equation during the application of the JOIN rule. In our example the transformation is:

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{5.127}$$

The following Theorem states the relationship between a permutation of the data access equation, the transformation $\mathbf{T}$ and the query schedule:

Theorem 5.8 *Let $\mathbf{f} = \mathbf{Fi} + \mathbf{Gs} + \mathbf{f}^{(0)}$ be the data access equation of a loop nest. Let $\mathbf{u}$ be the loop indices of the one-dimensional joins in the result of query scheduling. Let $\mathbf{T}$ be the transformation between the original loop indices $\mathbf{i}$ and the final loop indices $\mathbf{u}$: $\mathbf{u} = \mathbf{Ti}$. Then there exists a permutation matrix $\mathbf{P}$ such that the matrix $\mathbf{H} = \mathbf{PFT}^{-1}$ is in column echelon form.*

Conversely, let $\mathbf{H} = \mathbf{PFU}$ be the column echelon form of $\mathbf{F}$. Then there exists a query schedule such that the transformation between the original loop indices $\mathbf{i}$ and the final loop indices $\mathbf{u}$ is given by $\mathbf{u} = \mathbf{U}^{-1}\mathbf{i}$.

The theorem tells us that there is a correspondence between query schedules and echelon forms of the matrix $\mathbf{F}$ in the data access equation.

Proof: We prove the second part of the theorem first. Let $\mathbf{H} = \mathbf{PFU}$ be the echelon form of $\mathbf{F}$. By definition of the echelon form $\mathbf{H}$ has the following block structure:

$$\begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ \boxed{\mathbf{c}_1} & 0 & 0 & \dots & 0 & 0 \\ \boxed{\mathbf{M}_2} & \boxed{\mathbf{c}_2} & 0 & \dots & 0 & 0 \\ \multicolumn{2}{c}{\boxed{\mathbf{M}_3}} & \boxed{\mathbf{c}_3} & \dots & 0 & 0 \\ \vdots & & & \ddots & \vdots & \vdots \\ \multicolumn{4}{c}{\boxed{\mathbf{M}_r}} & \boxed{\mathbf{c}_r} & 0 \end{pmatrix} \tag{5.128}$$

where each blocks of rows has the same length as a corresponding subsequence of the equal numbers in the profile of $\mathbf{H}$. The entries of the vector $\mathbf{c}_k$, $1 \leq k \leq r$, all non-zero. Each matrix $\mathbf{M}_k$, $2 \leq k \leq r$, has $k-1$ columns. As we have mentioned r is the rank of the matrix $\mathbf{H}$.

We can rewrite the data access equation as:

$$\mathbf{Pf} = (\mathbf{PFU})(\mathbf{U}^{-1}\mathbf{i}) + \mathbf{PGs} + \mathbf{Pf}^{(0)} \tag{5.129}$$

By definition $\mathbf{H} = \mathbf{PFU}$. Let $\mathbf{h} = \mathbf{Pf}$ be the vector of the fields in the new order. Let $\mathbf{u} = \mathbf{U}^{-1}\mathbf{i}$, $\mathbf{G}' = \mathbf{PG}$ and $\mathbf{h}^{(0)} = \mathbf{Pf}^{(0)}$. (5.129) becomes:

$$\mathbf{h} = \mathbf{Hu} + \mathbf{G}'\mathbf{s} + \mathbf{h}^{(0)} \tag{5.130}$$

We can now read off a query schedule from this equation and the block structure of $\mathbf{H}$, as follows. Let:

$$\mathbf{H} = \begin{pmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \\ \mathbf{H}_2 \\ \vdots \\ \mathbf{H}_r \end{pmatrix} \tag{5.131}$$

be the partition of $\mathbf{H}$ into blocks of row according to (5.128). The block $\mathbf{H}_0$ is all zero. For for $1 \leq k \leq r$ the block $\mathbf{H}_k$ is given by:

$$\mathbf{H}_k = \begin{pmatrix} \mathbf{M}_k & \mathbf{c}_k & 0 \end{pmatrix} \tag{5.132}$$

Let us partition the vectors $\mathbf{h}$ and $\mathbf{h}^{(0)}$ and the matrix $\mathbf{G}'$ in accordance with the partition of $\mathbf{H}$:

$$\mathbf{h} = \begin{pmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_r \end{pmatrix} \quad \mathbf{h}^{(0)} = \begin{pmatrix} \mathbf{h}_0^{(0)} \\ \mathbf{h}_1^{(0)} \\ \vdots \\ \mathbf{h}_r^{(0)} \end{pmatrix} \quad \mathbf{G}' = \begin{pmatrix} \mathbf{G}_0' \\ \mathbf{G}_1' \\ \vdots \\ \mathbf{H}_r \end{pmatrix} \tag{5.133}$$

Altogether this partitions the transformed data access equation (5.130) into the following blocks ($0 \leq j \leq r$):

$$\mathbf{h}_j = \mathbf{H}_j \mathbf{u} + \mathbf{G}'_j \mathbf{s} + \mathbf{h}_j^{(0)} \tag{5.134}$$

Because $\mathbf{H}_0$ is all zero we get:

$$\mathbf{h}_0 = \mathbf{G}'_0 \mathbf{s} + \mathbf{h}_0^{(0)} \tag{5.135}$$

In "plain English" the fields in the block $\mathbf{h}_0$ do not depend on the loop indices and are, in fact, determined. We generate searches for all exposed field in the block (see Definition 5.2), and insert searches for the others as they become exposed later in the process. Now lets consider the next block:

$$\mathbf{h}_1 = \begin{pmatrix} \mathbf{c}_1 & 0 \end{pmatrix} \mathbf{u} + \mathbf{G}'_1 \mathbf{s} + \mathbf{h}_1^{(0)} \tag{5.136}$$

Or equivalently ($\mathbf{c}_1$ is just one column-vector):

$$\mathbf{h}_1 = \mathbf{c}_1 \mathbf{u}(1) + \mathbf{G}'_1 \mathbf{s} + \mathbf{h}_1^{(0)} \tag{5.137}$$

This is nothing but a one-dimensional affine join between the fields in the block $\mathbf{h}_1$ and the first element of $\mathbf{u}(1)$ of $\mathbf{u}$. Not all of the fields in $\mathbf{h}_1$ are joinable, because some of them might not be exposed, yet. We schedule a one-dimensional affine join for those fields that are exposed and generate searches for the rest later, as they become exposed. For the blocks $k > 1$, assume inductively that we have scheduled the all the previous blocks. This means that the values of the sub-vector $\mathbf{u}(1 : k-1)$ are determined by the surrounding loops. Consider block k:

$$\mathbf{h}_k = \begin{pmatrix} \mathbf{M}_k & \mathbf{c}_k & 0 \end{pmatrix} \mathbf{u} + \mathbf{G}'_k \mathbf{s} + \mathbf{h}_k^{(0)} \tag{5.138}$$

Since the matrix $\mathbf{M}_k$ has $k-1$ columns, we can rewrite (5.138) as:

$$\mathbf{h}_k = \mathbf{M}_k \mathbf{u}(1 : k-1) + \mathbf{c}_k \mathbf{u}(k) + \mathbf{G}'_k \mathbf{s} + \mathbf{h}_k^{(0)} \tag{5.139}$$

This defines a one-dimensional join between the fields in the block $\mathbf{h}_k$ and the index $\mathbf{u}(k)$:

$$\mathbf{h}_k = \mathbf{c}_k \mathbf{u}(k) + \text{invariants} \tag{5.140}$$

At this point, we schedule a join for all the field in $\mathbf{h}_k$ that are exposed and generate searches for those fields that are determined by previous blocks but become exposed only now. Observe that the data access equation only gives joins for the first r indices in $\mathbf{u}$. For the rest, we just generate dense loops. This proves the second part of the Theorem.

To prove the first part of the Theorem, consider a schedule shown in Program 5.50. Let's stack the blocks of fields joined and searched in the schedule in the data access equation in terms of the transformed loop indices $\mathbf{u} = \mathbf{Ti}$:

$$\begin{pmatrix} \mathbf{h}_0 \\ \mathbf{j}_1 \\ \vdots \\ \mathbf{j}_r \\ \mathbf{h}_r \\ \mathbf{g} \end{pmatrix} = \mathbf{h} = \mathbf{Hu} + \mathbf{G}'\mathbf{s} + \mathbf{h}^{(0)} \tag{5.141}$$

Consider a row of the matrix $\mathbf{H}$ that corresponds to a field in the join block $\mathbf{j}_k$. This row of $\mathbf{H}$ has a non-zero in the k-th column and is all zero afterwards, because the fields in the block $\mathbf{j}_k$ are joined with the index $\mathbf{u}(k)$. Therefore, for every k between 1 and r there is a row of $\mathbf{H}$ with the last non-zero in the k-th column. Observe that the columns $r+1$ through n in $\mathbf{H}$ are all zero: the fields are dependent only on the first r indices in $\mathbf{u}$.

Let $\mathbf{Q}$ be the permutation that sorts the rows of $\mathbf{H}$ in the increasing order of the last non-zero column. Let us apply this permutation to the equation (5.141):

$$\mathbf{Qh} = \mathbf{QHu} + \mathbf{QG}'\mathbf{s} + \mathbf{Qh}^{(0)} \tag{5.142}$$

By construction the matrix $\mathbf{H}' = \mathbf{QH}$ is in echelon form: its rows are sorted by the last non-zero columns and the profile can not increase by more than 1. Observe that the vectors $\mathbf{h}$ and $\mathbf{h}^{(0)}$ and the matrix $\mathbf{G}'$ are the permutations of the vectors $\mathbf{f}$ and $\mathbf{f}^{(0)}$ and the matrix $\mathbf{G}$, respectively, in the data access equation. Let $\mathbf{Z}$ be this permutation. Then we can rewrite (5.142) as:

$$\mathbf{QZf} = \mathbf{H}'\mathbf{u} + \mathbf{QZGs} + \mathbf{QZf}^{(0)} \tag{5.143}$$

Let $\mathbf{P}$ be the composite permutation $\mathbf{P} = \mathbf{QZ}$. Then, (5.143) is equivalent to:

$$\mathbf{Pf} = \mathbf{H}'\mathbf{u} + \mathbf{PGs} + \mathbf{Pf}^{(0)} = \mathbf{P}(\mathbf{P}^{-1}\mathbf{H}'\mathbf{T})(\mathbf{T}^{-1}\mathbf{u}) + \mathbf{PGs} + \mathbf{Pf}^{(0)} =$$

$$= \mathbf{P}(\mathbf{P}^{-1}\mathbf{H}'\mathbf{T})\mathbf{i} + \mathbf{PGs} + \mathbf{Pf}^{(0)} \tag{5.144}$$

We can now drop $\mathbf{P}$ to rewrite the last identity as:

$$\mathbf{f} = (\mathbf{P}^{-1}\mathbf{H}'\mathbf{T})\mathbf{i} + \mathbf{Gs} + \mathbf{f}^{(0)} \tag{5.145}$$

Compare this to the original data access equation:

$$\mathbf{f} = \mathbf{Fi} + \mathbf{Gs} + \mathbf{f}^{(0)} \tag{5.146}$$

Clearly:

$$\mathbf{F} = \mathbf{P}^{-1}\mathbf{H}'\mathbf{T} \tag{5.147}$$

or, equivalently:

$$\mathbf{PFT}^{-1} = \mathbf{H}' \tag{5.148}$$

which proves the Theorem, since $\mathbf{H}'$ is in echelon form. ■

The main corollary of the Theorem is that we can transform the matrix $\mathbf{F}$ in the data access equation into its echelon $\mathbf{H} = \mathbf{PFU}$ form and then mechanically construct the corresponding query schedule. Moreover, the placement of the fields in the resulting one-dimensional joins is given precisely by the permutation matrix $\mathbf{P}$ and the "step" structure of the echelon form matrix $\mathbf{H}$.

As we have alluded to in our example, the resulting loop transformation $\mathbf{T} = \mathbf{U}^{-1}$ is not necessarily legal. However, it is possible to adjust this transformation to a new one $\mathbf{W}$ such that (a) the echelon form $\mathbf{H}' = \mathbf{PFW}^{-1}$ has the same structure as the original, and thus produces essentially the same query schedule and (b) the transformation $\mathbf{W}$ is legal. We now describe how this is done.

5.9.4 Adjusting query scheduling

Here is the outline of our technique. We start with some query schedule and the corresponding echelon form of the data access equation $\mathbf{H} = \mathbf{PFU}$. The loop transformation $\mathbf{T} = \mathbf{U}^{-1}$ is not necessarily legal. In particular, we use the direction/distance vector representation of the dependencies. Let

$$\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_q \tag{5.149}$$

be the dependence vectors for the loop nest. The transformation is not legal because some of the transformed vectors are not lexicographically positive:

$$\exists k : \mathbf{T}\mathbf{d}_k \not\succ 0 \tag{5.150}$$

We find a non-singular matrix $\mathbf{M}$ such that the loop transformation given by $\mathbf{W} = \mathbf{MT}$ is legal:

$$\forall k : \mathbf{W}\mathbf{d}_k \succ 0 \tag{5.151}$$

However, not any matrix is suitable. We want to keep the data access equation of the new transformed loop in the echelon form with the same structure (profile). Let

$$\mathbf{H}' = \mathbf{PFW}^{-1} \tag{5.152}$$

be the transformed data access matrix. It is related to the matrix $\mathbf{H}$ by the inverse of the "correction" $\mathbf{M}$:

$$\mathbf{H}' = \mathbf{PFW}^{-1} = \mathbf{PFT}^{-1}\mathbf{M}^{-1} = \mathbf{PFUM}^{-1} = \mathbf{HM}^{-1} \tag{5.153}$$

It turns out that the matrix $\mathbf{M}$ has special structure.

Here is our main result:

Theorem 5.9 *Let $\mathbf{H}_1$ and $\mathbf{H}_2$ be two $m \times n$ matrices of full column rank such that $\mathbf{H}_1 = \mathbf{H}_2\mathbf{M}$ for some non-singular matrix $\mathbf{M}$. If the matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ are in column-echelon form, then (a) the matrix $\mathbf{M}$ is* **lower-triangular** *and (b) $\mathbf{H}_1$ and $\mathbf{H}_2$ have the same profile.*

Proof By definition of the echelon form, the matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ can be partitioned into blocks of rows ("steps"):

$$\mathbf{H}_1 = \begin{pmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_n \end{pmatrix} \quad \mathbf{H}_2 = \begin{pmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_n \end{pmatrix} \tag{5.154}$$

Since the matrices have full column rank, they can be partitioned into these $n+1$ steps. Our goal is to prove that $\mathbf{M}$ (and $\mathbf{M}^{-1}$) is lower-triangular. We do this by simultaneous induction on the number k of the blocks in the echelon form and the number of rows in $\mathbf{M}$ (and $\mathbf{M}^{-1}$).

Before we proceed, lets define the following quantities:

$$\begin{cases} p_k & \text{The number of rows in block } A_k \\ q_k & \text{The number of rows in block } B_k \end{cases}$$

Observe that p_0 and q_0 can be zero, but $p_k > 0$ and $q_k > 0$ for all $k \geq 1$. It is easy to see that $p_0 = q_0$: take a row $\mathbf{b} = 0$ within the $\mathbf{B}_0$ block. When multiplied by the matrix $\mathbf{M}$ it produces a zero row $\mathbf{a} = \mathbf{M}\mathbf{b}$ at the same position in $\mathbf{A}_0$. Therefore $p_0 \geq q_0$. Converse is also true. Therefore $p_0 = q_0$. The all-zero blocks do not pose any restrictions on the structure of $\mathbf{M}$. However, the subsequent blocks force $\mathbf{M}$ to be lower-triangular. Moreover, we will show that in fact $p_k = q_k$ for all $1 \leq k \leq n$, which implies that $\mathbf{H}_1$ and $\mathbf{H}_2$ have the same profile.

Basis of induction. The first row $\mathbf{a}^T$ of $\mathbf{A}_1$ has exactly one non-zero – the first element. Same is true for the first row $\mathbf{b}^T$ of $\mathbf{B}_1$. Since $\mathbf{a}^T = \mathbf{b}^T\mathbf{M}$, the first row of $\mathbf{M}$ must have exactly one non-zero element: M_{11}. Now $\mathbf{B}_1\mathbf{M}$ has non-zeroes only in its first column, and, therefore, must have at least as many rows as $\mathbf{A}_1$ ($\mathbf{A}_1$ is a sub-matrix of $\mathbf{B}_1\mathbf{M}$). It follows that $p_k \leq q_k$. A symmetric argument about the structure of $\mathbf{M}^{-1}$ tells us that $q_k \leq p_k$, and, therefore $p_k = q_k$.

Inductive step. Assume that the first $k-1$ rows of $\mathbf{M}$ and $\mathbf{M}^{-1}$ form a lower-triangular matrix. Also assume that $p_j = q_j$ for $1 \leq j \leq k-1$. The first row $\mathbf{a}$ of $\mathbf{A}_k$ is of the form

$$\mathbf{a} = (a_1, a_2, \dots, a_k, 0, \dots, 0)^T \tag{5.155}$$

Similarly, the first row of $\mathbf{B}_k$ is of the form:

$$\mathbf{b} = (b_1, b_2, \dots, b_k, 0, \dots, 0) \tag{5.156}$$

Because the sizes of the blocks 1 through $k-1$ are the same, we have that $\mathbf{a}$ and $\mathbf{b}$ occupy the same positions in $\mathbf{H}_1$ and $\mathbf{H}_2$, respectively. Therefore:

$$\mathbf{a} = \mathbf{b} \cdot \mathbf{M} \tag{5.157}$$

By induction hypothesis $\mathbf{M}(1 : (k-1), :)$ is lower-triangular. This means that $\mathbf{M}(1 : (k-1), k : n) = \mathbf{0}$. This, together with (5.157), gives us the following equalities:

$$\begin{cases} b_k & = & a_k \cdot M_{kk} & & \\ b_{k+1} & = & a_k \cdot M_{k,k+1} & = & 0 \\ b_{k+2} & = & a_k \cdot M_{k,k+2} & = & 0 \\ & & \vdots & & \\ b_n & = & a_k \cdot M_{kn} & = & 0 \end{cases} \tag{5.158}$$

Because $a_k \neq 0$, it follows that $\mathbf{M}(k, (k+1) : n) = \mathbf{0}$. This proves that $\mathbf{M}(1 : k, :)$ is lower-triangular. To complete the proof we need to show that $p_k = q_k$.

Since $\mathbf{M}(1 : k, :)$ is lower-triangular, $\mathbf{B}_k\mathbf{M}$ has the same non-zero structure as $\mathbf{B}_k$ and must be a sub-matrix of $\mathbf{A}_k$. Therefore, $\mathbf{A}_k$ has at least as many rows as $\mathbf{B}_k$. By a symmetric argument about $\mathbf{M}^{-1}$, we can show that $\mathbf{B}_k$ must have at least as many rows as $\mathbf{A}_k$. That is $p_k = q_k$. ■

We can now state the relationship between various echelon forms of the data access matrix $\mathbf{F}$:

Corollary 5.10 *Let $\mathbf{F}$ be an $m \times n$ matrix of rank $r \leq n$. Let $\mathbf{H}_1 = \mathbf{PFU}_1$ and $\mathbf{H}_2 = \mathbf{PFU}_2$ be two column echelon forms of the matrix $\mathbf{F}$ with the same row permutation matrix $\mathbf{P}$. Then there exists a non-singular matrix $\mathbf{M}$ such that $\mathbf{H}_1 = \mathbf{H}_2\mathbf{M}$ and $\mathbf{M}$ has the structure:*

$$\mathbf{M} = \begin{pmatrix} \mathbf{M}_{11} & 0 \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \tag{5.159}$$

where $\mathbf{M}_{11}$ is an r-by-r lower-triangular matrix. Moreover, $\mathbf{H}_1$ and $\mathbf{H}_2$ have the same profile.

We will say that such matrix $\mathbf{M}$ is *partially block lower-triangular* or PBLT.

Proof: Observe that because $\mathbf{U}_1$ and $\mathbf{U}_2$ are non-singular the echelon form matrices are related via a column transformation (multiplication on the right):

$$\mathbf{H}_1 = \mathbf{PFU}_1 = \mathbf{PFU}_2\mathbf{U}_2^{-1}\mathbf{U}_1 = \mathbf{H}_2(\mathbf{U}_2^{-1}\mathbf{U}_1) \tag{5.160}$$

That is the matrix $\mathbf{M}$ is given by:

$$\mathbf{M} = \mathbf{U}_2^{-1}\mathbf{U}_1 \tag{5.161}$$

Now we have to show that $\mathbf{M}$ is PBLT. Since the rank of $\mathbf{F}$ is $r \leq n$ the echelon forms have the structure:

$$\mathbf{H}_1 = \begin{pmatrix} \mathbf{L}_1 & \mathbf{0} \end{pmatrix} \tag{5.162}$$

$$\mathbf{H}_2 = \begin{pmatrix} \mathbf{L}_2 & \mathbf{0} \end{pmatrix} \tag{5.163}$$

where $\mathbf{L}_1$ and $\mathbf{L}_2$ are $m \times r$ matrix of rank r which are in echelon form. $\mathbf{0}$ denotes $n - r$ all zero columns. Let us partition $\mathbf{M}$ into the blocks:

$$\begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \tag{5.164}$$

where $\mathbf{M}_{11}$ has r rows and r columns. Then:

$$\mathbf{H}_2\mathbf{M} = \begin{pmatrix} \mathbf{L}_2 & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_2\mathbf{M}_{11} & \mathbf{L}_2\mathbf{M}_{12} \end{pmatrix} = \mathbf{H}_1 = \begin{pmatrix} \mathbf{L}_1 & \mathbf{0} \end{pmatrix} \tag{5.165}$$

We have the following restrictions on the blocks of $\mathbf{M}$:

$$\mathbf{L}_2\mathbf{M}_{11} = \mathbf{L}_1 \tag{5.166}$$

$$\mathbf{L}_2\mathbf{M}_{12} = \mathbf{0} \tag{5.167}$$

$\mathbf{M}_{11}$ is non-singular and it follows from Theorem 5.9 that it is also lower-triangular. Because $\mathbf{L}_2$ has full column rank only $\mathbf{M}_{12} = \mathbf{0}$ can satisfy (5.167). This proves that $\mathbf{M}$ is PBLT. ■

It is a well known fact that the inverse of a lower-triangular matrix is also lower-triangular. It is easy to show that the inverse of a PBLT matrix is also PBLT:

$$\begin{aligned} &\begin{pmatrix} \mathbf{M}_{11} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{N}_{11} & \mathbf{N}_{12} \\ \mathbf{N}_{21} & \mathbf{N}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{I}_r & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{n-r} \end{pmatrix} \Rightarrow \mathbf{M}_{11}\mathbf{N}_{11} = \mathbf{I}_r \wedge \mathbf{M}_{11}\mathbf{N}_{12} = \mathbf{0} \\ &\qquad \Rightarrow \mathbf{N}_{11} = \mathbf{M}_{11}^{-1} \wedge \mathbf{N}_{12} = \mathbf{0} \Rightarrow \mathbf{N}_{11} \text{ is lower tri.} \wedge \mathbf{N}_{12} = \mathbf{0} \end{aligned} \tag{5.168}$$

where $\mathbf{I}_k$ is a $k \times k$ identity matrix.

These results give us an idea for finding the correction matrix $\mathbf{M}$: restrict $\mathbf{M}$ to be PBLT! That is, given an query schedule with an illegal loop transformation $\mathbf{T}$ we try to find a PBLT $\mathbf{M}$ such that the new transformation $\mathbf{W} = \mathbf{MT}$ is legal. However, the problem is even simpler. As we show in Theorem 5.11 below, if there is a PBLT correction matrix $\mathbf{M}$ then there exists another correction matrix $\mathbf{N}$ which is defined by:

$$\mathbf{N} = \begin{pmatrix} \text{sign}(\text{diag}(\mathbf{M}_{11})) & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \tag{5.169}$$

Where the function diag($\mathbf{A}$) returns the diagonal matrix with the same diagonal as $\mathbf{A}$ and the function sign($\mathbf{A}$) returns a matrix which the entries set to $+1$, -1 or 0 depending on the sign of the corresponding entries in $\mathbf{A}$. In other words, we have to look for a correction matrix of the form:

$$\mathbf{N} = \begin{pmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \tag{5.170}$$

where $\mathbf{J}$ is a diagonal matrix with ± 1 on the diagonal. We say the matrix $\mathbf{N}$ is *partially block-diagonal* (PBD). First, we describe how to find the correction PBD matrix, then we state and prove the Theorem 5.11 that states that we can use a PBD matrix instead of PBLT one.

Let $\mathcal{D} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_q\}$ be the set of dependence vectors for the loop in question. We want to find a PBD matrix $\mathbf{N}$ such that:

$$\forall k : \mathbf{NTd}_k \succ 0 \tag{5.171}$$

The dimension r of the diagonal block $\mathbf{J}$ in $\mathbf{N}$ equals to the rank of the data access matrix $\mathbf{F}$. Let $j_p = \pm 1$, $1 \leq p \leq r$, be the (diagonal) elements of $\mathbf{J}$. We start by determining the entries, as follows.

Let $\mathbf{W} = \mathbf{NT}$. Let $\mathbf{w}_i^T$ be the i-th row of the matrix $\mathbf{W}$. Let $\mathbf{t}_i^T$ be the i-th row of $\mathbf{T}$. The first r rows of the matrices are related by:

$$\mathbf{w}_i = j_i \mathbf{t}_i \quad 1 \leq i \leq r \tag{5.172}$$

We need to determine the signs of j_i's in order to correct the first r rows of the transformation. Consider the first row. Let's divide the dependence vectors into two sets:

$$\begin{aligned} \mathcal{D}_1 &= \{\mathbf{d} \in \mathcal{D} | \mathbf{t}_1^T \mathbf{d} \neq 0\} \\ \mathcal{D}_2 &= \mathcal{D} \setminus \mathcal{D}_1 \end{aligned} \tag{5.173}$$

The legality of the first row of the new transformation is determined only by the vectors in the first set $\mathcal{D}_1$. Now, if the signs of the products $\mathbf{t}_1^T\mathbf{d}$ are the same for all $\mathbf{d} \in \mathcal{D}_1$, the we can choose the sign of j_1 appropriately: set $j_1 = 1$ is the products are all positive, $j_1 = -1$ is they are all negative. If the signs of the products are not the same, then it is not possible to correct the transformation $\mathbf{T}$. Setting j_1 (if possible) satisfies the dependence vectors in $\mathcal{D}_1$. For the rest, we recursively repeat the same procedure with the remaining dependence vectors. The algorithm for finding the matrix $\mathbf{J}$ is summarized below:

Program 5.52

```
let 𝒟 be the set of dependence vectors
for i=1, r
    𝒟_i := {d ∈ 𝒟 | t_i^T d ≠ 0}
    if the products are all positive or zero then
        j_i := 1
    else if the products are all negative or zero then
        j_i := -1
    else
        no correction is possible
    endif
```

$\mathcal{D}$:= $\mathcal{D} \setminus \mathcal{D}_i$

end

Now that we have found the matrix $\mathbf{J}$, we essentially have computed the first r rows of the transformation $\mathbf{W}$:

$$\mathbf{W}_1 = \mathbf{W}(1:r,:) = \begin{pmatrix}\mathbf{J} & \mathbf{0}\end{pmatrix}\mathbf{T} \tag{5.174}$$

This *partial transformation* does not violate the dependencies:

$$\forall \mathbf{d} \in \mathcal{D} : \mathbf{W}_1\mathbf{d} \succ 0 \wedge \mathbf{W}_1\mathbf{d} = 0 \tag{5.175}$$

How do we compute the remaining row of the correction matrix ($\mathbf{M}_{21}$ and $\mathbf{M}_{22}$ in (5.170))? Observe that the remaining rows of $\mathbf{W}$ are related to the rows of the correction matrix by:

$$\mathbf{W}_2 = \mathbf{W}(r+1:m,:) = \begin{pmatrix}\mathbf{M}_{21} & \mathbf{M}_{22}\end{pmatrix}\mathbf{T} \tag{5.176}$$

We can either compute $\mathbf{M}_{21}$ and $\mathbf{M}_{22}$ and then set $\mathbf{W}_2$, or compute the remaining rows $\mathbf{W}_2$ of the new transformation and rest assured that there exists a corresponding PBD correction matrix, because:

$$\begin{pmatrix}\mathbf{M}_{21} & \mathbf{M}_{22}\end{pmatrix} = \mathbf{W}_2\mathbf{T}^{-1} \tag{5.177}$$

We find the remaining rows of $\mathbf{W}$ by applying the completion procedure of Li and Pingali [55, 56]!

Observe that the first r rows of the transformation $\mathbf{T}$ correspond to the loops over one-dimensional joins. The remaining rows of $\mathbf{T}$ correspond to the inner dense loops in the schedule that enumerate over the loop indices unconstrained by the fields of the relations because the data access equation does not have full rank. In order to correct the schedule we need to perform two tasks:

- Reverse those of the first r loops for this the element j_i of the diagonal $\mathbf{J}$ in the correction matrix is -1. Other of the first r loops are unchanged.
- Generate the rest $n-r$ loops according to the completed rows of $\mathbf{W}$.

What if it is not possible to correct the current schedule? This can happen either because we can not compute the signs of the elements of $\mathbf{J}$ or because we can not complete the transformation $\mathbf{W}$. The simple answer is to generate another schedule using the query rewrite rules. We are guaranteed to find a correct schedule simply because the schedule that enumerates over the original loops and then searches into all of the relations can be generated by the rewrite rules. At this point we do not have enough experience to provide a heuristic search procedure.

To finish this Section, we prove the Theorem that relates correction matrices in PBLT and PBD forms.

Theorem 5.11 *Let* $\mathbf{M}$ *be an* $n \times n$ *non-singular matrix in PBLT form:*

$$\mathbf{M} = \begin{pmatrix}\mathbf{L} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22}\end{pmatrix} \tag{5.178}$$

Let $\mathbf{N}$ *be the corresponding matrix in PBD form:*

$$\mathbf{N} = \begin{pmatrix}\mathbf{J} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22}\end{pmatrix} \quad \mathbf{J} = \operatorname{sign}(\operatorname{diag}(\mathbf{L})) \tag{5.179}$$

Let $\mathbf{T}$ *be an* $n \times n$ *non-singular matrix. Let* $\mathbf{i}_1, \mathbf{i}_2 \in \mathbb{Z}^n$ *be two integer vectors. Then their images under the transformation* $\mathbf{MT}$ *are ordered exactly as their images under* $\mathbf{NT}$*:*

$$\mathbf{MTi}_1 \succ \mathbf{MTi}_2 \ \Leftrightarrow \mathbf{NTi}_1 \succ \mathbf{NTi}_2 \tag{5.180}$$

In "plain English" the Theorem says that the off-diagonal elements of lower-triangular block $\mathbf{L}$ in the PBLT matrix do not matter. This is also an expression of the well-known fact that skewing an inner loop nest by a multiple of an outer loop nest does not change the order of the enumeration of the iteration points.

Proof: Let $\mathbf{u} = \mathbf{MTi}$ for some $\mathbf{i} \in \mathbb{Z}^n$. Let $\mathbf{v} = \mathbf{NTi}$. These vectors are related by:

$$\mathbf{v} = \mathbf{NTi} = \mathbf{NTT}^{-1}\mathbf{M}^{-1}\mathbf{u} = \mathbf{NM}^{-1}\mathbf{u} \tag{5.181}$$

Consider the matrix $\mathbf{A} = \mathbf{NM}^{-1}$:

$$\mathbf{A} = \mathbf{NM}^{-1} = \begin{pmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{L}^{-1} & \mathbf{0} \\ \mathbf{M}'_{21} & \mathbf{M}'_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{JL}^{-1} & \mathbf{0} \\ \mathbf{M}_{21}\mathbf{L}^{-1} + \mathbf{M}_{22}\mathbf{M}'_{21} & \mathbf{M}_{22}\mathbf{M}'_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{JL}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{n-r} \end{pmatrix} \tag{5.182}$$

where the last equality follows from the definition of the inverse of the matrix $\mathbf{M}$:

$$\mathbf{MM}^{-1} = \begin{pmatrix} \mathbf{I}_r & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{n-r} \end{pmatrix} = \begin{pmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{L}^{-1} & \mathbf{0} \\ \mathbf{M}'_{21} & \mathbf{M}'_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{I}_r & \mathbf{0} \\ \mathbf{M}_{21}\mathbf{L}^{-1} + \mathbf{M}_{22}\mathbf{M}'_{21} & \mathbf{M}_{22}\mathbf{M}'_{22} \end{pmatrix} \tag{5.183}$$

Let us consider the block $\mathbf{K} = \mathbf{JL}^{-1}$ of $\mathbf{A}$. Since $\mathbf{J}$ is diagonal and $\mathbf{L}^{-1}$ is lower-triangular, $\mathbf{K}$ is lower-triangular, as well. Moreover, it has positive diagonal entries. It is easy to show that diagonal elements of $\mathbf{L}^{-1}$ have the same signs as the corresponding diagonal elements of $\mathbf{L}$ and, therefore, of $\mathbf{J}$. It follows that the diagonal of $\mathbf{K}$ is all positive.

Now we have that the images $\mathbf{u}$ and $\mathbf{v}$ under the transforms $\mathbf{M}$ and $\mathbf{N}$ are related by the matrix $\mathbf{A}$:

$$\mathbf{v} = \mathbf{Au} \tag{5.184}$$

and the matrix has the form:

$$\mathbf{A} = \begin{pmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{5.185}$$

where $\mathbf{K}$ is lower-triangular with positive diagonal. Observe that the inverse of $\mathbf{A}$ has the same form:

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{K}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{5.186}$$

because $\mathbf{K}^{-1}$ is also lower-triangular with positive diagonal.

It remains for us to prove that $\mathbf{A}$ preserves lexicographic order. The same will follow for $\mathbf{A}^{-1}$, we well. In particular, we have to prove that if a vector $\mathbf{c}$ is lexicographically positive than so is its image $\mathbf{Ac}$. By the definition of the lexicographic order:

$$\mathbf{c} \succ 0 \Leftrightarrow \exists 0 \le k < n : \mathbf{c}(1:k) = \mathbf{0} \wedge \mathbf{c}(k+1) > 0 \tag{5.187}$$

That is $\mathbf{c}$ has a (possible empty) prefix of zeros followed by a positive element. Let k be the length of the prefix. We have two cases:

- $k < r$, where r is the dimension of the lower-triangular block $\mathbf{K}$ in $\mathbf{A}$. Then:

$$\mathbf{Ac} = \begin{pmatrix} \mathbf{Kc}(1:r) \\ \cdots \end{pmatrix} = \begin{pmatrix} \mathbf{K}(1:k, 1:k)\mathbf{c}(1:k) \\ \mathbf{K}(k+1, 1:k)\mathbf{c}(1:k) + \mathbf{K}(k+1, k+1)\mathbf{c}(k+1) \\ \cdots \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{K}(k+1, k+1)\mathbf{c}(k+1) \\ \cdots \end{pmatrix} \quad (5.188)$$

  where "$\ldots$" denotes the part of the product that we do not care about. Observe that $\mathbf{K}(k+1, k+1) > 0$ and $\mathbf{c}(k+1) > 0$, therefore $\mathbf{Ac} \succ 0$.

- $r \leq k < n$. Then:

$$\mathbf{Ac} = \begin{pmatrix} \mathbf{Kc}(1:r) \\ \mathbf{c}(r+1, n) \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{c}(r+1, n) \end{pmatrix} \succ 0 \quad (5.189)$$

  because the leading non-zero element of $\mathbf{c}$ appears at the position $k+1 > r$.

■

Chapter 6

Experimental studies

In the preceding chapters we have discussed sequential sparse matrix code generation. In this Chapter we compare the performance of the compiler-generated code against that of several sparse matrix libraries:

- *Sparse BLAS* library developed at the National Institute of Standards and Technology (NIST) by Pozo and Remington [68]. The library contains the minimal set of routines that our compiler should be able to support. Experiments described in Section 6.1 demonstrate the quality of the output code.
- IBM Engineering and Scientific Software Library (ESSL, version 2, release 2) [28]. In Section 6.2 we study performance of Preconditioned Conjugate Gradient algorithm as provided by the library and generated by our compiler.
- BlockSolve library from Argonne National Laboratories [46,47]. The library makes extensive use of dense sub-matrices that arise in certain application domains. In Section 6.3 we demonstrate the impact of applying BLAS pattern-matching (Section 5.8.1) in this case.

6.1 NIST Sparse BLAS library

In this Section we discuss Sparse Basic Linear Algebra Subroutines (SPBLAS) library that is under development at the National Institute of Standards and Technology (NIST) [68]. The library provides implementations of matrix-vector product and the solution of triangular systems:

$$\mathbf{y} = \alpha\mathbf{y} + \beta\mathbf{A}\mathbf{x} \tag{6.1}$$

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y} \tag{6.2}$$

In the computations the vectors are dense. The sparse matrix formats targeted by the library are listed in Table 6.1. Matrix-vector products are provided for all formats. Some formats, as indicated by the "No TS" column in the table, are inappropriate for triangular system solution. We have developed Black Boxes for a subset of the formats, as indicated by the "BB" column and have compared the performance of the code generated by the compiler with the code provided in the library. We start in Section 6.1.1 by describing this subset of formats. Performance measurements are presented in Section 6.1.2.

6.1.1 Formats

Point formats

The formats in the library can be divided into *point* formats and *block* formats. Point formats maintain the indices for individual non-zero elements. Block formats maintain indices for dense blocks of non-zero elements. We start with point formats. We use the naming conventions from the SPBLAS library specification for the formal arguments to the Black Boxes.

Table 6.1: Sparse matrix formats in the SPBLAS library

Name	Description	No TS	BB
Point formats			
COO	Coordinate	X	X
CSC	Compressed Sparse Column		X
CSR	Compressed Sparse Row		X
DIA	Sparse Diagonal	X	
ELL	Ellpack/Itpack		X
SKY	Skyline		
Block formats			
BCO	Block coordinate	X	
BSC	Block compressed sparse column		
BSR	Block compressed sparse row		X
BDI	Block sparse diagonal	X	
BEL	Block Ellpack/Itpack		
VBR	Variable block compressed sparse row		X

COO COO format is the coordinate storage format, illustrated in Figure 4.6 on page 59. Here is the Black Box for this format:

Program 6.1
Formal arguments: `n, nnz, indx, jndx, val`

`CCO.schema()` $\equiv \langle\ \langle i, int\rangle,\ \langle j, int\rangle,\ \langle v, real\rangle\ \rangle$

`COO.hierarchy()` $\equiv \langle\ \langle i, j\rangle^* \prec v$:`Real` $\rangle$

`COO.bounds()` $\equiv \langle\ 1 \leq i, j \leq n\ \rangle$

`COO.h()` $\equiv \langle$ `(COO1 n=n indx=indx jndx=jndx val=val)` $\rangle$

`COO1.fields` $\equiv \langle\ \langle i, j\rangle\ \rangle$

`COO1.special` $\equiv \langle$ `Indirect` $\rangle$

`COO1.lb()` $\equiv$ `Expr`$\langle$ `1` $\rangle$

`COO1.ub()` $\equiv$ `Expr`$\langle$ `nnz` $\rangle$ *iterator:* $1 \leq k \leq$ `nnz`

`COO1.sorted_p()` $\equiv \langle$ `false` $\rangle$

`COO1.ideref`$_i$ $\equiv$ `Expr`$\langle$ `indx` $\rangle$

`COO1.ideref`$_j$ $\equiv$ `Expr`$\langle$ `jndx` $\rangle$

`COO1.hderef(k)` $\equiv$ `Expr`$\langle$ `(REAL_VAL deref=val[k])` $\rangle$

CSR Compressed Sparse Row format is a slight variation on the CRS format illustrated in Figure 1.12 on page 23. Instead of keeping a single `ROWPTR` array, this format uses two arrays: `pntrb` and `pntre` – to indicate the start and end of the non-zeros for each row. Here is the Black Box:

Program 6.2
Formal arguments: `n, nnz, val, indx, pntrb, pntrb`

`CSR.schema() =` $\equiv \langle\ \langle i, int\rangle,\ \langle j, int\rangle,\ \langle v, real\rangle\ \rangle$

`CSR.hierarchy()` $\equiv \langle\ i^* \prec j^* \prec v$:`Real` $\rangle$

`CSR.bounds()` $\equiv \langle\ 1 \leq i, j \leq n\ \rangle$

```
CSR.h() ≡ ⟨ (CSR_ROW n=n val=val indx=indx pntrb=pntrb pntre=pntre) ⟩
CSR_ROW.fields() ≡ ⟨ i ⟩
CSR_ROW.special() ≡ ⟨ Dense ⟩
CSR_ROW.lb() ≡ Expr⟨ 1 ⟩
CSR_ROW.ub() ≡ Expr⟨ n ⟩
CSR_ROW.hderef(i) ≡ ⟨
    (CSR_COL
        start=pntrb[i] end=pntre[i]-pntrb[1]
        indx=indx val=val) ⟩
CSR_COL.fields() ≡ ⟨ j ⟩
CSR_COL.special() ≡ ⟨ Indirect ⟩
CSR_COL.lb() ≡ Expr⟨ start ⟩
CSR_COL.ub() ≡ Expr⟨ end ⟩
CSR_COL.sorted_p() ≡ ⟨ false ⟩
CSR_COL.ideref(jj) ≡ Expr⟨ indx[jj] ⟩
CSR_COL.hderef(jj) ≡ Expr⟨ (REAL_VAL deref=val[jj]) ⟩
```

CSC Compressed Sparse Column is a slight variation on the CCS format illustrated in Figure 1.13 on page 23. Instead of keeping a single `COLPTR` array, this format uses two arrays: `pntrb` and `pntre` – to indicate the start and end of the non-zeros for each row. Here is the Black Box:

Program 6.3
Formal arguments: `n, nnz, val, indx, pntrb, pntrb`

CSC.schema() = ≡ ⟨ $\langle i, int\rangle$, $\langle j, int\rangle$, $\langle v, real\rangle$ ⟩

CSC.hierarchy() ≡ ⟨ $j^* \prec i^* \prec v$:Real ⟩

CSC.bounds() ≡ ⟨ $1 \leq i, j \leq n$ ⟩

```
CSC.h() ≡ ⟨ (CSC_COL n=n val=val indx=indx pntrb=pntrb pntre=pntre) ⟩
CSC_COL,fields() ≡ ⟨ j ⟩
CSC_COL,special() ≡ ⟨ Dense ⟩
CSC_COL,lb() ≡ Expr⟨ 1 ⟩
CSC_COL,ub() ≡ Expr⟨ n ⟩
CSC_COL,hderef(j) ≡ ⟨
    (CSC_ROW
        start=pntrb[j] end=pntre[j]-pntrb[1]
        indx=indx val=val) ⟩
CSC_ROW.fields() ≡ ⟨ i ⟩
CSC_ROW.special() ≡ ⟨ Indirect ⟩
CSC_ROW.lb() ≡ Expr⟨ start ⟩
CSC_ROW.ub() ≡ Expr⟨ end ⟩
CSC_ROW.sorted_p() ≡ ⟨ false ⟩
CSC_ROW.ideref(jj) ≡ Expr⟨ indx[jj] ⟩
CSC_ROW.hderef(jj) ≡ Expr⟨ (REAL_VAL deref=val[jj]) ⟩
```

ELL Ellpack/Itpack format is used to the matrices in the Ellpack [43, 74] system for the solution of elliptic PDE problems. Consider the matrix[1]:

$$\mathbf{A} = \begin{pmatrix} 11 & 0 & 13 & 0 & 0 \\ 21 & 0 & 0 & 24 & 0 \\ 31 & 32 & 33 & 0 & 35 \\ 0 & 42 & 0 & 44 & 0 \\ 0 & 0 & 53 & 0 & 55 \end{pmatrix} \tag{6.3}$$

Let n be the number of rows in the matrix. Let μ be the largest number of non-zero elements in any row of the matrix. In our example $\mu = 4$. The matrix is stored in two dense two-dimensional arrays with n rows and μ columns:

$$\texttt{val} = \begin{pmatrix} 11 & 13 & 0 & 0 \\ 21 & 24 & 0 & 0 \\ 31 & 32 & 33 & 35 \\ 42 & 44 & 0 & 0 \\ 53 & 55 & 0 & 0 \end{pmatrix} \quad \texttt{indx} = \begin{pmatrix} 1 & 3 & 1' & 1' \\ 1 & 4 & 2' & 2' \\ 1 & 2 & 3 & 5 \\ 2 & 4 & 4' & 4' \\ 3 & 5 & 5' & 5' \end{pmatrix} \tag{6.4}$$

`val` stores the non-zero values "shifted to the left". `indx` stores the corresponding column indices. Unless all rows of the matrix have exactly the same number of non-zeros, the array `val` stores zero elements. There many choices as to how to set the corresponding indices in the `indx` array. According the the SPBLAS specification, these indices are set to the row index (cf. the primed indices in (6.4)).

The arrays `val` and `indx` are stored in column-major order. There are two possible ways of implementing sparse matrix-vector products for this format. The simplest one is to treat the format as a variation on CRS:

Program 6.4

```
do i=1,n
    do jj=1,μ
        Y[i] := Y[i] + val[i,jj]*X[indx[jj]]
```

However, this produces stride-n access to `val` array and, subsequently, poor cache performance. A better code is obtained by reordering the loops:

Program 6.5

```
do jj=1,μ
    do i=1,n
        Y[i] := Y[i] + val[i,jj]*X[indx[jj]]
```

In order to expose the compiler to this choice, the `ELL` Black Box exposes the jj field. The index hierarchy is $(i \times jj) \prec j^1 \prec v$: i and jj indices can be enumerated independently and together they fix a single value for the j index.

Program 6.6

Formal arguments: `n, maxnz, val, indx` -- *maxnz is* μ --

`ELL.schema()` $\equiv \langle\ \langle i, int\rangle,\ \langle jj, int\rangle,\ \langle j, int\rangle,\ \langle v, real\rangle\ \rangle$

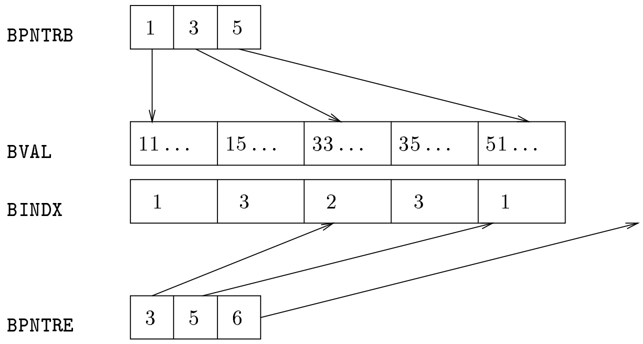
`ELL.hierarchy()` $\equiv \langle\ (i \times jj) \prec j^1 \prec v{:}\texttt{Real}\ \rangle$

`ELL.bounds()` $\equiv \langle\ 1 \leq i, j \leq n \wedge 1 \leq jj \leq \texttt{maxnz}\rangle$

`ELL.h()` $\equiv$ `Expr`$\langle$ `(ELL_PROD n=n maxnz=maxnz val=val indx=indx)` $\rangle$

`ELL_PROD.fields()` $\equiv \langle$ `i, jj` $\rangle$

[1]Example matrices are taken from [24]



BNZ = 5
LB = 2
NB = 5
N = 10

Figure 6.1: BSR format

ELL_PROD.special$_i$() ≡ ⟨ Dense ⟩

ELL_PROD.lb$_i$() ≡ Expr⟨ 1 ⟩

ELL_PROD.ub$_i$() ≡ Expr⟨ n ⟩

ELL_PROD.special$_{jj}$() ≡ ⟨ Dense ⟩

ELL_PROD.lb$_{jj}$() ≡ Expr⟨ 1 ⟩

ELL_PROD.ub$_{jj}$() ≡ Expr⟨ maxnz ⟩

ELL_PROD.hderef(i,jj) ≡ Expr⟨
    (ELL_ENTRY row=i offset=jj val=val indx=indx) ⟩

ELL_ENTRY.fields() ≡ ⟨ j^1 ⟩

ELL_ENTRY.deref() ≡ Expr⟨ indx[row,offset] ⟩

ELL_ENTRY.hderef() ≡ Expr⟨ (REAL_VAL deref=val[row,offset]) ⟩

To specify

Block formats

BSR Block Sparse Row data structure effectively stores a sparse matrix with each entry being a fixed-size square block. Consider an example matrix:

$$\mathbf{A} = \left(\begin{array}{cc|cc|cc} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ \hline 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ \hline 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{array}\right) \tag{6.5}$$

Storage layout for the matrix is illustrated in Figure 6.1. The array VAL stores the non-zero blocks. A block is “non-zero” is any element within it is not a zero. Individual blocks are stored in column-major order. BINDX array stores the block-column indices of the blocks. For example, the block:

$$\begin{pmatrix} 35 & 36 \\ 45 & 46 \end{pmatrix}$$

has the block-column index 3 and block-row index 2. BPNTRB[bi] points to the start of the block-row bi in the VAL and BINDX arrays. BPNTRE[bi]$-BPNTRB[1]$ points to the end of the block-row. LB is the dimension of the blocks. BNZ is the number of non-zero blocks. BNZ$\cdot$LB^2 is the length of the VAL array. NB is the number of block rows and N is the number of rows in the matrix.

The BSR Black Box exposes the block row and column indices bi and bj:

Program 6.7

Formal arguments: n, nb, lb, bnz, bpntrb, bpntre, val, bindx

```
BSR.schema() ≡ ⟨ ⟨bi,int⟩, ⟨bj,int⟩, ⟨i,int⟩, ⟨j,int⟩, ⟨v,real⟩ ⟩
BSR.hierarchy() ≡ ⟨ bi* ≺ bj* ≺ (i × j) ≺ v:Real ⟩
BSR.bounds() ≡ ⟨ 1 ≤ i,j ≤ n ⟩
BSR.h() ≡ Expr⟨
    (BSR_BROW nb=nb lb=lb bpntrb=bpntrb bpntre=bpntre
        val=val bindx=bindx) ⟩
BSR_BROW.fields ≡ ⟨ bi ⟩
BSR_BROW.special() ≡ ⟨ Dense ⟩
BSR_BROW.lb() ≡ Expr⟨ 1 ⟩
BSR_BROW.ub() ≡ Expr⟨ nb ⟩
BSR_BROW.hderef(bi) ≡ Expr⟨
    (BSR_BCOL start=bpntrb[bi] end=bpntre[bi]-bpntrb[1] lb=lb
        val=val bindx=bindx) ⟩
BSR_BCOL.fields ≡ ⟨ bj ⟩
BSR_BCOL.special ≡ ⟨ Indirect ⟩
BSR_BCOL.lb() ≡ Expr⟨ start ⟩
BSR_BCOL.ub() ≡ Expr⟨ end ⟩
BSR_BCOL.ideref() ≡ Expr⟨ bindx ⟩
BSR_BCOL.sorted_p() ≡ ⟨ false ⟩
BSR_BCOL.hderef(bjj) ≡Expr⟨
    (DENSE_MATRIX start=val[1+(bjj-1)*lb*lb] colmajor=true leading=lb
        istart=(1+(bi-1)*lb) jstart=(1+(bindx[bjj]-1)*lb)
        nrow=lb ncol=lb
        fields=⟨ i, j, v ⟩) ⟩
```

The code to enumerate the non-zero entries in the matrix is:

Program 6.8

```
do bi=1,nb
    do bjj=bpntrb[bi], (bpntre[bi]-bpntrb[1])
        istart = (1+(bi-1)*lb)
        jstart = (1+(bindx[bjj]-1)*lb)
        blockstart = 1+(bjj-1)*lb*lb
        do i = istart, istart+lb-1
            do j = jstart, jstart+lb-1
                v = val[blockstart + (j-jstart)*lb + istart]
                print(i, j, v)
```

VBR Variable Block Compressed Sparse Row format is a generalization of the BSR format: the dimensions of the blocks can vary, albeit in a restricted manner.

$$\mathbf{A} = \left(\begin{array}{ccc|c|cc} 11 & 12 & 13 & 0 & 15 & 16 \\ 21 & 22 & 23 & 0 & 25 & 26 \\ \hline 0 & 0 & 0 & 34 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 45 & 46 \\ 0 & 0 & 0 & 0 & 55 & 56 \\ \hline 61 & 62 & 63 & 0 & 65 & 66 \end{array}\right) \tag{6.6}$$

The rows of the matrix are partitioned into mb intervals and the rows of the matrix are partitioned into nb intervals. The intervals are defined by two integer vectors: $\mathbf{R} = (i_1 \ \ldots \ i_{mb} \ i_{mb+1})^T$ and $\mathbf{C} = (j_1 \ \ldots \ j_{nb} \ j_{nb+1})^T$ such that

$$j_1 = i_1 = 1 \tag{6.7}$$
$$i_{mb+1} = m + 1 \tag{6.8}$$
$$j_{nb+1} = n + 1 \tag{6.9}$$

where m is the number of rows in the matrix and n is the number of columns. Let $\mathbf{A}^{(p,q)}$ be the block of the matrix that lies at the intersection of the row interval p and the column interval q. Then, it stored the elements with row indices i such that:

$$i_p \leq i < i_{p+1} \tag{6.10}$$

and column indices j such that:

$$j_q \leq j < j_{q+1} \tag{6.11}$$

In our example, the partitioning vectors are:

$$\mathbf{R} = (1\ 3\ 4\ 6\ 7)^T \quad \mathbf{C} = (1\ 4\ 5\ 7)^T \tag{6.12}$$

The matrix is stored using the following arrays:

$$\begin{aligned} \mathtt{VAL} &= \begin{pmatrix} 11,\ldots,23 & 15,\ldots,26 & 34 & 45,\ldots,46 & 61,\ldots,63 & 65,66 \end{pmatrix} \\ \mathtt{INDX} &= \begin{pmatrix} 1 & 7 & 11 & 12 & 16 & 19 \end{pmatrix} \\ \mathtt{BINDX} &= \begin{pmatrix} 1 & 3 & 2 & 3 & 1 & 3 \end{pmatrix} \\ \mathtt{CPNTR} &= \begin{pmatrix} 1 & 4 & 5 & 7 \end{pmatrix} \\ \mathtt{RPNTR} &= \begin{pmatrix} 1 & 3 & 4 & 6 & 7 \end{pmatrix} \\ \mathtt{BPNTRB} &= \begin{pmatrix} 1 & 3 & 4 & 5 \end{pmatrix} \\ \mathtt{BPNTRE} &= \begin{pmatrix} 3 & 4 & 5 & 7 \end{pmatrix} \end{aligned} \tag{6.13}$$

The `VAL` array stores the values of the blocks with each block laid out in column-major order. The `INDX` array stores the index of the beginning of each block in the `VAL` array. Let bjj be the number of the block in the linearized row-compressed order. Then `VAL[INDX[bjj]]` is the first entry in the block. `BINDX` stores block-column indices bj of the blocks. Consider the block:

$$\mathbf{A}^{(4,1)} = \begin{pmatrix} 61 & 62 & 63 \end{pmatrix} \tag{6.14}$$

Its number in the linearized order it 5. INDX[5] is 16, which means that the block is stored starting with the element VAL[16]. BINDX[5] is 1, which is the block-column index of the block.

CPNTR array basically stores the **C** vector. The first column in the block column bj is given by

$$\texttt{CPNTR}[bj] - \texttt{CPNTR}[1] + 1 \tag{6.15}$$

And the number of columns in the block is

$$\texttt{CPNTR}[bj+1] - \texttt{CPNTR}[bj] \tag{6.16}$$

The last element $\texttt{CPNTR}[\texttt{nb}+1]$ is set to $nb + \texttt{CPNTR}[1]$. The difference between the array and the vector **C** is that the first element of the array is not guaranteed to be 1. RPNTR stores the array **R** in a similar fashion. The first row of the block row bi is given by:

$$\texttt{RPNTR}[bi] - \texttt{RPNTR}[1] + 1 \tag{6.17}$$

And the number of rows in the block is

$$\texttt{RPNTR}[bi+1] - \texttt{RPNTR}[bi] \tag{6.18}$$

BPNTRB array points to where each block row starts in the linearized order: $\texttt{BPNTRB}[bi] - \texttt{BPNTRB}[1] + 1$ is the location in BINDX of the first block in the row. BPNTRE points to where each block ends: $\texttt{BPNTRE}[bi] - \texttt{BPNTRB}[1] + 1$ is the location in BINDX of the last block in the row.

The information in the arrays in somewhat redundant. Here is a quote from [24]:

> "The array INDX is not essential for reconstructing the matrix. However, it is essential for good performance of most matrix operations and should be computed one time and kept for later use."

Similar to the BSR Black Box, the VBR Black Box exposes the block indices:

Program 6.9

Formal arguments: `n, nb, mb, cpntr, rpntr,`
`bpntrb, bpntre, bindx, indx, val`

VBR.schema() $\equiv \langle\ \langle bi, int\rangle,\ \langle bj, int\rangle,\ \langle i, int\rangle,\ \langle j, int\rangle,\ \langle v, real\rangle\ \rangle$

VBR.hierarchy() $\equiv \langle\ bi^* \prec bj^* \prec (i \times j) \prec v{:}\mathtt{Real}\ \rangle$

VBR.bounds() $\equiv \langle\ 1 \le i, j \le n\ \rangle$

VBR.h() $\equiv$ Expr$\langle$

```
    (VBR_BROW nb=nb mb=mb bpntrb=bpntrb bpntre=bpntre
        cpntr=cpntr rpntr=rpntr
        val=val bindx=bindx) ⟩
```

VBR_BROW.fields() $\equiv \langle$ bi $\rangle$

VBR_BROW.special() $\equiv \langle$ Dense $\rangle$

VBR_BROW.lb() $\equiv$ Expr$\langle$ 1 $\rangle$

VBR_BROW.ub() $\equiv$ Expr$\langle$ mb $\rangle$

VBR_BROW.hderef(bi) $\equiv$ Expr$\langle$

```
    (VBR_BCOL start=(bpntrb[bi]-bpntrb[1]+1)
        end=(bpntrb[be]-bpntrb[1]+1)
        rowstart=(rpntr[bi]-rpntr[1]+1)
        rowsize=(rpntr[bi+1]-rpntr[bi])
```

```
        indx=indx cpntr=cpntr val=val) ⟩
VBR_BCOL.fields() ≡ ⟨ bj ⟩
VBR_BCOL.special() ≡ ⟨ Indirect ⟩
VBR_BCOL.lb() ≡ Expr⟨ start ⟩
VBR_BCOL.ub() ≡ Expr⟨ end ⟩
VBR_BCOL.ideref() ≡ Expr⟨ bindx ⟩
VBR_BCOL.hderef(bjj) ≡ Expr⟨
    (DENSE_MATRIX start=val[indx[bjj]] colmajor=true
        leading=rowsize istart=rowstart
        jstart=(cpntr[bindx[bjj]]-cpntr[1]+1)
        nrow=rowsize ncol=(cpntr[bindx[bjj]+1]-cpntr[bindx[bjj]])
        fields=⟨ i, j, v ⟩)
```

The code to enumerate the non-zero entries in the matrix is:

Program 6.10

```
do bi=1,mb
    rowstart=(rpntr[bi]-rpntr[1]+1)
    rowsize=(rpntr[bi+1]-rpntr[bi])
    do bjj=(bpntrb[bi]-bpntrb[1]+1), (bpntrb[be]-bpntrb[1]+1)
        bj = bindx[bjj]
        jstart=(cpntr[bj]-cpntr[1]+1)
        do i = rowstart,(rpntr[bi+1]-rpntr[1])
            do j = jstart,(cpntr[bj+1]-cpntr[1])
                v = val[indx[bjj] + ...
                         ..  + rowsize*(j-jstart) + (i-rowstart)]
                print (i, j, v)
```

Accommodating triangular matrices

We add the `uplo` argument to all of the above black boxes. The argument indicates whether the matrix stored is lower, strict-lower, upper, strict-upper triangular or general (square). A strict triangular matrix only stores the non-zero entries strictly below or above the diagonal. In the unit-triangular solution subroutines of the SPBLAS library such storage is used for *unit*-triangular matrices.

The `bounds` metric must return the bounds in accordance with the setting of the `uplo` argument, as indicated in Table 6.2. If the `uplo` argument is nor specified it is assumed to be set to `G` (general matrix).

Table 6.2: Triangular bounds

Value of `uplo`	Results of `bounds()`
`G` (default)	$1 \leq i, j \leq n$
`L` (lower)	$1 \leq i \leq n \wedge 1 \leq j \leq i$
`LS` (strict lower)	$1 \leq i \ leqn \wedge 1 \leq j < i$
`U` (upper)	$1 \leq j \leq n \wedge 1 \leq i \leq j$
`US` (strict upper)	$1 \leq j \ leqn \wedge 1 \leq i < j$

6.1.2 Experiments

SPBLAS provides implementations of sparse matrix-vector product (MV) for all of the above formats. It also provides triangular system solution (TS) for all of the formats, except COO. In this Section we compare the performance of library code and the code generated by the compiler for both MV and TS (unit lower triangular) benchmarks. We start by describing the input programs given to the compiler and then we present the experimental data.

Input programs

Matrix-vector product is a DOANY computation. Therefore, the input code for all of the formats is of the form:

Program 6.11
```
integer n
declarations of low-level data structures
declaration of the matrix A
real X[n], Y[n]
doany i=1,n; j=1,n
    Y[i] := Y[i] + A[i,j]*X[j]
```

The declarations for point formats (except ELL) have the form:

Program 6.12
```
abstract A(black box name, args ..)
```

The declaration for the ELL format is:

Program 6.13
```
abstract B(ELL, args ..)
projection A⟨i,j,v⟩ of B⟨i,jj,j,v⟩
```

Since there is a functional dependence from $\langle i, j\rangle$ to jj, we can use the projection view (see Section 5.7). The declarations for the block formats have the form:

Program 6.14
```
abstract B(black box name, args ..)
projection A⟨i,j,v⟩ of B⟨bi,bj,i,j,v⟩
```

Recall that the Black Boxes for the block formats define relations with the schema $\langle bi, bj, i, j, v\rangle$ where bi and bj are block coordinates.

The solution of unit lower-triangular systems $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$ is a DOACROSS computation. We write the input programs by making the outer sequential loop explicit. For CSR format the program is:

Program 6.15
```
integer n
declarations of low-level data structures
abstract A(CSR uplo=LS ...)
real X[n], B[n]
doany i=1,n
    X[i] := B[i]
enddo
for i=1,n    sequential!
```

```
    doany j=1,i-1
        X[i] := X[i] - A[i,j]*X[j]
```

We use the uplo = LS flag to indicate that only the elements below the diagonal are stored.
For the CSC format the program is:

Program 6.16

```
integer n
declarations of low-level data structures
abstract A(CSC uplo=LS ...)
real X[n], B[n]
doany i=1,n
    X[i] := B[i]
enddo
for j=1,n    sequential!
    doany i=j+1,n
        X[i] := X[i] - A[i,j]*X[j]
```

Programs 6.15 and 6.16 can be derived using the techniques from Section 5.9.
For the ELL format the program is:

Program 6.17

```
integer n, maxnz
declarations of low-level data structures
abstract A(ELL ...)
-- A has the schema ⟨i,jj,j,v⟩ --
real X[n], B[n]
doany i=1,n
    X[i] := B[i]
enddo
for i=1,n    sequential!
    doany jj=1,maxnz; j=1,i-1
        X[i] := X[i] - A[i,jj,j]*X[j]
```

For the block formats we use the program:

Program 6.18

```
integer n, mb, nb
declarations of low-level data structures
abstract A(...)  has schema ⟨bi,bj,i,j⟩
real X[n], B[n]
doany i=1,n
    X[i] := B[i]
enddo
for bi=1,mb    sequential!
    doany bj=1,nb; i=1,n;j=1,n
        X[i] := X[i] - A[bi,bj,i,j]*X[j]
```

Performance measurements

We have measured the performance of the library and compiler-generated codes on the five matrices from the Harwell-Boeing collection (obtained through Matrix Market [22]). The properties of the matrices are summarized in Table 6.3. N is the number of rows and columns and NNZ is the number of non-zeros.

The experiments were run on a single thin node of an IBM SP-2 at Cornell, which is essentially an RS/6000 workstation with 120MHz POWER2 Super Chip (P2SC) processor and 256 Mbytes of memory [65]. Figures 6.2 and 6.3 show the comparative performance of the library and compiler-generated code for the point formats. The data for the compiler-generated code is marked with the "BERNOULLI" name. (Current release of the SPBLAS library does not contain the implementation of TS and MV for the ELL format.)

For block formats we have created matrices with block entries by "blowing-up" each non-zero in the above matrices into a block of dimension between 1 and 25. This has limited the initial sizes of the benchmark matrices. For example, the BCSPWR10 matrix with each entry replaced by the 25×25 block contains about 13M non-zeros. Figures 6.4 and 6.5 demonstrate the performance of the MV benchmark for BSR and VBR formats. Figures 6.6 and 6.7 demonstrate the performance of the TS benchmark.

In the case of point formats, the performance of the compiler-generated code is quite close to that of the library code. We believe that the differences are explained by the removal of some address computations in the hand-written code. In the case of block formats, the compiler-generated code performs as well or better than the library code. The reason is that the library code checks whether the entry $\mathbf{X}$[j] being multiplied within the loop is zero. If so, then a whole column of the dense block can be skipped. However, in our experiments most of the elements of the vector $\mathbf{X}$ were non-zero, thus making the conditional redundant. Without this conditional, the codes are identical.

6.2 Iterative methods

In this Section we measure the performance of the Preconditioned Conjugate Gradients algorithm (PCG) with level-0 Incomplete Cholesky Factorization preconditioner (ICC(0)) described in Section 1.2.1. One of the disadvantages of the SPBLAS library (and similar approaches) is that it is infeasible to produce a library of sparse routines that supports multiple formats and multiple sparse operands. A case in point is the ICC computation.

6.2.1 Input to the compiler

The following program computes the Incomplete Cholesky factor $\mathbf{H}$ of a matrix $\mathbf{A}$:

Program 6.19
```
initialize H to the lower triangle of A
for k=1 to n
    H(k,k) = √H(k,k)
    for i=k+1 to n
        H(i,k) = H(i,k) / H(k,k)
    for j=k+1,n
        for i=j,n
            if H(i,j) ≠ 0 then
                H(i,j) = H(i,j) - H(i,k) * H(j,k)
```

This is clearly a DOACROSS computation. However, it can be decomposed into a combination of sequential loops and DOANY loops or, equivalently, BLAS operations:

Program 6.20
```
initialize H to the lower triangle of A
```

Table 6.3: Summary of benchmark matrices

Name	N	NNZ	Application
BCSPWR10	5300	21842	Power network analysis
CAN_1072	1072	12444	Structural analysis in aircraft design
BLKHOLE	2132	14872	Structural engineering
DWT_1242	1242	10426	FEM analysis
BCSSTM13	2003	21181	Structural engineering

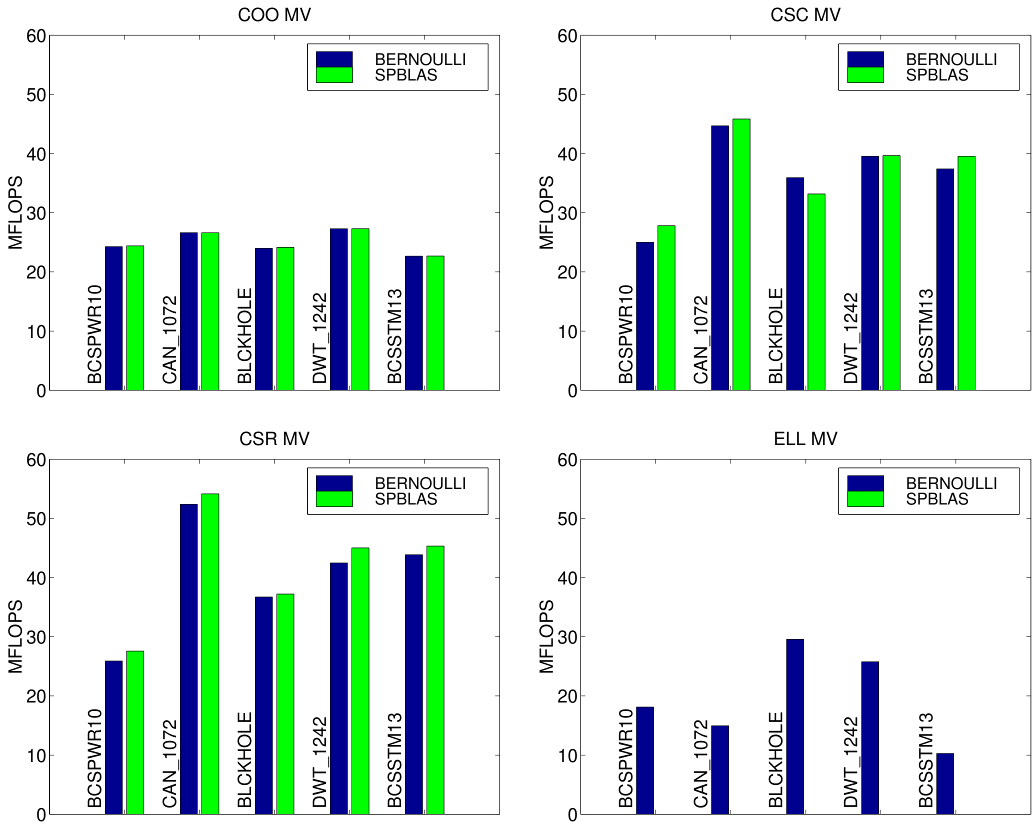


Figure 6.2: Performance of sparse matrix-vector product (point formats)

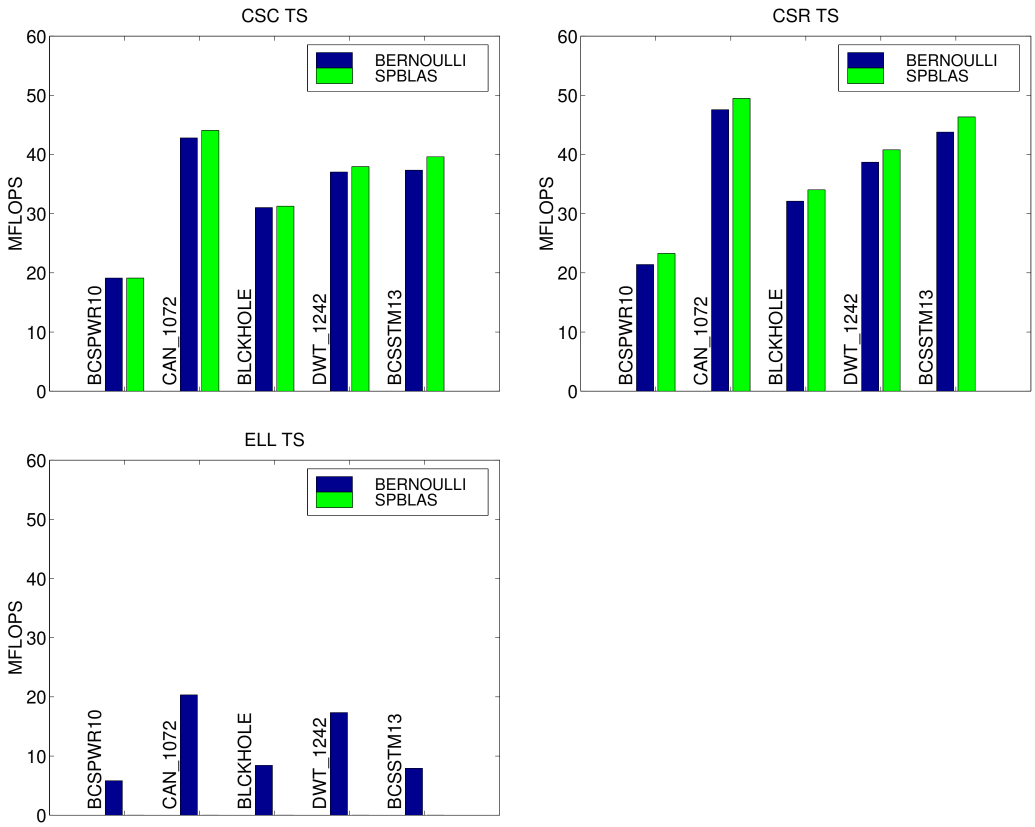


Figure 6.3: Performance of lower-triangular solution (point formats)

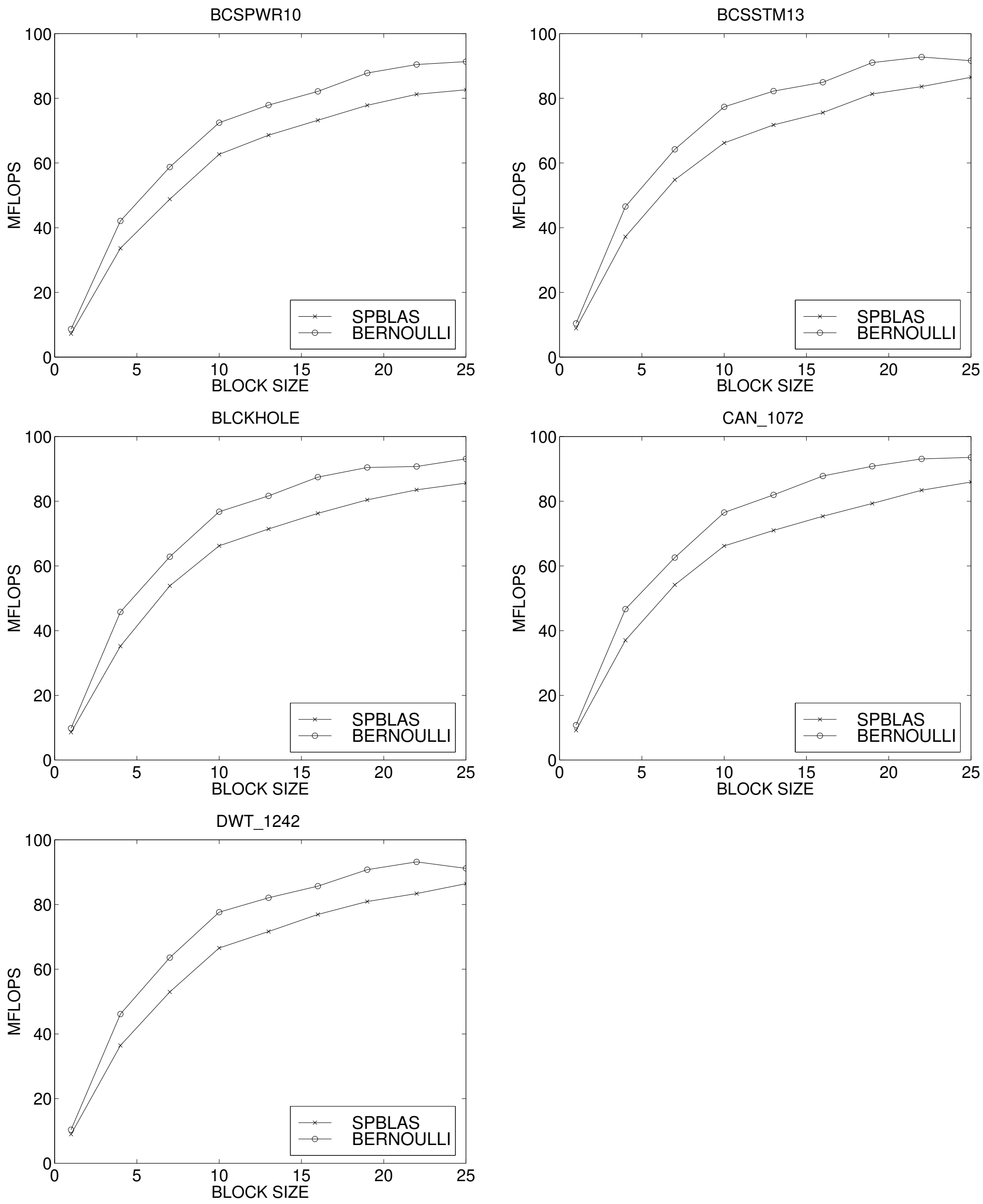


Figure 6.4: Performance of sparse matrix-vector product (BSR format)

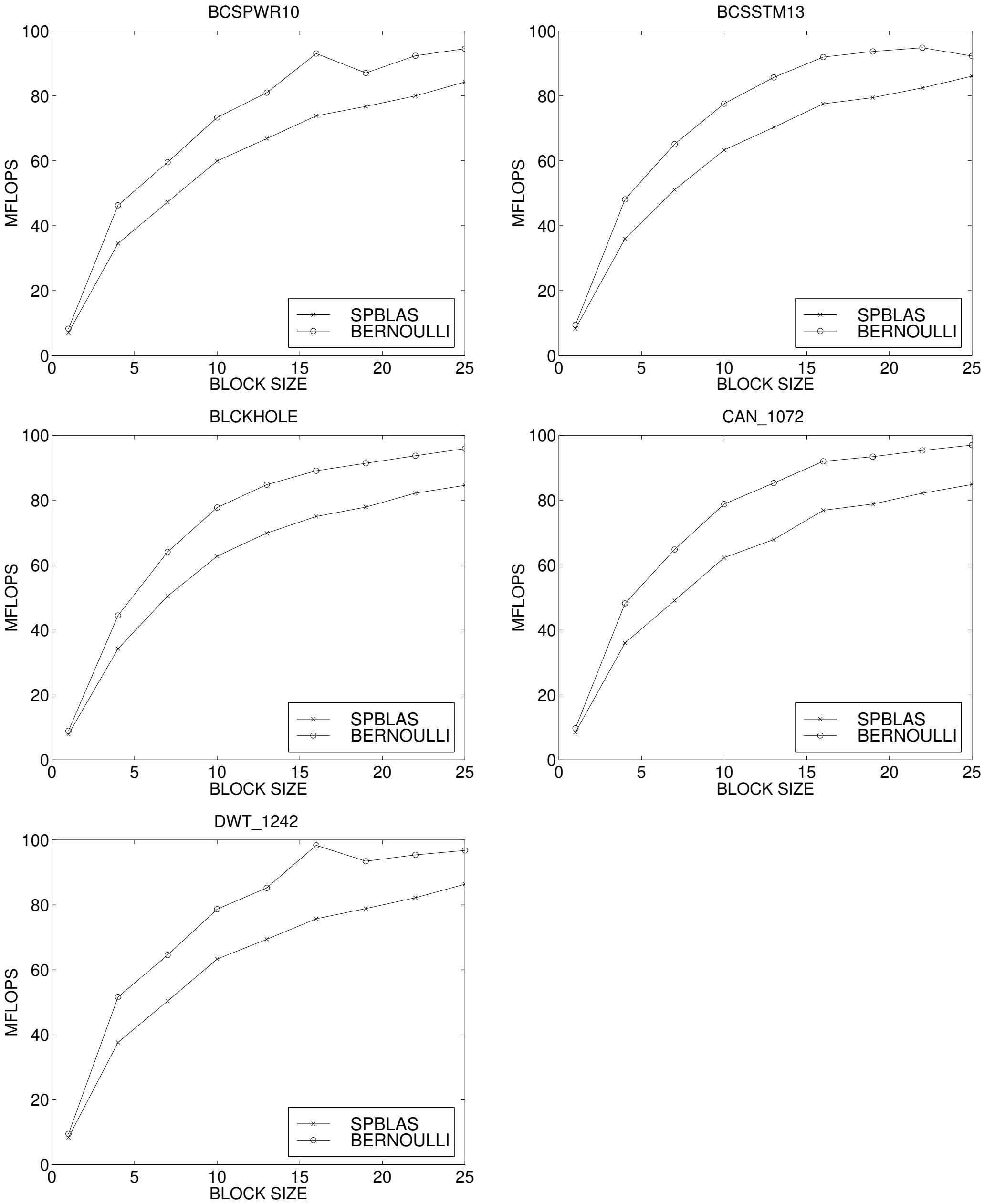


Figure 6.5: Performance of sparse matrix-vector product (VBR format)

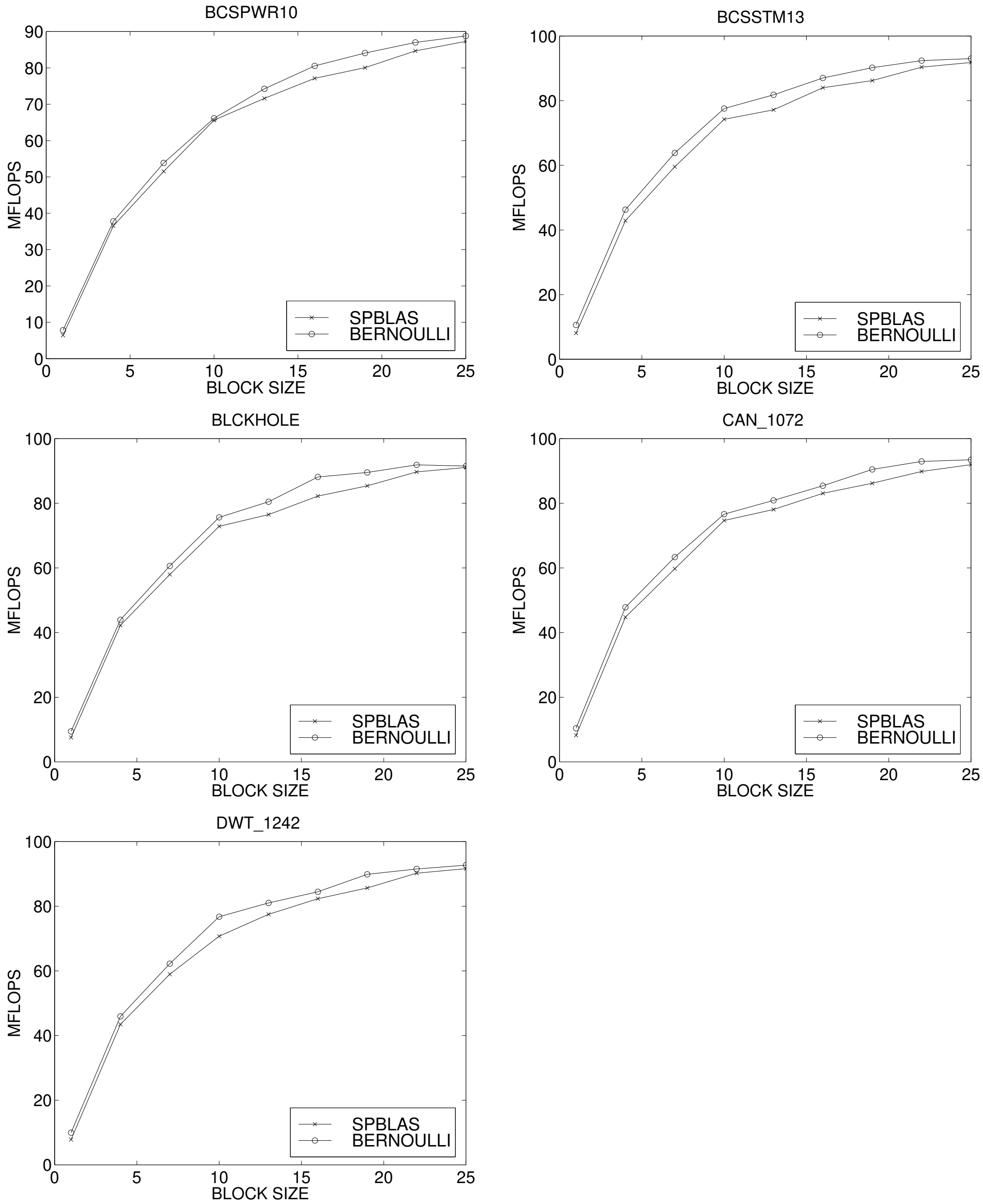


Figure 6.6: Performance of lower-triangular solution (BSR format)

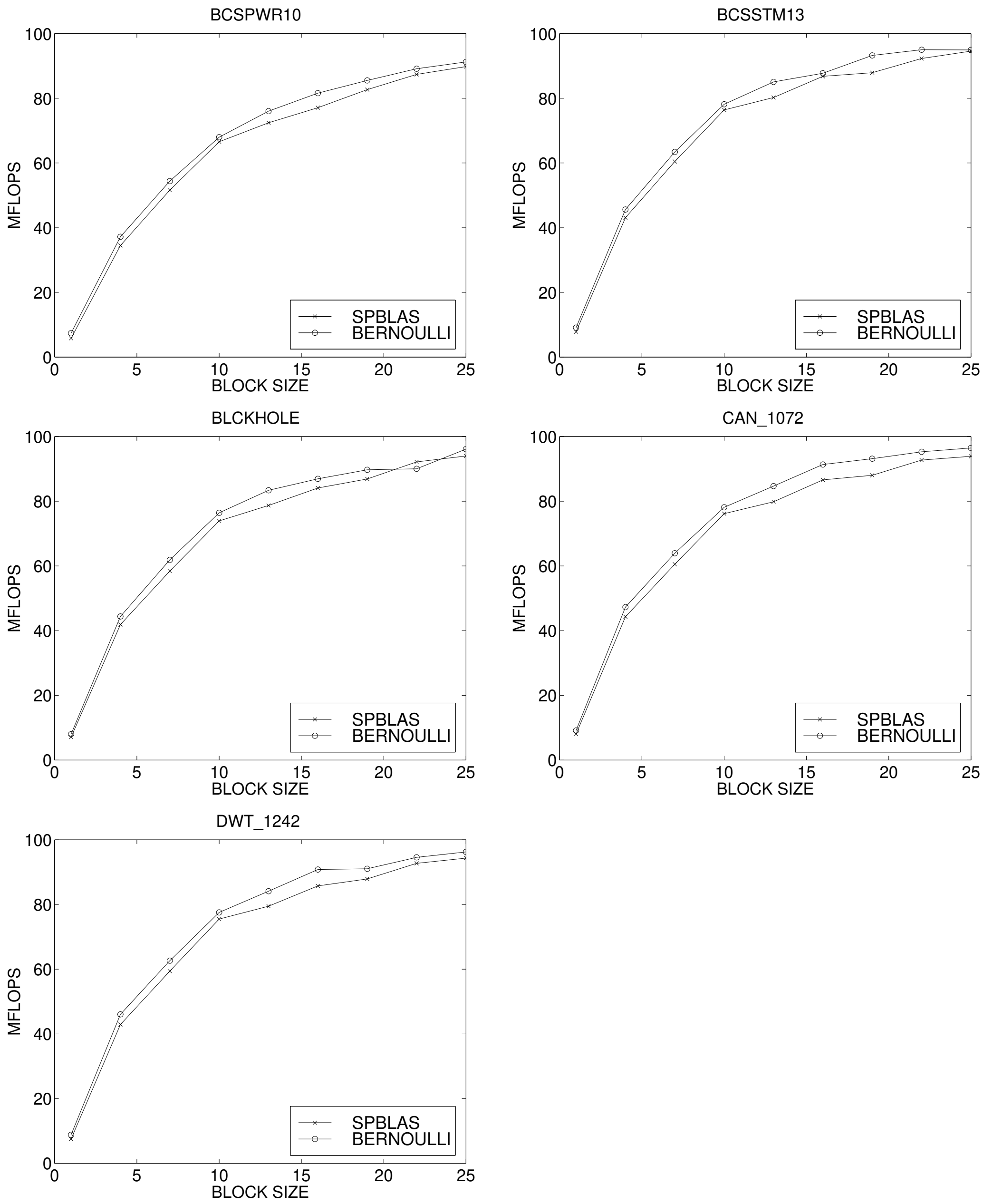


Figure 6.7: Performance of lower-triangular solution (VBR format)

```
for k=1 to n   sequential
    H(k,k) := √H(k,k)
    sparse vector scaling:
    H(:,k) := H(:,k) / H(k,k)
    incomplete outer product update:
    H(k+1:n,k+1:n) ≈ H(k+1:n,k+1:n) + H(:,k) * H(:,k)'
```

(Of course, all updates are applied only to the lower triangle of H.)

Let's focus on the incomplete outer product update. We call it "incomplete" because only the elements already stored are updated. This is a conjunctive DOANY computation with no fill and its sparse implementation can be generated by the BERNOULLI compiler. Dense outer product update is part of the dense BLAS interface [32]. However, it is hard to provide an incomplete sparse update in a general form, because it has 3 sparse operands. To demonstrate this advantage of our compiler we have implemented a PCG solver and compared its performance to the DSRIS routine provided in the IBM ESSL library (version 2, release 2).

For the compiler generated code the matrix was split into it's strict lower triangular part $\mathbf{A}_L$, stored using the CSC format, and the dense diagonal $\mathbf{D}$. Since the matrix is symmetric, it equals to the sum:

$$\mathbf{A} = \mathbf{A}_L + \mathbf{D} + \mathbf{A}_L^T \tag{6.19}$$

The matrix-vector product in the PCG algorithm is specified as:

Program 6.21

```
integer n, nnz
real D[n], Y[n], X[n]
abstract A_L(CSC uplo=LS ...)
doany i=1,n
    Y[i] := D[i] * X[i]
enddo
doany i=1,n; j=1,n
    Y[i] := Y[i] + A_L[i,j]*X[j]
enddo
doany i=1,n; j=1,n
    Y[i] := Y[i] + A_L[j,i]*X[j]
enddo
```

In order to compute the ICC factor, we have created the copies of the diagonal and of the arrays that store $\mathbf{A}_L$. Let $\mathbf{F}$ be the diagonal for the factor and $\mathbf{H}$ be a CSC matrix that stores the copy of the lower triangle. The computation of the preconditioned is specified to the compiler as:

Program 6.22

```
real F[n]
abstract H(CSC uplo=LS ...)
do k=1,n   sequential !!
    if F[k] ≤ 0 then error!!!
    F[k] := √F[k]
    doany i=k+1,n
        H[i,k] := H[i,k] / F[k]
```

```
    enddo
    doany j=k+1,n   update the diagonals
        F[j] := F[j] - H[j,k]*H[j,k]
    enddo
    doany j=k+1,n; i=j,n   update off-diagonals
        if H[i,j] ≠ 0 then
            H[i,j] := H[i,j] - H[i,k] * H[j,k]
        endif
    enddo
enddo
```

The pseudo-code for the compiler-generated implementation of the last update loop is:

Program 6.23

```
scatter H[*,k] into a dense vector U
do ⟨j,w⟩ ∈ H[*,k]
    search for the j-th column of H (fast lookup in CSC format)
    do ⟨i,v⟩ ∈ H[*,j]
        v := v - w * U[i]
```

Basically, the k-th column is scattered into a dense vector $\mathbf{U}$. Then all columns j that are updated by k are updated by appropriate multiple of the vector.

The PCG algorithm involves the solution of lower and upper triangular systems involving the incomplete factor $\mathbf{H}$ (and its diagonal $\mathbf{F}$). The solution of the lower-triangular system $\mathbf{X} = (\mathbf{H}+\mathbf{F})^{-1}\mathbf{B}$ is specified as:

Program 6.24

```
X := B
do j=1,n
    doany i=j+1,n
        X[j] := X[j] - H[i,j]*X[i]
    enddo
    X[j] := X[j] / F[j]
enddo
```

The solution of the upper triangular system $\mathbf{X} = (\mathbf{H}+\mathbf{F})^{-T}\mathbf{B}$ is specified as:

Program 6.25

```
X := B
do j=n,1,-1
    X[j] := X[j] / F[j]
    doany i=1,j-1
        X[j] := X[j] - H[i,j]*X[i]
    enddo
enddo
```

Table 6.4: Statistics of the matrices

Name	N	NNZ
BCSSTK32	44609	1029655
BCSSTK30	28924	1036208
RAND2K	2000	81330
RAND4K	4000	207806
RAND8K	8000	417052
RAND16K	16000	835214

Table 6.5: Performance of the PCG solver

Matrix	Preprocessing time (sec.)			Single iteration time (sec.)		
Name	BERNOULLI	ESSL	RATIO	BERNOULLI	ESSL	RATIO
BCSSTK32	0.76	2.70	28%	0.177	0.148	119%
BCSSTK30	1.15	4.57	25%	0.168	0.140	120%
RAND2K	0.03	0.14	21%	0.007	0.006	115%
RAND4K	0.13	0.51	25%	0.017	0.014	119%
RAND8K	0.37	1.16	32%	0.035	0.032	111%
RAND16K	1.17	2.50	47%	0.081	0.139	58%

Performance

We have measured the performance of the compiler-generated code and the ESSL library code on two matrices from Harwell-Boeing collection: BCSSTK30 and BCSSTK32 – and on four random SPD matrices generated by the `sprandsym` Matlab function [59]. `sprandsym` produces symmetric positive definite matrices as shifted sums of outer product updates. It was necessary to add an additional shift to the diagonals of the matrices in order to guarantee the existence of the ICC factor. In the case of the BCSSTK30 and BCSSTK32 matrices we have used only the sparsity pattern and filled in the values. The diagonal was made "heavy enough" to guarantee the existence of the ICC factor. The statistics of the matrices are shown in Table 6.4. N is the number of rows/columns. NNZ is the number of non-zeros in the lower-triangle of the matrix (including the diagonal).

We have measured two aspects of the performance of the PCG solver: the time taken by the computation of the preconditioner and the time taken by a single Conjugate Gradient iteration. The performance data is shown in Table 6.5. Unfortunately, we do not have access at this time to the source code of the DSRIS routine in the ESSL library and can only measure some kind of "preprocessing" time, which includes the computation of the preconditioner, but might include other things, such as permutation or re-formatting of the matrix to improve performance of matrix-vector products [84]. Because of this uncertainty we use the term "preprocessing" rather than "preconditioning".

The rather startling fact is that the ICC computation in the compiler-generated code is about 2-5 times faster than the preprocessing stage in the ESSL code. However, this data should be taken with a grain of salt. Without source code, we have no way of telling what exactly happens in the library code. We suspect that the library implements a more general Incomplete LU preconditioner, which is much more expensive. If this is true, then the comparison is not entirely fair to the library code. On the other hand, this is yet another argument for using our compiler: it is fairly easy with it to generate specialized code, which might not be included in a library due to lack of development time or other such reasons.

The CG iterations are 10-20% faster in the library code. There has been some very interesting work out of IBM recently on optimizing the performance of sparse matrix-vector products [84] by reordering and reformatting the matrix. It is reasonable to expect that matrix-vector product and triangular solves are hand-tuned for the RS6000 architecture.

6.3 BlockSolve library

The BlockSolve library from Argonne National Laboratories is designed to solve large sparse linear systems that arise in the solution of PDE's with multiple degrees of freedom. Figure 6.8 (adapted from [46]) 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 be rich in cliques and so-called *identical nodes* (or *i-nodes*). Assume that the stiffness matrix $\mathbf{A}$ is symmetric positive definite. Let n be the number of rows of $\mathbf{A}$. Let $G_A = (V, E)$ be the undirected graph that reflects the non-zero structure of the matrix:

$$V = \{1, 2, \ldots, n\} \tag{6.20}$$
$$E = \{(u, v) | \mathbf{A}_{uv} \neq 0\} \tag{6.21}$$

For each node $u \in V$ define the *adjacency set*:

$$S_u = \{v | (u, v) \in E\} \tag{6.22}$$

Equivalently, S_u is the set of the column indices of non-zeros in row u. An i-node is a set D of nodes u which have the same adjacency set. Since an i-node is an equivalence class, the rows of the matrix can be partitioned into maximal i-nodes. A simple greedy algorithm can used to discover maximal i-nodes.

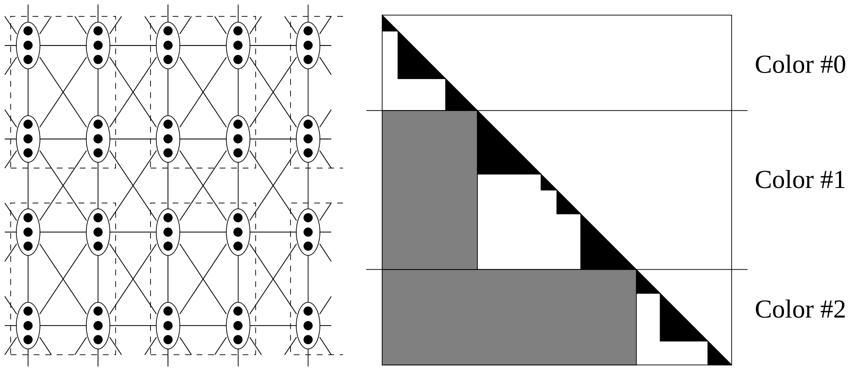


Figure 6.8: A multi-component grid and the color/clique/inode reordering of a sparse matrix

Cliques and i-nodes induce dense sub-matrices of $\mathbf{A}$ as follows. Let P be the set $\{i_1, i_2, \ldots, i_p\}$ of integers between 1 and n. Let Q be the set $\{j_1, j_2, \ldots, j_q\}$ of integers between 1 and n. Let $\mathbf{A}(P, Q)$ be the p-by-q matrix defined by:

$$\mathbf{A}(P, Q)_{st} = \mathbf{A}_{i_s, j_t}, \quad 1 \leq s \leq p, 1 \leq t \leq q \tag{6.23}$$

Let C be a set of nodes that form a clique. Then the matrix $\mathbf{A}(C, C)$ is dense. Let D be a set of nodes that form an i-node. Let S be the adjacency set of the i-node. Then the matrix $\mathbf{A}(D, S)$ is also dense.

BlockSolve library reorders the matrix by forming the quotient graphs based on i-nodes and then on cliques:

- Maximal i-nodes are computed. The nodes of the graph G_A are collapsed into individual i-nodes to obtain the quotient graph H_A.

- The graph H_A is heuristically partitioned into maximal cliques. The corresponding quotient graph K_A is formed.
- The nodes of K_A are (heuristically) colored. (Coloring is used in the parallel implementation.)
- Nodes of K_A are numbered consequitively within each color.
- Nodes of H_A are numbered consequitively within each clique and the ordering of the numbers between the cliques respects the order of the cliques in the numbering of K_A.
- Similarly, the nodes in G_A are numbered consequitively within each i-node and the numbering across i-nodes respects the order of i-nodes in H_A.

The structure of the resulting matrix is illustrated in Figure 6.8. 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 grouped into i-nodes.

I-nodes and cliques are stored separately. We can think of the matrix as being represented by a sum:

$$\mathbf{A} = \mathbf{A}_{inode} + \mathbf{A}_{clique} \tag{6.24}$$

where $\mathbf{A}_{clique}$ is the matrix of the blocks along the diagonal and $\mathbf{A}_{inode}$ is the matrix of the off-diagonal sparse blocks. $\mathbf{A}_{clique}$ is stored a a list of dense matrices and $\mathbf{A}_{inode}$ is stored as shown in Figure 1.15. The matrix-vector product $\mathbf{Y} = \mathbf{A} \cdot \mathbf{X}$ can be split into the sequence: $\mathbf{Y} := \mathbf{A}_{clique} \cdot \mathbf{X}$ followed by $\mathbf{Y} := \mathbf{Y} + \mathbf{A}_{inode} \cdot \mathbf{X}$. Both products can benefit from a fast implementation of dense matrix-vector product.

The product with the cliques $\mathbf{Y} := \mathbf{A}_{clique} \cdot \mathbf{X}$ is just a sequence of dense matrix-vector products. One would expect each clique (and the induced dense matrix) to have on the order of few dozen nodes (rows). Therefore it is beneficial to call dense BLAS library DGEMV routine for each clique.

The product with each i-node requires gathering of the values of $\mathbf{X}$ to be multiplied by the i-node into a small dense vector. Let D be the set of rows in the i-node and S be its adjacency set. Observe that due to the matrix reordering D is just an interval. However, S is a general set. Let $\mathbf{B} = \mathbf{A}_{D,S}$. As we have mentioned, $\mathbf{B}$ is a dense matrix. The product $\mathbf{B}\mathbf{X}_S$ affects the interval of $\mathbf{Y}$ given by the set D. So in order to multiply the vector $\mathbf{X}$ by the i-nodes in $\mathbf{A}_{inode}$, we need, for each i-node, to *gather* the elements of $\mathbf{X}$ into $\mathbf{X}_S$ and then compute the dense product $\mathbf{BZ}$. BlockSolve library dispatches at run-time between calling a BLAS routine (DGEMV for i-nodes and DTRMV for cliques) and calling a macro. We have implemented a similar optimization in our compiler as described in Section 5.8.1.

To understand the impact of this optimization of the performance we have run three versions of sparse matrix-vector product with the matrix stored using cliques and i-nodes. The first version was generated by the compiler with the BLAS optimization disabled ("No BLAS" column in Table 6.6). The second was generated with the BLAS optimization enabled ("BLAS" column in Table 6.6). And the third version is the original library code ("BlockSolve" column in Table 6.7). The measurements were taken on a Dell XPS D300 workstation with 300MHz Pentium II processor and 128MBytes of RAM, 512K L2 cache and 32K L1 cache. The workstation was running under Red Hat Linux 2.0.34. Compiler-generated C code and the BlockSolve library code were compiled by egcs compiler version 1.0.3.

The codes were run on synthetic sparse matrices which were obtained by applying a first order 7-point stencil to a regular 3-dimensional grid with with n points in each dimension and c unknowns at each grid point. The i-nodes in the resulting matrices have size c and most cliques have 2 i-nodes ($2c$ rows). We can thus adjust the sizes of dense sub-matrices in order to measure the impact of BLAS optimizations under different conditions.

We have broken the numbers into two tables. Table 6.6 reports the impact of using BLAS pattern-matching on the performance of compiler-generated code. Table 6.7 compares the performance of compiler-generated code (with BLAS pattern marching) and the original library code. Clearly, BLAS pattern matching is critical to achieving high-performance on multi-component problems. The improvement ranges from about 10% for single-component problems (with cliques of size 2) to a factor of 2.5 for problems with multiple components. The performance of compiler-generated code with BLAS pattern-matching is within 2-4% of the library code with a few exceptions that we are currently investigating.

Table 6.6: Impact of BLAS pattern-matching

Grids		Performance (Mflops)		
n	c	No BLAS	BLAS	Improvement[(a)] (%)
10	1	10.0	11.1	11.0
10	3	14.0	36.6	161.4
10	5	12.8	31.5	146.1
10	7	13.1	33.5	155.7
17	1	8.1	9.9	22.2
17	3	10.9	20.8	90.8
17	5	12.2	27.4	124.6
17	7	12.9	31.2	141.9
25	1	6.2	6.7	8.1
25	3	10.6	19.9	87.7
25	5	11.9	26.6	123.5

[(a)]Improvement = 100% * (BLAS - NoBLAS)/NoBLAS

Table 6.7: Comparison with the BlockSolve library code

Grids		Performance (Mflops)		
n	c	BlockSolve	Bernoulli with BLAS	Difference[(b)] (%)
10	1	14.4	11.1	-22.9
10	3	32.5	36.6	12.6
10	5	31.5	31.5	0.0
10	7	33.0	33.5	1.5
17	1	10.1	9.9	-2.0
17	3	20.6	20.8	1.0
17	5	27.3	27.4	0.4
17	7	30.1	31.2	3.7
25	1	6.9	6.7	-2.9
25	3	20.0	19.9	-0.5
25	5	26.3	26.6	1.1

[(b)]Difference = 100% * (Bernoulli - BlockSolve)/BlockSolve

Part III

Parallel code generation

Preview

Parallel program development adds a new dimension to the difficulties encountered in writing sequential programs: not only can the matrices be stored in a variety of formats, but there are many way of storing the information about how the matrices are partitioned. The best storage format for the *partitioning information* is application dependent, just as is the best storage format for a matrix. Previous work on language support and compilation techniques for parallel sparse matrix codes lacks such an abstraction. In particular, language features and compilation techniques described in the literature are tied to specific data formats and are inadequate for specifying and compiling high-performance algorithms [67, 86, 88, 92]. 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 formats for storing partition information.
- 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 address these concerns we extend our relational approach to the generation of parallel sparse code starting from dense code, a specification of sparse matrix formats and partition 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 partition maps themselves as relations.

We start by discussing in Chapter 7 how storage for distributed matrices is specified in our framework. Chapter 8 describes how parallel code is generated. Experimental evaluation of our techniques is presented in Chapter 9.

Chapter 7

Representing distributed tensors

7.1 Alignment and distribution in HPF

High Performance Fortran (HPF) [38, 50] allows a user to define a mapping between array elements and the physical processors. We call this mapping a *placement function*. Placement function is usually specified as a composition of *alignment* and *distribution*. The alignment function maps data and computations to a set of virtual processors organized as a Cartesian grid of some dimension (also called a *template*). The distribution function folds the virtual processors into the physical processors. Consider the following program fragment written in pidgin HPF:

Program 7.1

```
 1.  template T(n)
 2.  real A[n,n], X[n]
 3.  align A[i,*] with T[i]
 4.  align X[i] with T[i]
 5.  distribute T[BLOCK]
 6.  ...
 7.  do i=1,n on home T[i]
 8.      do j=1,n
 9.          X[i] = X[i] + A[i,j]
10.      enddo
11.  enddo
```

Line 1 declares the template (i.e. the grid of virtual processors). Line 3 provides the alignment function D_A for the array `A`: the i-th row is mapped to the i-th virtual processor:

$$t = D_A(row, column) = row \tag{7.1}$$

Similarly the i-th element of the array `X` is mapped to the same virtual processor:

$$t = D_X(x) = x \tag{7.2}$$

Line 5 defines the distribution function ρ for `T`: the virtual processors are assigned to physical processors in equal sized blocks. Let P be the number of processors. Assuming that the physical processors are numbered from 0 to $P - 1$, the virtual processor t, $1 \leq t \leq n$, is mapped to:

$$p = \rho(t) = \lfloor (t-1)/P \rfloor \tag{7.3}$$

The placement functions μ_A and μ_X for `A` and `X`, respectively, are defined as the compositions of alignments (7.1), (7.2) and the distribution (7.3):

$$\mu_A(row, column) = \rho(D_A(row, column)) = \lfloor (row - 1)/P \rfloor \tag{7.4}$$

$$\mu_X(x) = \rho(D_X(x)) = \lfloor (x - 1)/P \rfloor \tag{7.5}$$

The "`on home`" directive on line 7 tells the compiler to execute the i-th iteration of the loop on the physical processor that owns the virtual processor `T(i)`. In fact, this defines *computation alignment*:

$$t = C(i, j) = i \tag{7.6}$$

By composing it with the distribution (7.3), we get the placement function for the loop nest:

$$\mu_{\texttt{loop}}(i, j) = \rho(C(i, j)) = \lfloor (i - 1)/P \rfloor \tag{7.7}$$

The advantage of separating alignment from distribution is that a compiler can easily determine when an array access refers to local data only. Suppose we have a d-dimensional loop nest with indices $\mathbf{i} = (i_1\ i_2\ \ldots i_d)^T$ and an array access `A(`$\phi(\mathbf{i})$`)`, where ϕ is an affine function. Assume that both the loop and the array are aligned to the same template. Let's define the following quantities:

$$\begin{array}{ll} \mathbf{i} \in \mathcal{B} & \text{loop bounds} \\ C(\mathbf{i}) & \text{computation alignment} \\ \mathbf{a} = \phi(\mathbf{i}) & \text{the vector of array indices} \\ D_A(\mathbf{a}) & \text{array alignment} \end{array} \tag{7.8}$$

The HPF standard allows only affine alignment functions. Then the virtual processor $\mathbf{t}_1$ that owns the array element accessed during the iteration $\mathbf{i}$ is given by:

$$\mathbf{t}_1 = D_A(\mathbf{a}) = D_A(\phi(\mathbf{i})) \tag{7.9}$$

And the virtual processor $\mathbf{t}_2$ that owns the iteration is given by:

$$\mathbf{t}_2 = C(\mathbf{i}) \tag{7.10}$$

The access to `A` is purely local if the following holds:

$$\forall \mathbf{i} \in \mathbb{Z}^d : \mathbf{i} \in \mathcal{B} \Rightarrow D_A(\phi(\mathbf{i})) = C(\mathbf{i}) \tag{7.11}$$

It is easier to test the stronger condition:

$$\forall \mathbf{i} \in \mathbb{Z}^d : D_A(\phi(\mathbf{i})) = C(\mathbf{i}) \tag{7.12}$$

Since D_A, ϕ and C are affine functions, testing this condition requires some simple linear algebra. See Appendix A.2 for details. In Program 7.1 the condition is satisfied for both `A(i,j)` and `X(i)` references:

$$D_A(\phi_A(i, j)) = D_A(i, j) = i \tag{7.13}$$

$$D_X(\phi_X(i, j)) = D_X(i) = i \tag{7.14}$$

$$C(i, j) = i \tag{7.15}$$

The HPF standard leaves it up to the compiler to allocate *local storage*. The set ${\mathcal{D}_A}^{(p)}$ of array elements mapped to the processor p is given by:

$${\mathcal{D}_A}^{(p)} = \{\mathbf{a} | \rho(D_A(\mathbf{a})) = p\} \tag{7.16}$$

In our example this set (for `A`) is given by:

$$\mathcal{D}_A{}^{(p)} = \Big\{ row, column \;\Big|\; 1 \le column \le n \wedge 1 + pB \le row \le \min(pB + B, n) \Big\} \tag{7.17}$$

where $B = \lfloor n/P \rfloor$ is the size of the blocks assigned to the processors (the "last" block might be smaller if P does not divide n). When the value of the block size B is known at compile-time, the constraints (7.17) describe a polyhedron. Ancourt et al [3] describe how to declare storage for a polyhedron. In our example, the local storage for `A` can be declared as:

```
real Alocal[B,n]
```

with the indices of the `Alocal` and `A` arrays being related by:

$$\begin{aligned} row &= row\_local + p * B \\ column &= column\_local \end{aligned} \tag{7.18}$$

This defines *distribution relation* δ between the *global* array indices $\langle row, column \rangle$, on one hand, and the processor number p and local array indices $\langle row\_local, column\_local \rangle$, on the other:

$$\delta(\langle row, column \rangle, p, \langle row\_local, column\_local \rangle) = \Big(row = row\_local + p * B \wedge column = column\_local \Big) \tag{7.19}$$

Observe that **the distribution relation is implicit** in HPF: the user defines the placement function via alignment and distribution and it is the duty of the compiler to declare local storage and compute the relation.

7.2 Making local storage explicit

The computation in the SPMD node programs is eventually performed on the local data. We have argued in Part I that the format of the matrices affects the performance of the sequential programs. This also applies to parallel programs and we would like to provide a user with the ability to specify how the data is stored locally.

For parallel execution, one also has to specify the distribution relation. It can be stored in in a variety of formats. Just as it is the case with sparse matrix formats, the distribution relation formats are application dependent. For example, regular block/cyclic distributions 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 [38] provides for two kinds of irregular distributions: *generalized block* and *indirect*. In generalized block distribution, each processor receives a single block of continuous rows. The sizes of the individual blocks can vary. 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 in a variety of ways. However, this can require communication to determine the ownership of non-local data.

The Chaos library [67] allows the user to specify partitioning information by providing the list of row indices assigned to each processor. The list of indices is transferred into *distributed translation table* which is equivalent to having the `MAP` array partitioned block-wise. 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.

However, the above examples do not cover the whole spectrum of distributions used in practice. In the BlockSolve library [46, 47] the matrix is partitioned by row with each processor receiving a small set of blocks of rows (blocks can have different sizes), as illustrated in Figure 7.1 (cf. Figure 6.8). This scheme

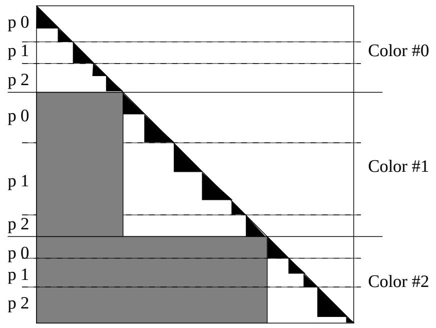


Figure 7.1: Partitioning in the BlockSolve library

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.

Given the above considerations, we would like to provide the user with the ability to specify the format of the distribution relation as well as of the local storage. In the uniprocessor case, we use the Black Box protocol to describe relations in terms of the underlying low-level objects. In the parallel case we define each relation as a *global view* of the partitions (or *fragments*) stored on each processor. The formats for the fragments are defined using the Black Box protocol. The problem we must address is that of describing distributed relations from the fragments.

Let's start with the following simple example (see Figure 7.2):

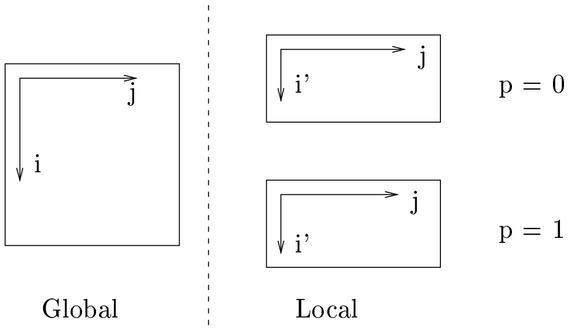


Figure 7.2: Distribution relation

- 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 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 $\mathbf{IND}^{(p)}\langle i, i' \rangle$. The local fragment of the matrix can also be viewed as a relation: $\mathbf{A}^{(p)}\langle i', j, v \rangle$. We can define the global matrix as follows:

$$\mathbf{A}\langle i,j,v\rangle = \bigcup_p \pi_{i,j,a}\left(\mathbf{IND}^{(p)}\langle i,i'\rangle \bowtie \mathbf{A}^{(p)}\langle i',j,v\rangle\right) \tag{7.20}$$

In "plain English" this formula describes the following recipe for computing the global view $\mathbf{A}$:

> On each processor p take the fragment $\mathbf{A}^{(p)}$, translate local row offsets i' into global row indices i via $\mathbf{IND}^{(p)}$, project the result onto global indices and the value, and then union the results across all processors.

We call $\mathbf{A}$ a *global view* of *local fragments* $\mathbf{A}^{(p)}$. In general, distribution relation is just another relation $\delta\langle\mathbf{a},p,\mathbf{a}'\rangle$ that stores the tuples of global indices $\mathbf{a}$, processor numbers p and local indices $\mathbf{a}'$. It relates the global view and the local fragments via the *fragmentation equation*:

$$\boxed{\mathbf{A}\langle\mathbf{a},v\rangle = \bigcup_p \pi_{\mathbf{a},v}\left(\delta\langle\mathbf{a},p,\mathbf{a}'\rangle \bowtie \mathbf{A}^{(p)}\langle\mathbf{a}',v\rangle\right)} \tag{7.21}$$

The distribution relation δ can be defined a variety of ways. In our example it is, in fact, distributed:

$$\delta\langle i,p,i'\rangle = \bigcup_p \{p\} \times \mathbf{IND}^{(p)}\langle i,i'\rangle \tag{7.22}$$

Here 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 [67]. Suppose that the global indices fall into the range $1 \le i \le N$ 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 local offset i' are stored on processor

$$q = \lfloor (i-1)/B \rfloor \tag{7.23}$$

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

$$h = (i-1) \bmod B \tag{7.24}$$

Then the relation is defined by:

$$\delta\langle i,p,i'\rangle = \bigcup_q \pi_{i,p,i'}\left(BLOCK\langle i,q,h\rangle \bowtie \mathbf{IND}^{(q)}\langle h,p,i'\rangle\right) \tag{7.25}$$

where $\mathbf{IND}^{(q)}\langle h,p,i'\rangle$ is the view of the above mentioned array of $\langle p,i'\rangle$ tuples and the relation $BLOCK\langle i,q,h\rangle$ is the shorthand for the constraints in (7.23) and (7.24).

The fragmentation equation (7.21) is general enough to specify all sorts of partitioning schemes. However, we would like to provide the distribution relation δ with more structure. Similar to the placement function in HPF, we decompose δ into alignment-like map that can be manipulated using simple linear algebra and distribution-like map, which keeps all of the non-linear information in the distribution relation.

We introduce the notion of the *global template space* and the *local template space*. Just as it is done in HPF, the indices $\mathbf{a}$ of the global views are aligned to the global template index $\mathbf{t}$: $\mathbf{t} = \mathrm{D}(\mathbf{a})$, where the *global alignment* D is an affine function. As we will discuss shortly, global alignment is used to help the compiler in discovering tensor accesses that do not require communication.

The indices $\mathbf{a}'$ of the local fragments are related to local template indices $\mathbf{t}'$ via *local alignment*: $\mathbf{t}' = D'(\mathbf{a}')$. The combination of global alignment and local alignment translates joins between global views into joins between local fragments. We discuss this point in Section 8.3 below.

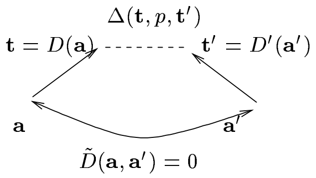


Figure 7.3: Relationship between global and local attributes and template spaces

---

Global and local template indices and the processor numbers p are related via *index translation* relation $\Delta\langle\mathbf{t}, p, \mathbf{t}'\rangle$. Just as a template in HPF can be shared by several arrays, the index translation relation can be shared by several global views.

The global and local alignment functions D and D' are not necessarily of full rank. Intuitively, not all global dimensions $\mathbf{a}$ have to be distributed. We need to specify how the "other" dimensions of $\mathbf{a}$ are related to the local indices $\mathbf{a}'$. For this we use the *co-alignment* relation $\tilde{D}(\mathbf{a}, \mathbf{a}') = 0$ where $\tilde{D}$ is an affine function. The relationship between various index spaces is illustrated in Figure 7.3.

For the correctness of the subsequent code generation algorithm, we need the following assumption:

Assumption 7.1 *Consider a global view $\mathbf{A}\langle\mathbf{a}, v\rangle$ and its local fragments $\mathbf{A}^{(p)}\langle\mathbf{a}', v\rangle$. Let $\mathcal{B}$ be the* dense *bounds of the indices $\mathbf{a}$. Let $\mathcal{B}^{(p)}$ be the dense bounds of each fragment. Then the distribution relation $\delta\langle\mathbf{a}, p, \mathbf{a}'\rangle$ must be a 1-1 and onto mapping between the global index $\mathbf{a}$ and the pair $\langle p, \mathbf{a}'\rangle$. Formally:*

- *for every global index $\mathbf{a} \in \mathcal{B}$ there exists a unique pair $\langle p, \mathbf{a}'\rangle$ such that $\mathbf{a}' \in \mathcal{B}^{(p)}$ and $\langle\mathbf{a}, p, \mathbf{a}'\rangle \in \delta$.*
- *for every local index $\mathbf{a}' \in \mathcal{B}^{(p)}$ on the processor p there exists a unique global index $\mathbf{a} \in \mathcal{B}$ such that $\langle\mathbf{a}, p, \mathbf{a}'\rangle \in \delta$.*

Observe that Assumption 7.1 implies that the index translation relation $\Delta\langle\mathbf{t}, p, \mathbf{t}'\rangle$ is also 1-1. Also Assumption 7.1 prohibits replication of data: each global index is mapped onto a unique processor.

We can now use the alignment constraints to refine the fragmentation equation (7.21):

$$\boxed{\mathbf{A}\langle\mathbf{a}, v\rangle = \bigcup_p \pi_{\mathbf{a},v} \mathop{\Theta}\limits_{\substack{\mathbf{t}=D_A(\mathbf{a})\wedge\mathbf{t}'=D'_A(\mathbf{t}')\wedge \\ \tilde{D}_A(\mathbf{a},\mathbf{a}')=0\wedge \\ p=q}} \begin{pmatrix} \mathcal{B}\langle\mathbf{a}\rangle, \{p\}, \\ \Delta\langle\mathbf{t}, q, \mathbf{t}'\rangle, \mathbf{A}^{(p)}\langle\mathbf{a}', v\rangle \end{pmatrix}} \tag{7.26}$$

where $\mathcal{B}$ are *dense* bounds on the global indices. The Θ-join translates the local indices through the alignment and global-to-local index translation Δ.

By mistake, the user may specify inconsistent relations δ when using the above scheme. 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 δ in fact provides a 1-1 and onto map. This problem is not unique to our framework – HPF-2 with irregular distributions [38] has a similar problem. Basically, if a function (or a "part" of it) is specified by its values at run-time, its properties can only be checked at run-time.

We need a general way of representing index translation relations Δ. One tempting option is to view the distribution relations as global views, just as distributed matrices. However, this is picture is too general and yet does not account for some special cases. We will return to this problem in Section 8.4 where, as we discuss communication generation, the requirements for the representation of distribution relations become clear. For now, we assume that Δ is an instance of an ADT which provides the following methods:

- Δ.`replicated`() returns on each processor an ADT expression for the whole relation. In general, this method is optional and we will discuss the alternatives in Section 8.4. Until then, we assume that the method is provided. For simplicity, we will use the term "Δ" to refer to the relation "Δ.`replicated`()", whenever this does not cause an ambiguity.

- Δ.`local`() returns on each processor p the query for the set of global template indices $\mathbf{t}$ and local template indices $\mathbf{t}'$ indices assigned to this processor p. In derivation below we will write this query as the relation $\Delta^{(p)}\langle \mathbf{t}, \mathbf{t}' \rangle$.

Other methods provided by ADTs for index translation relations are described in Section 8.4.

7.3 An Example

Consider the matrix-vector product:

Program 7.2

```
doany i=1,n; j=1,n
    Y[i] := Y[i] + A[i,j] * X[j]
```

Assume that the vectors are dense and the matrix is sparse. In many physical simulations the it is appropriate to partition the matrix **A** by row and to partition the elements of **X** and **Y** in the same way as the rows of **A**. According to our model we the the global views with the schema:

- $\mathbf{A}\langle r, c, a \rangle$
- $\mathbf{Y}\langle i_y, y \rangle$
- $\mathbf{X}\langle j_x, x \rangle$

And on each processor we have the local fragments:

- $\mathbf{A}^{(p)}\langle r', c', a \rangle$
- $\mathbf{Y}^{(p)}\langle i'_y, y \rangle$
- $\mathbf{X}^{(p)}\langle j'_x, x \rangle$

where the matrices $\mathbf{A}^{(p)}$ are stored using some sparse matrix format, such as CRS, and the vectors $\mathbf{Y}^{(p)}$ and $\mathbf{X}^{(p)}$ are dense (with respect to the local indices). We use the index translation relation $\Delta(t, p, t')$ to

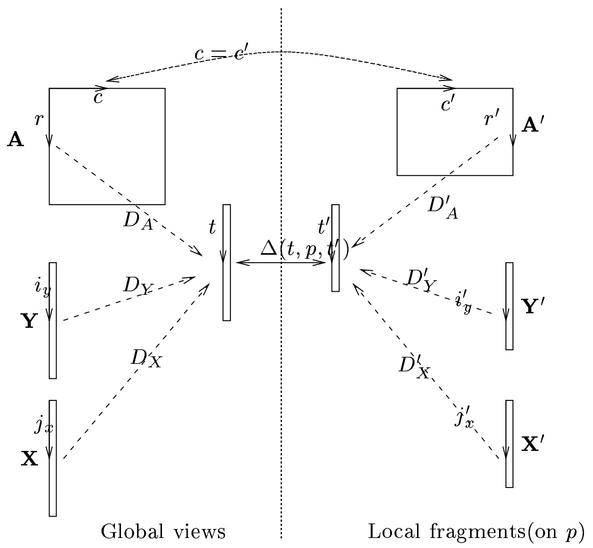


Figure 7.4: Alignments in sparse matrix-vector product

represent the global-to-local translation for the rows of $\mathbf{A}$ and the elements of the vectors. First, let's define the alignment constraints for $\mathbf{A}$:

$$\begin{array}{ll} t = D_A(r,c) = r & \text{global alignment} \\ t' = D'_A(r',c') = r' & \text{local alignment} \\ \tilde{D}_A(r,c,r',c') = c - c' & \text{co-alignment, i.e. } c = c' \end{array} \tag{7.27}$$

The co-alignment constraint $\tilde{D}_A(r,c,r',c') = 0$, or equivalently $c = c'$, expresses the unremarkable fact that the column indices are not partitioned and therefore, local column indices equal global column indices. The alignment constraints for $\mathbf{Y}$ are:

$$\begin{array}{ll} t = D_Y(i_y) = i_y & \text{global alignment} \\ t' = D'_Y(i'_y) = i'_y & \text{local alignment} \\ \tilde{D}_Y(i_y, i'_y) = 0 & \text{i.e. nothing to co-align} \end{array} \tag{7.28}$$

And the constraints for $\mathbf{X}$ are:

$$\begin{array}{ll} t = D_X(j_x) = j_x & \text{global alignment} \\ t' = D'_X(j'_x) = j'_x & \text{local alignment} \\ \tilde{D}_X(j_x, j'_x) = 0 & \text{i.e. nothing to co-align} \end{array} \tag{7.29}$$

The alignments are illustrated in Figure 7.4. We can write global view $\mathbf{A}$ as the result of the following query:

$$\mathbf{A}\langle r,c,a\rangle = \bigcup_p \pi_{r,c,a} \ominus_{r=t \wedge r'=t' \wedge c=c' \wedge p=q} \left(\begin{array}{c} \{1 \leq r,c \leq n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{A}^{(p)}\langle r',c',a\rangle \end{array} \right) \tag{7.30}$$

Observe that since we have no replication (follows from Assumption 7.1) the union is disjoint. $\mathbf{X}$ and $\mathbf{Y}$ are defined similarly:

$$\mathbf{Y}\langle i_y,y\rangle = \bigcup_p \pi_{i_y,y} \ominus_{i_y=t \wedge i'_y=t' \wedge p=q} \left(\begin{array}{c} \{1 \leq i_y \leq n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{Y}^{(p)}\langle i'_y,y\rangle \end{array} \right) \tag{7.31}$$

$$\mathbf{X}\langle j_x,x\rangle = \bigcup_p \pi_{j_x,x} \ominus_{j_x=t \wedge j'_x=t' \wedge p=q} \left(\begin{array}{c} \{1 \leq j_x \leq n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{X}^{(p)}\langle j'_x,x\rangle \end{array} \right) \tag{7.32}$$

Chapter 8

Code generation

8.1 Input program

8.1.1 Syntax for global views

For each global view the have to provide the following:

- the index translation relation (query)
- the alignment

To specify a distribution we need a syntax for Θ-joins:

`Theta(`P`, `$\{1 \leq x \leq N\}$`, `$\mathbf{A}\langle a_1, a_2\rangle$`, `$\mathbf{B}\langle b_1, b_2, b_3\rangle$`)`

simply describes the query:

$$\bigominus_P \left(\ \{1 \leq x \leq N\}, \mathbf{A}\langle a_1, a_2\rangle, \mathbf{B}\langle b_1, b_2, b_3\rangle \ \right) \tag{8.1}$$

The syntax for declaring an abstract variable to be an index translation relations is:

`abstract` Δ`(....)`

`distribution` Δ `global` $\mathbf{t}$ `local` $\mathbf{t}'$ `procs` p

Global views are declared by specifying the relation for the local fragments, the bounds of the global index, the index translation relation and the three components of alignment:

1. `global view` $\mathbf{A}\langle \mathbf{a}, v\rangle$ `of` $\mathbf{A}'\langle \mathbf{a}', v\rangle$
2.   `bounds <constraints on a>`
3.   `distribution` $\Delta\langle \mathbf{t}, p, \mathbf{t}'\rangle$
4.   `global-alignment` $\mathbf{t} = D(\mathbf{a})$
5.   `local-alignment` $\mathbf{t}' = D'(\mathbf{a}')$
6.   `co-alignment` $\tilde{D}(\mathbf{a}, \mathbf{a}')$
7. `end`

The functions D, D' and $\tilde{D}$ are specified using expressions in terms of the variables $\mathbf{a}$, $\mathbf{a}'$, $\mathbf{t}$ and $\mathbf{t}'$ that are scoped by the occurrences on lines 1 and 3 (see the example below).

To put it all together, here are the declarations for the matrix and the vectors in sparse matrix vector product:

Program 8.1

```
integer n, nlocal
-- concrete variables for the local fragment --
integer nnzlocal
...
integer rowp[nlocal], colind[nnzlocal], vals[nnzlocal]
-- definition of the local fragment --
abstract A'(CSR n=nlocal nnz=nnzlocal
        val=vals pntrb=colind[1] pntre=colind[2] indx=colind)
real X'[nlocal], Y'[nlocal]
abstract Δ(....)  details in Section 8.4
distribution Δ global ⟨t⟩ local ⟨t'⟩ procs $p$
global view A$\langle r, c, a\rangle$ of A'$\langle r', c', a\rangle$
    bounds $1 \le r, c \le n$
    distribution $\Delta\langle t, p, t'\rangle$
    global-alignment $t = r$
    local-alignment $t' = r'$
    co-alignment $c = c'$
end
global view Y$\langle i, y\rangle$ of Y'$\langle i', y\rangle$
    bounds $1 \le i \le n$
    distribution $\Delta\langle t, p, t'\rangle$
    global-alignment $t = i$
    local-alignment $t' = i'$
    co-alignment $0$
end
global view X$\langle i, x\rangle$ of Y'$\langle i', x\rangle$
    bounds $1 \le i \le n$
    distribution $\Delta\langle t, p, t'\rangle$
    global-alignment $t = i$
    local-alignment $t' = i'$
    co-alignment $0$
end
```

8.1.2 Computation decomposition

In addition to telling the compiler where the data is we have to specify which processor should execute which iterations. We use the following "`on home`" directive within a DOANY loop nest:

```
distribution Δ ...
...
doany i1=...;i2=...;...
    on home Δ(C(i))
```

```
    ...
enddo
```

where $\mathbf{i}$ is the vector of loop indices and C is an affine function. In fact, this directive specifies computation alignment. We do not need to specify local alignment and co-alignment for computation, since we do not have to provide "local storage" for it. Continuing Program 8.1, the full specification for sparse matrix-vector product is:

Program 8.2

```
integer n, nlocal
abstract A'(CSR ...)     (local fragment)
real X'[nlocal], Y'[nlocal]
abstract Δ(...)
distribution Δ global ⟨t⟩ local ⟨t'⟩ procs p
global view A⟨r,c,a⟩ of A'⟨r',c',a⟩
    ...
global view Y⟨i,y⟩ of Y'⟨i',y⟩
    ...
global view X⟨i,x⟩ of Y'⟨i',x⟩
    ...
doany i=1,n; j=1,n
    on home Δ(i)
    Y[i] := Y[i] + A[i,j] * X[j]
enddo
```

8.2 Preliminaries

In general, we start with a loop nest in affine Θ-join form (see Section 5.1):

Program 8.3

doany $\langle\ \mathbf{i}, \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m\ \rangle \in \bigodot_{\mathbf{a}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{a}^{(0)}} \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i} \rangle, \mathbf{A}_1\langle \mathbf{a}_1, v_1\rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m\rangle \end{array} \right)$

    S($\mathbf{s}$,$\mathbf{i}$,v$_1$,...,v$_m$)

where $\mathbf{i}$ is the vector of the loop indices, $\mathbf{s}$ is the vector of symbolic constants (invariants) and $\mathbf{a}$ is the vector of all tensor indices $\mathbf{a}_k$ stacked together. Additionally, we have a fragmentation equation for each relation $\mathbf{A}_k$, $1 \le k \le m$:

$$\mathbf{A}_k\langle \mathbf{a}_k, v_k\rangle = \bigcup_{p_k} \pi_{\mathbf{a}_k, v_k} \bigodot_{L_k(\mathbf{a}_k, \mathbf{a}'_k, \mathbf{t}_k, \mathbf{t}'_k) \wedge p_k = q_k} \left(\begin{array}{l} \mathcal{B}_k\langle \mathbf{a}_k\rangle, \{p_k\}, \\ \Delta_k\langle \mathbf{t}_k, q_k, \mathbf{t}'_k\rangle, \mathbf{A}^{(p_k)}\langle \mathbf{a}'_k, v_k\rangle \end{array} \right) \tag{8.2}$$

where the predicate L_k is the conjunction of all alignment constraints. The computation is aligned to the index translation relation $\Delta_0\langle \mathbf{t}_0, p_0, \mathbf{t}'_0\rangle$ by:

$$\mathbf{t}_0 = C(\mathbf{i}) \tag{8.3}$$

The query in Program 8.3 is a *distributed query* that involves distributed relations. In the database literature the optimization problem is: *find the sites that will evaluate parts of the query.* 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 [87] for a 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 the "`on home`" directive. This means that processor p has to execute the loop iterations given by:

$$\mathcal{B}^{(p)}\langle \mathbf{i} \rangle = \pi_{\mathbf{i}} \sigma_{p=p_0} \bigominus_{\mathbf{t}_0=C(\mathbf{i})} \left(\ \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0\langle \mathbf{t}_0, p_0, \mathbf{t}_0' \rangle \ \right) \tag{8.4}$$

Let $\Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}_0' \rangle$ be the set of global and local template indices that are assigned to processor p:

$$\Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}_0' \rangle = \pi_{\mathbf{t}_0,\mathbf{t}_0'} \sigma_{p=p_0} \Delta_0\langle \mathbf{t}_0, p_0, \mathbf{t}_0' \rangle \tag{8.5}$$

Then we can rewrite (8.4) as:

$$\mathcal{B}^{(p)}\langle \mathbf{i} \rangle = \pi_{\mathbf{i}} \bigominus_{\mathbf{t}_0=C(\mathbf{i})} \left(\ \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}_0' \rangle \ \right) \tag{8.6}$$

This gives us the local iteration set that satisfies the `on home` directive. On each processor we have to execute the following program:

Program 8.4

$$\texttt{doany} \ \left\langle \ \mathbf{i}, \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m \ \right\rangle \in \bigominus_{\mathbf{a}=\mathbf{Fi}+\mathbf{Gs}+\mathbf{a}^{(0)}} \left(\begin{array}{l} \mathcal{B}^{(p)}\langle \mathbf{i} \rangle, \mathbf{A}_1\langle \mathbf{a}_1, v_1 \rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m \rangle \end{array} \right)$$

$\quad\quad \texttt{S}(\mathbf{s}, \mathbf{i}, \mathrm{v}_1, \ldots, \mathrm{v}_m)$

Compared to Program 8.3, the loop enumerates over the local iteration set $\mathcal{B}^{(p)}$. We can expand the definition of the set to obtain:

Program 8.5

$$\texttt{doany} \ \left\langle \ \mathbf{i}, \mathbf{t}_0, \mathbf{t}_0', \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m \ \right\rangle \in \bigominus_{E \wedge \mathbf{t}_0=C(\mathbf{i})} \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}_0' \rangle, \\ \mathbf{A}_1\langle \mathbf{a}_1, v_1 \rangle, \ldots \\ \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m \rangle \end{array} \right)$$

$\quad\quad \texttt{S}(\mathbf{s}, \mathbf{i}, \mathrm{v}_1, \ldots, \mathrm{v}_m)$

where E denotes the data access equation $\mathbf{a} = \mathbf{Fi} + \mathbf{Gs} + \mathbf{a}^{(0)}$. Notice that we have omitted the projection "$\pi_{\mathbf{i}}$" (equation (8.4)) in the query in Program 8.5. In fact, the schema of the the query in Program 8.5 is different that that of the query in Program 8.4: we have added $\mathbf{t}_0$ and $\mathbf{t}_0'$ fields. However, it is easy to see that these programs perform the same computation. Because $\mathbf{t}_0$ (and $\mathbf{t}_0'$) is a function of $\mathbf{i}$, the *sequences* of iterations enumerated in Program 8.4 and Program 8.5 are the same, according to Theorem 5.7. Moreover, because the loop indices $\mathbf{a}_k$ are functions of the iterations $\mathbf{i}$, their sequence is correct, as well.

Program 8.5 describes the work to be done on each processor p, yet is can not be executed as is, because it contains references to global views. However, for each global view $\mathbf{A}_k$ we can compute its *localization* $\bar{\mathbf{A}}_k^{(p)}$:

$$\bar{\mathbf{A}}_k^{(p)}\langle \mathbf{a}_k, v_k \rangle = \pi_{\mathbf{a}_k, v_k} \bigominus_{\mathbf{a}_k=\phi_k(\mathbf{i})} \left(\ \mathcal{B}^{(p)}\langle \mathbf{i} \rangle, \mathbf{A}\langle \mathbf{a}_k, v_k \rangle \ \right) \tag{8.7}$$

which is the part of the global view that can potentially be accessed on processor p. Then Program 8.5 can be executed as:

Program 8.6

1. `allocate storage for` $\bar{\mathbf{A}}_k^{(p)}$, $1 \le k \le m$
2. `gather the values of those` $\bar{\mathbf{A}}_k^{(p)}$ `that appear on r.h.s of S`

3. doany $\left\langle\ \mathbf{i}, \mathbf{t}_0, \mathbf{t}_0', \mathbf{a}_1, v_1, \ldots, \mathbf{a}_m, v_m\ \right\rangle \in \bigominus_{E \wedge \mathbf{t}_0 = C(\mathbf{i})} \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}_0' \rangle, \\ \bar{\mathbf{A}}_1^{(p)}\langle \mathbf{a}_1, v_1 \rangle, \ldots \\ \ldots, \bar{\mathbf{A}}_m^{(p)}\langle \mathbf{a}_m, v_m \rangle \end{array} \right)$

4. $\quad$ `S(s,i,v1,...,vm)`

5. `enddo`

6. `scatter the values of the l.h.s relation`

The statement on line 1 computes the set of tensor indices $\mathbf{a}_k$ that can be accessed by processor p and allocates storage for them and for the values. The "gather" statement on line 2 communicates the values of those tensors that appear on the right-hand side of the assignment. The "scatter" statement communicates back the values of the tensor that is on the left-hand side of the assignment. Program 8.6 is rather inefficient:

- It can potentially communicate the values of the tensors that are not used in the computation.
- It can potentially communicate the tensors for which the accesses are purely local.

However, the communication statements are instances of more general localization problem: given a global view $\mathbf{A}\langle \mathbf{a}, v \rangle$ and a relation (or a query) $\mathbf{R}^{(p)}\langle \mathbf{i} \rangle$ on processor p we would like to compute the relation:

$$\boxed{\bar{\mathbf{A}}^{(p)}\langle \mathbf{a}, v \rangle = \bigominus_{\mathbf{a} = f(\mathbf{i})} \left(\ \mathbf{R}^{(p)}\langle \mathbf{i} \rangle, \mathbf{A}\langle \mathbf{a}, v \rangle\ \right)} \tag{8.8}$$

The first inefficiency is addressed by refining the relation $\mathbf{R}$. The second inefficiency is addressed by recognizing tensors with co-located references. We now address these points.

8.3 Exploiting co-location

8.3.1 Basic substitution

Consider the occurrence of the global view $\mathbf{A}_1\langle \mathbf{a}_1, v_1 \rangle$ in Program 8.5. The value of the index fields $\mathbf{a}_1$ is given by the appropriate portion of the data access equation:

$$\mathbf{a}_1 = \phi_1(\mathbf{i}, \mathbf{s}) \tag{8.9}$$

where ϕ_1 is an affine function of its arguments. Suppose that the global view is aligned to the same index translation relation $\Delta_0\langle \mathbf{t}_0, p, \mathbf{t}_0' \rangle$ as the loop iterations. Let $D_{A1}(\mathbf{a}_1)$ be the global alignment function for the view and $C(\mathbf{i})$ be the computation alignment. Because there is a functional dependence in the relation Δ_0 from the global template indices $\mathbf{t}_0$ to the processor number p, the following condition implies that the accesses to $\mathbf{A}_1$ are purely local:

$$\boxed{\forall \mathbf{s} \in \mathbb{Z}^b : \forall \mathbf{i} \in \mathbb{Z}^d : C(\mathbf{i}) = D_{A1}(\phi(\mathbf{s}, \mathbf{i}))} \tag{8.10}$$

where b is the dimension of the vector of symbolic constants. In "plain English" the condition says that both the loop iteration $\mathbf{i}$ and the tensor index $\mathbf{a}_1 = \phi_1(\mathbf{s}, \mathbf{i})$ are mapped to the same global template index and, because of the functional dependence, to the same processor. Verifying the condition (8.10) requires some simple linear algebra. For details, see Appendix A.2 or [8]. We will call condition (8.10) *co-location condition* and will say that the relation that satisfies the condition is *co-located*. Observe that this is the same condition as the one used in HPF compilers (statement (7.12) on page 180).

Without loss of generality, assume that the first k global views in Program 8.5 are co-located. This means that:

$$\forall \mathbf{s}, \mathbf{i} : \mathbf{t}_0 = C(\mathbf{i}) = D_{A1}(\phi_1(\mathbf{s}, \mathbf{i})) = D_{A2}(\phi_2(\mathbf{s}, \mathbf{i})) = \ldots = D_{Ak}(\phi_k(\mathbf{s}, \mathbf{i})) \tag{8.11}$$

Moreover, the localization $\bar{\mathbf{A}}_j^{(p)}$ on the processor p of each co-located global view $\mathbf{A}_j$, $1 \leq j \leq k$ can be described by the following query:

$$\bar{\mathbf{A}}_j^{(p)}\langle \mathbf{a}_j, v_j \rangle =$$

$$\pi_{\mathbf{a}_j, v_j} \bigominus_{\substack{\mathbf{t}_0 = D_{Aj}(\mathbf{a}_j) \wedge \mathbf{t}'_0 = D'_{Aj}(\mathbf{t}') \wedge \\ \tilde{D}_{Aj}(\mathbf{a}_j, \mathbf{a}'_j) = 0}} \Big(\mathcal{B}_j\langle \mathbf{a}_j \rangle, \Delta_0^{(p)}\langle \mathbf{t}, \mathbf{t}' \rangle, \mathbf{A}_j^{(p)}\langle \mathbf{a}'_j, v_j \rangle \Big) \tag{8.12}$$

where $\mathcal{B}_j$ is the polyhedron of the dense bounds of the global view index $\mathbf{a}_j$, as given by the view declaration in the program. In "plain English" the query (8.12) translates the indices of the local fragment $\mathbf{A}^{(p)}$ into the global indices. We let the predicate L_j denote the alignment constraints in (8.12):

$$L_j(\mathbf{a}_j, \mathbf{a}'_j, \mathbf{t}_0, \mathbf{t}'_0) \stackrel{\text{def}}{\equiv} \Big(\mathbf{t}_0 = D_{Aj}(\mathbf{a}_j) \wedge \mathbf{t}'_0 = D'_{Aj}(\mathbf{a}'_j) \wedge \tilde{D}_{Aj}(\mathbf{a}_j, \mathbf{a}'_j) = 0 \Big) \tag{8.13}$$

We can replace the occurrences of co-located views with appropriate queries in Program 8.5:

Program 8.7

$$\texttt{doany} \ \Big\langle \ \mathbf{i}, \mathbf{t}_0, \mathbf{t}'_0, \underline{\mathbf{a}_1, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}_k, \mathbf{a}'_k, v_k}, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m \ \Big\rangle \in \ldots$$

$$\ldots \bigominus_{\substack{E \wedge \mathbf{t}_0 = C(\mathbf{i}) \wedge \\ L_1 \wedge \ldots \wedge L_k}} \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}'_0 \rangle, \\ \mathcal{B}_1\langle \mathbf{a}_1 \rangle, \mathbf{A}^{(p)}\langle \mathbf{a}'_1, v_1 \rangle, \ldots, \mathcal{B}_k\langle \mathbf{a}_k \rangle, \mathbf{A}^{(p)}\langle \mathbf{a}'_k, v_k \rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{a}_{k+1}, v_{k+1} \rangle, \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m \rangle \end{array} \right)$$

$$\texttt{S}(\mathbf{s}, \mathbf{i}, \texttt{v}_1, \ldots, \texttt{v}_m)$$

Observe that (a) the schema of the query in Program 8.7 is different from the one in Program 8.5: we have added local indices $\mathbf{a}'_j$ of the co-located views (in the underlined part of the schema) – and (b) we have dropped the projections warranted by (8.12). This transformation is correct because there are functional dependencies form global indices to local indices and we can use Theorem 5.7 to remove the projections.

We can simplify the query in Program 8.7 by noticing that the global indices $\mathbf{a}_j$, $1 \leq j \leq k$, are computed from the loop indices $\mathbf{i}$ via tensor access functions:

$$\mathbf{a}_j = \phi_j(\mathbf{s}, \mathbf{i}), \quad 1 \leq j \leq k \tag{8.14}$$

Furthermore, we have assumed that the input program does not access tensors out of bounds, i.e.:

$$\forall \mathbf{i} \in \mathcal{B} : \phi_j(\mathbf{s}, \mathbf{i}) \in \mathcal{B}_j, \quad 1 \leq j \leq k \tag{8.15}$$

We can redefine the predicates L_j as:

$$L_j(\mathbf{s}, \mathbf{i}, \mathbf{a}'_j, \mathbf{t}_0, \mathbf{t}'_0) \stackrel{\text{def}}{\equiv} \Big(\mathbf{t}_0 = D_{Aj}(\phi_j(\mathbf{s}, \mathbf{i})) \wedge \mathbf{t}'_0 = D'_{Aj}(\mathbf{a}'_j) \wedge \tilde{D}_{Aj}(\phi_j(\mathbf{s}, \mathbf{i}), \mathbf{a}'_j) = 0 \Big) \tag{8.16}$$

and write Program 8.7 as:

Program 8.8

$$\texttt{doany}\ \left\langle\ \mathbf{i}, \mathbf{t}_0, \mathbf{t}_0', \underline{\mathbf{a}_1', v_1, \ldots, \mathbf{a}_k', v_k}, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m\ \right\rangle \in \ldots$$

$$\ldots \bigominus_{\substack{E' \wedge \mathbf{t}_0 = C(\mathbf{i}) \wedge \\ L_1 \wedge \ldots \wedge L_k}} \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i} \rangle, \Delta_0^{(p)} \langle \mathbf{t}_0, \mathbf{t}_0' \rangle, \mathbf{A}^{(p)} \langle \mathbf{a}_1', v_1 \rangle, \ldots, \mathbf{A}^{(p)} \langle \mathbf{a}_k', v_k \rangle, \\ \mathbf{A}_{k+1} \langle \mathbf{a}_{k+1}, v_{k+1} \rangle, \ldots, \mathbf{A}_m \langle \mathbf{a}_m, v_m \rangle \end{array} \right)$$

$$\texttt{S(s,}\mathbf{i}\texttt{,v}_1\texttt{,}\ldots\texttt{,v}_m\texttt{)}$$

Observe that this program does not involve global indices for the co-located relations (see the underlined part of the query). E' is the data access equation for the remaining global view indices. For further manipulation, it is convenient to transform the predicate of the Θ-join into explicit form. Overall, the linear constraints on the indices are:

$$\begin{array}{l} \mathbf{i} \in \mathcal{B} \\ \mathbf{a}_{j+1} = \phi_{j+1}(\mathbf{s}, \mathbf{i}) \wedge \ldots \wedge \mathbf{a}_m = \phi_m(\mathbf{s}, \mathbf{i}) \wedge \\ \mathbf{t}_0 = C(\mathbf{i}) \wedge \\ L_1(\mathbf{s}, \mathbf{i}, \mathbf{a}_1', \mathbf{t}_0, \mathbf{t}_0') \wedge \ldots \wedge L_k(\mathbf{s}, \mathbf{i}, \mathbf{a}_k', \mathbf{t}_0, \mathbf{t}_0') \end{array} \tag{8.17}$$

Using the results from Appendix A.2, we can transform the equations into explicit form:

$$\begin{array}{l} \mathbf{s} = f(\mathbf{s}') \\ \mathbf{i} = g(\mathbf{s}', \mathbf{i}') \wedge \mathbf{i}' \in \mathcal{B}' \\ \mathbf{a}_{j+1} = \phi_{j+1}'(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}_m = \phi_m'(\mathbf{s}', \mathbf{i}') \wedge \\ \mathbf{t}_0 = C_1(\mathbf{s}', \mathbf{i}') \\ \mathbf{t}_0' = C_1'(\mathbf{s}', \mathbf{i}') \\ \mathbf{a}_1' = h_1(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}_k' = h_k(\mathbf{s}', \mathbf{i}') \end{array} \tag{8.18}$$

where $\mathbf{s}'$ and $\mathbf{i}'$ are the free parameters in the solution of the linear system (8.17). $\mathbf{s}'$ the those parameters that restrict symbolic constants. The new loop indices $\mathbf{i}'$ might have larger dimensionality than the old ones, because the linear system (8.17) does not represent all the constraints that exist between local and global indices. All the functions in (8.18) are affine. The bounds on the new loop index $\mathcal{B}'$ are obtained by mapping the polyhedron of the old bounds "$\mathcal{B}' = g(\mathbf{s}', \mathcal{B})$". We can now rewrite the program in affine join form:

Program 8.9

```
Solve s = f(s') for s'
if no solution then exit
```

$$\texttt{doany}\ \left\langle\ \mathbf{i}', \mathbf{t}_0, \mathbf{t}_0', \mathbf{a}_1', v_1, \ldots, \mathbf{a}_k', v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m\ \right\rangle \in \ldots$$

$$\ldots \bigominus_{P} \left(\begin{array}{l} \mathcal{B}'\langle \mathbf{i}' \rangle, \Delta_0^{(p)} \langle \mathbf{t}_0, \mathbf{t}_0' \rangle, \mathbf{A}^{(p)} \langle \mathbf{a}_1', v_1 \rangle, \ldots, \mathbf{A}^{(p)} \langle \mathbf{a}_k', v_k \rangle, \\ \mathbf{A}_{k+1} \langle \mathbf{a}_{k+1}, v_{k+1} \rangle, \ldots, \mathbf{A}_m \langle \mathbf{a}_m, v_m \rangle \end{array} \right)$$

$$\texttt{S'(s,}\mathbf{i}\texttt{',v}_1\texttt{,}\ldots\texttt{,v}_m\texttt{)}$$

where the predicate P denotes the constraints:

$$P \stackrel{\text{def}}{\equiv} \left\{ \begin{array}{l} \mathbf{a}_{j+1} = \phi_{j+1}'(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}_m = \phi_m'(\mathbf{s}', \mathbf{i}') \wedge \\ \mathbf{t}_0 = C_1(\mathbf{s}', \mathbf{i}') \\ \mathbf{t}_0' = C_1'(\mathbf{s}', \mathbf{i}') \\ \mathbf{a}_1' = h_1(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}_k' = h_k(\mathbf{s}', \mathbf{i}') \end{array} \right. \tag{8.19}$$

The new loop body $\texttt{S}'$ is obtained by substituting $g(\mathbf{s}', \mathbf{i}')$ for $\mathbf{i}$.

8.3.2 An example

We now illustrate this on an example. Consider the setup for sparse matrix-vector product in Section 7.3. In particular, here are the definitions of the global views (recall Figure 7.3):

$$\mathbf{A}\langle r,c,a\rangle = \bigcup_p \pi_{r,c,a} \mathop{\ominus}_{r=t\wedge r'=t'\wedge c=c'\wedge p=q} \left(\begin{array}{l} \{1 \le r,c \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{A}^{(p)}\langle r',c',a\rangle \end{array} \right) \tag{8.20}$$

$$\mathbf{Y}\langle i_y,y\rangle = \bigcup_p \pi_{i_y,y} \mathop{\ominus}_{i_y=t\wedge i'_y=t'\wedge p=q} \left(\begin{array}{l} \{1 \le i_y \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{Y}^{(p)}\langle i'_y,y\rangle \end{array} \right) \tag{8.21}$$

$$\mathbf{X}\langle j_x,x\rangle = \bigcup_p \pi_{j_x,x} \mathop{\ominus}_{j_x=t\wedge j'_x=t'\wedge p=q} \left(\begin{array}{l} \{1 \le j_x \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{X}^{(p)}\langle j'_x,x\rangle \end{array} \right) \tag{8.22}$$

The program to be executed on each processor p is:

Program 8.10

`doany` $\langle i,j,t_0,t'_0,r,c,a,i_y,y,j_x,x\rangle \in \ldots$

$$\ldots \mathop{\ominus}_{\substack{i=r=i_y\wedge j=c=j_x \\ \wedge t_0=i}} \left(\begin{array}{l} \{1 \le i,j \le n\}, \Delta^{(p)}\langle t_0,t'_0\rangle, \\ \mathbf{A}\langle r,c,a\rangle, \mathbf{Y}\langle i_y,y\rangle, \mathbf{X}\langle j_x,x\rangle \end{array} \right)$$

```
    y := y + a * x
```

The references to $\mathbf{A}$ and $\mathbf{Y}$ are co-located:

$$\begin{array}{l} t_0 = i \\ t_A = D_A(r,c) = D_A(i,j) = i = t_0 \\ t_Y = D_Y(i_y) = D_Y(i) = i = t_0 \\ t_X = D_X(j_x) = D_X(j) = j \ne t_0 \end{array} \tag{8.23}$$

The equivalent program is:

Program 8.11

`doany` $\langle i,j,t_0,t'_0,r,c,r'c',a,i_y,i'_y,,y,j_x,x\rangle \in \ldots$

$$\ldots \mathop{\ominus}_{\substack{i=r=i_y\wedge j=c=j_x\wedge t_0=i=i_y=r\wedge \\ t'_0=r'=i'_y\wedge c'=c}} \left(\begin{array}{l} \{1 \le i,j \le n\}, \Delta^{(p)}\langle t_0,t'_0\rangle, \\ \{1 \le r,c \le n\}, \mathbf{A}^{(p)}\langle r',c',a\rangle, \\ \{1 \le i_y \le n\}, \mathbf{Y}^{(p)}\langle i'_y,y\rangle, \\ \mathbf{X}\langle j_x,x\rangle \end{array} \right)$$

```
    y := y + a * x
```

In order to bring this program into affine join form (a.la Program 8.9) we collect all linear equations on the indices:

$$i = r = i_y = t_0 \wedge j = c = j_x = c' \wedge t'_0 = r' = i'_y \tag{8.24}$$

The solution space has three dimensions:

$$\begin{array}{lcl} i = r = i_y = t_0 & = & u_1 \\ j = c = j_x = c' & = & u_2 \\ t'_0 = r' = i'_y & = & u_3 \end{array} \tag{8.25}$$

The bounds on the new indices $\mathbf{u} = (u_1\ u_2\ u_3)^T$ are:

$$\{1 \le u_1, u_2 \le n \wedge -\infty < u_3 < \infty\} \tag{8.26}$$

u_3 is unconstrained, which is OK since it will be set by one of the fields $t'_0 = r' = i'_y$. We can rewrite the program as:

Program 8.12
doany $\langle u_1, u_2, u_3, t_0, t'_0, r'c', a, i'_y, y, j_x, x\rangle \in \ldots$

$$\ldots \bigominus_{\substack{r=i_y=t_0=u_1\wedge\\ c=j_x=c'=u_2\wedge\\ t'_0=r'=i'_y=u_3}} \left(\begin{array}{l} \{1 \le u_1, u_2 \le n \wedge -\infty < u_3 < \infty\}, \\ \Delta^{(p)}\langle t_0, t'_0\rangle, \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, \mathbf{X}\langle j_x, x\rangle \end{array} \right)$$

```
    y := y + a * x
```

8.3.3 Simplification of the localized query

Since the body of the loop does not use the new loop indices $\mathbf{u}$ we can rewrite Program 8.12 as:

Program 8.13
doany $\langle a, x, y\rangle \in \ldots$

$$\ldots \pi_{a,x,y} \bigominus_{\substack{r=i_y=t_0=u_1\wedge\\ c=j_x=u_2\wedge\\ t'_0=r'=i'_y=u_3}} \left(\begin{array}{l} \{1 \le u_1, u_2 \le n \wedge -\infty < u_3 < \infty\}, \\ \Delta^{(p)}\langle t_0, t'_0\rangle, \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, \mathbf{X}\langle j_x, x\rangle \end{array} \right)$$

```
    y := y + a * x
```

Theorem 5.7 tells us that the programs are equivalent: there is a functional dependence from the loop indices to all the other fields, so we can project them out and, by using Theorem 5.7, drop the projection without changing the execution of the loop body.

Program 8.13 seems to enumerate over lots of "useless" indices: loop indices, template indices and global indices. Moreover, it enumerates over $\Delta^{(p)}$. The question is: can we simplify Program 8.13? The answer in this case is "yes". Consider the data access equation:

$$r = i_y = t_0 = u_1 \wedge c = j_x = u_2 \wedge t'_0 = r' = i'_y = u_3 \tag{8.27}$$

The above equations give us the join $r' = i'_y$ between the local fragments $\mathbf{A}^{(p)}\langle r', c', a\rangle$ and $\mathbf{Y}^{(p)}\langle i'_y, y\rangle$. Observe that once we have the tuple $\langle r', c', i'_y\rangle$ of the indices that satisfy the join, the global index $j_x = c = c'$ of $\mathbf{X}$ is determined. Also because the distribution relations for $\mathbf{A}$ and $\mathbf{Y}$ are assumed to be 1-1 and onto, this tuple fixes particular values $\langle r, c, i_y\rangle$ of the global indices. We do not know these values – they are obtained by translating the local indices through Δ – but we can be sure that they exist and are unique. Moreover, once we fix the global indices $\langle r, c, i_y\rangle$, they determine the loop indices $\mathbf{u}$, which are guaranteed to be within bounds:

$$\left.\begin{array}{l} \{1 \le r, c \le n\} \\ r = u_1 \wedge c = u_2 \end{array}\right\} \Rightarrow \{1 \le u_1, u_2 \le n\} \tag{8.28}$$

In summary, the values of the local indices $\langle r', c', i'_y\rangle$ uniquely determine everything else. Therefore we can rewrite the program as:

Program 8.14
doany $\langle w_1, w_2, r', c', a, j_x, x, i'_y, y\rangle \in \ldots$

$$\ldots \bigodot_{r'=i'_y=w_1 \wedge c'=j_x=w_2} \left(\begin{array}{l} \{-\infty < w_1, w_2 < \infty\}, \\ \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, \mathbf{X}\langle j_x, x\rangle \end{array} \right)$$

```
y := y + a * x
```

where the loop indices w_1 and w_2 are unconstrained – they are completely determined by the fields of the relations. This program should not come as a surprise: row-partitioned implementations of sparse matrix-vector product compute the local fragment of the result by multiplying the whole global vector $\mathbf{X}$ by the local fragment of $\mathbf{A}$:

$$\mathbf{Y}^{(p)} = \mathbf{A}^{(p)}\mathbf{X} \tag{8.29}$$

Program 8.14 enumerates exactly the matrix and vector elements used in these local products.

How can we perform this simplification in general? Two constraints can be simplified away:

- The loop bounds $\mathcal{B}$ can be simplified or, as in the example above, replaced by the whole integer space $\mathbb{Z}^d$. This simplification is similar to the one done in join scheduling (Section 5.5.3): bounds on the tensors indices together with access functions can imply "part of" loop bounds. To recall, let $\mathbf{f} = F(\mathbf{i})$ be the data access equation, where $\mathbf{f}$ is the vector of all tensor fields. Then we have two kinds of bounds on the loop indices $\mathbf{i}$:

$$\begin{array}{ll} \mathcal{B} & \text{the explicit loop bounds} \\ \mathcal{D} = \{\mathbf{i} | \exists \mathbf{f} : \mathbf{f} \in \mathcal{C} \wedge \mathbf{f} = F(\mathbf{i})\} & \text{implied bounds from tensors} \end{array} \tag{8.30}$$

  where $\mathcal{C}$ denotes the dense bounds on the tensor fields. We can use the gist operation [69] to simplify the loop bounds $\mathcal{B}$.

- Now suppose that loop bounds have been completely eliminated and the loop nest in Program 8.9 is of the form:

  Program 8.15

$$\texttt{doany} \left\langle \ \mathbf{i}', \mathbf{t}_0, \mathbf{t}'_0, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m \ \right\rangle \in \ldots$$

$$\ldots \bigodot_P \left(\begin{array}{l} \mathbb{Z}^{d'}\langle \mathbf{i}' \rangle, \Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}'_0\rangle, \mathbf{A}^{(p)}\langle \mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}^{(p)}\langle \mathbf{a}'_k, v_k\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{a}_{k+1}, v_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m\rangle \end{array} \right)$$

```
S'(s,i',v1,...,vm)
```

  We would like to remove the relation $\Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}'_0\rangle$ from the query, because then it would not have to be computed. The first condition is for the loop body in the input program not to use the loop indices. If it does, then we will have to translate local indices to global indices in order to obtain the values of the loop indices.

  The relation $\Delta_0^{(p)}\langle \mathbf{t}_0, \mathbf{t}'_0\rangle$ only translates indices, but it does not really constrain the local template indices $\mathbf{t}_0$. According to Assumption 7.1, $\Delta_0^{(p)}$ must store all the indices to which the local indices $\mathbf{a}'_j$, $1 \le j \le k$, can be mapped. Recall the data access equation for the query:

$$P \stackrel{\text{def}}{\equiv} \left\{ \begin{array}{l} \mathbf{a}_{j+1} = \phi'_{j+1}(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}_m = \phi'_m(\mathbf{s}', \mathbf{i}') \wedge \\ \mathbf{t}_0 = C_1(\mathbf{s}', \mathbf{i}') \\ \mathbf{t}'_0 = C'_1(\mathbf{s}', \mathbf{i}') \\ \mathbf{a}'_1 = h_1(\mathbf{s}', \mathbf{i}') \wedge \ldots \wedge \mathbf{a}'_k = h_k(\mathbf{s}', \mathbf{i}') \end{array} \right. \tag{8.31}$$

  Let's trace how the indices are translated. We start with the local indices $\mathbf{a}'_j$, $1 \le j \le k$. For a fixed value of symbolic constants $\mathbf{s}'$, the local indices determine some subspace of the loop indices $\mathbf{i}'$. Through local alignment, they also determine the unique value for the local template indices $\mathbf{t}'_0$ – this information is implicit in the data access equation. In turn, $\mathbf{t}'_0$ uniquely determine the global template indices $\mathbf{t}_0$ – though $\Delta_0^{(p)}$. Global template indices constrain the loop indices some more.

Now suppose that the values of the indices $\mathbf{a}'_j$ together with the global template indices $\mathbf{t}_0$ uniquely determine the values of the loop indices **i**'. This means that there is a functional dependence from the indices $\mathbf{a}'_j$ to all the other indices in the query! Moreover, suppose that the indices $\mathbf{a}_j$ uniquely determine all the remaining global view indices – $\mathbf{a}_{k+1}$, – etc in the query. Altogether this means that we can now rewrite the data access equation as:

$$P' \stackrel{\text{def}}{\equiv} \begin{cases} \mathbf{i}' = f(\mathbf{s}', \mathbf{a}', \mathbf{t}_0) \\ \mathbf{a} = g(\mathbf{s}', \mathbf{a}') \\ \mathbf{t}'_0 = h(\mathbf{s}, \mathbf{a}') \end{cases} \tag{8.32}$$

where $\mathbf{a}$ is the vector of all remaining global view indices and $\mathbf{a}'$ is the vector of the local indices $\mathbf{a}'_j$, $1 \le j \le k$. Since the body of the loop does not use the iteration vector $\mathbf{i}'$ we drop the first row of the equation:

$$P'' \stackrel{\text{def}}{\equiv} \begin{cases} \mathbf{a} = g(\mathbf{s}', \mathbf{a}') \\ \mathbf{t}'_0 = h(\mathbf{s}, \mathbf{a}') \end{cases} \tag{8.33}$$

Now, our claim is that the loop nest

Program 8.16

doany $\langle\ \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m\ \rangle \in \ldots$

$$\ldots \bigominus_{P''} \left(\begin{array}{l} \mathbf{A}^{(p)}\langle \mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}^{(p)}\langle \mathbf{a}'_k, v_k\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{a}_{k+1}, v_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m\rangle \end{array} \right)$$

S'(s, v_1, ..., v_m)

performs the same computation as the one in Program 8.15. Take an iteration of the loop in Program 8.16 with the values of the indices $\langle \mathbf{a}, \mathbf{a}'\rangle$. These values satisfy the data access equation P''. The values of local indices $\mathbf{a}'$ uniquely determine the local and, therefore, global template indices $\mathbf{t}_0$. Moreover, by the assumption that the original data access equation can be rewritten as in (8.32), the loop indices are also determined. Therefore there is a corresponding iteration of the loop nest in Program 8.15. The converse is trivially true.

We discuss in Appendix A.2 how to verify that the data access equation can be brought into the form (8.32).

To summarize the available simplification:

- Loop bounds can be "gisted" given the bounds of the global views
- Index translation $\Delta_0^{(p)}$ can be avoided if the data access equation can be brought into the form (8.32).

8.4 Communication generation

8.4.1 Preliminaries

Let's return to the sparse matrix-vector product example:

Program 8.17

doany $\langle r', c', a, j_x, x, i'_y, y\rangle \in \ldots$

$$\ldots \bigominus_{r'=i'_y \wedge c'=j_x} \left(\ \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, \mathbf{X}\langle j_x, x\rangle\ \right)$$

```
y := y + a * x
```

In order to produce the SPMD node program we have to generate code to localize $\mathbf{X}$. Each processor p accesses *at most* the following indices of $\mathbf{X}$:

$$USED_X^{(p)}\langle j_x\rangle = \pi_{j_x} \bigominus_{r'=i'_y \wedge c'=j_x} \left(\ \{1 \leq j_x \leq n\}, \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle \ \right) \tag{8.34}$$

In "plain English": we take the portion of the query that has already been localized and project it onto the indices of the tensor in question. The bounds $\{1 \leq j_x \leq n\}$ are taken from the declaration of the global view.

In the general case, we start with a loop nest of the form:

Program 8.18

$$\texttt{doany} \ \left\langle \ \mathbf{i}, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m \right\rangle \ \right\rangle \in \ldots$$

$$\ldots \bigominus_E \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i}\rangle, \mathbf{A}'_1\langle \mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle \mathbf{a}'_k, v_k\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{a}_{k+1}, v_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m\rangle \end{array} \right)$$

$$\texttt{S}(\mathbf{s}, \mathbf{i}, \mathtt{v}_1, \ldots, \mathtt{v}_m)$$

where E is the data access equation. (If the simplification of the previous Section have been performed, then the loop indices are equal to the local indices and are unbounded.) Now we generate the code to localize the relation $\mathbf{A}_{k+1}$. The set of global indices of the relation that are being accessed is:

$$USED_{k+1}^{(p)}\langle \mathbf{a}_{k+1}\rangle = \pi_{\mathbf{a}_{k+1}} \bigominus_{E'} \left(\ \mathcal{B}\langle \mathbf{i}\rangle, \mathcal{B}_{k+1}\langle \mathbf{a}_{k+1}\rangle, \mathbf{A}'_1\langle \mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle \mathbf{a}'_k, v_k\rangle \ \right) \tag{8.35}$$

where the predicate E' is obtained by projecting the data access equation E onto the indices $\mathbf{a}'_1$ through $\mathbf{a}'_k$ and onto $\mathbf{a}_{k+1}$. The bounds $\mathcal{B}_{k+1}$ are taken from the declaration of the global view. We have to generate code to perform two tasks:

- Compute set $USED_{k+1}^{(p)}$ and allocate the storage for the localized relation $\bar{\mathbf{A}}_{k+1}^{(p)}$. Here we compute the set of global indices $\mathbf{a}_{k+1}$ for which the storage is to be allocated, but we don communicate or store the actual values, yet. We call task *communication buffer allocation* (CBA).
- Communicate the values. The code depends on whether the tensor $\mathbf{A}_{k+1}$ appears on the left-hand side of the assignment in the loop body. We call this task *communication execution* (CE).

8.4.2 ADT protocol for index translation relations

Let $\mathbf{A}$ be the global view being localized (i.e. $\mathbf{A}_{k+1}$, previously). CBA involves the following steps:

- Computation of the *view set* which is the set of $\langle \mathbf{a}, q, \mathbf{a}'\rangle$ where $\mathbf{a}$ is the global index of $\mathbf{A}$, q is the processor that owns $\mathbf{a}$ and $\mathbf{a}'$ is the corresponding local index (on processor q). Let $L(\mathbf{a}, \mathbf{a}', \mathbf{t}, \mathbf{t}')$ be the alignment of $\mathbf{A}$ and Δ be the distribution relation of $\mathbf{A}$. Then the view set of $\mathbf{A}$ on processor p is defined by:

$$VIEW^{(p)}\langle \mathbf{a}, q, \mathbf{a}'\rangle =$$

$$\pi_{\mathbf{a},q,\mathbf{a}'} \bigominus_{\mathbf{a}=\mathbf{b} \wedge L(\mathbf{a},\mathbf{a}',\mathbf{t},\mathbf{t}')} \left(\ \mathcal{B}\langle \mathbf{a}, \mathbf{a}'\rangle, USED^{(p)}\langle \mathbf{b}\rangle, \Delta\langle \mathbf{t}, q, \mathbf{t}'\rangle \ \right) \tag{8.36}$$

  where we can safely use the infinite integer space (of the appropriate dimension) for the bounds $\mathcal{B}$.

- Computation of the *receive set* (assuming the access to the tensor being localized does not appear on the left-hand side of an assignment). Each processor p has to receive the data from the owner q of the non-local indices:

$$RECV^{(p)}\langle q, \mathbf{a}' \rangle = \bigcup_q \{q\} \times \left((\pi_{\mathbf{a}'} \sigma_{r=q} VIEW^{(p)}\langle \mathbf{a}, r, \mathbf{a}' \rangle) \bowtie \mathbf{A}^{(q)}\langle \mathbf{a}', v \rangle \right) \tag{8.37}$$

The join with $\mathbf{A}^{(q)}$ guarantees that only those elements of the fragment that are actually stored on q are eventually communicated. Observe that the relation $VIEW^{(p)}$ resides on processor p, while the fragment is stored on processor q. It follows that the computation of the $RECV$ set requires communication, in general. When the fragment is dense we can avoid the computation:

$$RECV^{(p)}_{\text{dense}}\langle q, \mathbf{a}' \rangle = \bigcup_q \{q\} \times \left(\pi_{\mathbf{a}'} \sigma_{r=q} VIEW^{(p)}\langle \mathbf{a}, r, \mathbf{a}' \rangle \right) \tag{8.38}$$

In this case the $RECV$ set is just the projection of the $VIEW$ set onto the owner indices q and local indices $\mathbf{a}'$.

- Computation of the *send set*. This set is the dual of the receive set:

$$SEND^{(p)}\langle q, \mathbf{a}' \rangle = \bigcup_q \{q\} \times \sigma_{r=q} RECV^{(q)}\langle r, \mathbf{a}' \rangle \tag{8.39}$$

Processors have to communicate in order to exchange the sets.

- Allocation of the buffers. For the send and receive sets we need to allocate the space for the values. The storage for the localized relation $\bar{\mathbf{A}}^{(p)}$ is allocated at this point.

The CE task involves copying the the values to and from the communication buffers (depending on whether $\mathbf{A}$ appears on the left-hand side) and sending and receiving the values.

We have to address the following question:

- How are the sets $USED$, $VIEW$, $SEND$ and $RECV$ to be stored?
- How is the $VIEW$ set to be computed?
- How are the $SEND$ and $RECV$ sets to be computed?
- How are the communication buffers to be stored?

While there is a default way of computing the sets and allocating the storage (Section 8.4.4), there exist application specific optimizations. For example, in dense codes with regular (BLOCK/CYCLIC) distributions the sets do not have to be stored at all because they can be described by a formula [3].

A more interesting example is the the sets are computed in the Aztec library [44, 85]. The library implements Krylov space iterative solvers and sparse matrix-vector product $\mathbf{Y} = \mathbf{AX}$ is the core operation. The matrix and the vectors are partitioned as described in Section 7.3. The index translation relation δ is represented by storing on each processor the array of global indices assigned to it (as in our example in Section 7.2). Each processor accesses the element $\mathbf{X}[j]$ for every non-zero entry of the local fragment of the matrix $\mathbf{A}^{(p)}[i', j]$:

$$USED^{(p)}_X\langle j \rangle = \pi_j \mathbf{A}^{(p)}\langle i', j \rangle \tag{8.40}$$

In order to compute the $VIEW$ we essentially have to compute the join:

$$VIEW^{(p)}_X\langle j, q, j' \rangle = USED^{(p)}_X\langle j \rangle \bowtie \delta\langle j, q, j' \rangle \tag{8.41}$$

The problem is that the processor p does not know which processor r owns the tuple $\langle j, q, j' \rangle$. The tuple is, of course, stored on processor q, which is not known to p. Aztec library has a clever solution. It is targeted towards physical simulations such as finite element computation in which one expects the processor

p to communicate with only a few "neighbors". The heuristic used in the library selects a small random sample of the $USED^{(p)}$ set and sends it to every processor. Then each processor q replies to p only if it owns an index in the sample. The whole operation can be formulated as global broadcast operation followed by a global reduction. For the indices which were not in the sample, the processor p now only contacts the processors which replied to the broadcast. It then sends the remaining indices in the $USED$ set only to this set of neighbors. If some indices are left untranslated, the processors organize a second round of sampling followed by translation, and so on. It is expected that the first round of communication reveals the precise set of neighbors.

Chaos library [67] implements a deterministic scheme for index translation. Initially every processor stores the array of indices assigned to it (as in the previous example). The indices are then rearranged to form a *distributed translation table*: each processor p sends the tuple ⟨j,j'⟩ to processor $q = j/B$ for some block size B. This effectively creates another index translation relation $\delta'\langle j, q, j'\rangle$ which is distributed BLOCK-wise based on the j field. As a result each processor p knows which processor $r = j/B$ to query for the ownership information.

An alternative to Chaos and Aztec library for each processor to broadcast its $USED$ set. A pair of global broadcast/reduction operations can be used to compute the $VIEW$ set on each processor. If the communication sets are small, but each processor can have more than a few neighbors, then this scheme might perform better.

In the BlockSolve library [46, 47] the index translation relation is replicated and the $VIEW$ set can be computed without communication. However, communication is still required in order to compute send/receive sets. The same is true for the generalized-block distribution provided by HPF-2 standard [38]: the index translation relation can be replicated in this case, as well.

Observe that the decision as to which algorithm to use in order to compute the sets can be rather non-trivial (for a compiler). The choice between the Aztec library, the Chaos library and the alternative suggested above clearly depends on the actual values of the inputs.

We take the following position:

- The algorithm for the computation of the communication sets can be seen as the property of the index translation relation. In order to allow a user to plug in her own algorithm for the computation of the sets, we have developed an ADT protocol for specifying index translations (below).

- There are still many cases when the compiler can be relied upon to produce an efficient implementation, most notably when the index translation relation is replicated. Some of the methods in the protocol below are optional and their default implementations are generated by the compiler.

Based on the above considerations, here are the methods that have to be supplied by the ADT for an index translation relation Δ:

- `decls()` returns the necessary declarations

- `init()` performs whatever initialization is necessary

- `replicated()` is an ADT expression for the replicated copy of the relation.

- `local()` is the query for the relation $\Delta^{(p)}\langle \mathbf{t}, \mathbf{t}'\rangle$ that stores the indices assigned locally.

- $\mathtt{allocate}(\mathtt{prefix}, \mathbf{A}^{(\mathtt{p})}, \mathbf{a}, \mathbf{a}', \mathtt{p}, \mathbf{t}, \mathbf{t}', \mathtt{L}(\mathbf{a}, \mathbf{a}', \mathbf{t}, \mathbf{t}'), \mathtt{Theta}(...))$ computes the communication sets and allocates the buffers. `Theta(...)` is the query that computes the $USED$ set (8.35) on page 198. The projection is implicit. $\mathbf{A}^{(p)}$ is the local fragment. L is the alignment. $\mathbf{a}$, $\mathbf{a}'$, etc. are the lists of names used to scope the variables in the alignment and the query. `prefix` is a unique prefix to be given to the send/receive buffers.

- `send_buf(prefix)` is the ADT expression for the send buffer seen as the relation with the schema $\langle q, \mathbf{a}', v\rangle$ where $\mathbf{a}'$ are local indices of the relation $\mathbf{A}$, v are the values and q is the destination processor. The same local index $\mathbf{a}'$ might be sent to several processors.. If we let $\mathbf{S}$ denote this relation, then the code to copy the values from the local fragment $\mathbf{A}^{(p)}$ into the send buffer is:

Program 8.19
doany $\langle \mathbf{a}', q, v, w\rangle \in \mathbf{A}^{(p)}\langle \mathbf{a}', v\rangle \bowtie \mathbf{S}\langle q, \mathbf{a}', w\rangle$
```
    w := v
```

- `recv_buf(prefix)` is the ADT expression for the receive buffer. It has the schema $\langle \mathbf{a}, v\rangle$ where $\mathbf{a}$ is the global view index. The buffer is, in fact, the localized relation $\bar{\mathbf{A}}^{(p)}$.
- `send(prefix)` and `recv(prefix)` send and receive the communication buffers.

Not all of these methods have to be specified at the same time. Here are three possibilities:

1. Only `replicated()` is provided. Then the rest are generated by the compiler.
2. `replicated()` and `local()`. The rest are generated by the compiler.
3. `local()`, `allocate()`, `send_buf()`, `recv_buf()`, `send()` and `recv()` are provided. These are enough to generate the communication code.

The initialization methods `decls()` and `init()` are optional.

8.4.3 Communication generation

After co-located references are resolved we are left with the code:

Program 8.20
doany $\left\langle\ \mathbf{i}, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m\right\rangle\ \rangle \in \ldots$

$$\ldots \bigominus_E \left(\begin{array}{l} \mathcal{B}\langle \mathbf{i}\rangle, \mathbf{A}'_1\langle \mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle \mathbf{a}'_k, v_k\rangle, \\ \mathbf{A}_{k+1}\langle \mathbf{a}_{k+1}, v_{k+1}\rangle, \ldots, \mathbf{A}_m\langle \mathbf{a}_m, v_m\rangle \end{array} \right)$$

$\mathtt{S}(\mathbf{s}, \mathbf{i}, \mathtt{v}_1, \ldots, \mathtt{v}_m)$

We assume the statement `S` is an assignment statement in one of two forms:

- `var := var op update` where `op` is either addition or multiplication and `var` can be either a scalar or a tensor reference. `update` is an expression that does not involve `var`. In this case the loop nest computes a reduction.
- `var := rhs` where `var` is a tensor reference of the form $\mathbf{A}[\phi(\mathbf{i}, \mathbf{s})]$ and the access function ϕ is 1-1 between the loop indices $\mathbf{i}$ and tensor indices (for any fixed value of symbolic constants $\mathbf{s}$). `rhs` does not contain any references to $\mathbf{A}$. In this case each element $\mathbf{A}[\phi(\mathbf{i}, \mathbf{s})]$ is independently assigned a value and the loop is DOALL.

These conditions are sufficient for a loop nest to be DOANY and they cover all the computational kernels that we are interested in. There are a few cases on how communication is placed, depending on the position of the reference to the tensor $\mathbf{A}_{k+1}$ in the assignment statement.
$\mathbf{A}_{k+1}$ appears in `rhs` or `update`. Let Δ be the index translation relation for the global view $\mathbf{A}_{k+1}$ and let L be its alignment. Then we generate the following code:

Program 8.21

-- *allocate buffers* --

Δ.allocate(prefix, $\mathbf{A}^{(p)}$,

$\qquad \mathbf{a}_{k+1}, \mathbf{a}'_{k+1}, q, \mathbf{t}_{k+1}, \mathbf{t}'_{k+1},\ L(\mathbf{a}_{k+1}, \mathbf{a}'_{k+1}, q, \mathbf{t}_{k+1}, \mathbf{t}'_{k+1}),$

$\qquad$ -- *query* (8.35) *on page 198 goes here* --)

copy the values into the send buffer

doany $\langle \mathbf{a}'_{k+1}, q, \mathbf{b}', v, w\rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{b}'=\mathbf{a}'_{k+1}} \left(\ \Delta.\mathtt{send\_buf}(\mathit{prefix})\langle q, \mathbf{b}', w\rangle, \mathbf{A}^{(p)}_{k+1}\langle \mathbf{a}'_{k+1}, v\rangle\ \right)$$

    w := v
enddo
-- *communicate* --
Δ.send(prefix,Δ.send_buf(prefix))
Δ.recv(prefix,Δ.recv_buf(prefix))
-- *compute* --
doany $\langle\ \mathbf{i}, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m \rangle\ \rangle \in \ldots$
$$\ldots \bigominus_E \begin{pmatrix} \mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}'_1\langle\mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle\mathbf{a}'_k, v_k\rangle, \\ \Delta.\mathtt{recv\_buf}\langle\mathbf{a}_{k+1}, v_{k+1}\rangle, \\ \mathbf{A}_{k+2}\langle\mathbf{a}_{k+1}, v_{k+2}\rangle, \ldots, \mathbf{A}_m\langle\mathbf{a}_m, v_m\rangle \end{pmatrix}$$
    S(s,$\mathbf{i}$,v_1,...,v_m)

<u>$\mathbf{A}_{k+1}$ is the var in "var := var op update"</u>. The code is:

Program 8.22

1. -- *allocate buffers* --
2. Δ.allocate(prefix,$\mathbf{A}^{(p)}$,
3. $\quad\quad$ $\mathbf{a}_{k+1}$,$\mathbf{a}'_{k+1}$,q,$\mathbf{t}_{k+1}$,$\mathbf{t}'_{k+1}$, $L(\mathbf{a}_{k+1}, \mathbf{a}'_{k+1}, q, \mathbf{t}_{k+1}, \mathbf{t}'_{k+1})$,
4. $\quad\quad$ -- *query* (8.35) *on page 198 goes here* --)
5. -- *compute I: initialize the buffer* ---
6. doany $\langle\mathbf{a}_{k+1}, w\rangle \in \Delta.\mathtt{recv\_buf}\langle\mathbf{a}_{k+1}, v_{k+1}\rangle$
7. $\quad$ w := identity   *see the explanation below*
8. enddo
9. -- *compute II: compute the updates* --
10. doany $\langle\ \mathbf{i}, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m \rangle\ \rangle \in \ldots$
11. $\ldots \bigominus_E \begin{pmatrix} \mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}'_1\langle\mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle\mathbf{a}'_k, v_k\rangle, \\ \Delta.\mathtt{recv\_buf}\langle\mathbf{a}_{k+1}, v_{k+1}\rangle, \\ \mathbf{A}_{k+2}\langle\mathbf{a}_{k+1}, v_{k+2}\rangle, \ldots, \mathbf{A}_m\langle\mathbf{a}_m, v_m\rangle \end{pmatrix}$
12. $\quad$ v_{k+1} := v_{k+1} op update
13. enddo
14. -- *communicate: send and recv buffers are switched!!* --
15. Δ.send(prefix,Δ.recv_buf(prefix))
16. Δ.recv(prefix,Δ.send_buf(prefix))
17. -- *apply the updates into the receive buffer* --
18. doany $\langle\mathbf{a}'_{k+1}, q, \mathbf{b}', v, w\rangle \in \ldots$
19. $\ldots \bigominus_{\mathbf{b}'=\mathbf{a}'_{k+1}} \left(\Delta.\mathtt{send\_buf}(prefix)\langle q, \mathbf{b}', w\rangle, \mathbf{A}^{(p)}_{k+1}\langle\mathbf{a}'_{k+1}, v\rangle \right)$
20. $\quad$ v := v op w
21. enddo

Couple of observations about this code:

- The identity is 1 if the update operation is multiplication and 0 if the operation is addition (line 7).
- Notice that the roles of the send and receive buffers have been switched (lines 15 and 16).

$\mathbf{A}_{k+1}$ is the `var` in "`var := rhs`". The code is:

Program 8.23

-- *allocate buffers* --

Δ`.allocate(prefix,`$\mathbf{A}^{(p)}$`,`

$\quad\quad \mathbf{a}_{k+1}, \mathbf{a}'_{k+1}, q, \mathbf{t}_{k+1}, \mathbf{t}'_{k+1},\ L(\mathbf{a}_{k+1}, \mathbf{a}'_{k+1}, q, \mathbf{t}_{k+1}, \mathbf{t}'_{k+1}),$

$\quad\quad$ -- *query* (8.35) *on page 198 goes here* -- `))`

-- *compute* --

`doany` $\langle\ \mathbf{i}, \mathbf{a}'_1, v_1, \ldots, \mathbf{a}'_k, v_k, \mathbf{a}_{k+1}, v_{k+1}, \ldots, \mathbf{a}_m, v_m\rangle\ \rangle \in \ldots$

$$\ldots \bigominus_E \left(\begin{array}{l} \mathcal{B}\langle\mathbf{i}\rangle, \mathbf{A}'_1\langle\mathbf{a}'_1, v_1\rangle, \ldots, \mathbf{A}'_k\langle\mathbf{a}'_k, v_k\rangle, \\ \Delta.\mathtt{recv\_buf}\langle\mathbf{a}_{k+1}, v_{k+1}\rangle, \\ \mathbf{A}_{k+2}\langle\mathbf{a}_{k+1}, v_{k+2}\rangle, \ldots, \mathbf{A}_m\langle\mathbf{a}_m, v_m\rangle \end{array} \right)$$

$\quad$ `v`$_{k+1}$ `:= rhs`

`enddo`

-- *communicate* --

Δ`.send(prefix,`Δ`.recv_buf(prefix))`

Δ`.recv(prefix,`Δ`.send_buf(prefix))`

-- *perform the assignments* --

`doany` $\langle\mathbf{a}'_{k+1}, \mathbf{b}', v, w\rangle \in \ldots$

$$\ldots \bigominus_{\mathbf{b}'=\mathbf{a}'_{k+1}} \left(\Delta.\mathtt{send\_buf}(prefix)\langle q, \mathbf{b}', w\rangle, \mathbf{A}^{(p)}_{k+1}\langle\mathbf{a}'_{k+1}, v\rangle \right)$$

$\quad$ `v := w`

`enddo`

Now if the index translation relation Δ supplies all of the necessary access methods – we are done. The communication generation transformation is applied repetitively in order to resolve all non-local references in the loop. Then the following steps are performed in order to obtain the final code:

- The invocation of the `allocate` methods is moved (if possible) outside of surrounding loops. This technique is especially effective in iterative algorithms, because the cost of the method can be amortized over many iterations. Standard data-flow analysis techniques [1] are used to determine the legality of this code motion.
- The compiler then expands the `allocate`() method invocations, because they might expose sequential `doany` loops.
- Sequential code generation are related optimizations are applied in order to generate final code.

8.4.4 Generating the default index translation ADT

It is possible that the ADT for the index translation function provides the `replicated`() method only. We now describe how the other methods are generated by the compiler.

Δ`.local`()

Let $\Lambda\langle\mathbf{t}, p, \mathbf{t}'\rangle$ be the shorthand for the relation Δ`.replicated`()$\langle\mathbf{t}, p, \mathbf{t}'\rangle$. Then the `local`() method is defined simply as:

Δ`.local()` $\equiv$ `Expr`$\langle$

$\quad$ `Theta(p=q,` $\Lambda\langle\mathbf{t}, q, \mathbf{t}'\rangle$`)`

$\rangle$

where p is the number of the "current" processor. Technically, the definition requires a projection:

$$\Delta.\mathtt{local}()\langle \mathbf{t}, \mathbf{t}' \rangle = \pi_{\mathbf{t},\mathbf{t}'} \bigominus_{p=q} \left(\ \Lambda\langle \mathbf{t}, q, \mathbf{t}' \rangle \ \right) \tag{8.42}$$

However, in loop nests the projection can be dropped according to Theorem 5.7.

$\underline{\mathtt{allocate}(\mathtt{prefix}, \mathbf{A}^{(\mathtt{p})}, \mathbf{a}, \mathbf{a}', \mathtt{p}, \mathbf{t}, \mathbf{t}', \mathtt{L}(\mathbf{a}, \mathbf{a}', \mathbf{t}, \mathbf{t}'), \mathtt{Q})}$

The `allocate()` method takes as its arguments the list of the names of various indices, the boolean expression L for the alignment, the name of the local fragment $\mathbf{A}^{(p)}$ of the relation being communicated and the Θ-join Q such that:

$$USED\langle \mathbf{a} \rangle = \pi_{\mathbf{a}} Q \tag{8.43}$$

$USED$ set is stored in a hash table by running the loop:

Program 8.24

abstract $USED$(USED_HASH_TABLE ...)

$USED$.init()

doany $\langle \mathbf{a}, \ldots \rangle \in Q$

  $USED$.insert($\mathbf{a}$)

enddo

The $RECV\_BUFFER$ data structure, shown in Figure 8.1, is used to store the global indices of $\mathbf{A}$ and the corresponding local indices and processor numbers. The global indices stored in the $USED$ hash-table

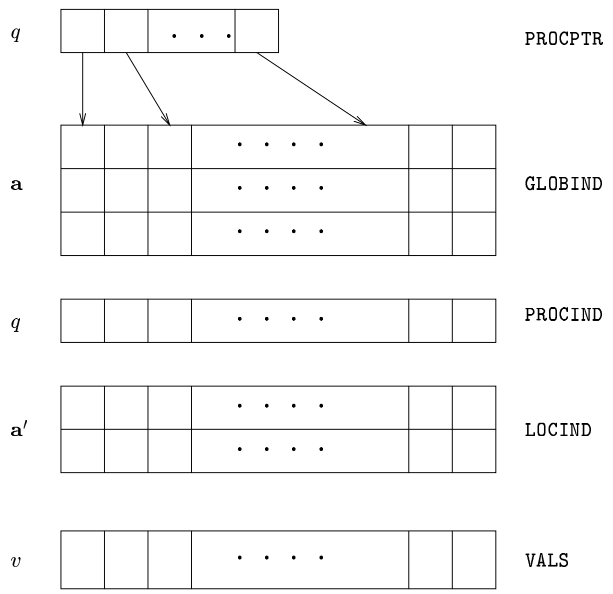


Figure 8.1: $RECV\_BUFFER$

are copied into the `GLOBIND` array (all names are, of course, prefixed in order to make them unique in the program). The array is viewed as the relation G with the schema $\langle \mathbf{a}, k \rangle$ where k is the offset into the array. Then the code executed to translate the indices is:

Program 8.25

doany $\langle \mathbf{a}, k, \mathbf{a}', \mathbf{t}, \mathbf{t}', q \rangle \in \bigominus_{L(\mathbf{a},\mathbf{a}',\mathbf{t},\mathbf{t}')} \left(\Lambda\langle \mathbf{t}, q, \mathbf{t}' \rangle, G\langle \mathbf{a}, k \rangle \right)$

    PROCIND[k] := q; LOCIND[k] := $\mathbf{a}'$;

enddo

The indices of the values to be received are then exchanged between processors. The result on each processor is the $SEND\_BUFFER$ data structure, shown in Figure 8.2. The S_LOCIND array stores the local indices

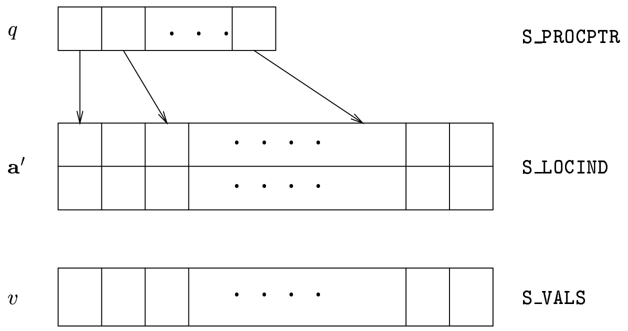


Figure 8.2: $SEND\_BUFFER$

that are requested by other processors (based on the S_PROCPTR array). Communication of values between the $SEND\_BUFFER$ and $RECV\_BUFFER$ is done by calling MPI send and receive routines with appropriate segments of the VAL and S_VAL arrays [51, 77].

In order to copy the values between the local fragments $\mathbf{A}^{(p)}$ and the communication buffers, the buffers can be viewed as essentially sparse matrices or relations. The GLOBIND and VALS arrays store the relation R with the schema $\langle \mathbf{a}, v \rangle$. It is used to replace the global view being localized in the original query and is returned by the Δ.recv_buf(prefix) method. The S_LOCIND and S_VALS arrays are viewed as the relation S with the schema $\langle q, \mathbf{a}', v \rangle$. The ADT expression for the relation is returned by the Δ.send_buf(prefix) method. The Black Boxes for the relations are quite simple: they generalize the coordinate storage format (Figure 4.6).

To summarize the generated access methods for Δ:

- allocate collects the $USED$ set in a hash table and then uses the $RECV\_BUFFER$ and $SEND\_BUFFER$ data structures to store and exchange the requests for data.
- send_buf method returns the relation S described above
- recv_buf method returns the relation R described above
- send and recv methods use MPI communication routines to exchange the values between the matching buffers in the $RECV\_BUFFER$ and $SEND\_BUFFER$ data structures.

8.5 An extended example

We illustrate the ideas of the previous section on an example. As before, we use sparse matrix-vector product $\mathbf{Y} = \mathbf{A}\mathbf{X}$. The matrix is partitioned by row and each fragment is stored in CSR format as specified in Section 6.1. The fragmentation equations for the matrix and the vectors are (from Section 8.3.2):

$$\mathbf{A}\langle r,c,a\rangle = \bigcup_p \pi_{r,c,a} \bigominus_{r=t\wedge r'=t'\wedge c=c'\wedge p=q} \left(\begin{array}{l} \{1 \le r,c \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{A}^{(p)}\langle r',c',a\rangle \end{array} \right) \tag{8.44}$$

$$\mathbf{Y}\langle i_y,y\rangle = \bigcup_p \pi_{i_y,y} \bigominus_{i_y=t\wedge i'_y=t'\wedge p=q} \left(\begin{array}{l} \{1 \le i_y \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{Y}^{(p)}\langle i'_y,y\rangle \end{array} \right) \tag{8.45}$$

$$\mathbf{X}\langle j_x,x\rangle = \bigcup_p \pi_{j_x,x} \bigominus_{j_x=t\wedge j'_x=t'\wedge p=q} \left(\begin{array}{l} \{1 \le j_x \le n\}, \{p\}, \\ \Delta\langle t,q,t'\rangle, \mathbf{X}^{(p)}\langle j'_x,x\rangle \end{array} \right) \tag{8.46}$$

The index translation relation Δ is stored in the format that is used in the BlockSolve library: each processor gets the same (small) number of contiguous blocks of rows. We start by describing the ADT for the index translation in this case. Then we discuss the major step in the code generated by the compiler.

Data structures

Let χ be the number of blocks of rows allocated to each processor. Let P be the number of processors and n be the number of rows and columns in the matrix. The index translation relation can be stored in two arrays of length $P \cdot \chi + 1$: `B` and `Bloc`. The element $\mathtt{B}[k]$ of the array keeps the starting index of the k-th block. Clearly, $\mathtt{B}[1] = 1$. Also we set $\mathtt{B}[P \cdot \chi + 1] = n + 1$. This way the k-th block is the interval:

$$\mathtt{B}[k] \le i < \mathtt{B}[k+1] \tag{8.47}$$

Given an index i we can find the corresponding block k in $O(\log \chi + \log P)$ time by using binary search, because the elements of `B` are sorted. Given the block index k we can find the processor p that owns the block as:

$$p = k \bmod \chi \tag{8.48}$$

(assuming that $0 \le p < P$). The array `Bloc` is used to find the local index i' for the given global index i, once we have found the block index k:

$$i' = i - \mathtt{Bloc}[k] \tag{8.49}$$

The values of the array are computed from the values of the `B` array as follows. Let S_k be the size of k-th block:

$$S_k = \mathtt{B}[k+1] - \mathtt{B}[k], \quad 1 \le k \le P \cdot \chi \tag{8.50}$$

Each processor p, $0 \le p < P$, owns the blocks:

$$1 + p, 1 + p + \chi, 1 + p + 2\chi, \ldots \tag{8.51}$$

And each block stores the elements with the local index starting at:

$$1,\ 1+S_{1+p},\ 1+S_p+S_{1+p+\chi},\ 1+S_{1+p}+S_{1+p+\chi}+S_{1+p+2\chi},\ \ldots \tag{8.52}$$

Overall the array Bloc is computed by the following program:

Program 8.26

```
for p = 1, P
    Bloc[p] := 0
for k = p+1, P * χ
    Bloc[k] := Bloc[k-χ] + S_{k-χ}
```

We can now define the Black Box for viewing the arrays B and Bloc as a relation that stores the $\langle i, q, i' \rangle$ tuples of the global indices, processor numbers and local indices. We call this Black Box MBLOCK (as in "multi-block"). It takes the following formal arguments:

- n is the range of the global index: $1 \le i \le$ n.
- nn is the maximum range of the local index (on any processor): $1 \le i' \le$ nn.
- nproc is the number of processors: $0 \le q <$ nproc.
- nblock is the number of blocks assigned to each processor (χ is the discussion above)
- bstart is the B array described above
- bloc is the Bloc array described above

The Black Box provides two index hierarchies:

- $q^* \prec \langle i, i' \rangle^*$ is used to enumerate the local and global indices for each processor, separately.
- $i^* \prec \langle q, i' \rangle^1$ is used to search for the processor and local index, given the global index.

The following listing defines all the ADTs for the Black Box:

Program 8.27

```
Formal arguments:  n, nn, nproc, nblock, bstart, bloc
MBLOCK.schema() ≡ ⟨ i, q, i' ⟩
MBLOCK.hierarchy() ≡ ⟨ q* ≺ ⟨i,i'⟩* ⊕ i* ≺ ⟨q,i'⟩¹ ⟩
MBLOCK.bounds() ≡ ⟨ 1 ≤ i ≤ n ∧ 1 ≤ i' ≤ nn ∧ 0 ≤ q < nproc ⟩
MBLOCK.h1() ≡ Expr⟨ -- ADT for the first hierarchy --
    (MBLOCK1 n=n nn=nn nproc=nproc
            nblock=nblock bstart=bstart bloc=bloc) ⟩
MBLOCK.h2() ≡ Expr⟨ -- ADT for the second hierarchy --
    (MBLOCK2 n=n nn=nn nproc=nproc nblock=nblock
            bstart=bstart bloc=bloc) ⟩

MBLOCK1.fields() ≡ ⟨ q ⟩
MBLOCK1.special_kind() ≡ ⟨ Dense ⟩
MBLOCK1.lb() ≡ Expr⟨ 0 ⟩
```

```
MBLOCK1.ub() ≡ Expr⟨ nproc - 1 ⟩
MBLOCK1.hderef(q) ≡ Expr⟨
    (MBLOCK1_1 proc=q nblock=nblock nproc=nproc
            bstart=bstart bloc=bloc) ⟩

MBLOCK1_1.fields() ≡ ⟨ i, i' ⟩
MBLOCK1_1.special_kind() ≡ ⟨ Stream ⟩
MBLOCK1_1.declare_iter(h) ≡ Decls⟨
    integer $k, $ii
    abstract h(MBLOCK1_ITER k=$k ii=$ii
            proc=proc nblock=nblock
            nproc=nproc bstart=bstart bloc=bloc)
⟩
MBLOCK1_1.open(h) ≡ Stmt⟨ h.open() ⟩
MBLOCK1_1.next(h) ≡ Stmt⟨ h.next() ⟩
MBLOCK1_1.valid(h) ≡ Expr⟨ h.valid() ⟩
MBLOCK1_1.deref_i(h) ≡ Expr⟨ h.deref_i() ⟩
MBLOCK1_1.deref_i'(h) ≡ Expr⟨ h.deref_i'() ⟩

MBLOCK1_ITER.open() ≡ Stmt⟨
    k := proc+1; ii := 1
⟩
MBLOCK1_ITER.next() ≡ Stmt⟨
    ii := ii + 1;
    if (ii = bloc[k+1]) then k := k + nblock endif
⟩
MBLOCK1_ITER.valid() ≡ Expr⟨ 1 ≤ k ≤ nproc * nblock ⟩
MBLOCK1_ITER.deref_i() ≡ Expr⟨ bloc[k]+ii ⟩
MBLOCK1_ITER.deref_i'() ≡ Expr⟨ ii ⟩
MBLOCK1_ITER.size() ≡ ⟨ 2 ⟩
MBLOCK1_ITER.marshall(b,kk) ≡ Stmt⟨
    b[kk] := k; b[kk+1] := ii
⟩
MBLOCK1_ITER.unmarshal(b,kk) ≡ Stmt⟨
    k := b[kk]; ii := b[kk+1]
⟩

MBLOCK2.fields() ≡ ⟨ i ⟩
MBLOCK2.special_kind() ≡ ⟨ Stream ⟩
MBLOCK2.declare_iter() ≡ Decls⟨
    integer $k, $i
```

```
    abstract h(MBLOCK2_ITER k=$k i=$i
            n=n nblock=nblock nproc=nproc
            bstart=bstart bloc=bloc)
⟩
MBLOCK2.open(h) ≡ Stmt⟨ h.open() ⟩
MBLOCK2.next(h) ≡ Stmt⟨ h.next() ⟩
MBLOCK2.valid(h) ≡ Expr⟨ 1 ≤ i ≤ n ⟩
MBLOCK2.deref(h) ≡ Expr⟨ h.deref_i() ⟩
MBLOCK2.search(h,j) ≡ Stmt⟨ h.search(j) ⟩
MBLOCK2.search_cost() ≡ ⟨ Olog ⟩
MBLOCK2.hderef(h) ≡ Expr⟨ (MBLOCK2_1 iter=h) ⟩

MBLOCK2_ITER.open() ≡ Stmt⟨
    i := 1; k := 1;
⟩
MBLOCK2_ITER.next() ≡ Stmt⟨
    i := i + 1;
    if (i = bstart[k+1]) then k := k + 1 endif
⟩
MBLOCK2_ITER.valid() ≡ Expr⟨ 1 ≤ k ≤ n ⟩
MBLOCK2_ITER.deref_i() ≡ Expr⟨ i ⟩
MBLOCK2_ITER.deref_q() ≡ Expr⟨ k mod nblock ⟩
MBLOCK2_ITER.deref_i'() ≡ Expr⟨ i - bloc[k] ⟩
MBLOCK2_ITER.search(j) ≡ Stmt⟨
    -- binary search to find the block k --
⟩

MBLOCK2_1.fields() ≡ ⟨ q, i' ⟩
MBLOCK2_1.deref_q() ≡ Expr⟨ iter.deref_q() ⟩
MBLOCK2_1.deref_i'() ≡ Expr⟨ iter.deref_i'() ⟩
```

The other relations used in the generated code are the send/receive buffers (Figures 8.1 and 8.2). These are viewed as a generalization of the coordinate storage format. Let us start with the *RECV_BUFFER* data structure. First we need a view of the `GLOBIND` array as the relation $G\langle \mathbf{a}, k\rangle$ that stores global indices $\mathbf{a}$ and the offset k into the array. Let d be the dimension of the global indices $\mathbf{a}$. The compiler generates a separate Black Box for each value of d. For a fixed value of d the Black Box is:

Program 8.28

```
Formal arguments: nind, globind
GLOBIND_BB.schema() ≡ ⟨ a_1, a_2, ..., a_d, k ⟩
GLOBIND_BB.hierarchy() ≡ ⟨ k* ≺ ⟨a_1, a_2, ..., a_d⟩^1 ⟩
GLOBIND_BB.bounds() ≡ ⟨ true ⟩
GLOBIND_BB.h() ≡ Expr⟨ (GBB nind=nind globind=globind) ⟩
```

```
GBB.fields() ≡ ⟨ k ⟩
GBB.special() ≡ ⟨ Dense ⟩
GBB.lb() ≡ Expr⟨ 1 ⟩
GBB.ub() ≡ Expr⟨ nind ⟩
GBB.hderef(k) ≡ Expr⟨ (GBB1 offset=k globind=globind) ⟩

GBB1.fields() ≡ ⟨ a_1, a_2, ..., a_d ⟩
-- deref() method for the j-th field --
GBB1.deref_{a_j} ≡ Expr⟨ globind[ (k-1)*d + j ] ⟩
```

Observe that we have left the bounds completely unrestricted. Since the `bounds` method is used only in simplifying some queries this does not affect the correctness of the derivation.

We also need a view of `GLOBIND` and `VALS` arrays as a relation $R\langle \mathbf{a}, v\rangle$ that stores global indices and values. Just as before, compiler generates a Black Box for each particular dimension of the global indices **a** and the value of the global bounds. We call this Black Box `RECV_BUF`:

Program 8.29

```
Formal arguments: nind, globind, vals
RECV_BUF.schema() ≡ ⟨ a_1, a_2, ..., a_d, v ⟩
RECV_BUF.hierarchy() ≡ ⟨ ⟨a_1, a_2, ..., a_d⟩* ≺ v^1 ⟩
RECV_BUF.bounds() ≡ ⟨ true ⟩
RECV_BUF.h() ≡ Expr⟨ (RBB nind=nind globind=globind) ⟩

RBB.fields() ≡ ⟨ a_1, a_2, ..., a_d ⟩
RBB.special() ≡ ⟨ Indirect ⟩
RBB.lb() ≡ Expr⟨ 1 ⟩
RBB.ub() ≡ Expr⟨ nind ⟩
-- deref() method for the j-th field --
RBB.deref_j(k) ≡ Expr⟨ globind[ (k-1)*d + j ] ⟩
RBB.hderef(k) ≡ Expr⟨
    (REAL_VAL name=v e=vals[k])
⟩
```

The arrays `S_PROCPTR`, `S_LOCIND` and `S_VALS` are viewed as a relation $S\langle q, \mathbf{a}', v\rangle$ that stores local indices $\mathbf{a}'$, values v to be sent and the destination processor indices q. Let d' be the dimension of the local indices. The `SEND_BUF` Black Box provides the index hierarchy:

$$q^* \prec \langle a'_1, a'_2, \ldots, a'_{d'}\rangle^* \prec v^1 \tag{8.53}$$

The compiler generates a different Black Box for each distinct dimensionality d' in the program:

Program 8.30

```
Formal arguments: nproc, procptr, locind, vals
SEND_BUF.schema() ≡ ⟨ q, a'_1, a'_2, ..., a'_{d'}, v ⟩
```

```
SEND_BUF.bounds() ≡ ⟨ true ⟩
SEND_BUF.h() ≡ Expr⟨ (SBB nproc=nproc
              procptr=procptr procind=locind vals=vals) ⟩

SBB.fields() ≡ ⟨ q ⟩
SBB.special() ≡ ⟨ Dense ⟩
SBB.lb() ≡ Expr⟨ 0 ⟩
SBB.ub() ≡ Expr⟨ nproc-1 ⟩
SBB.hderef(q) ≡ Expr⟨ (SBB1 proc=q
              procptr=procptr procind=locind vals=vals) ⟩

SBB1.fields() ≡ ⟨ a'_1, a'_2, ..., a'_d' ⟩
SBB1.special() ≡ ⟨ Indirect ⟩
SBB1.lb() ≡ Expr⟨ procptr[proc+1] ⟩
SBB1.ub() ≡ Expr⟨ procptr[proc+2]-1 ⟩
-- dereference for each field a'_j --
SBB1.deref_a'_j(kk) ≡ Expr⟨ locind[ (kk-1)*d' + j ] ⟩
SBB1.hderef(kk) ≡ Expr ⟨
    (REAL_VAL name=v e=vals[kk])
⟩
```

Generated code

The code generated by the compiler involves several queries:

- Computation of the $USED$ set (Program 8.24)
- Translation of the $USED$ set (Program 8.25)
- Gathering of the values from the local fragment into the send buffer
- Actual computation with the receive buffer

We now discuss them in more detail.

After the references to the global views for the matrix **A** and the vector **Y** have been translated into the references to the local fragments, we arrive at the loop nest:

Program 8.31
doany $\langle r', c', a, j_x, x, i'_y, y \rangle \in \ldots$

$$\ldots \bigominus_{r'=i'_y \wedge c'=j_x} \left(\mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, \mathbf{X}\langle j_x, x\rangle \right)$$

```
  y := y + a * x
```

We now have to generate code that will communicate the values of **X**. The loop that computes the $USED$ set is:

Program 8.32
abstract $USED$(USED_HASH_TABLE ...)
doany $\langle r', c', a, j_x, x, i'_y, y \rangle \in \ldots$

$\ldots \bigominus_{r'=i'_y \wedge c'=j_x} \left(\{1 \le j_x \le n\}, \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle \right)$

    $USED$.insert(j_x)

Since the index hierarchy of the $\mathbf{A}^{(p)}$ relation is $r' \prec c' \prec a$, the join between r' and i'_y is scheduled first. After query scheduling and join implementation the code is:

Program 8.33

do i' = 1, nlocal
    do jj = pntrb[i'], pntre[i']-pntrb[1]
        j_x := indx[jj]
        $USED$.insert(j_x)

At this point the $RECV\_BUFFER$ data structure is allocated and the global indices j_x stored in the $USED$ hash table are copied into the GLOBIND array.The next step is translating the global indices. The loop that performs the translation is (cf. Program 8.25):

Program 8.34

1. -- *definition for the index translation* --
2. integer nblock, nproc, B[...], Bloc[...]
3. abstract Λ(MBLOCK ...)
4. -- *globind array as a relation* --
5. abstract G(GLOBIND_BB ...)
6. doany $\langle j, k, j', t, t', q\rangle \in \bigominus_{j=t \wedge j'=t'} \left(\Lambda\langle t, q, t'\rangle, G\langle j, k\rangle \right)$
7.   procind[k] := q; locind[k] := j';

Let us consider possible schedules for the query on line 6. The index translation relation Λ provides us with two index hierarchies: $q^* \prec \langle t, t'\rangle^*$ and $t^* \prec \langle q, t'\rangle^1$. The relation G has the index hierarchy $k^* \prec j'$. Suppose we pick the first hierarchy in Λ. A pseudo-code for one of the schedules is:

Program 8.35

doany $q \in \Lambda$
    doany $k \in G$
        j := G_k
        doany $\langle t, t'\rangle \in \Lambda_q$
            if (j = t) then
                procind[k] := q; locind[k] := j';

There are two more schedules: the loops in Program 8.35 can be reordered with the constraint that q is enumerated before $\langle t, t'\rangle$. Regardless of the loop order the complexity of the code is:

$$O(P\eta\mu) \tag{8.54}$$

where P is the number of processors, η is the number of indices in G and μ is the maximum number of local indices assigned to each processor.

If we pick the second, $t^* \prec \langle q, t'\rangle^1$, hierarchy for Λ, then there are two possible schedules:

Program 8.36

doany $k \in G$

```
    j := G_k
    t := j
    search t in Λ -- binary search --
    if found then
        ⟨q,t'⟩ := Λ_t
        procind[k] := q; locind[k] := j';
    endif
```

and

Program 8.37

```
doany t ∈ Lambda
    ⟨q,t'⟩ := Λ_t
    doany k ∈ G
        j := G_k
        if (j = t) then
            procind[k] := q; locind[k] := j';
        endif
```

The complexity of this schedule in Program 8.36 is:

$$O(\eta \log(P + B)) \tag{8.55}$$

where B is the number of blocks of rows assigned to each processor. The complexity of the schedule in Program 8.37 is:

$$O(n\eta) \tag{8.56}$$

where n is the range of global indices. Clearly, the schedule in Program 8.36 has the lowest asymptotic complexity. But how does the compiler pick the best schedule? In this case, the simple heuristic that pick the schedule with the fewest `doany` loops does the trick. As we have discussed in Section 5.5.8, such schedule is expected to have the lowest asymptotic complexity. The concrete code for Program 8.36 is:

Program 8.38

```
do k=1,nind
    j := globind[k]
    perform binary search into Λ to find q and t'
    procind[k] := q; locind[k] := j';
enddo
```

Once the indices are translated, the tuples in `globind`, `locind` and `procind` array are sorted by processor number and the `procptr` is set up. Observe that the sorting can be done in $O(\eta + P)$ time. The indices are then exchanged between processors and the $SEND\_BUFFER$ data structure is set up. The only loop nest involving the data structure copies the values from the local fragment $\mathbf{X}^{(p)}\langle j', v\rangle$ into the send buffer $S\langle q, j', w\rangle$:

Program 8.39

doany $\langle j', v, q, j'', w\rangle \in \bigominus_{j'=j''} \left(\mathbf{X}^{(p)}\langle j', v\rangle, S\langle q, j'', w\rangle \right)$

```
    w := v
```

Since $\mathbf{X}^{(0)}$ is dense, the schedule for the loop nest is:

Program 8.40

doany $q \in S$
    doany $j'' \in S_q$
        j' := j''
        find the value v in $\mathbf{X}^{(p)}$
                for the index j'
        S_q,j'' := v

The concrete code is:

Program 8.41

```
do q = 0, nproc-1
    do kk=S_PROCPTR[q+1], S_PROCPTR[q+2]-1
        j' := S_LOCIND[kk]
        S_VALS[kk] := X'[j']
```

The last query is the one that performs the actual computation (cf. Program 8.31):

Program 8.42

doany $\langle r', c', a, j_x, x, i'_y, y \rangle \in \ldots$
        $\ldots \bigominus_{r'=i'_y \wedge c'=j_x} \left(\; \mathbf{A}^{(p)}\langle r', c', a\rangle, \mathbf{Y}^{(p)}\langle i'_y, y\rangle, R\langle j_x, x\rangle \; \right)$
    y := y + a * x

Instead of the global view $\mathbf{X}$ we use the receive buffer R. The schedule for this query is:

Program 8.43

1. doany $r' \in \mathbf{A}$
2.   doany $c' \in \mathbf{A}_{r'}$
3. i'_y := r'
4.   y := $\mathbf{Y}^{(p)}_{i'_y}$
5. j_x := c'
6.   search R for j_x and the value x
7.   y := y + $\mathbf{A}$_r',c' * x

The search on line 6 is quite expensive, because it takes time proportional to the size of the receive buffer R. However, in iterative applications we can precompute the search and amortize its cost over several iterations. The optimization is described in Section 5.8.2.

In iterative applications the code that computes communication sets and allocates storage (Programs 8.38 and 8.41) together with the code that precomputes the search in Program 8.43 can be executed only once. Following the terminology of the Chaos project [67] we call this portion of the final program the *inspector*. The code that performs the actual computation (Program 8.43 with the search precomputed) is called the *executor*. In the next Chapter we present the performance evaluation of the inspector and executor codes generated by our compiler.

Chapter 9

Experiments

In this Chapter, we present performance measurements on the IBM SP-2 at Cornell Theory Center. The experiments were done in the Fall of 1997 using 2 to 64 "thin" nodes of the multicomputer. Each node has a 120MHz POWER2 Super Chip (P2SC) architecture RS/6000 processor with 256 Mbytes of memory [65]. The goal of the experiments has been to demonstrate the necessity of exploiting user-defined formats for the index translation relation. We have compared the performance of three versions of parallel CG algorithm with diagonal preconditioning:

- The code provided by the BlockSolve library. This is the base line for the comparisons. In the graphs this code is referred to as "BS95".
- Same code, but with communication and index translation routines provided by the Chaos library [67]. We have chosen to use Chaos library as it is often sited as the target run-time layer for the compilation of irregular codes written in HPF-2. The library supports the general INDIRECT distributions. This code is referred to as "CHAOS".
- Code generated by the compiler. It is referred to as "BERNOULLI" in the graphs.

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 7.1, and the off-diagonal sparse portions of the matrix shown using gray rectangles in the figure. The sparse portions are stored using the INODE format (Figure 1.15).

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:

$$\begin{aligned} \mathbf{y} &= \mathbf{A}_D \cdot \mathbf{x} \\ \mathbf{y} &= \mathbf{y} + \mathbf{A}_{SL} \cdot \mathbf{x} \\ \mathbf{y} &= \mathbf{y} + \mathbf{A}_{SNL} \cdot \mathbf{x} \end{aligned} \tag{9.1}$$

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:

$$\begin{aligned} &\text{local:} && \mathbf{y}^{(p)} = \mathbf{A}_{D^{(p)}} \cdot \mathbf{x}^{(p)} \\ &\text{local:} && \mathbf{y}^{(p)} = \mathbf{y}^{(p)} + \mathbf{A}_{SL}^{(p)} \cdot \mathbf{x}^{(p)} \\ &\text{global:} && \mathbf{y} = \mathbf{y} + \mathbf{A}_{SNL} \cdot \mathbf{x} \end{aligned} \tag{9.2}$$

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 [38].

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.

We ran the different implementations of the solver on two larger matrices from the Boeing-Harwell collection obtained through the MatrixMarket collection [22]: BCSSTK30 and BCSSTK32. These matrices are representative of those occurring in finite-element analysis. The matrices were partitioned among the processors using the Chaco graph partitioner [41].

In order to understand how the inspector and the executor scale for large numbers of processors, we also ran the implementations 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. We kept the problem size *per processor* constant by placing a $30 \times 30 \times 30$ grid fragment on each processor. The corresponding sparse matrix has 135×10^3 rows with about 4.5×10^6 non-zeroes total on each processor.

We have used two measures of performance

- The time taken by the inspector.
- The ratio of the time taken by the inspector to the time taken by a *single iteration* of the executor.

The total number of iterations of the CG algorithm and, thus, the impact of the cost of the inspector depend on the actual values of the elements of the matrix. Therefore, the above separation of time measurements provides a problem-independent assessment of performance of the implementations of the algorithm.

Figure 9.1 shows the comparison of the time taken by the executors in the native BlockSolve implementation and in the compiler generated code. Overall, the performance of compiler-generated code is within 2-4% of that of the library code. We do not provide the executor time for the implementation using the Chaos library, since the executor in this case is merely a copy of the BlockSolve library code and does not reflect any compiler technology.

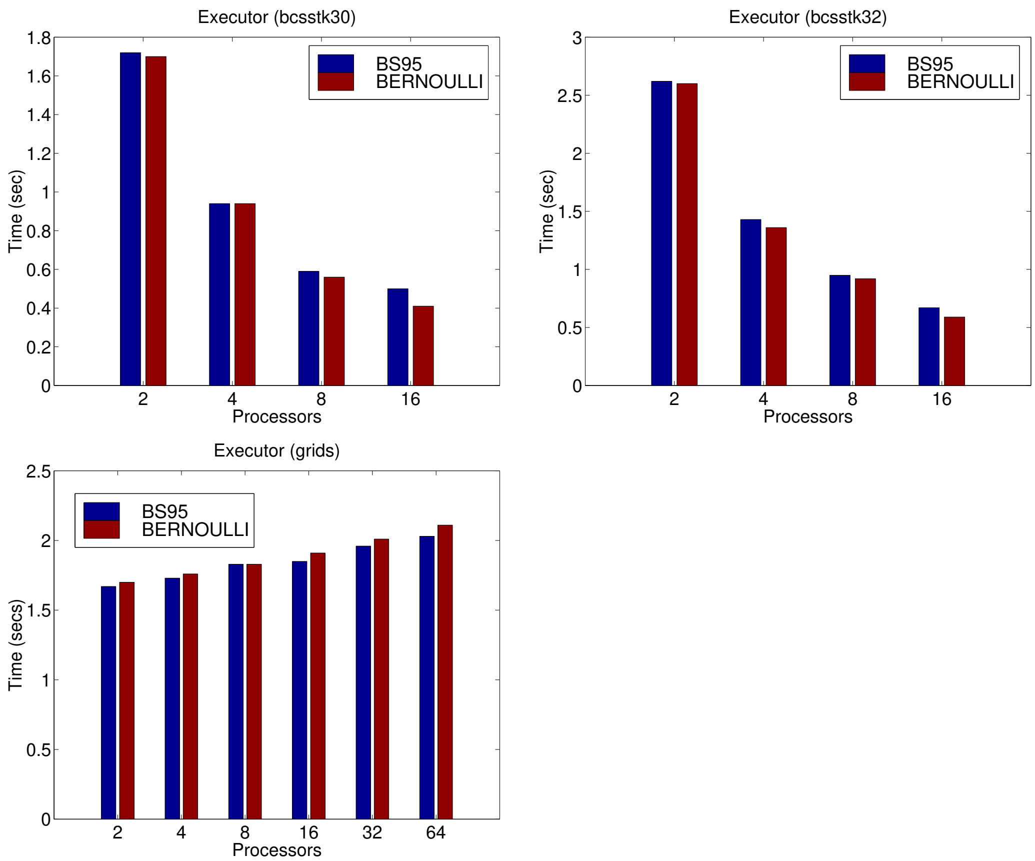


Figure 9.1: Executor performance

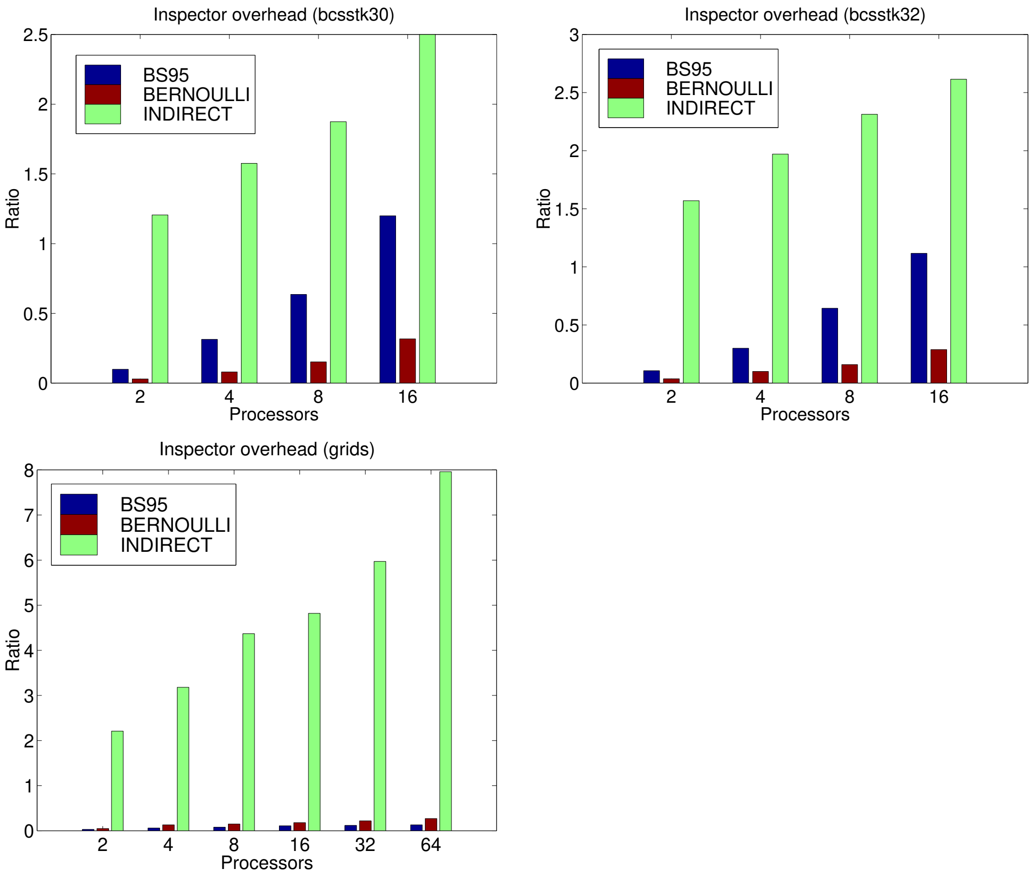


Figure 9.2: Inspector overhead

Figure 9.2 shows the comparison of inspector ratios for the three implementations. For large grid problems the inspector provided by the Chaos library is an order of magnitude more costly than either the one generated by the compiler or the native one. The reason is that in order to setup index translation the Chaos library has to communicate $O(n)$ indices, where n is the number of rows in the matrix. At the same time, the inspectors in the compiler-generated code and in the native library code do not require only $O(r)$ communication where $r$is the number of data elements actually communicated. The performance results for the grid problems clearly demonstrate this difference in asymptotic complexity.

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 and CHAOS implementations for a range of problems we have plotted in Figure 9.3 the ratios of the time that the CHAOS implementation would take to the time that the BERNOULLI implementation would take on 8 and 64 processors for a range of iteration counts $5 \leq k \leq 100$. The lines in the figure plot the values of the ratio:

$$\frac{k + r_I}{k + r_B} \tag{9.3}$$

where r_B is the inspector overhead for the BERNOULLI version, r_I is inspector overhead for the CHAOS version and k is the iteration count. A simple calculation shows that it would take 77 iterations of a CHAOS solver on 64 processors to get within 10% of the performance of the BERNOULLI solver. 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 CHAOS 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 (9.2), 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.

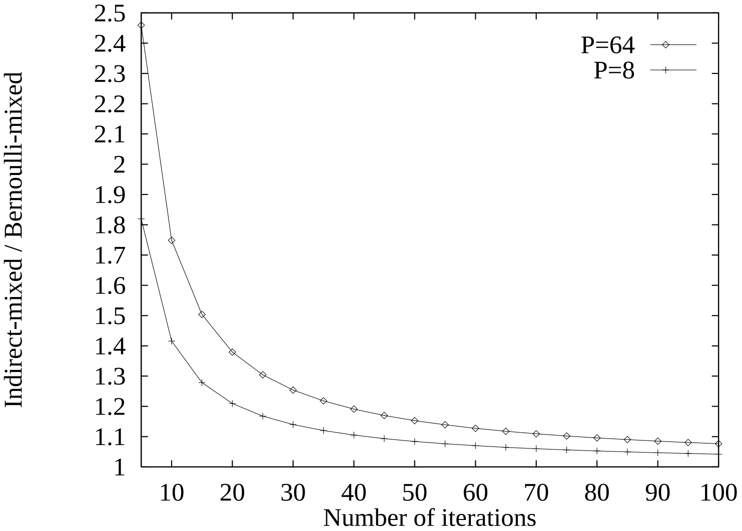


Figure 9.3: Effect of problem conditioning on the relative performance

Part IV

Conclusions

Chapter 10

Summary of related work

Previous research has been discussed in the appropriate portions of this thesis. In Chapter we summarize the related work and provide "backward pointers" to the relevant sections.

Compilers: dense matrix (regular) computations

The major advance in the compilation of dense matrix computations has been the abstraction of loop nests as polyhedra in the integer spaces of loop indices. To the best of our knowledge the model has been first discussed in a 1974 paper by Leslie Lamport [52]. Since then a considerable amount of work has been done on applying the model to dependence analysis [34, 70], automatic parallelization [5, 8, 35, 36] and locality optimizations [49, 55, 56, 89]. Under certain conditions compilation of data-parallel languages, such as HPF, can be described in term of operations on polyhedra [3].

In Section 2.2.1 of this thesis we discuss some of highlights of the linear loop model. In Section 7.1 we discuss the relationship between the model and compilation of data-parallel languages.

Compilers: irregular computations

Bik and Wijshoff were first to suggest automatic generation of sparse codes starting from dense codes . Bik's thesis [9] describes all of the techniques in detail. The work was published in [10–20]. Besides the computation of sparsity guard, discussed in Section 2.3.5, and the access reshaping algorithm, discussed in Section 2.2.2, this work includes non-zero structure analysis, sparse BLAS Web server [23] and heuristics for data structure selection [18]. We have implemented the algorithm for the computation of sparsity guard in our compiler.

The major difference with our work is that Bik's sparse compiler is limited to the formats which are a variation on the CRS and CCS schemes. No provisions are made for extending the compiler with new storage formats. Additionally, our algorithm for handling DOACROSS loop nests (Section 5.9) produces more general transformations that Bik's access reshaping algorithm.

There has been a lot of work in the data-parallel languages and compilers community on supporting irregular computations. Most notably, Saltz and colleagues have developed the inspector/executor methodology for communication generation [30, 67]. One of the results of this work in the Chaos library, which we discuss in throughout Chapter III. There is on-going work on extending the High Performance Fortran standard with features for specifying irregular computations and, specifically, on supporting sparse matrix computations [86, 88]. However, this work is limited to simple sparse matrix formats, such as CRS or CCS, and does not address handling of high-performance data structures. Moreover, the proposed HPF-2 standard [38] limits the formats for distribution relation. Experiments in Section 9 demonstrate the shortcomings of this approach.

Generic programming

C++ Standard Template Library is a classical example of a generic programming system [61, 62, 78, 94]. We discuss the library in Section 1.5. Our representation of ADTs as essentially collections of macros is closest to the mechanism described by Gordon Novak [63, 64] for the creation and manipulation of *views*, as discussed in Section 3.2.3.

Chapter 11

Contributions and future work

We close this thesis by, first, summarizing its contributions. Then we the limitations of our techniques and the directions of future research.

11.1 Contributions

Scientific software has traditionally been developed in layers. Code specific to a particular physical problem uses a library of high-level algebraic primitives (such as solution of linear systems). In turn the high-level library uses a set of low-level basic linear algebra subroutines (BLAS). In many applications most of the time is spent in BLAS computations, such as matrix-vector or matrix-matrix products. Libraries of *dense* BLAS are widely used because there are very few storage formats for dense matrices and, consequently, dense BLAS interface consists of a few routines which can be hand-tuned for particular hardware. In fact, computer hardware vendors supply hand-tuned implementations of dense BLAS and thus enable modular development of high-performance numerical software.

The situation is radically different in the domain of sparse matrix computations. Sparse matrices are used in a variety of applications and, as we have argued in Chapter 1, the choice of the data structures greatly affects program performance. More that a few different sparse matrix data structures, each with special time and space saving properties, are in use today in computational science software. And there is no reason to believe that that the that these will remain limited to the 13 defined in NIST Sparse BLAS library or the format used in BlockSolve library. In this context it is hard to maintain a comprehensive set of efficient Sparse BLAS, because (a) the multitude of formats implies the multitude of code and (b) adding a format implies developing from scratch of more that a few library functions.

We view the problem of supporting effective development of high-performance sparse matrix codes as one of *generic programming*. Generic programming is a discipline of designing and implementing software components which can be used when there is a set of *related data structures* supporting a common semantics described by an API or protocol, and a set of *common algorithms* that can be formulated in terms of this API.

In our case, sparse matrix formats form the set of related data structures and dense matrix codes are the natural specification of the algorithms. In this thesis we address the problem of automatically generating efficient sparse matrix codes for the basic computational kernels automatically from dense code and the specification of storage formats. As we argue in Chapter 2, the most widely used kernels fall in the class of *conjunctive DOANY* loops, which includes such computations as matrix-vector products. By being able to automatically generate these code for a variety of formats, our code generator provides an extensible set of sparse BLAS.

Another point of departure has been the prevalence of algebraic abstractions in restructuring compilers. Lattice algebras have been used in data flow analysis [1]. Polyhedral algebra (integer linear programming) has been used in analysis and restructuring of dense matrix (regular) codes [2, 34, 49, 70]. Besides being esthetically pleasing, these abstraction facilitate the development of new algorithms and simplify compiler implementations. The question for us has been: what is the right abstraction for generating efficient sparse matrix codes?

The main contribution of this thesis is **the abstraction of the computations and data access in dense and sparse loops as relational queries**. We view arrays in a loop nest as relations that store the tuples of arrays indices and values. Then we convert a the loop nest in a relational query that describes which loop indices, array indices and values have to be enumerated in during the computation. The benefit of such abstraction has long been recognized in the database community. Here is quote from the seminal paper by E.F.Codd [26]:

> [Relational model] provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

In other words, a relational query is the declarative specification of *what* has to be computed. Based on the identities of relational algebra, a query can be transformed into several equivalent expressions. The costs of the evaluation of each of the expressions depend on the *high-level* properties of the underlying data. The high-level properties include the metrics such as the cost of searching, the availability of random access and the order of enumeration of the tuples in a relation.

Based on these ideas we have developed the Bernoulli sparse compiler that takes conjunctive dense DOANY loop nests and specifications of data structures as its input and produces parallel or sequential sparse code. The major contributions in the development of the compiler are:

- Separation of specification of formats into low-level abstract data types, which hide the details of the implementation, and Black Boxes which present to the compiler a view of the high-level properties of the data structures, such as random access, ordering and cost of searching.
- Separation of the compilation process into query scheduling, join scheduling and instantiation. Query scheduling transforms the input (sequential) query into an equivalent nesting of loops based on identities of relational algebra and high-level properties of the data structures. The loops run over simpler join queries which usually have several possible implementations. Join scheduling selects the appropriate implementation for a query based on the high-level properties of the data structure. Instantiation then macro-expands the definitions of the ADTs into executable code. This separation makes the main optimization steps independent from the low-level details of data storage.
- Formulation of the problem of parallel message-passing code generation as that of distributed query evaluation. Distributed matrices and vectors are viewed as distributed relations. Loop nests enumerate over the results of queries over the relations. The parallel code generator accepts used-specified formats for partitioning information, as well, thus allowing the compiler the exploit the structure present in some applications.

On a theoretical side we developed a simple technique to extend query scheduling to DOACROSS loop nest:

- Query scheduling is associated with a special echelon form of the data access equation. We describe the space of possible transformations that can correct a schedule that violates the dependencies. Our algorithm is complete in that it is guaranteed to find a correct schedule if there is one. This is an improvement on the access reshaping method of Bik [10], which can generate only a limited set of transformations.

11.2 Limitations and future work

11.2.1 Specification of data structures

Simultaneous enumeration

The current version of the Black Box protocol together with the mechanism for specifying projection views does not convey all of the information that can be used in optimizing sparse matrix codes. Let's consider an example. The following code fragment adds up the products of the elements of the matrix **A** and **B**, both stored in Block Sparse Row format (Section 6.1):

Program 11.1

abstract $\mathbf{A}'$(BSR ...)

abstract $\mathbf{B}'$(BSR ...)

projection $\mathbf{A}\langle r,c,v\rangle$ of $\mathbf{A}'\langle br,bc,r,c,v\rangle$

projection $\mathbf{B}\langle r,c,v\rangle$ of $\mathbf{B}'\langle br,bc,r,c,v\rangle$

doany $\langle i,j,r_a,c_a,a,r_b,c_b,b\rangle \in \bigominus_{\substack{r_a=r_b=i\wedge\\ c_a=c_b=j}} \left(\begin{array}{l} \{1\le i,j\le n\}, \\ \mathbf{A}\langle r_a,c_a,a\rangle, \mathbf{B}\langle r_b,c_b,b\rangle \end{array} \right)$

    sum := sum + a * b

$\mathbf{A}'$ and $\mathbf{B}'$ are the relations with the block indices made explicit. The block indices are the hidden in the projection views $\mathbf{A}$ and $\mathbf{B}$. One possible schedule for the loop nest is:

Program 11.2

doany $br_a \in \mathbf{A}'$

    doany $bc_a \in \mathbf{A}'_{br_a}$

        doany $br_b \in \mathbf{B}'$

            doany $bc_b \in \mathbf{B}'_{br_b}$

                doany $\langle r_a,r_b\rangle \in \mathbf{A}'_{br_a,bc_a} \bowtie \mathbf{B}'_{br_b,bc_b}$

                    doany $\langle c_a,a,c_b,b\rangle \in \mathbf{A}'_{br_a,bc_a,r_a} \bowtie \mathbf{B}'_{br_b,bc_b,r_b}$

Other schedules can be obtained by interchanging some of the first four loops. The code basically enumerates over all possible combinations of blocks and then tries to join the blocks on row and column indices. Let η be the bound on number of blocks in the matrices. Let β be the bound on the number of elements in each block. The complexity of the above schedule is:

$$O(\eta^2\beta) \tag{11.1}$$

Now suppose that we know that the block sizes of $\mathbf{A}$ and $\mathbf{B}$ are the same. This still does not make any difference for the compiler, since we have no way of conveying this information. On the other hand, we could start with a different code:

Program 11.3

abstract $\mathbf{A}'$(BSR ...)

abstract $\mathbf{B}'$(BSR ...)

doany $\langle i,j,br_a,bc_a,r_a,c_a,a,br_b,bc_b,r_b,c_b,b\rangle \in \ldots$

$\ldots \bigominus_{\substack{r_a=r_b=i\wedge\\ c_a=c_b=j\wedge\\ br_a=br_b=bi\wedge\\ bc_a=bc_b=bj}} \left(\begin{array}{l} \{1\le i,j\le n \wedge 1\le bi,bj\le nb\}, \\ \mathbf{A}'\langle br_a,bc_a,r_a,c_a,a\rangle, \mathbf{B}\langle br_b,bc_b,r_b,c_b,b\rangle \end{array} \right)$

    sum := sum + a * b

Here we explicitly instruct the compiler to join the block indices. The resulting schedule is:

Program 11.4

doany $\langle br_a,br_b\rangle \in \mathbf{A}' \bowtie \mathbf{B}'$

    doany $\langle bc_a,bc_b\rangle \in \mathbf{A}'_{br_a} \bowtie \mathbf{B}'_{br_b}$

        doany $\langle r_a,r_b\rangle \in \mathbf{A}'_{br_a,bc_a} \bowtie \mathbf{B}'_{br_b,bc_b}$

```
doany ⟨c_a, a, c_b, b⟩ ∈ A'_{br_a,bc_a,r_a} ⋈ B'_{br_b,bc_b,r_b}
```

The complexity of this schedule is:

$$O(\eta\beta) \tag{11.2}$$

The still unresolved problem is: how to specify the views **A** and **B** in Program 11.1 so that (a) the relationship between the block indices is conveyed to the compiler and (b) the compiler can automatically convert Program 11.1 into Program 11.3?

Dependencies

Similar problem exists when compiling DOACROSS codes. For example, consider the solution of a unit lower triangular system $\mathbf{LX} = \mathbf{B}$ where the matrix $\mathbf{L}$ is stored in BSR format. While the schedule in Program 11.5 is legal, it can not be verified by the compiler.

Program 11.5

```
dolex br ∈ L
    dolex bc ∈ L_{br}
        dolex r ∈ L_{br,bc}
            dolex c ∈ L_{br,bc,r}
                B[r] := B[r] - L_{br,bc,r,c} * B[c]
```

In this case, the compiler does not know that enumerating the blocks of the matrix top-down and left-to-right does not violate the dependencies. Again, we do not have a way of relating the order of the enumeration of the blocks to the order of the rows and columns.

Composite data structures

The third problem is the specification of composite data structures, such as the one used in BlockSolve library. Recall from the discussion in Section 6.3 that the matrix can be thought of as the sum $\mathbf{A} = \mathbf{A}_D + \mathbf{A}_S$ where $\mathbf{A}_D$ represents the dense blocks of the matrix along the diagonal (Figure 6.8) and $\mathbf{A}_S$ represents off-diagonal blocks. Instead of specifying two sparse matrix-vector products:

```
Y := A_D * X
Y := Y + A_S * X
```

we would like to specify a single one that involves the composite matrix:

```
Y := A * X
```

and let the compiler decompose it appropriately.

Insertions (fill)

Currently, the Black Box protocol has no interface for specifying how new elements can be added to a data structure. The main problem is that while some data structures (such as a hash table) allow insertions at the level of individual elements, others (such as the i-node storage) have to be fully recomputed. The protocol for insertions must account for such differences. This problem will be addressed in Nikolay Mateev's thesis [58].

11.2.2 Sequential compilation techniques

Disjunctive loops

Bernoulli sparse compiler currently generates code for *conjunctive* DOANY loops. While disjunctive loops can be handled in out relational framework, as described in Section 2.3.2, not all of the details of the compilation of such loops have been worked out. We illustrate the main ideas on an example. The following loop computes the sum of the elements of the vectors $\mathbf{X}$ and $\mathbf{Y}$:

Program 11.6

```
doany i=1,n
    sum := sum + X[i] + Y[i]
```

Sparse code should enumerate those iterations i for which $\mathbf{X}[i] \neq 0$ or $\mathbf{Y}[i] \neq 0$:

Program 11.7

```
doany i=1,n
    if (X[i] ≠ 0 ∧ Y[i] ≠ 0) then
        sum := sum + X[i] + Y[i]
```

The query is (cf. (2.51) on page 44):

$$\sigma_{b_x \vee b_y} \bigominus_{i_x = i_y = i} \left(\ \{1 \leq i \leq n\}, \mathbf{X}\langle i_x, v_x, b_x \rangle, \mathbf{Y}\langle i_y, v_y, b_y \rangle \ \right) \tag{11.3}$$

In this model, the relations *logically* store all values – both zero and non-zero. The "presence bits" b_x and b_y indicate whether a particular element is *physically* stored in the data structure. The methods in the Black Boxes still access only those elements that are stored. But we can convert the enumerations over these elements into the enumerations over non-stored elements, as well. We can simplify away the loop bounds, as is done in Section 5.5.3:

$$\sigma_{b_x \vee b_y} \bigominus_{i_x = i_y} \left(\ \mathbf{X}\langle i_x, v_x, b_x \rangle, \mathbf{Y}\langle i_y, v_y, b_y \rangle \ \right) = \tag{11.4}$$

$$\sigma_{b_x \vee b_y} \Big(\mathbf{X}\langle i, v_x, b_x \rangle \bowtie \mathbf{Y}\langle i, v_y, b_y \rangle \Big) \tag{11.5}$$

The query in (11.5) is a join modified by the selection on the presence bits. Given the predicate in the selection, we can specialize join implementations: enumerate/test, hash and merge. For example, suppose the vectors are stored in the SPVO format (Section 4.5 on page 64). We can implement the join using the merge algorithm:

Program 11.8

```
X.declare_iter(h); Y.declare_iter(g)
X.open(h); Y.open(g)
while (X.valid(h) ∧ Y.valid(g)) do
    i_x := X.deref(h)
    i_y := Y.deref(g)
    if (i_x = i_y) then
        -- both bits are true --
        sum := sum + X.hderef(h).deref() + Y.hderef(g).deref()
    else if (i_x < i_y) then
```

```
        -- b_x is true --
        sum := sum + X.hderef(h).deref() + 0.0
    else
        -- b_y is true --
        sum := sum + 0.0 + Y.hderef(g).deref()
    endif
    -- now advance the pointers --
    if (i_x ≤ i_y) then
        X.next(h)
    endif
    if (i_y ≤ i_x) then
        Y.next(g)
    endif
enddo
```

Instead of having just one branch where the loop body gets executed, we have three branches – for all the combinations of the presence bits that satisfy the selection predicate in (11.5).

At this point we do not have enough experience with the tradeoffs between various join implementations when more than one combination of the presence bits is active. And just as is the case with compiling of DOACROSS loops the lack of structural information in the Black Box protocol can lead to inefficient code. Take a matrix **A** stored in the BSR format. How can the compiler convert the enumeration over the non-zeros only in the Program 11.9 below into the enumeration of some combination of zeros and non-zeros?

Program 11.9

```
doany br ∈ A
    doany bc ∈ A_br
        doany r ∈ A_br,bc
            doany c ∈ A_br,bc,r
```

In particular, the compiler has no way of knowing during which iteration of the loops over the blocks a particular zero element with the indices $\langle r_0, c_0 \rangle$ should appear. However, *we* know that if the block size is B, then the iteration is:

$$\Big\langle br = 1 + \lfloor (r_0 - 1)/B \rfloor, bc = 1 + \lfloor (c_0 - 1)/B \rfloor \Big\rangle \tag{11.6}$$

Insertions (fill)

Situation with fill is similar. Suppose we have the code:

Program 11.10

```
doany i=1,n
    X[i] := Y[i]
```

The relational model for this loop is:

Program 11.11

```
doany ⟨i_x, v_x, b_x, i_y, v_y, b_y⟩ ∈ σ_{b_x ∨ b_y} ( X⟨i, v_x, b_x⟩ ⋈ Y⟨i, v_y, b_y⟩ )
    if b_y then
        if ¬b_x then v_x = insert(X,i) endif
```

```
        vx := vy
    else if bx then
        delete(X,i)
    endif
enddo
```

Just as before, we can try to generalize join implementation to include insertions: Program 11.11 still tells us to *simultaneously enumerate* **X** and **Y** – the computation to be done for each tuple in the join is different, though. Suppose that **X** and **Y** were both stored in sorted order in bi-directional linked lists that support the `insert_before(h,i)` and `delete(h)` methods. The former inserts an element just before the current one and returns the l-value of the inserted element. The latter deletes the current element and advances the iterator. Then the merge join would be:

Program 11.12

```
X.declare_iter(h); Y.declare_iter(g)
X.open(h); Y.open(g)
while (X.valid(h) ∧ Y.valid(g)) do
    ix := X.deref(h)
    iy := Y.deref(g)
    if (ix = iy) then
        -- both bits are true --
        X.hderef(h).deref() := Y.hderef(g).deref()
        X.next(h); Y.next(g)
    else if (iy < ix) then
        -- by is true, insert --
        X.insert_before(h,iy) := vy
        Y.next(g)
    else
        -- bx is true, delete --
        X.delete(h)
    endif
enddo
```

Just as is the case with disjunctive queries, we have no experience with the relative merits of various implementations of joins that involve fill. It is also possible to convert the loops with fill into loops without fill by pre-allocating storage. In the above example, suppose that **X** has the method `assemble` that takes the set **S** of tuples in some default representation (say, a hash table) and allocates the storage for **X**. Then the above loop nest is transformed into:

Program 11.13

```
doany iy ∈ Y
    insert iy into S
enddo
X.allocate(S)
doany ⟨ix, vx, bx, iy, vy, by⟩ ∈ σ_{bx∧by} ( X⟨i, vx, bx⟩ ⋈ Y⟨i, vy, by⟩ )
    vx := vy
```

The second loop nest now does not involve insertions. Overall, the details of handling fill are still an open problem [58].

11.2.3 Parallel code generation

There are two limitations in our parallel code generation algorithm. First, the layout of messages in send and receive buffers (Figures 8.1 and 8.2) lacks structure necessary in some applications. Second, the default choice of sites that compute communication sets might not work well in some applications. We illustrate both points on an example. Consider the loop nest that computes the a single update in sparse Cholesky:

Program 11.14

```
doany i=1,n; j=1,n
    if (A(i,j) ∧ B(i,j)) then
        A[i,j] := A[i,j] + B[i,j]
```

The conditional in the loop ensures that only those elements of **A** that are already stored are updated. But there is more information coming from the application. Because the storage is pre-allocated, existence of an $\langle i, j\rangle$ entry in **B** implies the existence of an $\langle i, j\rangle$ entry in **A**:

$$\forall i, j : \langle i, j\rangle \in \mathbf{B} \Rightarrow \langle i, j\rangle \in \mathbf{A} \tag{11.7}$$

For the purposes of this example, we can think of **A** and **B** as being stored locally in the `ELEMENT` format (Section 4.5.4). In some parallel implementations [82] both **A** and **B** are partitioned by column, although they use different index translation relations δ_A and δ_B, which are replicated:

$$\mathbf{A}\langle i, j, a\rangle = \pi_{i,j,a} \bigcup_p \delta_A\langle j, p, j'\rangle \bowtie_{j'} \mathbf{A}^{(p)}\langle i, j', a\rangle \tag{11.8}$$

$$\mathbf{B}\langle i, j, b\rangle = \pi_{i,j,a} \bigcup_p \delta_B\langle j, p, j'\rangle \bowtie_{j'} \mathbf{B}^{(p)}\langle i, j', b\rangle \tag{11.9}$$

Computation is aligned with δ_A. That is the processor p executes those iterations $\langle i, j\rangle$ for which $\langle j, p, j'\rangle \in \delta_A$ for some local index j'. This means that the elements of **B** have to be communicated. We can informally define the set *Comm* which tells which columns j of **B** have to be sent from processor p to processor q:

$$\mathit{Comm}\langle q, p, j\rangle = \pi_{q,p,j} \left(\ \mathbf{A}\langle i, j, a\rangle \bowtie \delta_A\langle j, q, j'\rangle \bowtie \mathbf{B}\langle i, j, b\rangle \bowtie \delta_B\langle j, p, j''\rangle \ \right) \tag{11.10}$$

In "plain English" this says that the processor p has to send to the column j of **B** that it owns to processor q if the processor q owns the matching column of **A**.

Layout of communication buffers

Suppose a particular processor p has to send the set $C = \{j_1, j_2, \ldots, j_c\}$ of columns of **B** to the particular processor q. Since **B** is stored in the `ELEMENT` format, each column has the same set of non-zero elements with row indices $R = \{i_1, i_2, \ldots, i_r\}$. If we store the messages as described in Section 8.4.4 then the processors have to exchange $c \cdot r$ indices in the inspector. On the other hand, we really only need to exchange $c + r$ indices that define the set $C \times R$ of the elements of **B** being communicated. Moreover, the structure present in the storage of **B**, namely the availability of separate enumeration by rows as well as by columns, is destroyed once we "flatten" **B** into the communication buffers. Both of these considerations will adversely impact the quality of generated code when compared to hand-written library codes. To address this problem, parallel code generator has to inspect the Black Boxes for the relations being communicated and try to layout the messages in a similar way. Moreover, once the protocol for handling insertions is in place, we might be able to use the same Black Box for the message buffers as for the original relation, thus retaining the structure.

Sequencing of communication

One of the functions of the inspector is effectively to evaluate the *Comm* set in (11.10). The expression for the *Comm* set is, in fact, a distributed query, since it involves the global views **A** and **B**. When we evaluate such query we have to determine which processor should compute which part of the query. In the language of distributed databases, we have to choose the "sites" for the evaluation of the query [87]. The order of the computation of the $USED$ set and others in Section 8.4.4 is just one heuristic for doing this: we bring all the necessary information to the receiver processor q. In this application it means enumerating through the local fragment $\mathbf{A}^{(q)}$ and for each column j asking the processor p for the matching column of $\mathbf{B}$. This turns out to be rather inefficient. Usually, $\mathbf{A}$ is much larger that $\mathbf{B}$ and many column indices will be rejected by the owner of $\mathbf{B}$. Moreover, we have additional information that can help us make an intelligent decision: the implication (11.7). Using this implication, we can simplify the definition of the $Comm$ set:

$$Comm\langle q, p, j\rangle = \pi_{q,p,j}\left(\ \delta_A\langle j, q, j'\rangle \bowtie \mathbf{B}\langle i, j, b\rangle \bowtie \delta_B\langle j, p, j''\rangle\ \right) \tag{11.11}$$

In "plain English" this says that the owner of the column j of $\mathbf{B}$ can safely send it to the owner p of the index j, the matching column of $\mathbf{A}$ is guaranteed to exist.

In order to deal with this problem, we have to revisit the generation of communication sets as a distributed query evaluation problem, itself.

Part V

Appendices

Appendix A

Useful results from linear algebra

Throughout the thesis we use various linear algebraic "utilities" in order to manipulate systems of equalities and inequalities. While usually we only have to deal with simple constraints like $i = j$, occasionally we come across more general affine constraints. Therefore it is useful to have a set of well-defined primitives for dealing with systems of affine equalities and inequalities. In this chapter of the appendix we describe how the primitives used in the thesis are implemented.

We start with affine constraints:

$$F(\mathbf{x}, \mathbf{y}) = 0 \tag{A.1}$$

where F is an affine function and $\mathbf{x}$ and $\mathbf{y}$ are integer vectors. We say the constraints in (A.1) are *implicit*, since we can not compute the solutions directly. The primitives that involve equalities only are:

- Transform the system (A.1) into *explicit* form:

$$\mathbf{x} = f(\mathbf{t}), \quad \mathbf{y} = g(\mathbf{t}) \tag{A.2}$$

  where $\mathbf{t}$ is a vector of free parameters and f and g are affine functions.

- Transform the system (A.1) into *echelon* form:

$$\mathbf{x} = f(\mathbf{u}), \quad \mathbf{y} = h(\mathbf{u}) + g(\mathbf{t}) \tag{A.3}$$

  where the vector of parameters $\mathbf{u}$ has the smallest possible dimension. In "plain English" this echelon form gives us the solutions $\mathbf{x}$ to the system of constrains:

$$\exists \mathbf{y} : F(\mathbf{x}, \mathbf{y}) = 0 \tag{A.4}$$

  and provides a way to compute for every $\mathbf{x}$ that satisfies the (A.4) the set of $\mathbf{y}$'s that satisfy the system together with it.

- Decide the predicate:

$$\forall \mathbf{x} : \exists \mathbf{y} : F(\mathbf{x}, \mathbf{y}) = 0 \tag{A.5}$$

- Verify whether there is a functional dependence from $\mathbf{x}$ to $\mathbf{y}$ under the constraints (A.1). Recall, that there is a functional dependence if for a given value of $\mathbf{x}$ that satisfies (A.1) there is a unique companion value of $\mathbf{y}$.

We have also used one primitive that involves inequality constraints:

- Given affine equality and inequality constraints:

$$F(\mathbf{x}, \mathbf{y}) = 0 \wedge G(\mathbf{x}, \mathbf{y}) \leq 0 \tag{A.6}$$

  where F and G are affine functions, generate the loop nest over the solutions to the constraints with $\mathbf{x}$ being nested above $\mathbf{y}$:

  Program A.1
```
doany x ∈ some bounds
    doany y ∈ other bounds that depend on x
```

We now discuss how this primitives are implemented. Manipulations of equality constraints are based on (a) computing the explicit form of the solution of the constraints and (b) bringing this solution into echelon form.

A.1 Solution of integer linear systems

A system (A.1) of affine constraints can be equivalently written as a system of linear equations:

$$\mathbf{A}\mathbf{z} = \mathbf{b}, \quad \mathbf{z} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{A.7}$$

where $\mathbf{A}$ is an $m \times n$ integer matrix and $\mathbf{b}$ is an integer vector of length m. It is a well known fact [27, 75] that we can find the matrix $\mathbf{V}$ of unimodular column transformations that transforms the system into:

$$\begin{pmatrix} \mathbf{L} & \mathbf{0} \end{pmatrix} \mathbf{w} = \mathbf{b} \tag{A.8}$$

where:

$$\begin{pmatrix} \mathbf{L} & \mathbf{0} \end{pmatrix} = \mathbf{A}\mathbf{V}, \quad \mathbf{w} = \mathbf{V}^{-1}\mathbf{z} \tag{A.9}$$

Let r be the rank of the matrix $\mathbf{A}$. $\mathbf{L}$ is an $m \times r$ matrix in *weak column-echelon form*:

Definition A.1 *Let $(p_0 = 0, p_1, \ldots, p_m)$ be the profile (see Definition 5.4) of an $m \times r$ matrix $\mathbf{L}$. Let $\mu_k = \max_{0 \leq i \leq k} p_i$. The matrix $\mathbf{L}$ is said to be in* **weak echelon form** *if:*

$$\forall 1 \leq k \leq r : p_{k+1} \leq \mu_k + 1 \tag{A.10}$$

The difference between column-echelon form (Definition 5.5) and *weak* column-echelon form is that the latter can be computed with column operations only. The following matrix is in weak column-echelon form:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 0 & 0 \\ 3 & 4 & 5 \\ 6 & 0 & 0 \end{pmatrix} \tag{A.11}$$

However, a linear system $\mathbf{L}\mathbf{v} = \mathbf{c}$ can still be solved by forward substitution. The rank of the matrix in weak column-echelon form is given by the largest entry in its profile. In the rest of this Appendix we will refer to "weak column-echelon form" simply as "echelon form".

By construction, The matrix $\mathbf{L}$ in (A.9) has full column rank. The last $n - r$ columns of $\mathbf{V}$ form the basis for the null-space of $\mathbf{A}$. Let $\mathbf{V}_1$ be the first r columns of $\mathbf{V}$ and $\mathbf{V}_2$ be the rest $n - r$ columns. Let $\mathbf{w}_1$ be the first r elements of $\mathbf{w}$. Let $\mathbf{w}_2$ be the rest. Then the system (A.8) is equivalent to the following constraints:

$$\begin{aligned} &\mathbf{L}\mathbf{w}_1 = \mathbf{b} \\ &\mathbf{w}_1 \in \mathbb{Z}^r \\ &\mathbf{w}_2 \in \mathbb{Z}^{n-r} \end{aligned} \tag{A.12}$$

Observe that the vector $\mathbf{w}_2$ is unconstrained. Since $\mathbf{L}$ has full column rank, the solution to

$$\mathbf{L}\mathbf{w}_1 = \mathbf{b} \tag{A.13}$$

is either unique or does not exist at all. If there is no solution to (A.13), then there is no solution to the original system. The solution $\mathbf{w}_1^*$ to (A.13), if it exists, can be found by forward substitution. We now show how to transform $\mathbf{w}_1^*$ into the expression for all solution to (A.7). For any vector $\mathbf{w}_2 \in \mathbb{Z}^{n-r}$ the system system (A.8) is satisfied by all integer vectors $\mathbf{w}$ of the form:

$$\mathbf{w} = \begin{pmatrix} \mathbf{w}_1^* \\ \mathbf{w}_2 \end{pmatrix} \tag{A.14}$$

Using the relationship $\mathbf{z} = \mathbf{V}\mathbf{w}$, we obtain the expression for all solutions to $\mathbf{z}$ (if they exist at all):

$$\mathbf{z} = \mathbf{V}\mathbf{w} = \mathbf{V}_1\mathbf{w}_1^* + \mathbf{V}_2\mathbf{w}_2 \tag{A.15}$$

We can rewrite this as:

$$\mathbf{z} = \mathbf{z}^{(0)} + \mathbf{N}\mathbf{t} \tag{A.16}$$

where $\mathbf{z}^{(0)} = \mathbf{V}_1\mathbf{w}_1^*$ is the *particular solution* to the linear system (A.7) and $\mathbf{N} = \mathbf{V}_2$ is the basis for all solutions to the corresponding *homogeneous system* $\mathbf{A}\mathbf{w} = \mathbf{0}$. We can partition the individual equations in (A.16) into the ones that define $\mathbf{x}$ and $\mathbf{y}$:

$$\mathbf{x} = \mathbf{x}^{(0)} + \mathbf{N}_x\mathbf{t}, \quad \mathbf{y} = \mathbf{y}^{(0)} + \mathbf{N}_y\mathbf{t} \tag{A.17}$$

Thus we solve the problem of finding an explicit representation for $\mathbf{x}$ and $\mathbf{y}$.

A.2 Echelon form of the solution of a linear system

Other primitives on our list are defined by bringing the explicit solution (A.17) to the constraints into echelon form. Let

$$\mathbf{M} = \mathbf{N}\mathbf{U} \tag{A.18}$$

be the column echelon form of $\mathbf{N}$: the matrix $\mathbf{U}$ is unimodular and the matrix $\mathbf{M}$ is in (weak) echelon form. $\mathbf{M}$ has full column rank because $\mathbf{N}$ has full columns rank. Observe that $\mathbf{M}$ is also a basis for the null space of $\mathbf{A}$:

$$\mathbf{A}\mathbf{M} = \mathbf{A}\mathbf{N}\mathbf{U} = \mathbf{0}\mathbf{U} = \mathbf{0} \tag{A.19}$$

Therefore, we can rewrite (A.16) as:

$$\mathbf{z} = \mathbf{z}^{(0)} + \mathbf{M}\mathbf{u} \tag{A.20}$$

where $\mathbf{u}$ is the new vector of free parameters. Given the partition of $\mathbf{z}$ into $\mathbf{x}$ and $\mathbf{y}$ we can rewrite this as:

$$\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} = \mathbf{z}^{(0)} + \mathbf{M}\mathbf{u} = \mathbf{z}^{(0)} + \begin{pmatrix}\mathbf{M}_{11} & \mathbf{0}\\ \mathbf{M}_{21} & \mathbf{M}_{22}\end{pmatrix}\begin{pmatrix}\mathbf{u}_1\\ \mathbf{u}_2\end{pmatrix} = \begin{pmatrix}\mathbf{x}^{(0)} & + & \mathbf{M}_{11}\mathbf{u}_1 & & \\ \mathbf{y}^{(0)} & + & \mathbf{M}_{21}\mathbf{u}_1 & + & \mathbf{M}_{22}\mathbf{u}_2\end{pmatrix} \tag{A.21}$$

where the rows of matrix $\mathbf{M}$ are partitioned according to the partition of $\mathbf{z}$ and the columns of $\mathbf{M}$ are partitioned in such a way that $\mathbf{M}_{11}$ has full column rank. It immediately follows that $\mathbf{M}_{22}$ has full column rank, as well.

We interested in (A.21) because it gives us a recipe for computing the second part of the solution $\mathbf{y}$ given the value of the first part of the solution $\mathbf{x}$:

```
find u1 that satisfies x = x(0) + M11 u1
if (found the solution u1*) then
    y := y(0) + M21 u1 + M22 u2 (for any u2)
else
    no solution
endif
```

The value of $\mathbf{u}_1$ can be found by forward substitution, because $\mathbf{M}_{11}$ is in echelon form. This also allows us to constructively decide the predicate (A.5). The predicate is true if and only if $\mathbf{M}_{11}$ is unimodular: only then can be find an integer value of $\mathbf{u}_1$ for any integer value of $\mathbf{x}_1$. The matrix $\mathbf{M}_{11}$ is unimodular if (a) it is square and (b) its determinant is ± 1. In our case the matrix is also in echelon form. A square, non-singular matrix in echelon form is simply lower-triangular. A lower-triangular matrix has determinant ± 1 if and only if its diagonal entries are ± 1. Overall, we have to test whether (a) $\mathbf{M}_{11}$ is square and (b) has ± 1 on the diagonal.

The above results can be easily extended to partitions of $\mathbf{x}$ into more than 2 sub-vectors. Let

$$\mathbf{x} = \begin{pmatrix}\mathbf{x}_1\\ \mathbf{x}_2\\ \vdots\\ \mathbf{x}_k\end{pmatrix} \tag{A.22}$$

be the partition of $\mathbf{x}$. Then we can express the sub-vectors in echelon form as:

$$\begin{aligned} \mathbf{x}_1 &= f_1(\mathbf{u}_1)\\ \mathbf{x}_2 &= f_2(\mathbf{u}_1, \mathbf{u}_2)\\ &\dots\\ \mathbf{x}_k &= f_k(\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k) \end{aligned} \tag{A.23}$$

where f_i are affine functions of the arguments. Given the values of $\mathbf{x}_1$ through $\mathbf{x}_{i-1}$, we can compute the value of $\mathbf{x}_i$ by, first, solving for the values of $\mathbf{u}_1$ through $\mathbf{u}_{i-1}$ and, second, picking any value for $\mathbf{u}_i$.

Deciding if there is a functional dependence from $\mathbf{x}$ to $\mathbf{y}$ is trivial, once we have the explicit solution in echelon form (A.21). If $\mathbf{M}_{22}$ is non-zero (and the vector $\mathbf{u}_2$ is non-empty) then $\mathbf{y}$ has additional degree of freedom. On the other hand, if $\mathbf{M}_{22}$ is zero (and has no columns, since it must be of full column rank), then both $\mathbf{x}$ and $\mathbf{y}$ are uniquely determined by the value of $\mathbf{u}_1 = \mathbf{u}$, and there is a unique value of $\mathbf{y}$ for every feasible value of $\mathbf{x}$.

A.3 Block elimination

Suppose we have a system (A.6) of integer linear equalities and inequalities in terms of two integer vectors $\mathbf{x}$ and $\mathbf{y}$. We would like to produce a loop nest which would in the outer loops enumerate *all* the solutions

$\mathbf{x}$ and then in the inner loops enumerate all the solutions $\mathbf{y}$ for the given $\mathbf{x}$, as in Program A.1.

We start by expressing the solution to the equalities in the echelon form by using the results of the previous section:

$$\begin{array}{rcl} \mathbf{x} & = & f_x(\mathbf{u}) \\ \mathbf{y} & = & f_y(\mathbf{u}, \mathbf{w}) \end{array} \tag{A.24}$$

Second, we substitute these expression for $\mathbf{x}$ and $\mathbf{y}$ into the inequalities in (A.6):

$$G(f_x(\mathbf{u}), f_y(\mathbf{u}, \mathbf{w})) = H(\mathbf{u}, \mathbf{w}) \leq 0 \tag{A.25}$$

Then we can use projection techniques such as Fourier-Motzkin elimination [29, 34, 70] to find the bounds on $\mathbf{u}$:

$$H_1(\mathbf{u}) \leq 0 \tag{A.26}$$

and we can rearrange the inequalities in H to form the bounds on $\mathbf{w}$ in terms of the $\mathbf{u}$:

$$H_2(\mathbf{w}) \leq H_3(\mathbf{w}) \tag{A.27}$$

(See [70] for details.) The constraints on $\mathbf{x}$ are:

$$\{\mathbf{x} \mid \exists \mathbf{u} : H_1(\mathbf{u}) \leq 0 \land \mathbf{x} = f_x(\mathbf{u})\} \tag{A.28}$$

The constraints on $\mathbf{y}$ are parametrized by $\mathbf{u}$:

$$\{\mathbf{y} \mid \exists \mathbf{w} : H_2(\mathbf{w}) \leq H_3(\mathbf{u}) \land \mathbf{y} = f_y(\mathbf{u}, \mathbf{w})\} \tag{A.29}$$

We can use (A.28) and (A.29) in order to produce the desired loop nest:

```
doany u such that H1(u) ≤ 0
    x := fx(u)
    doany w such that H2(w) ≤ H3(u)
        y := fy(u,w)
```

Moreover, suppose we have a fixed value of $\mathbf{x}$ and want to enumerate all matching $\mathbf{y}$'s. The following code fragment does the trick:

```
find u such that fx(u)
if found then
    doany w such that H2(w) ≤ H3(u)
        y := fy(u,w)
```

Appendix B

Relational algebra notation

In this chapter of the Appendix we formalize the relational algebra notation used in the thesis. Relational queries have three components:

$$Q_{\mathbf{s}}\langle\mathbf{a}\rangle \tag{B.1}$$

Q is the relational expression, $\mathbf{s}$ are *parameters* and $\mathbf{a}$ is the schema of the output of the query. The set of names of the parameters and the schema must be disjoint. We now define the meaning of the three by the induction on the structure of the expression:

- A single relation $R_{\mathbf{s}}\langle\mathbf{a}\rangle$ is evaluated by taking each tuple of the relation $\mathbf{R}_{\mathbf{s}}$ parametrized by $\mathbf{s}$, and renaming the attributes from the schema defined by the Black Box into the names in $\mathbf{a}$. For example, if a matrix $\mathbf{A}$ is stored in CSR format (Section 6.1.1), then for a given row index i we get the relation $\mathbf{A}_i$ for the elements of the row. The schema of this relation, as defined by the `CSR` Black Box is $\langle j, v\rangle$. In the expression $\mathbf{A}_i\langle j', v'\rangle$ the field j becomes the field j' in the output and the field v becomes the field v'.
- The cross product of two relations $R_{\mathbf{s}}\langle\mathbf{a}\rangle \times S_{\mathbf{s}}\langle\mathbf{b}\rangle$ is taken by concatenating all possible pairs of tuples that result from evaluating the inputs $R_{\mathbf{s}}\langle\mathbf{a}\rangle$ and $S_{\mathbf{s}}\langle\mathbf{b}\rangle$. It is assumed that the schema $\mathbf{a}$ and $\mathbf{b}$ and disjoint. Otherwise, the result is undefined. The schema of the result of the cross-product is the union of the schemata of the inputs.
- The selection $\sigma_{P(\mathbf{s},\mathbf{a})} R_{\mathbf{s}}\langle\mathbf{a}\rangle$ is computed by evaluating the predicate P for each tuple of the input. The variables that appear in the predicate are bound by the schema of the input and the parameters.
- The Θ-join is evaluated according to its definition as a cross product followed by a selection:

$$\bigominus_{P(\mathbf{s},\mathbf{a},\mathbf{b},\ldots)} \left(\; \mathbf{R}_{\mathbf{s}}\langle\mathbf{a}\rangle, S_{\mathbf{s}}\langle\mathbf{b}\rangle, \ldots \; \right) = \sigma_{P(\mathbf{s},\mathbf{a},\mathbf{b},\ldots)} \left(\; \mathbf{R}_{\mathbf{s}}\langle\mathbf{a}\rangle \times S_{\mathbf{s}}\langle\mathbf{b}\rangle \times \ldots \; \right) \tag{B.2}$$

  In particular, the variables in the predicate P are bound by the parameters $\mathbf{s}$ and the schema of the input relations. The schema of the input relations are assumed to be disjoint.
- The natural join $R_{\mathbf{s}}\langle\mathbf{a},\mathbf{c}\rangle \bowtie_{\mathbf{c}} S_{\mathbf{s}}\langle\mathbf{b},\mathbf{c}\rangle$ is evaluated by matching the tuples from the inputs on the common attributes. The subscript in $\bowtie_{\mathbf{c}}$ is purely illustrative and can be omitted. The schema of the output is the union of the schema of the inputs.
- The *parametrized union*:

$$\bigcup_{\mathbf{a}\in R_{\mathbf{s}}} S_{\mathbf{s},\mathbf{a}}\langle\mathbf{b}\rangle \tag{B.3}$$

  is evaluated by, for each tuple $\mathbf{a}$ in the result of the query R, evaluating the query S with the tuple $\mathbf{a}$ added to the set of parameters and taking the union of the results:

```
for a ∈ R_s
    for b ∈ S_{s,a}
        output b
```

The schema of the output is **b**.

We often write definitions of relations as:

$$R\langle \mathbf{a} \rangle = Q\langle \mathbf{a} \rangle \tag{B.4}$$

The schema notation on the left-hand side is purely illustrative and can be omitted:

$$R = Q\langle \mathbf{a} \rangle \tag{B.5}$$

because the schema of R is determined by the schema of the query Q. Similarly, we write loops as:

`doany` $\langle \mathbf{a} \rangle \in Q\langle \mathbf{a} \rangle$

$\quad \ldots \mathbf{a} \ldots$

Again the schema notation on the left-hand side of the inclusion sign is purely illustrative. The loop nest can be written as:

`doany` $Q\langle \mathbf{a} \rangle$

$\quad \ldots \mathbf{a} \ldots$

because the schema of the query determines the names of the fields in the output.

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Saman P. Amarasinghe and Monica Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the PLDI '93*, June 1993.

[3] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the 4th Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. This paper has subsequently been revised.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* Society for Industrial and Applied Mathematics, 1994.

[5] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of PLDI'93*, June 1993.

[6] Satish Balay, William Groopp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 – Revision 2.0.15, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[7] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA., 1993.

[8] 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.

[9] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, University of Leiden, Leiden, The Netherlands, May 1996.

[10] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In *The Seventh International Workshop on Languages and Compilers for Parallel Computing*, pages 406–422, Ithaca, NY, August 8–10, 1994.

[11] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Supercomputing '93*, pages 430–441, Portland, OR, November 15–19, 1993.

[12] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993.

[13] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 12–14, 1993.

[14] Aart J.C. Bik and Harry A.G. Wijshoff. A sparse compiler. Technical Report 93–04, Dept. of Computer Science, Leiden University, 1993.

[15] Aart J.C. Bik and Harry A.G. Wijshoff. Non-zero structure analysis. In *International Conference on Supercomputing*, pages 226 – 235, 1994.

[16] 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.

[17] Aart J.C. Bik and Harry A.G. Wijshoff. Annotations for a sparse compiler. In *The Eight International Workshop on Languages and Compilers for Parallel Computing*, pages 500–514, Columbus, OH, August 10–12, 1995.

[18] 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.

[19] Aart J.C. Bik and Harry A.G. Wijshoff. Simple quantitative experiments with a sparse compiler. In *Parallel algorithms for irregularly structured problems: third international workshop, IRREGULAR '96*, pages 249–262, Santa Barbara, CA, August 19–21 1996.

[20] Aart J.C. Bik and Harry A.G. Wijshoff. The use of iteration sparse partitioning to construct representative simple sections. *Journal of Parallel and Distributed Computing*, 34:95 – 110, 1996.

[21] C. Bischof, A. Carle, P. Khademi, and G. Pusch. Automatic differentiation: Obtaining fast and reliable derivatives–fast. In *Proceedings of the SIAM Symposium on Control Problems in Industry*, July 1994. `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P484.ps.Z`.

[22] 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.

[23] Peter Brinkhaus, Aart J.C. Bik, and Harry A.G. Wijshoff. Subroutine on demand-service, sparse blas 2 & 3. `http://hp137a.wi.leidenuniv.nl:8080/blas-service/blas.html`. Accessed January 28, 1997.

[24] Sandra Carney, Michael A. Heroux, Guangye Li, and Kesheng Wu. A revised proposal for a sparse BLAS toolkit. SPARKER Working Note #3, June 1994. `http://www.cerfacs.fr/~douglas/mgnet/papers/SparKer/sparker3.ps.gz`.

[25] D. D. Chamberlin et al. A history and evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981.

[26] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[27] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1993.

[28] IBM Corporation. *Engineering and Scientific Subroutine Library*. `http://www.austin.ibm.com/software/Apps/essl.html`.

[29] George B. Dantzig and Curtis Eaves B. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[30] Raja Das, Joel Saltz, and Reinhard van Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 152–168, Portland, Oregon, August 12–14, 1993. Also as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[31] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSD-95-883, Computer Science Division, University of California, Berkeley, 1995.

[32] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[33] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):18–32, March 1988.

[34] Paul Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[35] Paul Feautrier. Some efficient solutions to the affine scheduling problem, i, one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[36] Paul Feautrier. Some efficient solutions to the affine scheduling problem, ii, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[37] Paul Feautrier. Fine-grain scheduling under resource constraints. In *The Seventh International Workshop on Languages and Compilers for Parallel Computing*, number 892 in LNCS, Ithaca, NY, August 8–10, 1994. Springer-Verlag.

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

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

[40] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[41] Bruce Hendrickson and Rober Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[42] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[43] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C Karathanasis, P.N. Papachiou, M.K. Samartizis, E.A. Vavalis, Ko Yamg Wang, and S. Weerawarana. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 96–107, Amsterdam, The Netherlands, June 11–15, 1990.

[44] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro. Aztec user's guide: Version 1.1. Technical Report SAND95-1559, Sandia National Laboratories, October 1995. `http://www.cs.sandia.gov/CRF/Aztec_pubs.html`.

[45] Claes Johnson. *Numerical solution of partial differential equations by the finite element method.* Cambridge University Press, 1987.

[46] Mark T. Jones and Paul E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and Its Applications*, pages 229–245. Springer-Verlag, 1993. Also MSC Preprint P314-0692.

[47] 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.

[48] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the 9th ACM symposium on discrete algorithms*, 1998.

[49] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 16–18, 1997.

[50] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1993.

[51] Argonne National Laboratory. MPI standard page. `http://www.mcs.anl.gov/mpi`.

[52] L. Lamport. The parallel execution of DO loops. *Communications of ACM*, 17(2):83–93, February 1974.

[53] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[54] P. E. Lewis and J. P. Ward. *The Finite Element Method: Principles and Applications*. Addison-Wesley, Reading, Massachusetts, 1991.

[55] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for numa compilers. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–295, Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 12–15, 1992.

[56] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[57] Joseph W. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*, 34:82–109, 1992.

[58] Nikolay Mateev. *Compilation of sparse matrix codes*. PhD thesis, Cornell University, Department of Computer Science, to appear.

[59] The MathWorks Inc. *MATLAB Reference Guide*, 1992.

[60] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 2–15, Charleston, SC, January 10-13, 1993.

[61] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996.

[62] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice and Experience*, 24(7):623–642, July 1994.

[63] Gordon Novak. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[64] Gordon Novak. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering*, 23(7):1–17, 1997.

[65] Stacy Pendell. Sp configuration. `http://www.tc.cornell.edu/UserDoc/SP/config.html`, September 9, 1996. Accessed June 22, 1997.

[66] Sergio Pissantezky. *Sparse Matrix Technology*. Academic Press, London, 1984.

[67] 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.

[68] Roldan Pozo and Karin A. Remington. `http://math.nist.gov/spblas`.

[69] Willam Pugh and David Wonnacott. Going beyond integer programming with the omega test to eliminate false dependencies. Technical Report CS-TR-2993, Department of Computer Science, Universoty of Maryland, College Park, December 1992.

[70] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[71] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, LNCS #768, pages 546–566, Portland, Oregon, August 12–14, 1993. Springer-Verlag.

[72] Mukund Raghavachari and Anne Rogers. A study of the effects of ordering, paritioning and factorization algorithms on distributed sparse cholesky factorization. Technical Report CS-TR-505-96, Department of Computer Science, Princeton University, 1996.

[73] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, New York, NY, beta edition, 1996.

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

[75] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, England, 1986.

[76] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[77] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[78] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[79] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, Department of Computer Science, July 1997. Available as a technical report TR97-1635.

[80] Michael Stonebraker, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 37–53. Morgan Kaufmann Publishers, Inc., 2nd edition, 1994.

[81] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

[82] Chuanguang Sun. Efficient parallel solutions of large spare spd systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, August 1992.

[83] Chunguang Sun. Parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Technical Report 95-212, Cornell University Theory Center, 1995.

[84] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Eigth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997. SIAM.

[85] R. S. Tuminaro, J. N. Shadid, and S. A. Hutchinson. Parallel sparse matrix vector multiply software for matrices with data locality. *Concurrency: Practice and Experience*, 1997. `http://www.cs.sandia.gov/CRF/Aztec_pubs.html`.

[86] 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. Available at http://www.vcpc.univie.ac.at/activities/language.

[87] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, MD, 1988.

[88] 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.

[89] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 26–28, 1991.

[90] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

[91] Michael Wolfe. *High Performance Compilers.* Addison-Wesley, Redwood City, CA, 1996.

[92] 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.

[93] M. Yannakakis. Computing the minimum fill is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 1981.

[94] Dan Zigmond. Generic programming and the c++ stl. *Dr. Dobb's Journal*, August 1995.