Proc. 7th Int. Workshop on Program Comprehension, May 5-7, 1999, Pittsburgh, USA.

The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations*

Christoph W. Keßler Craig H. Smith

Fachbereich IV - Informatik, Universität Trier, 54286 Trier, Germany
e-mail: {kessler,smith}@psi.uni-trier.de

Abstract

Automatic program comprehension is particularly useful when applied to sparse matrix codes*, since it allows to abstract e.g. from specific sparse matrix storage formats used in the code. In this paper we describe SPARAMAT, a system for speculative automatic program comprehension suitable for sparse matrix codes, and its implementation.*

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.

Irregular sparsity patterns are usually defined by runtime data. Here 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 supports 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.

[4] 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, these problems have been attacked for *sparse* matrix computations only in a very conservative way, e.g., by run–time parallelization techniques such as the inspector–executor method [25] or run–time analysis of sparsity patterns for load–balanced array distribution [36]. 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.

In this paper we describe SPARAMAT, a system for concept comprehension that is particularly suitable to *sparse* matrix codes. We started by studying 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 [13, 15, 22, 34, 32, 39] and recorded a list of basic computational kernels for sparse matrix computations, together with their frequently occurring syntactical and algorithmic variations.

Basic terminology. A *concept* is an abstraction of an externally defined procedure. It represents the (generally infinite) set of concrete procedures coded in a given programming language that have the same type and that we consider to be equivalent in all occurring calling contexts. Typically we give a concept a *name* that we associate with the type and the operation that we consider to be implemented by these procedures. An *idiom* of a concept c is such a concrete

*Research partially funded by DFG, project SPARAMAT

procedure, coded in a specific programming language, that has the same type as c and that we consider to implement the operation of c. An *occurrence* of an idiom i of a concept c (or short: an occurrence of c) in a given source program is a fragment of the source program that matches this idiom i by unification of program variables with the procedure parameters of i. Thus it is legal to replace this fragment by a call to c, where the program objects are bound to the formal parameters of c. The (compiler) data structure representing this call is called an *instance* I of c; the fields in I that hold the program objects passed as parameters to c are called the *slots* of I. Beyond the Fortran77 parameter passing, SPARAMAT allows procedure–valued parameters as well as higher–dimensional and composite data structures to occur as slot entries.

After suitable preprocessing transformations (inlining all procedures) and normalizations (constant propagation), the intermediate program representation — abstract syntax tree and/or program dependence graph — is submitted to the concept recognizer. The concept recognizer, described in Section 4, identifies code fragments as concept occurrences and annotates them by concept instances.

When applied to parallelization, we are primarily interested in recognizing concepts for which there are particular parallel routines available, tailored to the target machine. In the back–end phase, the concept instances can be replaced by suitable parallel implementations. The information derived in the recognition phase also supports automatic data layout and performance prediction.

Problems with sparse matrix computations. One problem we were faced with is that there is no standard data structure to store a sparse matrix. Rather, there is a set of about 15 competing formats in use that vary in their advantages and disadvantages, in comparison to the two–dimensional array which is the "natural" storage scheme for a dense matrix.

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. As it is generally not possible to statically evaluate this condition, a concept recognition engine can only *suspect*, based on its observations of the code while tracking the live ranges of program objects, that a certain set of program objects implements a sparse matrix; the final *proof* of this hypothesis must either be supplied by the user in an interactive program understanding framework, or equivalent run-time tests must be generated by the code generator. Unfortunately, such run–time tests, even if parallelizable, incur some overhead. Nevertheless, static program flow analysis [18] can substantially support such a *speculative* comprehension and parallelization. Only at program points where insufficient static information is available, run–time tests or user prompting is required to confirm (or reject) the speculative comprehension.

Application areas. The expected benefit from successful recognition is large. For automatic parallelization, the back-end should generate two variants of parallel code for the recognized program fragments: (1) an optimized parallel library routine that is executed speculatively, and (2) a conservative parallelization, maybe using the inspector–executor technique [25], or just sequential code, which is executed non–speculatively. These two code variants may even be executed concurrently and overlapped with the evaluation of run-time tests: If the testing processors find out during execution that the hypothesis allowing speculative 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.

Beyond automatic parallelization, the abstraction from specific data structures for the sparse matrices also supports 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 [4]. 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.

The SPARAMAT implementation focuses on sparse matrix computations coded by indirect array accesses. This is because, in order to maintain an achievable goal in a university project, it is necessary to limit oneself to a language that is rather easy to analyze (Fortran), to only a handful of sparse matrix formats (see Section 2), and to a limited set of most important concepts [19]. 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. Due to the flexibility of the generative approach, more concepts and templates may be easily added by any SPARAMAT user. Furthermore, it appears that we can reuse some techniques from our earlier PARAMAT project [17] more straightforwardly for indirect array accesses than for pointer accesses.

The remainder of this paper is organized as follows: Section 2 deals with vectors and sparse matrix storage schemes; Section 3 summarizes concepts for (sparse) matrix computations. Section 4 discusses concept recognition and describes our implementation. We close with related work and conclusions. A larger example for a neural network simulation code is given in [19].

2 Vectors and (sparse) matrices

Vectors. A *vector* is an object in the intermediate program representation that summarizes a one–dimensional view of some elements of an array. For instance, a vector of reals accessing the first 5 elements in column 7 of a two–dimensional array `a` of reals is represented as `V(a,1,5,1,7,7,0)`. For ease of notation we assume that the "elements" of the vector itself are consecutively numbered starting at 1. `IV(...)` denotes integer vectors.

An *indexed vector* summarizes a one–dimensional view of some elements of an array whose indices are specified in a second (integer) vector, e.g. `VX(a,IV(x,1,n,2))`.

(Sparse) Matrices. A *matrix* summarizes a two–dimensional view of an array according to the conventions of a specific storage format. Dense matrices appear as a special case of sparse matrices:

The *dense storage format* (**DNS**) uses a two–dimensional array `a(1:n,1:m)` to store all elements. A DNS matrix access to the full array `a` would be summarized as

```
DNS( a, 1,n,1, 1,m,1, 0 )
```

where the last entry specifies that the matrix access to `a` is not transposed. — In Fortran, multiplication of a DNS matrix by a vector typically looks like

```
DO i = 1, n
   b(i) = 0.0
   DO j = 1, m
      b(i) = b(i) + a(i,j) * x(j)
   ENDDO
ENDDO
```

As an example data structure that uses index vectors to represent sparse matrices we describe the *row–compressed sorted storage format* (**CSR**): A data array `a(1: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 array `col(1:nz)` gives the column index for each element in `a`, and an integer array `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 full CSR matrix access could be summarized as

```
CSR( V(a,1,nz,1), IV(firstinrow,1,n+1,1),
     IV(col,1,nz,1), n, nz )
```

Such storage formats are typical for Fortran77 implementations. CSR is used e.g. in the SLAP package [34].
Multiplication of a CSR matrix by a vector 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
```

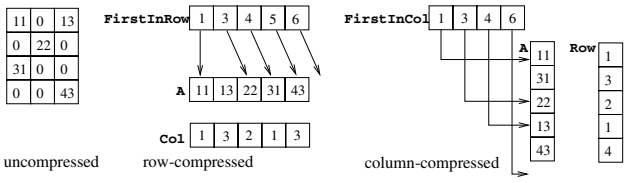

Figure 1: Row–compressed (CSR) and column–compressed (CSC) storage formats for sparse matrices.

Further formats, like COO (coordinate format), MSR (a CSR extension), CUR (unsorted CSR variant), XSR/XUR (sorted/unsorted CSR extension), CSC (column–compressed format), JAD (jagged diagonal format) and LNK (linked list format) are explained in [19]. The format names are partially adapted from [32]. See also [2] and [38].

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. For instance, 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.

3 Concepts

This section gives a survey of concepts that are frequently encountered in sparse matrix codes. Although this list is surely not exhaustive, it should at least illustrate the application domain. The extension of this list by more concepts to cover an even larger part of numerical software is the subject of on-going research.

We have developed a concept specification language, CSL [19], that allows one to describe concepts and matching rules on a level that is (more or less) independent from a particular source language or compiler. A concept specification consists of the following components: its name (naming conventions are discussed below), an ordered and typed list of its parameters, and a set of matching rules (called *templates*). A matching rule has several fields: a field for structural pattern matching, specified in terms of intermediate representation constructs (loop headers, conditions, assignments, and instances of the corresponding subconcepts), fields specifying auxiliary predicates (e.g., structural properties or dataflow relations), fields for the specification of pre- and postconditions for the slot entries implied by this concept (see Section 4), and a field creating a concept instance after successful matching. For an example specification see Figure 2.

Our naming conventions for concepts are as follows: The shape of operands is denoted by shorthands `S` (scalar), `V` (vector), `VX` (indexed vector), and `YYY` (matrix in storage

```
concept SDOTVV {
  param(out)  @r: real;
  param(none) @L: range;
  param(in)   @u: vector;
  param(in)   @v: vector;
  param(in)   @init: real;

  templateVertical {
    pattern {
       node  DO_STMT($i,$lb,$ub,$st)
       child    INCR($s,MUL($e1,$e2))
    }
    where {   $e1.isSimpleArrayAccess( $i )
           && $e2.isSimpleArrayAccess( $i )
           && $s.isVar()
           && $i.notOccurIn( $s )
    }
    instance SDOTVV($s,   newRange( $i, $lb, $ub, $st),
                    newVector( $e1, $i, $lb, $ub, $st),
                    newVector( $e2, $i, $lb, $ub, $st),
                    $s )
  }
  templateHorizontal {
    pattern {
       node(s): SINIT($x,$c)
       fill(f)
       node(n): SDOTVV($r1,$L1,$u1,$v1,$init1)
    }
    where(s) { $x.isEqual($init1) }
    where(f) {
      f.isEmpty() || f.notMayOut($x) && f.notMayIn($x)
    }
    instance(s) EMPTY()
    instance(n) SDOTVV( $L1, $u1, $v1, $r1, $c )
} }
```

Figure 2: A CSL specification for the SDOTVV concept (simple dot product) with two templates.

format YYY). 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. The default type is real; integer concepts and objects are prefixed with an I.

We extend our earlier approach [17] to representing concepts and concept instances in several aspects.

Operator parameters. Some concepts like VMAPVV (elementwise application of a binary operator to two operand vectors) take an operator (i.e., a function pointer) as a parameter. 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.

Functional composition. We are still discussing arbitrary functional composition of concepts to form new concepts. This idea is inspired by the work of Cole on algorithmic skeletons [11]. Nevertheless, there should remain at least some "flat" concepts for important special cases, e.g. SDOTVV for dot product, VMATVECMV for matrix–vector multiplication, etc. These may be regarded as "syntactic sugar" but are to be preferred as they enhance readability and speed up the program comprehension process.

No in–place computations. Most of our concepts represent not–in–place computations. In general, recognized in–place computations are represented by using temporary variables, vectors, or matrices. This abstracts even further from the particular implementation. It is the job of the back-end to reuse (temporary array) space where possible. 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 INCR (increment) or SCAL (scaling).

Concept instances as parameters. Nesting of concept instances is a natural way to represent a tree–like computation without having to specify temporary variables. As an example, we may denote a DAXPY–like computation as

```
VMAPVS( V(tmp,1,N,1), MUL, V(c,1,N,1), 3.14 )
VINCRV( V(b,1,N,1), V(tmp,1,N,1) )
```

which is closer to the internal representation in the compiler, or as

```
VINCR( V(b,1,N,1), VMAPVS(MUL,V(c,1,N,1),3.14))
```

which is more readable for humans. If the computation structure is a directed acyclic graph (DAG), then we may also obtain a DAG of concept instances, using temporary variables and arrays for values used multiple times. In order to support nesting, our notation of concept instances allows to have the result parameter (if there is exactly one) of a concept instance appear as the "return value" of a concept instance, rather than as its first parameter, following the analogy to a call to a function returning a value.

Concepts for scalar computations. There are concepts for binary expression operators, like ADD, MUL, MAX, EQ etc., for unary expression operators like NEG (negation), ABS (absolute value), INV (reciprocal), SQR (squaring) etc., The commutative and associative operators, ADD, MUL, MAX etc., may also have more than two operands. STAR is a special version of a multi–operand ADD denoting difference stencils [17]. 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. Assignments to scalars are either SCOPY where the assignee is a variable, or SINIT where the assignee is a constant, or an expression operator where the assignee is a recognized expression. For technical reasons there are some auxiliary concepts like EMPTY (no operation) and RANGE (to summarize a loop header). For more details, see [19].

Vector and matrix computations. We give here an informal description of some concepts. See [19] for a more complete list. v, v_1, v_2 denote (real) vectors, a a real array, iv an integer vector, m, m_1, m_2 matrices in some format.

VMAPVV$(v, \oplus, v_1, v_2)$	elementwise appl. of binary operator $\oplus$
VMAPV$(v, \ominus, v_1)$	elementwise appl. of unary operator $\ominus$
VMAPVS$(v, \oplus, v_1, r)$	elementwise appl. with a scalar operand r
VINCRV(v, v_1)	$v(i) = v(i) + v_1(i),\ i = 1, ..., \lvert v_1 \rvert$
VCOPYV(v, v_1)	copy vector elements
VINIT(v, c)	initialize vector elements by a constant c
SREDV$(r, \otimes, v)$	reduction $r = \bigotimes_{j=1}^{\lvert v \rvert} v(j)$
VGATHERVX$(v,$ VX$(a, v_1))$	$v(i) = a(v_1(i)),\ i = 1, ..., \lvert v_1 \rvert$
VXSCATTERV(VX$(a, iv), v_1)$	$a(iv(i)) = v_1(i),\ i = 1, ..., \lvert v_1 \rvert$
MMAPMM$(m, \oplus, m_1, m_2)$	elementwise appl. of binary operator $\oplus$

For similar concepts like matrix format conversion, matrix copy, matrix initialization, matrix transpose etc., see [19].

VMATVECMV(v, m_1, v_2)	matrix-vector product $v = m_1 \cdot v_2$.
VVECMATMV(v, m_1, v_2)	vector-matrix product $v = m_1^T \cdot v_2$.
MMATMULMM(m, m_1, m_2)	matrix-matrix-product $m = m_1 \cdot m_2$

More concepts, e.g. searching and sorting on vectors and matrices, prefix calculations, Givens rotation, for-

ward/backward substitution, LU decomposition and their subconcepts, submatrix extraction etc. are listed in [19].

I/O concepts. `READ` and `WRITE` are the concepts for reading and writing a scalar value to a file.

`VREAD`(v, F)	read a vector v from file F
`VWRITE`(v, F)	write a vector v to file F
`MREAD`(m, F, f)	read m from file F in file storage format f
`MWRITE`(m, F, f)	write 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 [7].

4 Speculative concept recognition

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

The static part of our concept matching method is based on a bottom–up rewriting approach using a deterministic finite bottom–up[1] tree–automaton that works on the program's intermediate representation (IR) as an abstract syntax tree or control flow graph, augmented by concept instances and data–flow edges computed during the recognition. Normalizing transformations, such as loop distribution or rerolling of unrolled loops, are done whenever applicable.

The matching rules for the concept idioms to be recognized, called *templates*, are specified as far as possible in terms of subconcept occurrences (see Fig. 2), following the natural hierarchical composition of computations in the given programming language, by applying loops and sequencing to subcomputations. Since at most one template may match an IR node, identification of concept occurrences is deterministic. For efficiency reasons the applicable templates are selected by a hashtable lookup: each rule to match an occurrence of a concept c is indexed by the most characteristic subconcept c' (called the *trigger concept*) that occurs in a matching rule. The graph induced by these edges (c', c) is called the *trigger graph*. Hence, concept recognition becomes a path finding problem in the trigger graph. Matched IR nodes are annotated with concept instances. If working on an abstract syntax tree, a concept instance holds all information that would be required to reconstruct an equivalent of the subtree it annotates.

[1] To be precise, for the unification of objects *within* a matching rule we apply a top–down traversal of (nested) concept instances for already matched nodes.

Vertical matching proceeds along the hierarchical nesting structure (statements, expressions) of the program's IR, starting with the leaf nodes. Matching a node is only possible when all its children have been matched. The trigger concept used When applying vertical matching to an IR node, the concept that has been matched for its first child is used as the trigger concept.

As a running example, consider the following code excerpt:

```
    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 program's IR (e.g. 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 vectors `V(a,first(i),first(i+1)-1,1)` and `IV(col,first(i),first(i+1)-1,1)`. By a template similar to the first one in Fig. 2, the entire `j` loop is matched as an occurrence of `SDOTVVX` (dot product with one indexed operand vector); the unparsed program is now

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

Horizontal matching tries to merge several matched IR nodes v_1, v_2, ... belonging to the body of the same parent node (e.g., a loop body). 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 that 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 horizontal matching, the values of outer loop variables can be considered as constant. Horizontal matching, following the corresponding template

(similar to the second template in Fig. 2), "merges" [2] the two nodes and generates a "shared" concept instance:

```
    DO i = 1, n
S'':  SDOTVVX( b(i), V(a,first(i),first(i+1)-1,1),
       VX(x,IV(col,first(i),first(i+1)-1,1)), 0.0)
    ENDDO
```

Speculative concept matching. 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 V(first,1,n+1,1)>
   <assume injectivity of V(col,first(i),
                first(i+1)-1,1) forall i in 1:n>
S: VMATVECMV( V(b,1,n,1),
              CSR(a, IV(first,1,n+1,1),
                  IV(col,first(1),first(n+1)-1,1),
                  n, first(n+1)-1),  V(x,1,n,1) );
```

where the first three lines summarize the assumptions guiding our speculative concept recognition. If they cannot be statically eliminated, these three preconditions would, at code generation, result in three run-time tests being scheduled before or concurrent to the 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.

Now we have a closer look at these pre- and postconditions:

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, `iv(`i`)`$\leq$`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 $i \neq j \implies$ `iv(`i`)` $\neq$ `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)` equals 1,

(2) vector `IV(first,1,`n`+1,1)` is monotonic, and

(3) vectors `IV(col,first(`i`),first(`$i+1$`)-1,1)` are injective for all $i \in \{1, ..., n\}$.

These properties may be checked for separately.

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 preconditions 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 (i.e., a postcondition). 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 summarized by the program comprehension engine in the form of pre- and postcondition annotations to the concept instances. Note that the preservations are the complementary set of the deletions; thus we renounce on listing them. If existing program objects may be overwritten, their old properties are clearly deleted. Note that the `install` and `propagate` annotations are postconditions that 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 IV(FirstB,1,M,1)>
<assume injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                        forall i in 1:M>
<delete all of FirstA>
<delete all of ColA>
<install FirstA(1)=1>
<propagate (monotonicity of IV(FirstB,1,M,1))
 then (monotonicity of IV(FirstA,1,N,1))>
<propagate (monotonicity of IV(FirstB,1,M,1)
 and (injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                     forall i in 1:M)
 then(injectivity of IV(ColA,FirstA(i),FirstA(i+1),1)
                     forall i in 1:N)>
MCNVTM( CSR( V(A,1,NZ,1), IV(FirstA,1,N+1,1),
             IV(ColA,1,N,1), N, NZ),
        CSC( V(B,1,NZ,1), IV(FirstB,1,M+1,1),
             IV(RowB,1,M,1), M, NZ) )
```

4.2 Run time tests / user interaction

Once the static concept recognition phase is finished, these properties, summarized as pre- and postconditions for each

[2] Technically, one node is hidden from further matching and code generation by annotating it with an instance of `EMPTY`, see Fig. 2.

concept instance in the (partially) matched program, are optimized by a dataflow framework presented in [18]. This method allows to eliminate redundant conditions and to schedule the resulting run-time tests (or user interactions) appropriately.

Ideally, there are, once a sparse matrix has been constructed or read from a file, no changes to its organizational variables any more during execution of the program, i.e., its sparsity pattern remains static. In that case SPARAMAT can completely eliminate all but one test on monotonicity and injectivity, which is located immediately after constructing the organizational data structures. An example program exhibiting such a structure is shown in the neural network simulation code given in [19].

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.

If applied to an interactive program comprehension framework, these run-time tests correspond to prompting the user for answering yes/no questions about the properties.

The run-time tests on monotonicity and injectivity can be parallelized, as shown in [19].

4.3 SPARAMAT implementation

SPARAMAT is currently being implemented using the Polaris Fortran compiler [6] as a front end. SPARAMAT is conceptually broken up into two major components, the driver and the generator (see Fig. 3). When configuring the SPARAMAT system, the generator accepts specification files describing concepts and templates, and creates C++ files containing the trigger graph (TG) and the templates. These are then linked with some SPARAMAT concept matching core routines to form the driver.

When submitting a Fortran program to SPARAMAT, its control flow graph is passed to the driver from the Polaris front end. The driver, upon completion, passes to the optimizer the control flow graph annotated with concept instances. The pre- and postconditions are optimized in a separate pass described in [18]. This optimizer passes the modified control flow graph to the back end.

If applied to parallelization, the back end may generate for the concept instances (speculative) calls to parallel routines and replace the remaining conditions by run-time checks, see Fig. 3.

If applied in an interactive framework, SPARAMAT could instead compute and present a certainty factor for specific matrix formats and then ask the user, displaying the code, if the detected format is correct. This requires a user interface to display the code highlighting matched concepts. As DOT [20] files are already generated for testing, the GraphViz package can be easily leveraged to implement such a user interface.

To date we have implemented a subset of the basic concepts and some of their templates, in particular for dot product and dense and sparse matrix–vector multiplication and their subconcepts for CSR, MSR and COO format. Due to the generative approach, the implementation can be easily extended for further formats, concepts, and templates by any user of the SPARAMAT system.

In some situations the matrix format might not be clear until a discriminating piece of code is encountered. Until then, it may be necessary to store a set of possible formats in the concept instance and postpone the final identification of the format to a later point in the concept matching process. To support this, we use a special nested concept summarizing the same matrix object in a set of different possible formats within the same slot of the concept instance.

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 [35], Pinter and Pinter [28], Paul and Prakash [27], diMartino [23] and Keßler [17]. Some work focuses on recognition of induction variables and reductions [1, 29, 14] and on linear recurrences [10, 30]. General techniques designed mainly for non–numerical codes have been proposed by Wills et al. [31] and Ning et al. [16, 21].

Concept recognition has been applied in some commercial systems, e.g. EAVE [8] for automatic vectorization, or CMAX [33] and a project at Convex [24] for automatic parallelization. Furthermore there are several academic applications [17, 23] and proposals for application [9, 3] 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 [17] or [12].

Our former PARAMAT project (1992–94) [17] kept its focus on dense matrix computations only, because of their static analyzability. The same decision was also made by other researchers [28, 23, 24, 3] and companies [33] investigating general concept recognition with the goal of automatic parallelization. According to our knowledge, there is currently no other framework that is actually able to recognize sparse matrix computations in the sense given in this paper.

6 Conclusion and future work

We have described a framework for applying program comprehension techniques to sparse matrix computations and

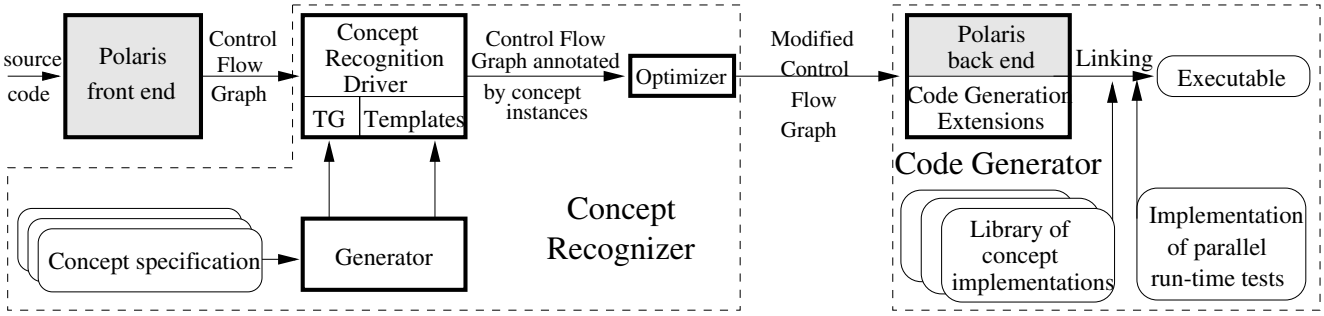

Figure 3: SPARAMAT implementation (left hand side), with a possible application to parallel code generation from concept instances (right hand side).

its implementation. We see that it is possible to perform speculative program comprehension even where static analysis does not provide sufficient information; in these cases the static tests on the syntactic properties (pattern matching) and consistency of the organizational variables are complemented by user prompting or run-time tests whose placement in the code can be optimized by a static data flow framework.

If applied to parallel code generation, speculatively matched program parts may be optimistically replaced by suitable parallel library routine calls, together with the necessary (parallel) run-time tests. Only if the tests are passed, parallel execution may continue with the optimized parallel sparse matrix library routine. Otherwise, it must fall back to a conservative code variant.

Our automatic program comprehension techniques for sparse matrix codes can also be used in a non–parallel environment, e.g. for program flow analysis, for program maintenance, debugging support, and for more freedom of choice for a suitable data structure for sparse matrices.

Current work on the SPARAMAT implementation focuses on CSL and the generator. Once operational, we will implement the complete list of concepts given in [19] with the most important templates.

References

[1] Z. Ammarguellat and W. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. *Proc. Conf. on Progr. Language Design and Implementation*, pp. 283–295. ACM, June 1990.

[2] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[3] S. Bhansali et al. Parallelizing sequential programs by algorithm-level transformations. In V. Rajlich and A. Cimitile, eds., *Proc. 3rd IEEE Workshop on Program Comprehension*, pp. 100–107. IEEE CS Press, Nov. 1994.

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

[5] A. J. C. Bik. *Compiler support for sparse matrix computations*. PhD thesis, Leiden University, 1996.

[6] W. Blume et al. Polaris: The next generation in parallelizing compilers,. *Proc. 7th Wksh. on Languages and Compilers for Parallel Computing*, 1994.

[7] R. Boisvert et al. Matrix-market: a web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pp. 125–137. Chapman and Hall, 1997.

[8] P. Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. *Proc. Int. Conf. on Supercomputing*, pp. 119–130, July 1988.

[9] T. Brandes, M. Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. *16th Int. Conf. on Par. Processing*, 446–448, 1987.

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

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

[12] B. DiMartino and C. W. Keßler. Program comprehension engines for automatic parallelization: A comparative study. In I. Jelly, I. Gorton, and P. Croll, eds., *Proc. 1st Int. Wksh. on Software Eng. for Par. and Distr. Systems*, pp. 146–157. London: Chapman&Hall, Mar. 1996.

[13] I. S. Duff. `MA28` – a set of Fortran subroutines for sparse unsymmetric linear equations. Tech. rept. AERE R8730, HMSO, London. Sources at [26], 1977.

[14] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Prog. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[15] R. Grimes. `SPARSE-BLAS` basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [26], 1984.

[16] M. Harandi and J. Ning. Knowledge-based program analysis. *IEEE Software*, pp. 74–81, January 1990.

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

[18] C. W. Keßler. Applicability of Program Comprehension to Sparse Matrix Computations. *Proc. 3rd Int. Euro-Par Conference*. Springer LNCS, Aug. 1997.

[19] C. W. Keßler, H. Seidl, and C. H. Smith. The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations. Technical Report 99-??, Universität Trier, FB IV - Mathematik/Informatik, 54286 Trier, Germany, 1999.

[20] E. Koutsofios and S. North. Drawing graphs with *dot*. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, Sept. 1991.

[21] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. *Proc. KBSE'92 7th Knowledge-Based Software Eng. Conf.*, pp. 216–225, 1992.

[22] K. Kundert. `SPARSE 1.3` package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [26], 1988.

[23] B. D. Martino and G. Iannello. Pap Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns. *Proc. 4th Wksh. on Program Comprehension*. IEEE CS Press, Mar. 1996.

[24] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*, Aug. 1995.

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

[26] 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.

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

[28] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. *Proc. ACM Symp. on Principles of Progr. Languages*, pp. 79–92, 1991.

[29] B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proc. 9th ACM Int. Conf. on Supercomputing*, pp. 444–448, 1995.

[30] X. Redon and P. Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *PARLE'93, Springer LNCS vol. 694*, pp. 132–145, 1993.

[31] C. Rich and L. M. Wills. Recognizing a Program's Design: A Graph–Parsing Approach. *IEEE Software*, pp. 82–89, Jan. 1990.

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

[33] G. Sabot and S. Wholey. Cmax: a Fortran Translator for the Connection Machine System. *Proc. 7th ACM Int. Conf. on Supercomputing*, 147–156, 1993.

[34] M. K. Seager and A. Greenbaum. Slap: Sparse Linear Algebra Package, Version 2. Source code available via netlib [26], 1989.

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

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

[37] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison–Wesley, 1990.

[38] Z. Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer 1991.

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