The Sparse Compiler MT1: A Reference Guide

Aart J.C. Bik * Peter J.H. Brinkhaus Harry A.G. Wijshoff

Contents

1 Introduction 1

2 Motivation 1
- 2.1 Sparse Matrix Computations 1
- 2.2 Compiler Support 2

3 The Sparse Compiler MT1 3
- 3.1 Terminology 3
- 3.2 Organization of MT1 3
- 3.3 Available Commands 3

4 Annotation Mechanism 4
- 4.1 Declaration Annotations 4
- 4.2 File Annotations 4
- 4.3 Nonzero Structure Annotations 5

5 Phases of MT1 5
- 5.1 Annotation Analysis 6
  - 5.1.1 File Annotations 6
  - 5.1.2 Nonzero Structure Annotations 6
- 5.2 Procedure Cloning 7
- 5.3 Program Analysis 8
- 5.4 Transformation Application 9
- 5.5 Data Structure Selection 10
  - 5.5.1 Reshaping Access Patterns 10
  - 5.5.2 Iteration Space Partitioning 11
- 5.6 Sparse Code Generation 12
  - 5.6.1 Initialization Code Generation 12
  - 5.6.2 Actual Sparse Code Generation 13

6 Final Remarks 14

1 Introduction

Initiated as a small research tool in 1992/1993 [6, 21], MT1 has been extended to a full FORTRAN 77 restructuring compiler at the Leiden University during the time period 1993-1996. In contrast with early restructuring compilers, having a fixed set of internal program transformations that are inaccessible for programmers, MT1 enables the programmer to define a set of *program transformations* by means of a simple transformation language.

*Contact Information: High Performance Computing Division, Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands, `ajcbik@cs.leidenuniv.nl`.

The defined transformations may be applied interactively, so that explicit control over the order in which transformations are applied can be achieved. Although, in principle, machine code could be generated directly after all program transformations have been applied, MT1 performs a **source-to-source translation**. Hence, the output of MT1 must be used as input for another FORTRAN compiler for the desired target architecture to obtain executable code.

Although initially MT1 was intended to gain more insight in how the phase-ordering problem should be addressed, MT1 has also been used to prototype a novel approach to the development of sparse codes that has been proposed by Bik and Wijshoff in 1993 [10]. Rather than dealing with the sparsity of matrices at the programming level, as is done traditionally, the sparsity should be dealt with at the compilation level by a special kind of restructuring compiler, referred to as a **sparse compiler**. Elaboration of these ideas have resulted in the development of an *automatic data structure selection and transformation method* [7] and [12, 15]. The method has been incorporated into MT1 to actually test the feasibility of automatically generating sparse codes. Hence, although MT1 can be used as a conventional restructuring compiler, MT1 can also be used as a sparse compiler to automatically convert a dense program into semantically equivalent sparse code.

This document serves as a reference guide for programmers that want to use MT1 as a sparse compiler. A detailed presentation of the other features of MT1 is given in the MT1 Reference Manual [22]. A thorough presentation of the automatic data structure selection and transformation method can be found in [7].

2 Motivation

Although this document is not intended to be a tutorial on sparse matrix computations, a brief motivation for using a sparse compiler to generate sparse codes is given.

2.1 Sparse Matrix Computations

If many elements in a matrix are zero, then this matrix is called a **sparse matrix**. In contrast, a matrix containing many nonzero elements is referred to as a **dense matrix**. Both the *storage requirements* and *computational time* of an application that operates on sparse matrices can be reduced substantially in comparison with an application that

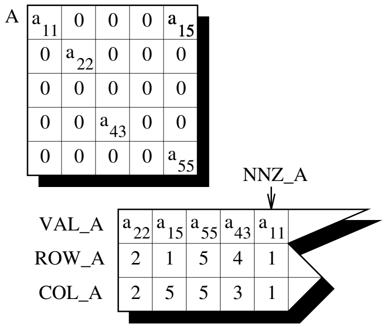


Figure 1: Dense vs. Sparse Storage

operates on dense matrices by only storing nonzero elements and avoiding redundant operations on zero elements (see e.g. [28, 29, 30, 31, 32, 34]).

EXAMPLE: Below, two FORTRAN fragments performing the operation $\vec{b} \leftarrow \vec{b} + A\vec{x}$ are given. In the dense fragment, a two-dimensional array `A` is used to store all elements of the matrix A, whereas a more complex sparse storage scheme (data structure) is used in the sparse fragment to avoid redundant operations on zero elements:

Dense Fragment:

```
REAL A(M,N)
...
DO I = 1, M
  DO J = 1, N
    B(I) = B(I) + A(I,J)*X(J)
  ENDDO
ENDDO
```

Sparse Fragment:

```
REAL    VAL_A(SZ)
INTEGER ROW_A(SZ), COL_A(SZ)
INTEGER NNZ_A
...
DO IJ = 1, NNZ_A
  I    = ROW_A(IJ)
  J    = COL_A(IJ)
  B(I) = B(I) + VAL_A(IJ)*X(J)
ENDDO
```

Possible contents of these storage schemes are illustrated in figure 1. For this example, 25 elements are stored and operated upon in the dense fragment, whereas only 5 nonzero elements are stored and operated upon in the sparse fragment. However, some additional storage, referred to as overhead storage, is required in the sparse storage scheme to reconstruct the underlying matrix. The row and column index of each nonzero element of A are stored as well, while an additional scalar records the total number of elements that are actually stored in the arrays (because some additional space may be present to allow for the insertion of more nonzero elements). Nevertheless, the total storage requirements are reduced with respect to dense storage of A (viz. 16 vs. 25 memory cells). For larger sparse matrices, more extensive savings can be expected.

Note that no substantial savings in computational time arise from protecting the loop-body of the dense fragment with the test '`A(I,J).NE.0.0`', because this test would still be executed `M`·`N` times. In contrast, the loop-body of the sparse fragment is only executed `NNZ_A` times. Keeping the storage requirements as well as the amount of work truly proportional to the number of nonzero elements in a sparse matrix is one of the most important objectives in sparse matrix computations [25][30, ch2][31, ch2][32, p1-3][34].

As already illustrated by this small example, however, achieving this objective may be a complex and cumbersome task for the programmer. The use of complicated sparse storage schemes usually obscures the actual functionality of the code, making both the development and maintenance of sparse codes a non-trivial task. Furthermore, the occurrence of subscripted subscripts induced by sparse storage schemes (cf. the use of `ROW_A` and `COL_A` in the example) usually disables most compiler optimizations because the compiler must make very conservative estimates about the actual data dependences that may occur at run-time. Despite all these problems, however, only limited compiler support for sparse matrix computations has been developed in the past. The sparse compiler, however, provides an alternative approach to develop sparse codes.

2.2 Compiler Support

Because restructuring compilers are very useful to automatically detect and exploit implicit parallelism in serial software, the question arises whether it is also possible to let a restructuring compiler convert code that operates on simple data structures into a format that exploits certain characteristics of the data operated on. In contrast to conventional restructuring compilers, mainly focusing on program transformations, this approach must allow for the application of *data structure transformations* as well. In particular, a restructuring compiler that can automatically convert a dense program into a sparse program that operates on more complex sparse data structure to reduce both the storage requirements as well as the computational time of the original program is called a **sparse compiler**. The automatic data structure selection and transformation method of [7] and [12, 15] that can be used for this purpose has been incorporated into MT1 to test the feasibility of using a sparse compiler to generate sparse codes.

Besides the fact that dealing with sparsity of matrices at the compilation level rather than at the programming is less error-prone, this approach has a number of other advantages. First, the complexity of writing and maintaining sparse codes is reduced substantially, which enables programmers who are not familiar with sparse matrix computations to easily produce sparse code. Second, applying data dependence analysis to the dense code usually yields more accurate information, which allows for more program transformations. Moreover, because the sparse compiler can account for characteristics of both the nonzero structure and a particular target machine (provided that these characteristics are made available in some manner), as well as the actual operations performed, one dense program can be converted into a range of sparse versions, each of which is tailored to a particular instance of the same problem. Finally, just as traditional restructuring compilers enable the re-use of existing serial software, a sparse compiler enables the re-use of (parts of) existing dense code.

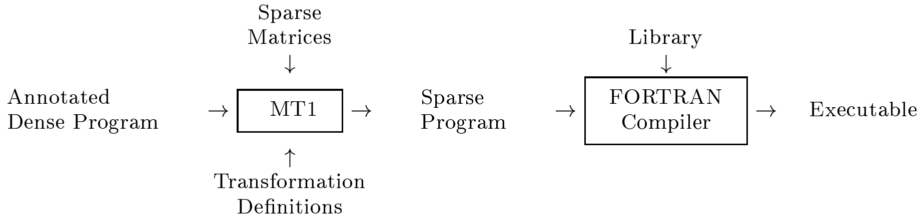


Figure 2: Using MT1 as a Sparse Compiler

3 The Sparse Compiler MT1

In this section, some terminology and the organization of the sparse compiler MT1 are discussed.

3.1 Terminology

For each matrix, the sparsity of which is not explicitly dealt with in the original dense program, the following three concepts can be distinguished:

- An $m \times n$ **implicitly sparse matrix** A.
- A two-dimensional array `REAL A(M,N)`, used as **enveloping data structure** of A.
- A **sparse storage scheme** for A, selected by the sparse compiler.

The concept of an implicitly sparse matrix is introduced to reason about programs at a logical level (perform the operation $\vec{b} = A\vec{x}$, consider the nonzero structure of A, etc.). At the programming level, all operations on an implicitly sparse matrix are defined on the enveloping data structure, for which a two-dimensional array of appropriate size is used. To emphasize the correspondence between an implicitly sparse matrix and its enveloping data structure, identical names will be used for both (i.e. two-dimensional arrays `A`, `B`, `C`, etc., denote the enveloping data structures of implicitly sparse matrices A, B, C, etc.).

In fact, an implicitly sparse matrix is an ordinary sparse matrix for which a simple storage scheme is used to reduce the complexity of developing and maintaining the original dense program. The burden of sparse code generation is placed on the sparse compiler, which selects a suitable sparse storage scheme for each implicitly sparse matrix and transforms all occurrences of the corresponding enveloping data structure in the original dense program accordingly.

3.2 Organization of MT1

As depicted in figure 2, the input of MT1 consists of a dense FORTRAN in which a few annotations are used to supply sparsity related information to the compiler. Moreover, the input may also consist a set of programmer-defined program transformations, and some sparse matrices that are available on file at compile-time.

These matrices on file are automatically analyzed by the sparse compiler to determine characteristics of the nonzero structures. Subsequently, the sparse compiler selects a suitable sparse storage scheme for each implicitly sparse matrix. This selection is based on analysis of annotations, nonzero structure analysis of sparse matrices available on file, and analysis of the input program. While selecting a data structure, MT1 may apply conventional program transformations if this selection cannot be resolved efficiently.

Finally, a target program that is semantically equivalent to the input program but that operates on the selected storage schemes is generated by means of program and data structure transformations. This generated sparse program together with a small run-time library can be used as input for a conventional FORTRAN compiler for the desired target architecture to obtain executable sparse code.

3.3 Available Commands

Commands are simply given at the '`=>`' prompt of MT1. The following commands are relevant with respect to using MT1 as a sparse compiler:

- **auto/start**: Starts applying the programmer-defined program transformations automatically or with user interaction, respectively.
- **bounds**: Toggles between an expensive simplification method for loop bounds or a less expensive and less precise simplification method.
- **edit/readprg/readtrf** `<filename>`: Either edits or reads a program/transformation file.
- **help**: Provides a listing of all MT1 commands.
- **noverbose/verbose**: Controls the amount of messages that are prompted to the programmer.
- **query**: Toggles between user interaction or supplying (conservative) default answers to all queries.
- **save** `<filename>`: Saves the current FORTRAN program in a readable form to the given file.
- **showprg/showsym**: Prompts the current program or symbol-table to the programmer.

- **smin**: Toggles between enabling or disabling prompting the ' sparse matrix information node' to the programmer after automatic nonzero structure analysis.

- **sparse**: Instructs the sparse module of MT1 to convert the current dense program into sparse code.

A detailed description of all the commands that are available in MT1 is given in the MT1 Reference Manual [22].

4 Annotation Mechanism

Information that cannot be expressed in the original dense program is supplied to the sparse compiler by means of annotations. All annotations have the form of comments, which enables direct compilation and testing of the original dense program with other FORTRAN compilers (provided that the implicitly sparse matrices are not too large).

4.1 Declaration Annotations

Because MT1 cannot distinguish between ordinary arrays and arrays that are used as enveloping data structures of implicitly sparse matrices, a mechanism to provide the compiler with this kind of information must be available. The identification of enveloping data structures is done by means of declaration annotations.

A declaration annotation starts at the beginning of a line with '`C_SPARSE` followed the specification of the identifier of an enveloping data structure using '`ARRAY( id )`' within parentheses. The following declaration annotation, for instance, informs the compiler about the fact that the array `A` is used as enveloping data structure of an implicitly sparse matrices A of size 100×100:[1]

```
      REAL A(100,100)
C_SPARSE( ARRAY(A) )
```

Each annotation that identifies an enveloping data structure must follow the actual declaration of the array. Such annotations are only allowed before the first executable statement *in the main program* (although sparsity information can be propagated automatically to subroutines and functions). Moreover, each enveloping data structure must be a two-dimensional array with basis type `INTEGER`, `REAL`, `DOUBLE PRECISION`, or `COMPLEX`, and index set $[1..\mathtt{M}] \times [1..\mathtt{N}]$ for suitable constants `M` and `N`. If any of these constraints is violated, an appropriate error message is generated, and the program is not accepted as input for the sparse compiler.

EXAMPLE: In the following program, the annotations involving arrays `A` and `B` in the main program, and the local array `G` of subroutine `PROC` violate these constraints. Hence, although the annotation involving `C` is correct, the program is rejected as input for the sparse compiler:

[1]The syntax of annotations has been slightly modified with respect to the syntax given in [7].

```
      PROGRAM ANNOT
      REAL A(-5:5,10), B(10)
      REAL C(10,10), D(100)
C_SPARSE( ARRAY(A) )
C_SPARSE( ARRAY(B) )
C_SPARSE( ARRAY(C) )
      CALL PROC(C)
      CALL PROC(D)
      END
```

```
      SUBROUTINE PROC(F)
      REAL F(10,10)
      REAL G(10,10)
C_SPARSE( ARRAY(G) )
      ...
      RETURN
      END
```

Within each declaration annotation, more information about the corresponding implicitly sparse matrix may be supplied in a comma-separated list, as further elaborated upon in the following sections.

4.2 File Annotations

The file in which an implicitly sparse matrix resides can be specified in a file annotation as '`FILE( filename )`'. Obviously, only one file may be specified for each implicitly sparse matrix.

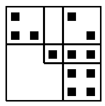
If this file is already available at compile-time in the directory defined by the environment variable `SPARSEDIR`, then the file is examined by the automatic nonzero structure analyzer. Because this analyzer expects all matrices in coordinate-scheme, sparse matrices generated in a program or stored in alternative storage schemes must first be converted into coordinate-scheme in order to enable this automatic analysis. On file, the coordinate-scheme consists of three integers, indicating the number of rows, columns, and entries in the matrix, respectively, followed by the row and column index and numerical value of each nonzero element in arbitrary order. If the size of the stored matrix does not match the declaration of the enveloping data structure, the program is rejected.

EXAMPLE: The following annotation indicates that the implicitly sparse matrix A having array `A` as enveloping data structure can be found in the file '`mat1.cs`':

```
      REAL A(5,5)
C_SPARSE( ARRAY(A),  FILE('mat1.cs') )
```

Below, an example file and the corresponding nonzero structure of the matrix A are shown, annotated with the block form detected by the analyzer:

contents of file 'mat1.cs'
5 5 12
1 1 5.0 2 2 5.0
3 3 5.0 4 4 5.0
5 5 5.0 2 1 1.0
3 4 1.0 1 4 1.0
2 5 1.0 5 4 1.0
4 5 1.0 3 5 1.0



Nonzero Structure of A

Because the programmer is unaware of the sparse storage scheme that will be selected by the sparse compiler, the compiler is responsible for the generation of appropriate initialization code for each selected data structure. Independent of whether a file specified in a file annotation is available and analyzed at compile-time, or not, this file will be used in the automatically generated initialization code of the selected sparse storage scheme. This code is generated before the first executable statement in the main program, and, in contrast with automatic analysis of the nonzero structure, will only read the file at run-time.

4.3 Nonzero Structure Annotations

Because it is unlikely that, at compile time, all implicitly sparse matrices are available on file, nonzero structure information can also be supplied directly to the compiler by means of annotations.

An approximation of the density of an implicitly sparse matrix can be supplied as '`DENSITY( expr )`', where symbol denotes an arbitrary FORTRAN expression. The approximated density is only recorded if the expression can be evaluated at compile-time and has a real value in the range $(0, 1]$. Obviously, only one approximated density may be supplied for an implicitly sparse matrix.

If a particular region in an implicitly sparse matrix is either sparse, dense (or will become so at run-time), or completely zero (and will remain so at run-time), then this information can be supplied to the compiler by specifying this property followed by a description of the index set of the region. Moreover, a preferred access direction for this region can optionally be supplied.

The property of each region is simply defined as '`SPARSE`', '`DENSE`', and '`ZERO`'. The index set of a region is described using a, possibly empty, comma-separated list of boundary pairs, defining lower and upper bounds for any of the expressions '`I`', '`J`', '`I+J`', and '`I-J`'.[2] Symbolic constants may be used in the definition of each bound to increase the flexibility of the program, provided that all expressions can be evaluated at compile-time. For an $m \times n$ implicitly sparse matrix A, the region consists of all elements $a_{\mathtt{I},\mathtt{J}}$ with indices $(\mathtt{I}, \mathtt{J}) \in [1..m] \times [1..n]$ that satisfy all the constraints in the list of boundary pairs simultaneously. In this manner, regions can be described with an index set that can be expressed in terms of a two-dimensional simple section [2, 3].

Supplying empty or overlapping regions causes the sparse compiler to reject the program. All regions that are not explicitly specified are assumed to be sparse.

EXAMPLE: The following annotations indicate that the nonzero structure of an implicitly sparse matrix B with enveloping data structure `B` has the characteristics shown in figure 3, where the preferred access direction of the first dense region is $(1, 1)^T$:

```
      INTEGER   N
      PARAMETER (N=100)
      REAL      B(N,N)
C_SPARSE(ARRAY(B), DENSE( -5 <= I-J <=    5)(1,1)           )
C_SPARSE(ARRAY(B), ZERO (1-N <= I-J <= 20-N)                )
C_SPARSE(ARRAY(B), DENSE(N-4 <= I   <=    N, 6 <= I-J <= N-1))
```

[2]The indices `I` and `J` only serve a notational purpose, and have no relation with FORTRAN variables.

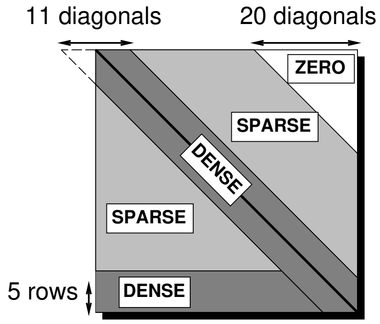


Figure 3: Nonzero Structure of B

For this matrix B, the sparse compiler will attempt to isolate operations on the different regions, to eliminate redundant operations on the upper right corner, and to reshape the access patterns of occurrences that access the band along the preferred access direction. If these attempts are successful, a storage scheme will be selected in which the band and border are stored statically in two rectangular full-sized arrays, and the entries in the sparse regions are stored dynamically in a pool of sparse vectors.

5 Phases of MT1

In [7] and [12, 15], the automatic data structure selection and transformation method of is presented as a three-phase method, consisting of (i) Program Analysis, (ii) Data Structure Selection, and (iii) Sparse Code Generation. At the implementation level of MT1, however, a slightly different subdivision into phases has been made, as is illustrated in figure 4. These phases are discussed in more detail in the following sections.

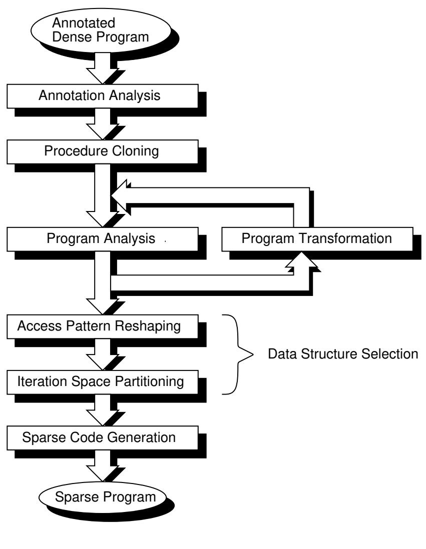


Figure 4: Phases of the Sparse Compiler

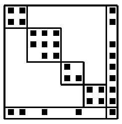


Figure 5: Example Matrix

5.1 Annotation Analysis

Directly after the programmer has directed MT1 to read an annotated program with the **readprg**-command, MT1 analyses the annotations in the program.

For each implicitly sparse matrix, MT1 records the properties supplied in annotations, such as the file in which this matrix resides, an approximated density, and a description of a number of regions together with the properties and preferred access direction for these regions. After all information has been gathered from the program, the annotations are stripped of the program, i.e. annotations do not appear in the abstract syntax tree representation of the program.

5.1.1 File Annotations

If the programmer defines a file for an implicitly sparse matrix A, then, independent on whether this file is already available at compile-time or not, this file is used in the initialization code for A that is generated by MT1.

If this file already exists at compile-time in the directory defined by the environment variable `SPARSEDIR`, then the file is also examined by the automatic nonzero structure analyzer that is described in detail in [7, ch6] and [11]. The results of this analysis are used by MT1 to select a suitable sparse storage scheme for A. If the analyzer has detected some zero regions, then MT1 inquires the programmer whether these regions will remain zero at run-time, as illustrated below:

```
=> readprg prg.f
** Zero regions of A preserved (y/n/q) ==> y
 :
 :
```

The programmer can use '**y**' or '**n**' to indicate that the zero regions are preserved or not, respectively. If the response '**q**' is given, MT1 switches into the mode in which all user interaction is disabled and default answers are given to all queries (cf. the **query**-command).

To provide some feedback to the programmer, the **smin**-command can be used to enable prompting the results of the analysis in the form of a 'sparse matrix information node'. Below, the output for the 10×10 matrix shown in figure 5 is given.

```
=> smin
Info: Show sparse matrix information
=> readprg program.f
Info: +--- THRESHOLD = 0.50 --------------------------------------+
Info: | Size             :     10 x 10                            |
Info: | #Entries         :          32                            |
Info: | Av #Entries/row  :        3.20   Density : 0.3200         |
Info: | Av #Entries/col  :        3.20   #Blocks : 10             |
Info: |                                                           |
Info: | Semi-Bandwidths  :      9-9      #Elts   : 100            |
Info: | #Used Diagonals  :     14        #Elts   : 75             |
Info: | #Full Diagonals  :      3        2-2 SB  : 68.75  %       |
Info: |                                  >= 90% B: 17             |
Info: |                                                           |
Info: | Block-D/L/U      :          40          69          70    |
Info: |                         1/4         1/5         1/4       |
Info: |                                                           |
Info: | Type             : Bordered Block Diagonal Form   (0.8000)|
Info: +-----------------------------------------------------------+
 :
 :
```

First, some general information is prompted which is computed under the assumption that each nonzero element is stored exactly once. Moreover, the number of detected dense blocks is given. Subsequently, some band related information is given. In the band with lower and upper semi-bandwidth 9, 100 elements appear. There are 14 used diagonals containing 75 elements in total, whereas only 3 of these diagonals are full. Furthermore, 68.75% of the entries appears in a band with bandwidth 5, and the smallest band containing more than 90% of all entries has bandwidth 17. Thereafter, the total number of elements in the nonzero blocks of minimum block partitions into (bordered) block diagonal and triangular form are shown in combination with a pair containing the corresponding border size and number of diagonal blocks in the remaining sub-matrix. Finally, the classification of this matrix is shown, together with the density within the corresponding nonzero regions. For the example, the bordered block diagonal form form has been selected, illustrated in figure 5. This form has a density of 0.8 (viz. 32/40).

5.1.2 Nonzero Structure Annotations

As discussed in section 4.3, the programmer can supply nonzero structure information to MT1 by specifying the property and preferred direction for a number of regions. Because a **simple section** [2, 3] provides a convenient representation of the index set of a part in an array (at programming level) or, likewise, the index set of a region in a matrix (at logical level), all regions in MT1 are described in terms of simple sections.

A two-dimensional simple section $S \subseteq \mathcal{Z}^2$ consists of all discrete points $(i, j) \in Z^2$ in a convex polygon defined by the 8 inequalities shown below, where $\sigma_i, \tau_i \in \mathcal{Z}$:

$$S = \{(i,j) \in \mathcal{Z}^2 \mid \left[\begin{array}{ccccc} \sigma_1 & \leq & i & \leq & \tau_1 \\ \sigma_2 & \leq & j & \leq & \tau_2 \\ \sigma_3 & \leq & i+j & \leq & \tau_3 \\ \sigma_4 & \leq & i-j & \leq & \tau_4 \end{array} \right] \}$$

Obviously, a simple section can be represented in a compact manner by only storing the vectors $\vec{\sigma}$ and $\vec{\tau}$.

Moreover, many operations, such as taking the intersection of two simple sections, computing the number of elements in a simple section, or refining the inequalities are relatively easy to perform [7, p116-124] and [20].

Each time MT1 encounters a nonzero structure annotation for an $m \times n$ implicitly sparse matrix A, it starts by constructing a new simple section S that is represented by the vectors $\vec{\sigma} = (1, 1, 2, 1-n)$ and $\vec{\tau} = (m, n, m+n, m-1)$. Since in this manner S consists of the whole index set $[1, m] \times [1, n]$ of A, effectively the region consisting of the whole matrix is described. Thereafter, for each construct defining boundary values $\sigma_i = l$ and $\tau_i = u$, the following assignment statements are executed:

$$\left\{ \begin{array}{lcl} \sigma_i & := & \max(\sigma_i, l); \\ \tau_i & := & \max(\tau_i, u); \end{array} \right.$$

Eventually, the inequalities of S are further refined if possible and the vectors $\vec{\sigma}$ and $\vec{\tau}$ are stored together with the property and preferred access direction of the corresponding region. If the programmer has specified a preferred access direction, then this direction is stored after the components have been made relatively prime. If no preferred access direction has been specified, MT1 uses the shape of the simple section to assign a suitable preferred access direction to the region, as illustrated in figure 6.

EXAMPLE: Consider, for instance, what happens if the following annotation appears in a program:

```
      REAL A(100,100)
C_SPARSE( ARRAY(A), DENSE(90<=I<=100, 20<=I-J<=99)(0,2) )
```

Because array `A` is used as enveloping data structure of an 100×100 implicitly sparse matrix, MT1 constructs a new simple section S represented by the vectors $\vec{\sigma} = (1, 1, 2, -99)$ and $\vec{\tau} = (100, 100, 200, 99)$. The part '`90<=I<=100`' sets $\sigma_1 = 90$ and leaves $\tau_1 = 100$ unaffected. Likewise, '`20<=I-J<=99`' sets $\sigma_4 = 20$ and leaves $\tau_4 = 99$ unaffected. Boundary refinement yields the following simple section S:

$$S = \{(i,j) \in \mathcal{Z}^2 \mid \begin{bmatrix} 90 & \leq & i & \leq & 100 \\ 1 & \leq & j & \leq & 80 \\ 91 & \leq & i+j & \leq & 180 \\ 20 & \leq & i-j & \leq & 99 \end{bmatrix} \}$$

Finally, MT1 stores the vector representation of S together with the property 'dense' and the preferred access direction $(0, 1)$ (viz. $2/\gcd(0, 2) = 1$). Effectively, this representation describes the region illustrated in figure 7.

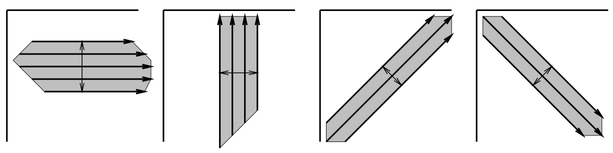


Figure 6: Some Selected Preferred Access Directions

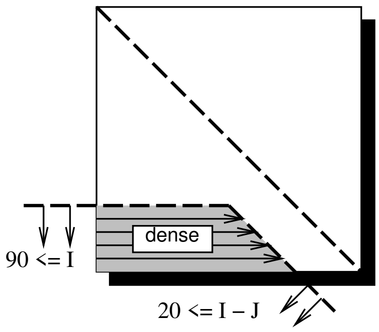


Figure 7: Region Described by `90`$\leq$`I` and `20`$\leq$`I-J`

5.2 Procedure Cloning

In MT1, enveloping data structures may be involved in the following parameter passing mechanisms:

- An element of an enveloping data structure is associated with a scalar *input*-argument.
- The first element of an enveloping data structure is associated with a two-dimensional formal argument that has exactly the same index set as the enveloping data structure.

These associations, and any association between variables of which the data structure remains unaffected and arbitrary formal arguments are referred to as **valid associations**. All other associations are invalid and must resolved explicitly by the programmer, as explained in more detail in [7, ch5].

If all associations in a annotated dense program are valid, MT1 uses **procedure cloning** [24, 23] *to enforce a unique association between enveloping data structures and formal arguments.* The algorithm is presented in detail in [7, ch5] and [18]. After cloning, all occurrences of a formal argument that are uniquely associated with an enveloping data structure can be handled as occurrences of this enveloping data structure. In this manner, different program and data structure transformations can be applied to each clone without interfering with other clones that support different kinds of associations for the original procedure.

EXAMPLE: In the following program, associating `A`, `B`, and `C` with the formal argument `M` is valid. However, because in subroutine `P` the enveloping data structure `A` can be associated with `M`, associating `M` as actual argument with the formal argument `L` in the call to subroutine `Q` is invalid:

```
      PROGRAM MAIN
      REAL A(10,10)
      REAL B(100), C(10,10)
C_SPARSE( ARRAY(A) )
      CALL P(A)
      CALL P(B)
      CALL P(C)
      END
```

```
      SUBROUTINE P(M)
      REAL M(10,10)
      IF (...) CALL Q(M)
      RETURN
      END
      SUBROUTINE Q(L)
      REAL L(20,10)
      ...
      RETURN
      END
```

```
Symbol_____Class______Type______________Length_Common_Offset_SMUVE_Dimension____

MAIN       PROGRAM                                              NY
A          VARIABLE   REAL                                   Y     1:10,1:10
B          VARIABLE   REAL                                         1:100
C          VARIABLE   REAL                                         1:10,1:10
...
P          SUBROUTINE                                           NN
M          DUMMY      REAL                                    NN   1:10,1:10
...
P_A        SUBROUTINE                                           NN
M          DUMMY      REAL                                   YNN   1:10,1:10
...
```

Figure 8: Symbol-Table

The sparse compiler uses a flow insensitive algorithm to determine all required procedure clones in a program and to detect potential invalid associations. For the previous example, this implies that even if the IF-statement is used to ensure that the call to `Q` in subroutine `P` will never occur while `A` is associated with `M`, the algorithm records the requirement of a clone `Q_A`, and reports a potential invalid association.

Only after the declaration '`REAL L(20,10)`' has been rewritten into '`REAL L(10,10)`', all associations become valid. In this case, MT1 generates the clones `P_A` and `Q_A` in which `A` is uniquely associated with `M` and `L` respectively. MT1 also preserves the original subroutines `P` and `Q` to handle the remaining calls. In the verbose-mode, MT1 informs the programmer about the clones required, as illustrated below:

```
=> verbose
Info: Verbose on
=> readprg main.f
Info: Parsing
  :
Info: Cloning
Info: Cloning P to P_A
Info: Cloning Q to Q_A
  :
```

After all clones have been generated, each procedure call in the program in which enveloping data structures appear as actual arguments are replaced by a call to the appropriate clone. Each actual and corresponding formal argument used to pass a whole implicitly sparse matrix are eliminated from the CALL-statement and header after the unique association has been recorded in the clone. Arguments used to pass a single element of an implicitly sparse matrix, on the other hand, are preserved.

If the program is examined, the unique association of an enveloping data structure `A` with a formal argument `F` that has been eliminated from the header is indicated by prompting '`<- Sparse(A)`' after the declaration of this formal argument. The same construct is used in the main program to indicate which arrays are used as enveloping data structures.

EXAMPLE: For example, after cloning the previous program (with the declaration '`REAL L(10,10)`') is prompted to the programmer as follows:

```
PROGRAM MAIN                     SUBROUTINE P_A()
REAL A(10,10) ! <- Sparse(A)     REAL M(10,10) ! <- Sparse(A)
REAL B(100),C(10,10)
                                 C9: IF (...) X10: CALL Q_A()
X1: CALL P_A()                   X16: RETURN
X2: CALL P(B)
X3: CALL P(C)                    END
X13: STOP
                                 SUBROUTINE Q_A()
END                              REAL L(10,10) ! <- Sparse(A)

...                              ...
original subroutine P and Q      X17: RETURN
...
                                 END
```

In the symbol-table, a '`Y`' appears below the '`S`' in the `SMUVE`-fields for each enveloping data structure or formal argument that is associated with an enveloping data structure (the other fields are explained in the MT1 Reference Manual [22]). In figure 8, parts of the symbol-table of the previous program that are prompted to the programmer if the **showsym**-command is given are shown.

After a sparse storage scheme has been selected for the implicitly sparse matrix having `A` as enveloping data structure, the selected data structure will be made available to all clones using a named COMMON-block, as is illustrated below:

```
PROGRAM MAIN                   SUBROUTINE P_A()
declarations ...               same declarations ...
COMMON /A/ identifiers ...     COMMON /A/ identifiers ...

CALL P_A( ... )                ...
...
END                            RETURN
                               END
```

5.3 Program Analysis

With respect to the sparse compiler, program analysis in MT1 consists of the following tasks:

- Conditions are associated with assignment statements, IF-statements, and DO-loops.
- The perfectly nested sub-loops in the program that can be reshaped are identified.
- Access summaries are constructed for all occurrences of enveloping data structures.

```
=> readprg feedb.f
Info: ** Matrix A          ******************************[rho=1.00e-01]***
Info: ** Size :  100 x 100   - /PROP/1    FILE : none                   **
Info: ** Occ. :    2 : 1    Row/Col/Diag/Scal/Comp :   1/  0/  0/  1/  0  **
Info: ***********************************************************************
```

Figure 9: Feedback to the Programmer after Program Analysis

In order to identify the statements that can exploit sparsity to reduce the computational time of the original dense program, MT1 associates a condition with each statement that defines the instances of the statement that have to be executed. For example, if $E(A)$ denotes the index set of the **entries** (stored elements) of A,[3] then condition '$(\mathtt{I},\mathtt{J}) \in E(A)$' can be associated with the following assignment statement, since this statement can safely be skipped for zero elements without affecting the semantics of the program:

```
ACC = ACC + A(I,J)
```

Likewise, conditions can be associated with IF-statements and DO-loops. The trivial condition '**true**' simply indicates that a statement cannot exploit sparsity. Condition '**false**' indicates that a statement can already be eliminated at compile-time. All conditions are computed by evaluating an attributed grammar that has been presented in [12, 15] and improved in [7, ch5].

During examination of an annotated dense program, all non-trivial conditions are prompted after each statement, thereby accounting for the unique association between formal arguments and enveloping data structures in a clone. An example is shown below:

```
       SUBROUTINE P_A()
       REAL M(100,100) ! <- Sparse(A)
       ...
       L9: DO I = <1>, <100>, 1
         L10: DO J = <1>, <100>, 1
           S11: X(<I>) = X(<I>)+M(<I>,<J>)
C_ASSIGN_COND: <I>,<J> <- E(A)
           C17: IF (M(<I>,<J>).GT.30) X13: STOP
C_IF_CONTROL: <I>,<J> <- E(A)
         ENDDO
       ENDDO
       ...
       END
```

During program analysis, MT1 also identifies the perfectly nested sub-loops in a program that can be reshaped according to the reshaping method of [7, ch7] and [9]. In addition, MT1 computes an **access summary** for each occurrence of an enveloping data structure, which consists of a normalized access direction and a simple section that approximates the part of the enveloping data structure that can be accessed by the occurrence. Details can be found in [7, ch5] and [20].

[3] The precise content of this set is usually not known at compile-time. However, index pairs belonging to dense regions are certainly included in this set, whereas index pairs belonging to zero regions are certainly not present.

Although most of the information gathered during program analysis is stored internally by MT1 for future use, for each implicitly sparse matrix, some feedback is given to the programmer.

EXAMPLE: Consider the following simple program:

```
      PROGRAM FEEDB
      PARAMETER (N=100)
      REAL       A(N,N), X
C_SPARSE(ARRAY(A), SPARSE()(0,1) )
C_SPARSE(ARRAY(A), DENSITY(0.1) )

      A(1,1) = 10.0
      DO I = 1, N
        DO J = 1, N
          X = X + A(I,J)
        ENDDO
      ENDDO
      END
```

The feedback that is given for this program to the programmer is shown in figure 9. First, the density and size of the implicitly sparse matrix are given. Thereafter, three fields are used to indicate whether nonzero structure information is available. The first field is either 'INF' or '-' to indicate that information has been obtained from automatic nonzero analysis of the matrix on file. Likewise, the second field is 'PROP' if nonzero structure annotations are used, or '-' otherwise. The last field indicates the number of regions of which some information is available. In the second line, first the total number of occurrences of the enveloping data structure and the number of occurrences that cannot be reshaped are given. Finally, the number of occurrences having row-, column-, diagonal-, scalar-wise, or complex access patterns (due to subscript functions that cannot be analyzed at compile-time) is given.

Note that the 'conventional' program analysis of MT1 consists of several other analyses as well (such as data dependence analysis). This is described in depth in the MT1 Reference Manual [22].

5.4 Transformation Application

The ability of MT1 to apply programmer-defined program transformations enables the programmer to define a number of preparatory program transformations that convert the original dense program into a form from which more efficient sparse code can be obtained. For example, loop distribution can be used to increase the number of perfectly nested loops in the program, thereby providing MT1 with more opportunities to apply access pattern reshaping. The same loop transformation can also be used to increase the number of loop-bodies in the program that are dominated by a particular guard.

A number of other program transformations that may enable the generation of more efficient sparse code is given in [7, p112-114]. The programmer may also define a set of ordinary program transformations that simply improve the efficiency of certain constructs in the dense program. The programmer must be careful, however, that such transformations do not interfere with the generation of efficient sparse code, for example, by obscuring the functionality of the original program or by disabling the reshaping method.

The programmer-defined transformations are defined using the **transformation definition language** described in [6, 17] and the MT1 Reference Manual [22]. After MT1 has been instructed to read a set of transformation definitions using the **readtrf**-command, program transformations can be either applied interactively or automatically using the **start**- or **auto**-command respectively.

EXAMPLE: Below, loop distribution is defined:

```
transform
 list( do ( !e1, !e2, !e3, !e4,
              follow(!s1, !s2) ),  !s3  )
into
 list( do ( !e1, !e2, !e3, !e4, !s1),
        list( do( !e1, !e2, !e3, !e4,  !s2 ), !s3 ) )
condition
    not isnil (!s2)  and
    not dep @ < ($.dobody.follow,$.dobody)
;
```

In the interactive mode, each matching fragment together with the fragment that would result after applying the corresponding transformation is prompted to the programmer. An example is given below:

```
=> start
** MATCH ON
      L2: DO J = <1>, <10>, 1
        S3: B(<I>) = B(<I>)+(A(<I>,<J>)*X(<J>))
            -----
        S4: D(<I>,<J>) = 0.
      ENDDO
** TRANSFORM INTO
      L8: DO J = <1>, <10>, 1
        S7: B(<I>) = B(<I>)+(A(<I>,<J>)*X(<J>))
      ENDDO
      L10: DO J = <1>, <10>, 1
        S9: D(<I>,<J>) = 0.
      ENDDO
      .....
** ACCEPT (y/n/q/e/s) ==> y
   :
```

Here, the programmer can instruct MT1 to either apply the transformation, not to apply the transformation, to quit the current transformation, to quit the transformation phase, or to go to the next program unit, with the answer '**y**', '**n**', '**q**', '**e**', or '**s**', respectively.

Since the current implementation does not allow for program transformations that alter function- or procedure-calls, there is no need to apply procedure cloning after programmer-defined transformations have been applied. However, the transformed program is analyzed again by MT1, as illustrated in figure 4. This implies that again some feedback may be given (cf. figure 9).

5.5 Data Structure Selection

If the programmer instructs the sparse compiler to generate sparse code with the **sparse**-command, MT1 enters the last three phases, as illustrated in figure 4. First, a sparse storage scheme must be selected for each implicitly sparse matrix. Although according to the philosophy behind MT1, the core of the compiler should be kept as empty as possible, two program transformations that support the data structure selection of the sparse compiler have been implemented internally in MT1. In this section, these transformations are discussed.

After these transformations have been applied, the sparse compiler selects a hybrid combination of static full-sized storage of dense regions and dynamic storage in a pool of sparse vectors of sparse regions. The layout of sparse vectors may be chosen arbitrarily over each different sparse region. Naturally, no storage is allocated for regions that remain zero at run-time. Instead, the sparse compiler will try to eliminate useless operations on elements of these regions already at compile-time [20]. All details of the sparse storage scheme and the data structure selection can be found in [7, ch7].

5.5.1 Reshaping Access Patterns

Although there are dynamic sparse storage schemes that support the fast generation of entries along different directions, for instance, along rows and columns (cf. linked list schemes), the overhead storage and run-time maintenance overhead of such storage schemes is usually substantial. Therefore, the sparse storage scheme that may be selected by MT1 to store the entries in the sparse regions of an implicitly sparse matrix are restricted to a dynamic data structure that consists of a pool of sparse vectors supporting only one direction for each individual sparse region. The layout of sparse vectors may be different for different sparse regions.

This implies that it becomes very important that each sparse region is accessed in a consistent manner, namely along the direction supported for that region. Only in this manner, the the overhead reducing techniques of section 5.6 may become enabled. Even dense regions should be accessed in a consistent manner, because this enables the selection of a full-sized array over this region in which elements along the most frequently occurring access patterns are stored along the columns, which enhances spatial locality in FORTRAN. Because it is likely that the enveloping data structures are accessed along arbitrary directions in the original dense program, a method to reshape the access patterns of two-dimensional arrays, based on the unimodular framework of [4, 5], has been incorporated in the sparse compiler.

In MT1, this reshaping method is implemented by scanning over all perfectly nested sub-loops that have been identified during program analysis.

For each loop, all occurrences of enveloping data structures that allow for reshaping are examined. If this occurrence may access a part of the array that corresponds to a region for which a preferred direction has been specified, then this occurrence will be involved in reshaping. Otherwise, the occurrence is simply discarded from further consideration. Subsequently, a reshaping method is applied. On failure, some user interaction is performed about whether certain prohibiting occurrences or data dependences may be ignored during reshaping, after which possibly another attempt is taken. On success, the new access summaries of all affect occurrences are constructed incrementally to avoid re-analysis of the program. Implementation details of the reshaping method can be found in [7, ch7] and [9].

EXAMPLE: Consider the following program:

```
      PROGRAM RESHAPE
      INTEGER N, I, J
      PARAMETER (N=100)
      REAL      A(N,N), B(N,N), C(N,N)
C_SPARSE(ARRAY(A), SPARSE()(0,1) )
C_SPARSE(ARRAY(B), SPARSE()(0,1) )
C_SPARSE(ARRAY(C), SPARSE()(0,1) )

      DO J = 1, N
        DO I = 1, N
          C(J,I) = A(I,J) + B(I,J)
        ENDDO
      ENDDO
      ...
```

Because the programmer has indicated that row-wise access patterns are preferred for the three implicitly sparse matrices A, B, and C with the enveloping data structures `A`, `B`, and `C`, respectively, MT1 tries to reshape the double loop accordingly. Obviously, however, the reshaping method fails to reshape all access patterns accordingly:

```
=> sparse
Warning: reshaping fails
      L1: DO J = <1>, <100>, 1
        L2: DO I = <1>, <100>, 1
          S3: C(<J>,<I>) = A(<I>,<J>)+B(<I>,<J>)
        ENDDO
      ENDDO
** C(<J>,<I>): IGNORE (n/y/q) ==> y
** A(<I>,<J>): IGNORE (n/y/q) ==> n
** B(<I>,<J>): IGNORE (n/y/q) ==> n
Info: Ignored : 1   Remaining : 2
⋮
```

Here, the programmer has indicated that the occurrence of `C` may be ignored during reshaping, so that the reshaping method can be successful by leaving the double loop unaffected.

EXAMPLE: If row-wise access patterns are preferred for A, then the following user interaction could occur:

```
=> sparse
Info: Dependences prohibit reshaping
      L1: DO J = <1>, <10>, 1
        L2: DO I = <1>, <10>, 1
          S3: X = X+A(<I>,<J>)
        ENDDO
      ENDDO
** Dependence: S3 d-anti <* S3 X: IGNORE (n/y/q) ==> y
** Dependence: S3 d-anti =< S3 X: IGNORE (n/y/q) ==> y
** Dependence: S3 d-flow <* S3 X: IGNORE (n/y/q) ==> ...
⋮
```

Here, MT1 informs the programmer about the fact that data dependences prohibit loop interchanging. After the programmer has indicated that all data dependences due to the accumulation can safely be ignored, the reshaping method will interchange the loops.

Recall that user interaction may be disabled and enabled again using the **query**-command. If the verbose mode has been enabled using the **verbose**-command, eventually MT1 provides some feedback by reporting the number of times the reshaping method was successful or failed:

```
=> verbose
Info: Verbose on
=> sparse
⋮
Info: Reshaping (succ=1, fail=0)
⋮
```

With the **bounds**-command, the programmer can toggle between an aggressive but expensive simplification method of loop bounds after applying a unimodular loop transformation, or a less expensive but also less precise simplification method.

5.5.2 Iteration Space Partitioning

Another important step for the sparse compiler is the construction of a number of mutually disjoint regions in each implicitly sparse matrix A such that each part of the corresponding enveloping data structure `A` that may be accessed by an arbitrary occurrence of `A` corresponds to a region in A that is confined to exactly one of these regions. Given these regions, MT1 can select a different storage organization for each region and convert the code accordingly. Since each occurrence can only access a part of the enveloping data structure corresponding to a region in A that is confined to *only one* of these regions, the need for run-time tests to determine which of the selected storage organizations must be accessed is avoided. Because usually only a limited number of such regions can be distinguished, the iteration space partitioning presented in [7, ch7] and [14, 16, 20] has been incorporated in MT1 to increase the resulting amount of fragmentation.

EXAMPLE: For example, for the following fragment, MT1 detects that peeling of the iteration `J=I` confines the overlap between the two occurrences of `A` to the main diagonal:

```
                                          DO I = 1, 100
DO I = 1, 100                               B(I) = A(I,I)
  B(I) = A(I,I)                             A(I,I) = D(I,I)
  DO J = I, 100             I = J           DO J = I+1, 100
    A(I,J) = D(I,J)           →               A(I,J) = D(I,J)
  ENDDO                                     ENDDO
ENDDO                                     ENDDO
```

Obviously, after application of a relatively simple transformation, different kind of storage can be selected for the main diagonal and the strict lower and upper triangular part of A.

Again, in the verbose mode eventually MT1 provides some feedback by reporting the number of times iteration space partitioning was successful or failed:

```
=> verbose
Info: Verbose on
=> sparse
.
.
.
Info: Reshaping (succ=1, fail=0)
Info: Partitioning (succ=3, fail=0)
.
.
.
```

Obviously, if one of these methods fails a substantial number of times, it becomes very likely that inefficient sparse code will be generated. In this cases, it may be useful to experiment with different annotations and formulations of the original dense algorithm.

5.6 Sparse Code Generation

In the final phase, the actual data structure transformations are applied by converting the (possibly adapted) dense code into a form that operates on the selected sparse storage schemes, thereby using overhead reducing techniques as much as possible. A small number of general primitives are supplied in a separate library, so that constructs that would otherwise appear frequently in the generated sparse code can be replaced by calls to the appropriate primitives. This approach tends to reduce the size of the generated sparse code whereas, on the other hand, the primitives are chosen general enough to leave the sparse compiler with enough flexibility in selecting data structures. Identifiers of all subroutines and functions in this library, as well as all compiler generated identifiers contain underscores to prevent conflicts with identifiers in the original dense program (assuming that identifiers in the original dense program adhere the ANSI FORTRAN 77-standard, whereas the compiler used to compile the generated sparse code supports the use of underscores in identifiers).

5.6.1 Initialization Code Generation

Because the programmer is unaware of the sparse storage scheme that is eventually selected by the sparse compiler, MT1 is responsible for generating appropriate initialization code for each selected data structure. This code is generated at the beginning of the main program, and expects each implicitly sparse matrix that is stored on file in coordinate-scheme.

Static full-sized arrays are initialized by a BLOCK DATA-construct. Moreover, if no file name has been specified, the whole dynamic pool is simply reset using the primitive `SINI__`. On the other hand, if a file name has been specified in a file annotation, then MT1 generates code that will read the entries of an implicitly matrix from that file and store these entries in the appropriate data structures.

Entries that belong to full-sized arrays are stored directly at the appropriate position. All entries belonging to the dynamic pool are first temporarily stored in coordinate-scheme like storage using an additional array `TMP__`, after which the primitive `SINI__` is used to initialize the dynamic pool of sparse vectors appropriately. Moreover, if the programmer has indicated that some regions remain zero at run-time, tests are generated that validate these assertions.

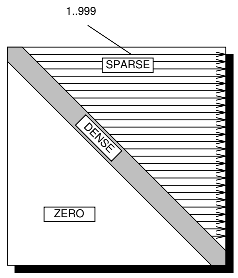


Figure 10: An Example Sparse Storage Scheme

EXAMPLE: Consider the following annotations that direct MT1 to select the data structure shown in figure 10:

```
      PARAMETER (N=100)
      REAL       A(N,N)
C_SPARSE(ARRAY(A), SPARSE(1-N <= I-J <=  -1)(0,1) )
C_SPARSE(ARRAY(A), DENSE (0   <= I-J <=   0)(1,1) )
C_SPARSE(ARRAY(A), ZERO  (1   <= I-J <= N-1)      )
C_SPARSE(ARRAY(A), FILE('FILEA'))
```

If this data structure is actually selected by MT1, then the following initialization code is generated automatically:

```
 INTEGER TMP__(693)
 REAL    DN1_A,VAL_A
 INTEGER IND_A,LOW_A,HGH_A,LST_A
 COMMON /A___/DN1_A(100),VAL_A(693),IND_A(693),
+             LOW_A(99),HGH_A(99),LST_A

 LST_A = 1
 OPEN (1,FILE='FILEA',STATUS='OLD')
 READ (1,*) M_, N_, NNZ_
 IF ((M_.NE.100).OR.(N_.NE.100)) STOP 'Incorrect Size'
 IF (NNZ_.GE.693) STOP 'Too many entries'
 DO K_ = 1, NNZ_, 1
   READ (1,*) I_,J_,V_
   IF ((I_-J_).EQ.0) THEN
     DN1_A(J_) = V_
   ELSE IF ((I_-J_).LE.(-1)) THEN
     LST_A = LST_A+1
     VAL_A(LST_A) = V_
     IND_A(LST_A) = J_
     TMP__(LST_A) = I_
   ELSE IF (1.LE.(I_-J_)) THEN
     STOP 'Entry occurs in zero region'
   END IF
 ENDDO
 CLOSE (1)
 CALL SINI__(VAL_A,TMP__,IND_A,LOW_A,HGH_A,99,693,LST_A)
```

The following BLOCK DATA-construct also appears in the generated program to reset the full-sized array `DN1_A` that is used to store the main diagonal of A:

```
 BLOCK DATA
 REAL DN1_A,VAL_A
 INTEGER IND_A,LOW_A,HGH_A,LST_A
 COMMON /A___/DN1_A(100),VAL_A(693),IND_A(693),
+      LOW_A(99),HGH_A(99),LST_A

 DATA DN1_A /100*0./

 END
```

5.6.2 Actual Sparse Code Generation

Because simply skipping operations by means of the conditions that have been computed by MT1 does not reduce the execution time on most machines [30, 32], overhead reducing techniques are required. The prototype sparse compiler has the ability to apply one of the following overhead reducing techniques:

1. replacing accesses to zero regions by zero,
2. avoiding overhead by using static full-sized storage for particular regions,
3. applying so-called guard encapsulation, where a construct that iterates over the entries along an access pattern is generated, or
4. applying so-called access pattern expansion, where a sparse vector is scattered into a full-sized array before operations are applied to this vector, and gathered back into the dynamic sparse storage thereafter.

Rather surprisingly, the first experiments with MT1 [8, 19] indicate that these four techniques, explained in more detail in [7], are sufficient to convert many dense primitives into code in which the sparsity of matrices is fully exploited. In this section, each technique is explained by means of a simple example,

EXAMPLE: In the following fragment, an annotation is used to inform the compiler that the main diagonal of an implicitly sparse matrix A with enveloping data structure `A` is zero, and will remain so at run-time:

```
      REAL A(N,N), X(N)
C_SPARSE(A : _ZERO(0 <= I-J <= 0))
      ...                                 ...
S1:   DG = 0.0                     →   DG = 0.0
      DO I = 1, N                         ...
S2:     DG = DG + A(I,I) * X(I)
      ENDDO
      ...
```

Hence, occurrence `A(I,I)` can be replaced by '`0.0`'. Thereafter, the condition '**false**' becomes associated with S_2 and, hence, with the `I`-loop (since MT1 always ignores the 'side-effect' that sets the final value of a loop index). This implies that both statements can be eliminated.

EXAMPLE: Below, an annotation is used to indicate that a small band in a 5×5 implicitly sparse matrix B is dense. If this band can be isolated in the whole program, full-sized storage of the band is selected and the following conversion is done:

```
   REAL B(5,5)
 C_SPARSE(B:_DENSE(-1<=I-J<=1))
   DO I = -1, 1
     DO J = MAX(1,1-I),MIN(5,5-I)
       B(I+J,J) = ...
     ENDDO              →
   ENDDO

REAL DN1_B(1:5,-1:1)
...
DO I = -1, 1
  DO J = MAX(1,1-I),MIN(5,5-I)
    DN1_B(J,I) = ...
  ENDDO
ENDDO
```

Note that unit-stride access appears in the resulting program.

EXAMPLE: To prevent the requirement for *ordered* storage of entries in the dynamic pool of sparse vectors (not supported in the current prototype sparse compiler), guard encapsulation may only be applied if no data dependences is carried by the DO-loop for which encapsulation is possible, and if no exit-branch or STOP-statement can be executed in the body of that loop. If a data dependences prohibits encapsulation, MT1 inquires the programmer whether the prohibiting data dependences may be ignored:

```
=> sparse
Info: Loop-carried dependences prohibit encapsulation
      L2: DO J = <1>, <100>, 1
        S3: Y(<I>) = Y(<I>)+(A(<I>,<J>)*X(<J>))
      ENDDO
** Dependence: S3 d-flow =< S3 Y: IGNORE (n/y/q) ==> y
** Dependence: S3 d-anti =< S3 Y: IGNORE (n/y/q) ==> y
** Dependence: S3 d-outp =< S3 Y: IGNORE (n/y/q) ==> y
⋮
```

Since here the programmer has indicated that all data dependences due to the accumulation may be ignored, MT1 converts his fragment into code that will only iterate over the entries in A at run-time:

```
DO J_ = LOW_A(I), HGH_A(I)
  J = IND_A(J_)
  Y(I) = Y(I)+VAL_A(J_)*X(J)
ENDDO
```

EXAMPLE: In the following example, access pattern expansion at nesting depth 1 is feasible for occurrence `A(I,J)`, which also affects the other occurrence of the enveloping data structure `A`:

```
   REAL A(100,100), D(100,100)
C_SPARSE(A : _SPARSE()(0,1))
   DO I = 1, 100
     DO J = 1, 100
       D(I,J) = A(I,I) * A(I,J)
     ENDDO
   ENDDO
```

This expansion is implemented using the primitives `SSCT__` and `SGTH__`. The full-sized array `SAP_10` is used to store the expanded access pattern. Array `SWT_10` is used as a switch array to record the position of entries in the full-sized array (not particularly useful for this example):

```
REAL    SAP_10(1:100)
LOGICAL SWT_10(1:100)
COMMON  /STOR__/ SAP_10, SWT_10
...
DO I = 1, 100
  CALL SSCT__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
  DO J = 1, 100
    D(I,J) = SAP_10(I) * SAP_10(J)
  ENDDO
  CALL SGTH__(VAL_A, IND_A, LOW_A(I), HGH_A(I), SAP_10, SWT_10)
ENDDO
```

Although lookup overhead in sparse data structures is avoided in the innermost DO-loop with access pattern expansion, this technique does *not* reduce the actual number of operations performed.

```
=> sparse
.
.
.
Info: ** Sparse Code Statistics **********************************
Info: ** Zero         : 4     Dense    : 1     Sparse    : 3     **
Info: ** Encapsulated : 1     Expanded : 1     Remaining : 1     **
Info: ************************************************************
```

Figure 11: Feedback to the Programmer after Sparse Code Generation

EXAMPLE: If none of the previous overhead reducing techniques is applicable, each remaining occurrence of an enveloping data structure is simply replaced by a lookup in the dynamic pool of sparse vectors.

Consider, for instance, the following fragment, in which annotations are used to enforce the selection of general sparse row- and column-wise storage for the implicitly sparse matrices A and B having array A and B respectively as enveloping data structures:

```
      REAL A(15,15), B(20,20)
C_SPARSE(A : _SPARSE()(0,1))
C_SPARSE(B : _SPARSE()(1,0))
      ...
      A(2,3) = 7.0 * B(15,4) + 1.0 - A(2,3)
```

Because occurrences at nesting depth 0 cannot be involved in any overhead reducing technique, eventually the scalar statement is replaced by the following construct, where primitive SINS__ is used to insert a new entry in a dynamic pool, and primitive LKP__ is used to fetch an element from a sparse vector (returning the reserved index 1 if this element is not an entry, where a zero value always appears at this location):

```
 L_ = LKP__(IND_A, LOW_A(2), HGH_A(2), 3)
 IF (L_ .EQ. 1) THEN
   CALL SINS__(VAL_A, IND_A, LOW_A, HGH_A, 2,
+              NP_A, SZ_A, LST_A, L_, 3)
 ENDIF
 VAL_A(L_) = 7.0 * VAL_B(  LKP__(IND_B, LOW_B(4),
+              HGH_B(4), 15)) + 1.0 - VAL_A(L_)
```

After sparse code has been generated, MT1 provides some feedback to the programmer by prompting the number of times each overhead reducing techniques was applicable. An example is shown in figure 11. Here, MT1 informs the programmer that 4 occurrences of enveloping data structures were replaced by zero, 1 occurrence has been replaced by a reference to a full-sized array, and 3 occurrences access regions that are stored in a dynamic pool of sparse vectors. The second line indicates how these 'sparse occurrences' are handled by prompting the number of times guard encapsulation or access pattern expansion was applicable. All remaining occurrences are simply replaced by an expensive lookup. Obviously, if many occurrences are handled this way, this strongly indicates that the conflicts in the program are not handled very well. In this case, it may be useful to experiment with different annotations and formulations of the original dense algorithm.

6 Final Remarks

Although, at least in principle, the ordinary program transformations of MT1 could be made more advanced to convert the input dense program into some other dense form from which eventually efficient sparse code can be obtained, the programmer can assist the sparse compiler by adhering to a few programming rules.

For example, the sparse compiler is not able to directly apply guard encapsulation to the following DO-loop taken from the Level 2 BLAS-routine SGEMV [26, 27], because condition '**true**' has been associated with the second statement in the body of the I-loop:

```
      IX   = KX                              ← true
      DO 110, I = 1, M
         TEMP = TEMP + A( I, J )*X( IX )     ← (I,J) ∈ E(A)
         IX   = IX   + INCX                  ← true
110   CONTINUE
```

Obviously, standard compiler techniques dealing with induction variables [1] can replace subscript IX with the expression 'KX+(I-1)*INCX', after which the prohibiting assignment statement can be eliminated. However, since these techniques are not part of the standard transformations performed by MT1, the programmer can assist the sparse compiler by using a formulation of the algorithm in which this conversion has already been done.

As another example, in the following fragment, MT1 associates the condition '**true**' with the loop-body of the J-loop:

```
DO I = 1, 100
  T = A(I,I)
  DO J = 1, I - 1
    D(I,J) = D(I,J) - T
  ENDDO
ENDDO
```

Again, although after scalar forward substitution [33, p178-179], and dead-code elimination [1], MT1 can associate the condition '(I,I) $\in E(A)$' with the only remaining statement 'D(I,J) = D(I,J) - A(I,I)', the programmer can assist the sparse compiler by directly formulating the algorithm in this manner. In general, some experimentation may be required to get acquainted with the kind of dense constructs that are handled efficiently by the sparse compiler. Unfortunately, however, there are also some dense construct that, independent of the way these constructs are formulated, are always converted into extremely inefficient sparse codes (cf. the transposition example in [7, p246]).

Finally, note that although some experimentation has been performed with the advanced transformations presented in [7, ch10] and [13], these transformations have not been incorporated in the current version of MT1.

Contact Information

Please, send all your comments and bug-reports on MT1 to `ajcbik@cs.leidenuniv.nl`. Information related to the sparse compiler MT1 is being maintained at `http://www.wi.leidenuniv.nl/home/ajcbik/`.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Department of Computer Science, Rice University, 1989.

[3] Vasanth Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[4] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Languages and Compilers for Parallel Computing*, 1990.

[5] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Boston, 1993.

[6] Aart J.C. Bik. A prototype restructuring compiler. Master's thesis, Utrecht University, 1992. INF/SCR-92-11.

[7] Aart J.C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Department of Computer Science, Leiden University, 1996. ISBN 90-9009442-3.

[8] Aart J.C. Bik, Peter J.H. Brinkhaus, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. The automatic generation of sparse primitives. Technical Report no 96-20, Department of Computer Science, Leiden University, 1996.

[9] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, No. 892*, pages 406–422. Springer-Verlag, Berlin/New York, 1995.

[10] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[11] Aart J.C. Bik and Harry A.G. Wijshoff. Nonzero structure analysis. In *Proceedings of the International Conference on Supercomputing*, pages 226–235, 1994.

[12] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, No. 768*, pages 57–75. Springer-Verlag, Berlin/New York, 1994.

[13] 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.

[14] Aart J.C. Bik and Harry A.G. Wijshoff. Construction of representative simple sections. In *Proceedings of the International Conference on Parallel Processing*, pages 9–18, 1995. Volume 2: Software.

[15] 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.

[16] Aart J.C. Bik and Harry A.G. Wijshoff. Iteration space partitioning. In H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *Lecture Notes in Computer Science, No. 1067*, pages 475–484. Springer-Verlag, 1996.

[17] Aart J.C. Bik and Harry A.G. Wijshoff. MT1: A prototype restructuring compiler. In *Proceedings of the second annual conference of the Advanced School for Computing and Imaging (ASCI)*, pages 78–83, 1996. Lommel, Belgium.

[18] Aart J.C. Bik and Harry A.G. Wijshoff. A note on dealing with subroutines and functions in the automatic generation of sparse codes. In *Proceedings of the second annual conference of the Advanced School for Computing and Imaging (ASCI)*, pages 96–101, 1996. Lommel, Belgium.

[19] Aart J.C. Bik and Harry A.G. Wijshoff. Simple *Quantitative* experiments with a sparse compiler. In *Irregular 1996*, 1996. To Appear.

[20] Aart J.C. Bik and Harry A.G. Wijshoff. The use of iteration space partitioning to construct representative simple sections. *Journal of Parallel and Distributed Computing*, 34:95–110, 1996.

[21] Peter J.H. Brinkhaus. Compiler analysis of procedure calls. Master's thesis, Utrecht University, 1993. INF/SCR-93-13.

[22] Peter J.H. Brinkhaus, Aart J.C. Bik, and Harry A.G. Wijshoff. *The MT1 Reference Manual.* Department of Computer Science, Leiden University, 1996.

[23] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 96–105, 1992.

[24] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8:491–523, 1986.

[25] David S. Dodson, Roger G. Grimes, and John G. Lewis. Sparse extensions to the Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 17:253–263, 1991.

[26] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[27] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[28] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Society for Industrial and Applied Mathematics, 1991.

[29] Iain S. Duff. A survey of sparse matrix research. In *Proceedings of the IEEE*, pages 500–535, 1977.

[30] Iain S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, Oxford, 1990.

[31] Alan George and Joseph W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, New York, 1981.

[32] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[33] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

[34] Zahari Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer, Dordrecht, 1991.