Applicability of Automatic Program Comprehension to Sparse Matrix Computations

Christoph W. Keßler

Fachbereich IV - Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: `kessler@psi.uni-trier.de`

Abstract

Automatic program comprehension techniques have been shown to improve automatic parallelization of dense matrix computations. The recognition of concepts in the code enables aggressive automatic program transformations up to local algorithm replacement, and supports automatic data layout and performance prediction.

We show how this approach can be generalized to sparse matrix codes. Program comprehension will be particularly useful in this case, since it should allow to abstract e.g. from specific storage formats used in the code. Unfortunately, space–efficient data structures for sparse matrices typically yield programs in which not all data dependencies can be determined at compile time.

We propose a speculative program comprehension and parallelization method suitable for sparse matrix codes. Only where static analysis does not supply sufficient information, a parallelized run time test must check whether a code fragment speculatively recognized at compile time as a sparse matrix computation really implements this type of computation. The placement of run–time tests is supported by a static data flow analysis framework.

Keywords: program comprehension, concept recognition, parallelizing compiler, sparse matrix computations, speculative parallelization, data flow analysis, run–time data dependence analysis

1 Introduction

Matrix computations constitute the core of many scientific numerical programs. A matrix is called *sparse* if so many of its entries are zero that it seems worthwhile to use a more space–efficient data structure to store it than a simple two–dimensional array; otherwise the matrix is called *dense*. Space–efficient data structures for sparse matrices try to store only the nonzero elements. This results in considerable savings in space for the matrix elements and time for operations on them, at the cost of some space and time overhead to keep the data structure consistent. If the spatial arrangement of the nonzero matrix elements (the *sparsity pattern*) is statically known to be regular (e.g., a blocked or band matrix), the matrix is typically stored in a way directly following this sparsity pattern; e.g., each diagonal may be stored as a one–dimensional array (vector).

Irregular sparsity patterns are usually defined by runtime data. In the remainder of this article, we have only this case in mind when using the term “sparse matrix”. Typical data structures used for the representation of sparse matrices in Fortran77 programs are, beyond a *data array* containing the nonzero elements themselves, several *organizational variables*, e.g. arrays with suitable row and/or column index information for each data array element. Linked lists are, if at all, simulated by index vectors, as Fortran77 supports no pointers nor structures. C implementations may also use explicit linked list data structures to store the nonzero elements, which enhances dynamic insertion and deletion of elements. However, on several architectures, a pointer variable needs more space than an integer index variable. As space is often critical in sparse matrix computations, explicit linked lists occur rather rarely in practice. Also, many numerical C programs are written in a near–Fortran77 style, because they were either directly transposed from existing Fortran77 code, or because the programming style is influenced by former Fortran77 projects or Fortran77–based numerics textbooks.

Matrix computations on these data structures are common in practice and often parallelizable. Consequently, numerous parallel algorithms have been invented or adapted for sparse matrix computations over the last decades for various parallel architectures.

[2] suggests the programmer to express, in the source code, parallel (sparse) matrix computations in terms of dense matrix data structures, which are more elegant to parallelize and distribute, and let the compiler select a suitable data structure for the matrices automatically. Clearly this is not applicable to (existing) programs that use hard–coded data structures for sparse matrices.

While the problems of automatic parallelization for *dense* matrix computations are, meanwhile, well understood and sufficiently solved (e.g. [3, 4, 5]), these problems are, for *sparse* matrix computations, solved, if at all, in a very conservative way (e.g., by run–time parallelization techniques such as the inspector–executor method [6] or run–time analysis of sparsity patterns for load–balanced array distribution [7]). This is not astonishing because such code looks quite awful to the compiler, consisting of indirect array indexing or pointer dereferencing which makes exact static access analysis impossible. Run-time analysis, though, usually incurs tremendous overhead which is not always likely to be weighed out by parallel speedup, because the good volume–to–surface ratio of computation to communication, which is typical for dense matrix computations, is usually not present in sparse matrix computations.

For automatic parallelization of dense matrix computations and other completely statically analyzable codes, automatic program comprehension techniques appeared to be useful [5, 8]. After suitable preprocessing transformations and normalizations, the intermediate program representation — abstract syntax tree and/or program dependence graph — is submitted to a concept recognition tool which locally identifies code fragments for which there exist special code transformations or particular parallel algorithms tailored to the target machine. In the back–end phase, these code pieces can be replaced by suitable parallel implementations. The information derived in the recognition phase also supports automatic data layout and performance prediction.

We are interested in how far this approach can be extended to sparse matrix codes. One problem we are faced with is that there is no standard data structure to store a sparse matrix, compared to the two–dimensional array which is the "natural" storage scheme for the dense matrix. We have examined several representative source codes for implementations of basic linear algebra operations like dot product, matrix–vector multiplication, matrix–matrix multiplication, or LU factorization for sparse matrices [9, 10, 11, 12, 13, 16] and recorded a preliminary list of basic computational kernels (also often called "templates" or "concepts") for sparse matrix computations, together with their various syntactical appearances ("idioms").

The other main difference is that space–efficient data structures for sparse matrices use either indirect array references or (if available) pointer data structures. Thus the array access information required for safe concept recognition and code replacement is no longer completely available at compile time. Regarding program comprehension, this means that it is no longer sufficient to consider only the declaration of the matrix and the code of the computation itself, in order to safely determine the semantics of the computation. Code can only be recognized as an occurrence of, say, sparse matrix–vector multiplication, subject to the condition that the data structures occurring in the code really implement a sparse matrix. Generally, it is not always possible to statically evaluate this condition. In such a case, a concept recognition engine can only *suspect*, based on its observations of the code while tracking the life ranges of program objects, that a certain set of program objects implements a sparse matrix; the final *proof* of this hypothesis has then to be done at run time. For instance, such a run time test has to be executed always when one of the organisational variables is modified. Unfortunately, such run–time tests, even if parallelizable, incur some overhead.

Nevertheless, static program flow analysis can substantially support such a *speculative* comprehension and parallelization. Only at program points where not sufficient static information is available, the run–time tests are applied to confirm (or reject) the speculative comprehension.

Because of the great benefit that could be taken from a positive test result, we propose to generate two variants of parallel code for the program sections involved: (1) an optimized parallel sparse matrix algorithm (e.g., a library routine) which is executed speculatively, and (2) a conservative parallelization, maybe using the inspector–executor technique, or just sequential code, which is executed non–speculatively. A simple way of generating parallel code for a speculatively recognized program fragment could be to let one of the p available processors execute the sequential code while the remaining $p-1$ processors are spent to execute the speculatively parallelized variant (including run time checks where required). If the latter processors find out during execution that the hypothesis allowing parallel execution was wrong, they abort and wait for the sequential variant to complete. Otherwise, they abort the sequential variant and return the computed results. — Nevertheless, if the sparsity pattern is static, it may be more profitable to execute the run time test once at the beginning and then branching to the suitable code variant.

The expected benefit from successful recognition is large. Beyond automatic parallelization, the abstraction from specific data structures for the sparse matrices may also support program maintenance and debugging, and could help with the exchange of one data structure for a sparse matrix against another, more suitable one. For instance, recognized operations on sparse matrices could be replaced by their counterparts on dense matrices, and thus, program comprehension may serve as a front end to [2]. Or, the information derived by concept recognition may just be emitted as mathematical formulas e.g. in LaTeX format, typeset in a mathematical textbook style, and shown in a graphical editor as annotations to the source code, in order to improve human program understanding.

In the sequel we will focus on sparse matrix computations coded by indirect array accesses. This spares us considering pointer alias analysis first. Furthermore, it appears that we can reuse some techniques from our earlier PARAMAT project [5] more straightforwardly for indirect array accesses than for pointer accesses.

The remainder of this article is organized as follows: Section 2 summarizes common storage schemes for sparse matrices; readers who are familiar with these may skip this section. Section 3 gives a preliminary list of common concepts in sparse matrix computations, which also covers the corresponding operations on dense matrices. Section 4 discusses the technical problems involved in recognition of these concepts. We close with related work and conclusions.

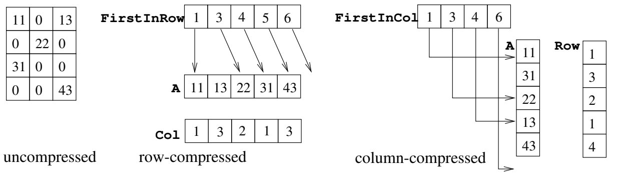

Figure 1: Row–compressed (CSR, CUR) and column–compressed (CSC, CUC) storage formats for sparse matrices.

2 Some common data structures for sparse matrices

There are several competing data structures for sparse matrices (with irregular sparsity pattern); each of them has advantages and disadvantages, and none of them has managed to constitute itself as *the* ideal data structure for a sparse matrix.

In this section we focus on some general data structures for irregular sparsity patterns, that use index vectors to represent sparse matrices. These constructs are typical for most Fortran77 implementations. Formats for special, more regular sparsity patterns, such as for band matrices, block sparse matrices, or skyline matrices, are not considered here. The abbreviations of format names are partially adapted from [13]. More details can be found in [13], [14], and [15].

- **DNS** (*dense storage format*): uses a two–dimensional array `A(N,M)` to store all elements. Due to the symmetric access structure of the two–dimensional array, a *leading dimension* flag `ld` tells us whether the matrix is stored row–major or column–major. In the following, we summarize all data referring to the dense matrix as an object

```
MDNS{ A, N, M, ld }
```

  Example: In Fortran, DNS–Matrix–vector multiplication may look like

```
DO i = 1, N
   b(i) = 0.0
   DO j = 1, N
      b(i) = b(i) + A(i,j) * x(j)
   ENDDO
ENDDO
```

- **COO** (*coordinate format*): A data array `A(NZ)` stores the `NZ` nonzero matrix elements in arbitrary order, and integer vectors `row(NZ)` and `col(NZ)` hold for each nonzero element its row and column index. The object representing the matrix is summarized as

```
MCOO{ A, row, col, N, NZ }
```

  Example: COO–Matrix–vector multiplication may look like

```
DO i = 1, N
   b(i) = 0.0
ENDDO
DO k = 1, NZ
   b(row(k)) = b(row(k)) + A(k) * x(col(k))
ENDDO
```

  The COO format occurs e.g. in the SLAP package [12].

- **CSR** (*row–compressed sorted storage format*): A data array `A(NZ)` stores the `NZ` nonzero matrix elements a_{ij} in row–major order, where within each row the elements appear in the same order as in the dense equivalent. An integer vector `Col(1:NZ)` gives the column index for each element in `A`, and an integer vector `FirstInRow(1:N+1)` gives indices to `A` such that FirstInRow(i) denotes the position in `A` where row i starts, $i = 1, ..., N$ and `FirstInRow(N+1)` always contains `NZ+1` (see Figure 1). Thus, FirstInRow($i+1$)-FirstInRow(i) gives the number of nonzero elements in row i. A CSR matrix object is summarized as

```
MCSR{ A, FirstInRow, Col, N, NZ }
```

  Example: CSR–Matrix–vector multiplication may look like

```
DO i = 1, N
   b(i) = 0.0
   DO k = FirstInRow(i), FirstInRow(i+1)-1
      b(i) = b(i) + A(k) * x(Col(k))
   ENDDO
ENDDO
```

  CSR is used e.g. in the SLAP package [12].

- **CUR** (*row–compressed unsorted storage format*): like CSR, but the order of nonzeros within each row is not important. CUR is used e.g. as the basic format in SPARSKIT [13]. CUR matrix–vector multiplication looks identical to the CSR version.

- **XSR / XUR**: an extension of CSR / CUR that additionally stores in an n–element integer array

`LastInRow` for each compressed row its last index within the data array `A`. This makes row interchanges and row reallocations due to fill–in more efficient. XUR is used e.g. in Y12M [16].

- **MSR** (*modified row–compressed storage format*): like CSR, but the elements of the main diagonal of the matrix are stored separately and regardless of whether they are zero or not. This is motivated by the fact that, often, most of the diagonal elements are a priori known to be nonzero, and are accessed more frequently than the other elements. Typically the diagonal elements are stored in the first `N` elements of `A` and `A(N+1)` is unused. The column indices of the diagonal elements need not be stored, thus the elements of the array `FirstInRow` of CSR are stored in the first `N+1` entries of a two–purpose integer array `FRCol`. The remaining nonzero elements are stored in `A(N+2:NZ+1)` and their column indices in `FRCol(N+2:NZ+1)`. A MSR matrix object is thus given as

  ```
  MMSR{ A, FRCol, N, NZ }
  ```

  MSR is used e.g. in the sparse matrix routines of the *Numerical Recipes* [17].

  Example: Matrix–vector multiplication may look as follows (routine `sprsax()` from [17]):

  ```
  DO i = 1, N
      b(i) = A(i) * x(i);
      DO k = FRCol(i), FRCol(i+1)-1
         b(i) = b(i) + A(k) * x(FRCol(k))
      ENDDO
  ENDDO
  ```

- **CSC** (*column–compressed format*): similar to CSR where the `A` contains the nonzero elements in column–major order and the other two arrays are defined correspondingly (see Figure 1). Thus, CSC format for a matrix A is equivalent to the CSR format for A^T, and vice versa. A CSC matrix object is summarized by

  ```
  MCSC{ A, FirstInCol, Row, N, NZ }
  ```

  Example: CSC–Matrix–vector multiplication may look like

  ```
  DO i = 1, N
     DO k = FirstInCol(i), FirstInCol(i+1)-1
        b(Row(k)) = b(Row(k)) + A(k) * x(i)
     ENDDO
  ENDDO
  ```

  CSC is used e.g. in the Harwell `MA28` package [9])

- **MSC** (*modified column–compressed storage format*): A MSC matrix object

  ```
  MCSC{ A, FCRow, N, NZ }
  ```

  is similar to the CSC representation, but the elements of the main diagonal of the matrix are stored separately, as for MSR.

- **JAD** (*jagged diagonal format*): First the rows of the matrix are permuted to obtain decreasing numbers n_i of nonzero elements for each row i. The data array `A(1:NZ)` is filled as follows: The first nonzero element of each row i (the first "jagged diagonal") is stored in `A(`i`)`, the second nonzero element of each row i in `A(N+`i`)` etc. The overall number `NJD` of jagged diagonals is at most `N`. An integer array `Col(1:NZ)` holds the column index of each element in `A`. An integer array `FirstInJDiag(1:N)` holds indices into `A` resp. `Col` indicating the beginning of a new jagged diagonal; thus `FirstInJDiag(`k+1`)-FirstInJDiag(`k`)` gives the number n_k of elements belonging to the kth jagged diagonal. Thus, a JAD matrix object is given by

  ```
  MJAD{ A, FirstInJDiag, Col, N, NZ, NJD }
  ```

  Example: JAD–Matrix–vector multiplication may look like

  ```
  DO r = 1, N
     b(r) = 0.0
  ENDDO
  DO k = 1, NJD
     DO i = FirstInJDiag(k), FirstInJDiag(k+1)-1
        r = i - FirstInJDiag(k)
        b(r) = b(r) + A(i) * x(Col(i))
     ENDDO
  ENDDO
  ```

  followed by re–permutation of vector `b` if necessary.

- **LNK** (*linked list storage format*): The data array `A(1:MAXNZ)` holds the `NZ` nonzero elements in arbitrary order, the integer array `Col(1:MAXNZ)` gives the column index of each nonzero element. An integer array `NextInRow(1:MAXNZ)` links the elements belonging to the same row in order of increasing `Col` index. A zero `NextInRow` entry marks the last nonzero element in a row. The list head element of each row i is indexed by the ith element of the integer array `FirstInRow(1:N)`. Empty rows are denoted by a zero `FirstInRow` entry. If required by the application, a similar linking may also be provided in the other dimension, using two more index vectors `NextInCol(1:NZ)` and `FirstInCol(1:N)`. Thus, a singly–linked LNK matrix object is summarized by

  ```
  MLNK{A,FirstInRow,NextInRow,Col,N,NZ,MAXNZ}
  ```

  Example: LNK–Matrix–vector multiplication may look like

```
DO i = 1, N
   b(i)=0.0
   k = FirstInRow(i)
   WHILE (k.GT.0)
      b(i) = b(i) + A(k) * x(Col(k))
      k = NextInRow(k)
   ENDWHILE
ENDDO
```

The LNK format requires more space than the previously discussed sparse matrix formats, but it supports efficient dynamic insertion and deletion of elements (provided that `A` and `NextInRow` have been allocated with sufficient space reserve, `MAXNZ`).

While matrix–vector multiplication codes for a sparse matrix look quite simple and seem to be somehow identifiable by concept matching techniques, implementations of matrix–matrix multiplication or LU decomposition look quite unstructured. This is mainly due to the fact that in the course of these algorithms, some matrix elements may become nonzero which were originally zero (*fill–in*), and thus additional storage has to be allocated for inserting them. Thus, the sparsity pattern may change in each step of these algorithms, while at matrix–vector multiplication, the sparsity pattern (and thus, the organizational variables) is read–only.

A simple work–around to cope with a limited number of fill–ins is to store fill–ins in a separate temporary data structure, or respectively, to allocate slightly more space for the data array and the index vectors. This is e.g. applied in `SPARSE` [11].

There are also many possibilities for slight modifications and extensions of these data structures. For instance, a flag may indicate symmetry of a matrix. Such changes are quite ad–hoc, and it seems generally not sensible to define a new family of concepts for each such modification. As an illustrating example, in the Harwell routines `MA30`, the sign bit of the row resp. column indices is "misused" to indicate whether a new column or row has just started, thus saving the `FirstInRow` resp. `FirstInCol` array when sequentially scanning through the matrix. Clearly such dirty tricks make program comprehension more difficult.

A main consequence that arises from these data structures is that the comfortable symmetry present in the two–dimensional arrays implementing dense matrices (DNS) is lost. Thus, we must explicitly distinguish between transposed and non–transposed matrix accesses, i.e. row–wise and column–wise linearization of the storage for the nonzero matrix elements.

Linked list data structures (e.g., the LNK format) cause operations on them, such as traversal or insert/delete, to be inherently sequential. Thus they are particularly good candidates to be completely replaced by other data structures more suitable for exploiting parallelism, e.g. linked lists with multiple heads for parallel access. Data structure replacement for a sparse matrix is possible if all operations on it have been recognized and if alias analysis can guarantee that there are no other variables which may be used to access one of these linked list elements in an unforeseen way.

3 Some common concepts in (sparse) matrix computations

The survey of concepts given in this section is far from being exhaustive. It should be regarded rather as an illustration of our intention. The formulation of the final list of frequently occurring concepts for a representative bunch of numerical software is the subject of on-going research.

The example computations given in each of the following subsections serve only as an illustration; there may be many more possibilities to express these concepts. E.g., loops may be reversed, unrolled, blocked, distributed or interchanged, temporary variables may be used, etc. It is the job of program comprehension to abstract from such syntactical variations.

The naming conventions for concepts are as follows: The shape of operands is denoted by shorthands S (scalar), V (vector), M (matrix), and VX (indexed vector operands). The result shape is given first, followed by a mnemonic for the type of computation denoted by the concept, and the shorthands of the operands.

We extend our earlier approach [5] to representing concepts and concept instances in two aspects:

- *Concepts are more general.*

  In previous work we had to define one pattern for each operation, even if the remaining structure of computation, type, size, and number of operands was the same. For instance, the former concept VADD denoting elementwise addition of two floatingpoint vectors of size N each was different from that denoting elementwise multiplication (VMUL). Our earlier organization of concepts (called "patterns" [5]) was thus rather oriented towards library routines like the Fortran90 intrinsics.

  Now we abstract from such differences by making the operator a parameter of a more general concept called VMAPVV. The usage of such nestable "meta–patterns" is inspired by the work of Cole on algorithmic skeletons [18]. This makes hierarchical program comprehension slightly more complicated, but greatly reduces the number of different concepts, and allows for a more lean code generation interface. Nevertheless, there remain "flat" concepts for important special cases, e.g. SCOUNTV (count elements fulfilling a condition), VDOTVV for dot product, VMATVECMV for matrix–vector multiplication, etc. These can be regarded as "syntactic sugar", as they may as well be expressed using other concepts, (e.g. `SCOUNTV(.gt.3.14,u(1:N))` may be expressed as

SREDV(ADD,VMAPVS(.gt., u(1:N), 3.14))) but are to be preferred because they enhance readability and speed up the program comprehension process.

Also in contrast to [5], we do not distinguish any more explicitly between in–place and not–in–place computations in the concept name. This abstracts even further from the particular implementation. It will be the job of the back-end to reuse (temporary array) space where possible[1]. In other words, we try to track *values* of objects rather than memory locations. Where it is unavoidable to have accumulating concepts, they can be specified using accumulative basic operations like the increment operators.

- *Concept instances can be nested.*

  Nesting reflects the hierarchy of computation in a straightforward way. [2] The internal representation of the annotation computed for a recognized part of code is a directed acyclic graph (DAG) of concept instances, very similar to expression DAGs for scalar computations.

  Also, our notation of concept instances is now given in a more functional style. This means that the result parameter(s) of a concept instance may also appear on the left hand side of an assignment of a concept instance, rather than as its first parameter.

  As an example, we may denote a DAXPY–like computation as

```
VMAPVS( tmp(1:N), MUL, c(1:N), 3.1415 )
VMAPVV( b(1:N), INCR, b(1:N), tmp(1:N) )
```

  which is closer to the internal representation of a compiler, and unavoidable if the hierarchy of computation forms a DAG rather than a tree, or as

```
b(1:N) = VMAPVV( INCR, b(1:N),
           VMAPVS( MUL, c(1:N), 3.1415))
```

  which is more readable for humans.

For better readability, array access strides other than +1 are not shown in the following lists, but a generalization is straightforward.

3.1 Concepts for scalar computations

Assignments to scalars are either SCOPY where the assignee is a variable, or SINIT where the assignee is a constant. Furthermore, FSUM stands for iterative calculations that compute a scalar value, and SWAP denotes a scalar swap. For more details, see [5].

[1] Note that this problem corresponds to register allocation applied to (sections of) arrays rather than scalars. [19]

[2] In previous work we always had to use explicitly declared temporary variables to simulate this.

For the binary expression operators there are the concepts ADD, MUL, ..., MAX, MIN, ..., EQ, NE, GT, GE, etc.; for the unary expression operators we have NEG for negation, ABS for absolute value, INV for reciprocal, SQR for squaring, SQRT for square root, LOG for logarithm, SIN for sine, etc. The commutative and associative operators ADD, MUL, MAX, MIN, OR, AND may also have more than two operands. STAR is a special version of a multi–operand ADD denoting difference stencils [5]. The increment operators INCR (for accumulating addition) and SCAL (for accumulating product) are used instead of ADD or MUL where the result variable is identical to one of the arguments.

In the sequel, $\oplus$ stands for any binary operator (concept) and $\ominus$ for any unary operator. $\otimes$ denotes associative binary operators. $\odot$ stands for MAX or MIN.

3.2 Concepts for elementwise computations on vectors

$v_0(s_0:t_0) = \texttt{VMAPVV}(\oplus, v_1(s_1:t_1), v_2(s_2:t_2))$
elementwise application of binary operator $\oplus$

$v_0(s_0:t_0) = \texttt{VMAPV}(\ominus, v_1(s_1:t_1))$
elementwise application of unary operator $\ominus$

$v_0(s_0:t_0) = \texttt{VMAPVS}(\oplus, v_1(s_1:t_1), r)$
elementwise application with one scalar operand r

$v_0(s_0:t_0) = \texttt{VCOPYV}(v_1(s_1:t_1))$ copy vector elements

$v_0(s_0:t_0) = \texttt{VINIT}(c)$
initialize vector elements by a constant c

$v_0(s_0:t_0) = \texttt{VASGNS}(r)$
initialize vector elements by a scalar r

$r = \texttt{SREDV}(\otimes, v_1(s_1:t_1))$ $\quad r = \bigotimes_{j=s_1}^{t_1} v_1(j)$

$k = \texttt{SREDLOCV}(\odot, v_1(s_1:t_1))$
some k with $v_1(k) = \bigodot_{j=s_1}^{t_1} v_1(j)$

$v_0(s_0:t_0) = \texttt{VPREFV}(\otimes, v_1(s_1:t_1))$
$v_0(i) = \bigotimes_{j=s_1}^{s_1-s_0+i} v_1(j),\ i = s_0, ..., t_0$

$v_0(s_0:t_0) = \texttt{VSUFFV}(\otimes, v_1(s_1:t_1))$
$v_0(i) = \bigotimes_{j=s_1-s_0+i}^{t_1} v_1(j),\ i = s_0, ..., t_0$

3.3 Searching and sorting on a vector

$k = \texttt{SSRCHV}(v_1(s_1:t_1), r)$ return rank of r in v_1

$v_0(s_1:t_1) = \texttt{VSORTV}(v_1(s_1:t_1))$ sort v_1

$v_0(n_0+1:n) = \texttt{VCOLLVV}(v_1(s_1:t_1), v_2(s_1:t_1), n_0)$
extract all $n-n_0$ elements $v_2(i)$ with $v_1(i) \neq 0$

$\texttt{VSWAPVV}(v_1(s_1:t_1), v_2(s_2:t_2))$ swap vectors v_1 and v_2

3.4 Indexed vector operations

$v_0(s_0:t_0) = \texttt{VGATHERVX}(v_1(:), v_2(s_2:t_2))$
$v_0(i-s_2+s_0) = v_1(v_2(i)),\quad i = s_2, ..., t_2$

$\texttt{VXSCATTERV}(v_0, v_1(s_1:t_1), v_2(s_2:t_2)$
$v_0(v_1(i-s_2+s_1)) = v_2(i),\quad i = s_2, ..., t_2$

3.5 Concepts for elementwise computations on matrices

In the following list, m_i for $i = 0, 1, 2, ...$ stands for matrix objects MXXX{ ... } in some format XXX.

$m_0 =$ MMAPMM$(\oplus, m_1, m_2)$
elementwise application of binary operator $\oplus$
$m_0 =$ MMAPM$(\ominus, m_1)$
elementwise application of unary operator $\ominus$
$m_0 =$ MMAPMV$(\oplus, m_1, v_2, d_2)$ map $\oplus$ across dim. d_2 of m_1
$m_0 =$ MMAPMS$(\oplus, m_1, r)$
elementwise apply $\oplus$ of r and any element of m_1
$m_0 =$ MMAPVV$(\oplus, v_1, d_1, v_2, d_2)$
map $\oplus$ across $v_1 \times v_2$, spanning dim.'s d_1, d_2 of m_0
$m_0 =$ MCOPYM(m_1) matrix copy, same format for m_0
$m_0 =$ MCNVTM(m_1)
like MCOPYM, but formats of m_0, m_1 differ
$m_0 =$ MEXPANDV(v_1, d_1)
blow up vector v_1 to a matrix along dimension d_1
$m_0 =$ MTRANSPM(m_1) matrix transpose
$m_0 =$ MINITS(r)
initialize all elements of m_0 by scalar expression r
$m_0 =$ MASGNS(r, i, j)
initialize all elements of m_0 by expression r indexed by a formal row index i and/or column index j

Note that outer product (MOUTERVV) is a special case of MMAPVV.

3.6 Searching and sorting on a matrix

In the following list, rv_1 denotes a matrix row RXXX{..} in some format XXX.

$m_0 =$ MCOLLM(m_1, f, i, j) filter out all elements $m_1(i, j)$ fulfilling a boolean condition $f(m_1, i, j)$, parameterized by formal row index i and/or formal column index j
$m_0 =$ MGETSUBM$(m_1, s_1, t_1, s_2, t_2)$ extract rectangular submatrix of m_1 in range $(s_1 : t_1, s_2 : t_2)$
$m_0 =$ MSETSUBM$(m_1, s_1, t_1, s_2, t_2, m_2)$
replace submatrix $m_1(s_1 : t_1, s_2 : t_2)$ by m_2
$r =$ SGETELM(m_1, i, j)
extract element $m_1(i, j)$ if it exists, and 0 otherwise
$m_0 =$ MSETELMS(m_1, i, j, r) set element $m_1(i, j)$ to r
$v_0 =$ VGETROWM(m_1, i) extract row i from matrix m_1
$m_0 =$ MSETROWMV(m_1, i, rv_1) set row i in matrix m_1 to rv_1
$v_0 =$ VGETCOLM(m_1, i) extract column i from a matrix
$m_0 =$ MSETCOLMV(m_1, i, cv_1)
set column i in matrix m_1 to cv_1
$v_0 =$ VGETDIAM(m_1, i) extract diagonal i from a matrix
$m_0 =$ MSETDIAMV(m_1, i, v_1)
set diagonal i in matrix m_1 to v_1
$m_0 =$ MGETLM(m_1) extract left lower triangular matrix (including diagonal) from m_1
$m_0 =$ MGETUM(m_1) extract right upper triangular matrix (including diagonal) from m_1
$m_0 =$ MPRMROWM(m_0, v_1)
permute rows of matrix m_0 with permutation vector v_1
$m_0 =$ MPRMCOLM(m_0, v_1)
permute columns of m_0 with permutation vector v_1

3.7 Matrix-vector and matrix-matrix product, decompositions

$v_0 =$ VMATVECMV(m_1, v_2)
matrix-vector product $v_0 = m_1 \cdot v_2$.
$v_0 =$ VVECMATMV(m_1, v_2)
vector-matrix product $v_0 = m_1^T \cdot v_2$.
$m_0 =$ MMATMULMM(m_1, m_2)
matrix-matrix-product $m_0 = m_1 \cdot m_2$
$v_0 =$ VUSOLVEMV(m_1, v_2)
backward subst. $v_0 = m_1^{-1} \cdot v_2$, m_1 upper triangular
$v_0(s_0 : t_0) =$ VLSOLVEMV$(m_1, v_2(s_2 : t_2))$
forward subst. $v_0 = m_1^{-1} \cdot v_2$, m_1 lower triangular
$v_0 =$ VROTVVM(v_0, v_1, m_2) Givens rotation
$\{m_0, m_1\} =$ MMLUDM(m_2, p, t) LU decomposition of m_2, pivot strategy p, drop tolerance t
$v_0 =$ VUPDROWM$(\oplus, m_1, pr, i, c, space, droptol)$
update row i of m_1 in LU decomp. for pivot row pr, start column c, dense result vector of size *space*

In order to express a transposed matrix-matrix product, the MTRANSP concept has to be applied to the operand matrix to be accessed in transposed order[3]. For dense matrices this can be skipped by toggling the leading dimension.

It is interesting to note that a matrix-vector multiplication for a matrix in CSR format (VMATVECMV(MCSR{...}, ...)) looks exactly like a transposed matrix-vector multiplication for CSC format (VVECMATMV(MCSC{...}, ...)) and vice versa. Furthermore, for matrix-vector product the order of nonzero elements within the same row resp. column is not important here, thus the concept variants for CSR and CUR resp. CSC and CUC matrices are equivalent. Thus, for each such pair of equivalent concept variants only one common implementation is required for the back–end.

3.8 I/O concepts

READ and WRITE are the scalar concepts for reading and writing to a file.

$v(s : t) =$ VREAD(F) read a $(t - s + 1)$–element vector v consecutively from file F
VWRITE$(v(s : t), F)$ write a $(t - s + 1)$–element vector v consecutively to file F
$m =$ MREAD(F, f) read a matrix m from file F in file storage format f
MWRITE(m, F, f) write a matrix m to file F in file storage format f

There are various file storage formats in use for sparse matrices, e.g. the Harwell–Boeing file format, the array format, or coordinate format [20].

[3]The reason why we do not define three more concepts for the combinations of transposed operand matrices is that executing a transpose, if not avoidable, is one order of magnitude less costly than a matrix-matrix product, while the execution time of a transpose is in the same order as a transposed matrix-vector product.

3.9 Exception slots

For some of the concepts listed above there exist additional slots containing actions specified by the programmer to cover cases when possible exceptions occur. For example, the INV concept (scalar reciprocal) offers a slot to enter a statement that handles the "division by zero" exception. As another example, for LU decomposition (LUD) on a sparse operand matrix an exception slot indicates what should be done if the allocated space is exceeded.

4 Static and dynamic matching of sparse matrix concepts

Safe identification of a sparse matrix operation consists of (1) a test for the syntactical properties of this operation, which can be performed by concept recognition at compile time, and (2) a test for the dynamic properties which may partially have to be performed at run time. Regarding (parallel) code generation, this implies that two versions of code for the corresponding program fragment must be generated: one version branching to an optimized sparse matrix library routine if the test is positive, and a conservative version (maybe using the inspector–executor technique, or just sequential) that is executed otherwise.

4.1 Compile–time concept matching

Testing the static part is, in principle, not hard. Just in the same way as we did for dense matrix operations [5], we work bottom–up on the abstract syntax tree, incrementally compute dataflow relations ("cross edges"), and deterministically identify concepts based on already matched subconcepts. For efficiency reasons, the matching rules are hierarchically organized ("concept hierarchy graph"), following the natural hierarchical composition of computations by applying loops and sequencing to subcomputations. Normalizing transformations, such as rerolling of unrolled loops, are done whenever applicable.

4.1.1 Vertical matching

Vertical matching proceeds along the hierarchical nesting structure (statements, expressions) of the program. It uses only control dependence information which may be given by a control flow graph or derived from the abstract syntax tree. For languages with strictly block–structured control flow, control dependence information can be directly obtained from the abstract syntax tree. Data flow information is not required for vertical matching. A program fragment matched by a concept P is annotated by a summary node ("concept instance") looking similar to a call to an externally defined function P(). The formal parameters ("slots") of the summary node are bound to the corresponding program objects occuring in the code fragment. If we are working on the abstract syntax tree, the summary node holds all information which would be required to reconstruct an equivalent of the subtree it annotates.

As an example, consider the following program fragment:

```
    DO i = 1, N
S1:    b(i) = 0.0
       DO j = first(i), first(i+1)-1
S2:       b(i) = b(i) + A(j) * x(col(j))
       ENDDO
    ENDDO
```

The syntax tree is traversed bottom–up from the left to the right. Statement S1 is recognized as a scalar initialization, summarized as SINIT(b(i),0.0). Statement S2 is matched as a scalar update computation, summarized as INCR(b(i), MUL(A(j),x(col(j)). Now the loop around S2 is considered. The index expressions of A and col are bound by the loop variable j which ranges from some loop–invariant value first(i) to some loop–invariant value first(i+1)-1. Thus the accesses to arrays A and col during the j loop can be summarized as vector accesses A(first(i):first(i+1)-1) and col(first(i):first(i+1)-1). The entire j loop is matched as

```
INCR( b(i), SREDV( ADD,
       VMAPVV( MUL, A(first(i):first(i+1)-1),
        VGATHER( x(:), col(first(i):first(i+1)-1))))
```

for which a special flat concept exists, namely SDOTVVX, dot product with one indexed operand vector. The unparsed program is now

```
    DO i = 1, n
S1':   b(i) = SINIT( 0.0 );
S2':   INCR(b(i), SDOTVVX(A(first(i):first(i+1)-1),
                  x(:),col(first(i):first(i+1)-1)))
    ENDDO
```

Although all statements in the body of the i loop are matched, there is no direct way to match the i loop at this point. We must first address the dataflow relations between S1' and S2':

4.1.2 Cross matching

Cross matching tries to merge several matched summary nodes v_1, v_2, ... belonging to the body of the same construct (which is, mostly, a loop header). If there is a common concept that covers the functionality of, say, v_i and v_j, there is generally some data flow relation between v_i and v_j which can be used to guide the matching process. For each summary node we consider the slot entries to be read or written, and compute data flow edges (also called cross–edges) connecting slots referring to the same value, e.g., Def–Use chains ("FLOW" cross edges).

Continuing on the example above, we obtain that the same value of `b(i)` is written (generated) by the `SINIT` computation in `S1'` and consumed (used and killed) by the `INCR` computation in `S2'`. Note that it suffices to consider the current loop level: regarding cross matching, the values of outer loop variables can be considered as constant. Cross matching, following the corresponding matching rule, merges the two summary nodes to a node

```
       b(i) = ADD( 0.0, SDOTVVX( .... ))
```

and after constant folding we obtain

```
    DO i = 1, N
S'':   b(i) = SDOTVVX( A(first(i):first(i+1)-1),
               x(:), col(first(i):first(i+1)-1)))
    ENDDO
```

4.2 Speculative concept matching and run time tests

In order to continue with this example, we now would like to apply vertical matching to the `i` loop. The accesses to `A` and `col` are supposed to be CSR matrix accesses because the range of the loop variable `j` binding their index expressions is controlled by expressions bound by the `i` loop. Unfortunately, the values of the `first(:)` elements are statically unknown. Thus it is impossible to definitively conclude that this is an occurrence of a CSR matrix vector product.

Nevertheless we continue, with assumptions based on syntactic observations only, concept matching in a *speculative* way. We obtain

```
      <assume first(1)=1>
      <assume monotonicity of first in 1:N+1>
      <assume injectivity of col
       in first(i):first(i+1)-1 forall i in 1:N>
S''': b(1:N)=VMATVECMV( MCSR{A(first(1):first(N+1)),
            first(1:N),col(first(1):first(N+1)-1),
            N, first(N+1)-1 }, x(1:N) );
```

where the first three lines summarize the assumptions guiding our speculative concept recognition. If they cannot be statically eliminated, these three assumptions will result in three run time tests being executed immediately before or during speculative parallel execution of `S'''` as a CSR matrix vector product. The range of the values in `col` needs not be bound–checked at run time since we can safely assume that the original program runs correctly in sequential.

We now look at these run time tests in more detail.

We call an integer vector `iv` *monotonic* over an index range $[L:U]$ at a program point q iff for any control flow path through q, $\texttt{iv}(i) \leq \texttt{iv}(i+1)$ holds at entry to q for all $i \in [L:U-1]$.

We call an integer vector `iv` *injective* over an index range `L:U` at a program point q iff for any control flow path through q, for all $i, j \in$`L:U` holds $i \neq j \Longrightarrow \texttt{iv}(i) \neq \texttt{iv}(j)$ at entry to q. Injectivity of a vector is usually not statically known, but is an important condition that we need to check at various occasions.

We must verify the speculative transformation and parallelization of a recognized computation on a set of program objects which are strongly suspected to implement a sparse matrix A. This consists typically of a check for injectivity of an index vector, plus maybe some other checks on the organizational variables. For instance, for non–transposed and transposed sparse matrix–vector multiplication in CSR or CUR row–compressed format, we have to check that

1. `first(1)==1`,
2. `first(:)` is monotonic in the index range $1:N+1$ and
3. array `col()` is injective in the index range $\texttt{first}(i)$:$\texttt{first}(i+1)-1$ for all i in the range $1,...,N$

These properties may be checked for separately.

4.3 Speculative loop distribution

Loop distribution is an important normalization applied in the concept recognizer. As an example, consider the following code fragment taken from the `SPARSE-BLAS` [10] routine `DGTHRZ`:

```
   DO 10 I = 1, NZ
       X(I)        = Y(INDX(I))
       Y(INDX(I)) = 0.0D0
10 CONTINUE
```

In order to definitely recognize (and also in order to parallelize) this fragment, we need to know the values of the elements of array `INDX`. Unfortunately, this information is generally not statically available. But similar as for the speculative recognition of sparse matrix operations we speculatively assume that `INDX` is injective in the range `1:NZ`. As now there remain no loop–carried dependencies, we can apply loop distribution [3] to the `I` loop:

```
   <assume injectivity of INDX(1:NZ)>
   DO I = 1, NZ
       X(I) = Y(INDX(I))
   ENDDO
   DO I = 1, NZ
       Y(INDX(I)) = 0.0D0
   ENDDO
```

Applying concept matching to each loop separately makes the speculatively matched copy of the program segment look as follows:

```
<assumes injectivity of INDX(1:NZ)>
X(1:NZ) = VGATHER( Y(:), INDX(1:NZ) )
VSCATTER( Y(:), INDX(1:NZ), VINIT(0.0) )
```

Speculative loop distribution saves the original program structure for the code generation phase. This allows to generate also the conservative code variant.

4.4 Preservation and propagation of format properties

Even if at some program point we are statically in doubt about whether a set of program objects really implements a sparse matrix in a certain storage format, we may derive static information about some format properties of a speculatively recognized concept instance.

For any concept (or combination of a concept and specific parameter formats) the format property requirements for its parameter matrices are generally known. If an instance I of a concept c generates a new (sparse) result matrix m, it may also be generally known whether m will have some format properties after execution of I. Such a property π of m may either hold in any case after execution of an instance of c, i.e. $\pi(m)$ is installed by c. Or, π may depend on some of the actual format properties $\pi_1, \pi_2, ...$ of the operand matrices $m_1, m_2, ...$ In this case, $\pi(m)$ will hold after execution of I only if $\pi_1(m_1)$, $\pi_2(m_2)$ etc. were valid before execution of I. In other words, this describes a propagation of properties $\pi_1(m_1) \wedge \pi_2(m_2) \wedge ... \rightarrow \pi(m)$. Also, it is generally known which properties of operand matrices may be (possibly) deleted by executing an instance of a concept c.

The assumptions, preservations, propagations and deletions of format properties associated with each concept instance are to be summarized by the program comprehension engine in the form of program annotations, as already shown for the previous example. Note that the preservations are the complementary set of the deletions; thus we renounce on listing them for better space economy. If existing objects may be overwritten by a concept instance, their old properties are clearly deleted. Note that the `install` and `propagate` annotations refer to the newly created values. The shorthand `all` stands for all properties considered.

For example, if a certain piece of program has been speculatively recognized as an occurrence of a CSC to CSR conversion concept, annotations are (conceptually) inserted before the concept instance as follows:

```
<assume FirstB(1)=1>
<assume monotonicity of FirstB in 1:M>
<assume injectivity of RowB
 in FirstB(i):FirstB(i+1) forall i in 1:M>
<delete all of FirstA>
<delete all of ColA>
<install FirstA(1)=1>
<propagate (monotonicity of FirstB in 1:M)
 then (monotonicity of FirstA in 1:N)>
<propagate (monotonicity of FirstB in 1:M)
 and (injectivity of RowB in ...)
 then(injectivity of ColA in FirstA(i):FirstA(i+1)
       forall i in 1:N)>
MCSR{A,FirstA,ColA,N,NZ} =
              MCNVTM( MCSC{B,FirstB,RowB,M,NZ} )
```

4.5 Placing run time tests

Program points are associated with each statement or concept instance, i.e. with the nodes in the control flow graph after concept matching. In an implementation all properties π of interest for all arrays or array sections A of interest may be stored for any program point q in bitvectors

$\mathit{ASSUME}_{\pi,A}(q) = 1$ iff $\pi(A)$ is `assumed` to hold at entry to q

$\mathit{DELETE}_{\pi,A}(q) = 1$, iff $\pi(A)$ may be `deleted` by execution of q

$\mathit{INSTALL}_{\pi,A}(q) = 1$ iff $\pi(A)$ is `installed` by execution of q. — Propagations are represented by sets

$\mathit{PROPAGATE}_{\pi,A}(q)$ containing all properties π_j of arrays A_j that must hold at entry to q in order to infer $\pi(A)$ at exit of q. — Moreover, we denote by

$\mathit{TEST}_{\pi,A}(q)$ whether a run time test of property π of array section A has been scheduled immediately before q. When starting the placement of run time tests, all $\mathit{TEST}_{\pi,A}(q)$ are zero.

For the placement of run time tests we compute an additional property

$\mathit{HOLD}_{\pi,A}(q)$ which tells whether $\pi(A)$ holds at entry of q. We compute it by a standard data flow technique (see e.g. [3]) iterating over the control flow graph $G = (V, E)$ of the program, using the following data flow equation:

$$\begin{aligned} \mathit{HOLD}_{\pi,A}(q) &= \mathit{TEST}_{\pi,A}(q) \\ &\vee \bigwedge_{(q',q)\in E} (\mathit{HOLD}_{\pi,A}(q') \wedge \neg \mathit{DELETE}_{\pi,A}(q') \\ &\qquad\qquad \vee \mathit{INSTALL}_{\pi,A}(q')) \\ &\vee \bigwedge_{(q',q)\in E} \quad \bigwedge_{(\pi',A')\in \mathit{PROPAGATE}_{\pi,A}(q')} \mathit{HOLD}_{\pi',A'}(q') \end{aligned} \tag{1}$$

For the data flow computation of HOLD, we initialize all HOLD entries by 1. Since the DELETE, $\mathit{INSTALL}$, $\mathit{PROPAGATE}$ and TEST entries are constants for each π, A, and q, the sequence of the values of $\mathit{HOLD}_{\pi,A}(q)$ during the iterative computation is monotonically decreasing and bounded by zero, thus the data flow computation converges.

Clearly, after all necessary run time tests have been placed, $\mathit{HOLD}_{\pi,A}(q)$ must fulfill

$$HOLD_{\pi,A}(q) \geq ASSUME_{\pi,A}(q) \quad \text{for all } \pi, A, q$$

in order to ensure correctness of the speculative program comprehension. Thus we arrive, as a very general method, at the following simple nondeterministic algorithm for placing run time tests:

Algorithm: *placing run time tests*
(1) **for all** π and A of interest **do**
(2) **for all** q **do** $TEST_{\pi,A}(q) = 0$;
(3) **forever do**
(4) initialize $HOLD_{\pi,A}(q') = 1$ for all program points q'
(5) recompute $HOLD_{\pi,A}(q)$ for all program points q according to equation (1)
(6) **if** $HOLD_{\pi,A}(q) \geq ASSUME_{\pi,A}(q)$ for all q **then break**;
(7) set $TEST_{\pi,A}(q') = 1$ for some suitably chosen program point q'
od
od

The goal is to place the run–time tests in step (7) in such a way that the total run time overhead induced by them in the speculatively parallelized program is minimized. A very simple strategy is to place a test for $\pi(A)$ *immediately after* each statement q' killing property π of A, i.e. where $KILL_{\pi,A}(q') = 1$, which is defined as follows:

$$KILL_{\pi,A}(q') := DELETE_{\pi,A}(q') \wedge \neg INSTALL_{\pi,A}(q') \wedge \neg \bigwedge_{(\pi',A')\in PROPAGATE_{\pi,A}(q')} HOLD_{\pi',A'}(q')$$

Of course, this initial setting will typically introduce superfluous tests. For instance, for a sequence of consecutive killings of $\pi(A)$ it is sufficient to schedule a test for $\pi(A)$ only after the last killing program point, provided that $\pi(A)$ may not be assumed to hold at some point within this sequence. Also, a test for $\pi(A)$ after program point q' is superfluous if $\pi(A)$ is not assumed at any point q'' that may be executed after q'. These optimizations can be carried out in a second phase by another data flow framework. [4]

If there are, once a sparse matrix has been generated, no changes to its organizational variables any more during execution of the program, i.e., its sparsity pattern remains static, we can completely eliminate all but one test on monotonicity and injectivity, which is located immediately after constructing its organizational data structures.

[4]Alternatively, one may as well start with a test of $\pi(A)$ being inserted immediately before each program point q with $ASSUME_{\pi,A}(q) \wedge \neg HOLD_{\pi,A}(q)$, and then eliminating all tests in a sequence of uses of $\pi(A)$ but the first one (provided that no kill of $\pi(A)$ may occur in between), which is just symmetric to the strategy described above.

Note that, due to speculative execution, a test can be executed *concurrently* with the subsequent concept implementation even if that is guarded by this test. In particular, possible communication delays of the test can be filled with computations from the subsequent concept implementation, and vice versa. The results of the speculative execution of the concept implementation are committed only if the test process accepts.

4.6 A parallel algorithm for the monotonicity tests

Monotonicity of a vector $x(1:n)$ is very easy to check in parallel. Each of the p processors considers a contiguous slice of x of size at most $\lceil n/p \rceil$. The slices are tested locally; if a processor detects non–monotonicity in its local part, it signals a FAIL and aborts the test on all processors. Otherwise, the values at the boundary to the next upper slice of x are checked concurrently. The test passes if no processor detects non–monotonicity for any pair of neighboring elements.

On a distributed memory architecture, the latter, global phase of this algorithm requires each processor (but the first one) to send the first element of its slice of x to the processor owning the next lower slice; thus each processor (but the last one) receives an element and compares it with the element at the upper boundary of its slice.

4.7 A parallel algorithm for the injectivity tests

A parallel algorithm may reduce the overhead of the injectivity test. For a shared memory parallel target machine we apply an algorithm similar to bucket sort[5] to test injectivity for an integer array `a` of n elements, as it is likely that the elements of `a` are within a limited range (say, $1:m$).[6] We hold a shared temporary array `counter` of m counters, one for each possible value, which are (in parallel) initialized by zero. Each processor k, $k = 1, ..., p$ increments[7] the corresponding counters for the elements $a((k-1)n/p + 1 : kn/p)$. If a processor detects a counter value to exceed 1, it posts a FAIL signal, the test returns FALSE. Otherwise, the test accepts. The test requires m additional shared memory cells. Concurrent write access to the same counter (which may sequentialize access to this location on most shared memory systems) occurs only if the test fails. Thus, the run time is $O((m+n)/p)$.

On a distributed memory system, we use an existing algorithm for parallel sorting of an integer array of size n on a processor network that may be appropriately

[5]A similar test was suggested in [21].

[6]m is to be chosen as a conservative overestimation of the extent of the compressed matrix dimension which is usually not statically known.

[7]This should be done by an atomic fetch&increment operation such as `mpadd(&(counter[a[j]]),1)` on the SB-PRAM, cf. [22].

embedded into the present hardware topology. As result, processor i holds the ith slice of the sorted array, of size n/p. Furthermore, each processor $i > 0$ sends the first element of its slice to its predecessor $i-1$ who appends it as $(n/p+1)$st element to its local slice. Each processor now checks its extended slice for duplicate entries. If the extended slices are injective, then so is the original array. The run time is dominated by the parallel sorting algorithm.

5 Related work

Several automatic concept comprehension techniques have been developed over the last years. These approaches vary considerably in their application domain, purpose, method, and status of implementation.

General concept recognition techniques for scientific codes have been contributed by Snyder [23], Pinter and Pinter [24], Paul and Prakash [25], diMartino [26] and myself [5]. Some work focuses on recognition of induction variables and reductions [27, 28, 29] and on linear recurrences [30, 31]. General techniques designed mainly for non–numerical codes have been proposed by Wills et al. [32] and Ning et al. [33, 34].

Concept recognition has been applied in some commercial systems, e.g. EAVE [35] for automatic vectorization, or CMAX [36] and a project at Convex [37] for automatic parallelization. Furthermore there are several academic applications [5, 26] and proposals for application [38, 39] of concept recognition for automatic parallelization. Today, most commercial compilers for high–performance computers are able to perform at least simple reduction recognition automatically.

A more detailed survey of these approaches and projects can be found e.g. in [5] or [8].

Our former PARAMAT project (1992–94) [5] kept its focus on dense matrix computations only, because of their static analyzability. The same decision was also made by other researchers [24, 26, 37, 40] and companies [36] investigating general concept recognition with the goal of automatic parallelization. According to our knowledge, there is currently no other framework which is actually able to recognize sparse matrix computations in the sense given in this paper.

6 Conclusion and future work

We have described the problems of applying program comprehension techniques to sparse matrix computations with the primary goal of automatic parallelization. We see that it is possible to perform speculative program comprehension where static analysis does not provide sufficient information; in these cases the static tests on the syntactic properties and consistency of the organizational variables are complemented by run–time tests whose placement in the code can be optimized by a static data flow framework. Only if these tests are passed, the target program may continue on the optimized parallel sparse matrix library routine. Otherwise, it must fall back to the conservative code variant. We have also given simple parallel algorithms for verifying monotonicity and injectivity at run time.

Future research will address the compilation of a more complete list of concepts for the operations on sparse matrices that are most often encountered in existing Fortran codes; we will continue our on-going domain analysis. In the same way the syntactic matching rules will be derived. An implementation based on POLARIS is just being started. These issues are the subject of a new research project called SPARAMAT which is funded by DFG and started in late 1997.

In order to maintain a half-way achievable goal in such a small university project, it is necessary to limit oneself to a language that is rather easy to analyze, to only a handful of sparse matrix formats, and to a limited set of most important concepts. For this reason, pointer alias analysis of C programs, as well as concepts and matching rules for pointer–based linked list data structures, are beyond the scope of this project.

However, when faced with distributed memory target architectures, we will have to address automatic array distribution and redistribution. It seems to be a reasonable idea to integrate run–time distribution schemes for sparse matrices [7] into this framework; for instance, the run–time analysis of the sparsity pattern, which is required for load–balancing partitioning of the sparse matrix, may be integrated into the run–time tests.

Acknowledgements The author thanks Prof. Dr. Emilio L. Zapata for an early discussion about the applicability of program comprehension techniques to sparse matrix computations. Prof. Dr. Helmut Seidl contributed some suggestions on specific recognition techniques.

This research is part of the SPARAMAT project funded by DFG.

References

[1] Aart Johannes Casimir Bik. *Compiler support for sparse matrix computations.* PhD thesis, Leiden University, 1996.

[2] Aart J.C. Bik and Harry A.G. Wijshoff. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Trans. on Parallel and Distributed Systems*, 7(2):109–126, February 1996.

[3] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.

[4] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic Data Layout Using 0–1 Integer Programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1993.

[5] Christoph W. Keßler. Pattern-driven Automatic Parallelization. *Scientific Programming*, 5:251–274, 1996.

[6] R. Mirchandaney, J. Saltz, R.M. Smith, D.M. Nicol, and K. Crowley. Principles of run-time support for parallel processors. In *Proc. 2nd ACM Int. Conf. on Supercomputing*, pages 140–152. ACM Press, July 1988.

[7] M. Ujaldon, E.L. Zapata, S. Sharma, and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *Journal of Parallel and Distributed Computing*, 38(2), Nov. 1996.

[8] Beniamino DiMartino and Christoph W. Keßler. Program comprehension engines for automatic parallelization: A comparative study. In I. Jelly, I. Gorton, and P. Croll, editors, *Proc. First Int. Workshop on Software Engineering for Parallel and Distributed Systems*, pages 146–157. London: Chapman&Hall, March 25–26 1996.

[9] Iain S. Duff. `MA28` – a set of Fortran subroutines for sparse unsymmetric linear equations. Tech. rept. AERE R8730, HMSO, London. Source code available via netlib [41], 1977.

[10] R.G. Grimes. `SPARSE-BLAS` basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [41], 1984.

[11] K.S. Kundert. `SPARSE 1.3` package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [41], 1988.

[12] Mark K. Seager and Anne Greenbaum. Slap: Sparse Linear Algebra Package, Version 2. Source code available via netlib [41], 1989.

[13] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations, Version 2. Research report, University of Minnesota, Minneapolis, MN 55455, June 1994.

[14] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[15] Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publisher, 1991.

[16] Z. Zlatev, J. Wasniewsky, and K. Schaumburg. *Y12M - Solution of Large and Sparse Systems of Linear Algebraic Equations.* Springer LNCS vol. 121, 1981.

[17] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition.* Cambridge University Press, 1992.

[18] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman and MIT Press, 1989.

[19] Christoph W. Keßler, Wolfgang J. Paul, and Thomas Rauber. Scheduling Vector Straight Line Code on Vector Processors. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 77–91. Springer Worksh. in Computing Series, 1992.

[20] R. Boisvert, R. Pozo, K. Remington, and J. Dongarra. Matrix-market: a web resource for test matrix collections. In R. Boisvert et al., editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, 1997.

[21] Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. 8th ACM Int. Conf. on Supercomputing*, pages 33–43. ACM Press, July 1994.

[22] Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Programming*, 25(1):17–50, Feb. 1997.

[23] Lawrence Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[24] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In *Proc. ACM SIGPLAN Symp. on Principles of Programming Languages*, pages 79–92, 1991.

[25] Santanu Paul and Atul Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Trans. on Software Engineering*, 20(6):463–475, 1994.

[26] B. Di Martino and G. Iannello. Pap Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns. In *Proc. 4th IEEE Workshop on Program Comprehension.* Los Alamitos: IEEE Computer Society Press, Mar. 1996.

[27] Zahira Ammarguellat and W.L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 283–295. ACM Press, June 20–22 1990.

[28] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proc. 9th ACM Int. Conf. on Supercomputing*, pages 444–448, July 1995.

[29] Michael Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.

[30] David Callahan. Recognizing and parallelizing bounded recurrences. In *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, 1991.

[31] Xavier Redon and Paul Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *PARLE 93, Springer LNCS vol. 694*, pages 132–145, 1993.

[32] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph–Parsing Approach. *IEEE Software*, pages 82–89, Jan. 1990.

[33] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, January 1990.

[34] Wojtek Kozaczynski, Jim Ning, and Tom Sarver. Program concept recognition. In *Proc. of KBSE'92 Seventh Knowledge-Based Software Engineering Conference*, pages 216–225, 1992.

[35] Pradip Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. In *Proc. Int. Conf. on Supercomputing*, pages 119–130, July 1988.

[36] Gary Sabot and Skef Wholey. Cmax: a Fortran Translator for the Connection Machine System. In *Proc. 7th ACM Int. Conf. on Supercomputing*, pages 147–156, 1993.

[37] Robert Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. In *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*. sponsored jointly by IJCAII/AAAI/CSCSI, Aug. 1995.

[38] Thomas Brandes and Manfred Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. In *16th Int. Conf. on Parallel Processing*, pages 446–448, 1987.

[39] S. Bhansali, J.R. Hagemeister, C.S. Raghavendra, and H. Sivaraman. Parallelizing sequential programs by algorithm-level transformations. In V. Rajlich and A. Cimitile, editors, *Proc. 3rd IEEE Workshop on Program Comprehension*, pages 100–107. Los Alamitos: IEEE Computer Society Press, Nov. 1994.

[40] S. Bhansali and J.R. Hagemeister. A pattern-matching approach for reusing software libraries in parallel systems. Technical report, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164-2752, Oct. 1995.

[41] NETLIB. Collection of free scientific software. Accessible by anonymous ftp to `netlib2.cs.utk.edu` or `netlib.no` or e-mail `"send index"` to `netlib@netlib.no`.