On the Applicability of Program Comprehension Techniques to the Automatic Parallelization of Sparse Matrix Computations

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

Abstract

Space–efficient data structures for sparse matrices are an important concept in numerical programming because they allow for considerable savings in space and time compared with common two–dimensional arrays. Unfortunately, for such programs it is usually impossible to statically determine all data dependencies. Thus, automatic parallelization of such codes is usually done at run time by applying the inspector–executor technique, incurring tremendous overhead.

Program comprehension techniques exploit knowledge on frequently occurring implementation variations of important computations. They have been shown to improve many important fields of automatic parallelization of dense matrix computations, such as automatic program transformation and local algorithm replacement, data flow analysis, array distribution, and performance prediction.

In this study we investigate up to which degree this approach could be generalized to sequential codes implementing sparse matrix computations, and how much additional information, e.g. on run–time values of some variables, is needed. We propose a speculative program comprehension and parallelization method combined with compile–time program flow analysis techniques. Only where static analysis does not supply sufficient information, a parallelized run time test must check whether a code fragment speculatively recognized and parallelized at compile time as a potential sparse matrix computation, really implements this computation. Beyond automatic parallelization, program comprehension for sparse matrix computations may also support program maintenance and debugging, and could guide the selection of a more suitable data structure for a sparse matrix.

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 or, at least, below a near–zero threshold, that it seems worthwile 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. Only if this overhead is too large because the fraction of nonzero elements is rather high, one should use the usual data structure for dense matrices, a two–dimensional array.

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 band may be stored as a one–dimensional array (vector).

Irregular sparsity patterns are usually defined by run-time data. In the remainder of this paper, we have only this case in mind when using the term "sparse matrix". Typical data structures in Fortran77 are, beyond a *data array* containing the nonzero elements themselves, several *organizational variables*, e.g. arrays of indices into the data array, together with suitable row and/or column index information for each data array element. C implementations typically use linked lists to store the nonzero elements.

Matrix computations on these data structures are common in practice. Thus, it is a reasonable idea to port them on a parallel computer to obtain further speed–up. Consequently, numerous parallel algorithms have been invented or adapted for sparse matrix computations over the last decades for various parallel architectures.

[BW96] 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.

The more comfortable way to obtain a parallel program is automatic parallelization. This approach tries to make the compiler recognize the parallelism in an existing sequential source code and generate parallel code for it. While the problems of automatic parallelization for dense matrix computations are, meanwhile, well understood and sufficiently solved [ZC90, KN90, KLS90, LC90, KK93, BKK93, Keß96], these problems are, for sparse matrix computations, solved, if at all, in a very conservative way (e.g., by run–time determination of communication by the inspector–executor technique, see Fig. 1, or run–time analysis of sparsity patterns for load–balanced array distribution [RZ93]). This is not astonishing because the corresponding code looks quite awful to the compiler, consisting of indirect array indexing or pointer dereferencing which makes exact static access analysis impossible. Dynamic 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), program comprehension techniques such as pattern matching appeared to be a good idea [Keß96]. After suitable preprocessing transformations and normalizations, the intermediate representation of the code — abstract syntax tree and/or program dependence graph — is submitted to a pattern 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, this information is used to replace these code pieces by suitable parallel implementations. The information derived in the recognition phase also supports the global optimization of array distributions, and the prediction of parallel execution times which is required for automatic array distribution.

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.

The other main difference is that (time– and) 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 pattern recognition and code replacement is no longer completely

Figure 1: **The inspector–executor technique for run–time parallelization**
The inspector–executor technique [SCMB90] generates two passes for each loop:
The *inspector* computes the values that all index expressions will have for each iteration of the loop. With this information, it determines all loop–carried dependences and then computes a schedule (a sequence of parallel "wavefronts") preserving these dependences. If all induction variables have been eliminated and there is only one level of indirection, the inspector may run in parallel but nevertheless incurs high overhead.
The *executor* uses this schedule for the parallel execution of the loop.
The inspector–executor scheme can be extended to message passing architectures [KMR90, DSv93]; it is a very general technique but incurs much overhead in the inspector phase. This may in some cases be partially optimized, e.g. at nested loops by hoisting the inspector out of a loop if the dependency structure of its iterations is loop invariant, thus amortizing the inspector's overhead by reusing the schedules [ESH93, LB96]. [KW94] and [LB96] try to characterize the run–time access patterns in indirect array accesses and exploit emerging regularities for speeding up the inspector phase.
We suppose that an approach which is more tailored towards a specific application domain (such as the one described in this paper) may perform better compared to the general inspector–executor method.

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

A simple way of generating 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, maybe, some 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.

The expected benefit from successful recognition is large. Beyond automatic parallelization, it 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 [BW96].

We have examined several representative source codes for implementations of basic linear algebra subroutines like dot product, matrix–vector multiplication, matrix–matrix multiplication, or LU factorization for sparse matrices [Gri84, Kun88, NAG88]. We describe our observations and their implications for the design of an automatic parallelizer tailored towards sparse matrix computations.

The remainder of this paper is organized as follows: Section 2 contains a *bestiarium* of common storage schemes for sparse matrices that we encountered so far. Section 3 shows, at the example of a preliminary list of computation patterns for two sparse matrix formats based on index vectors, how patterns to be matched in sparse matrix computations may look like. Section 4 discusses the technical problems involved in recognition of these patterns. Section 5 extends the issue to computations based on linked list storage formats. Section 7 concludes.

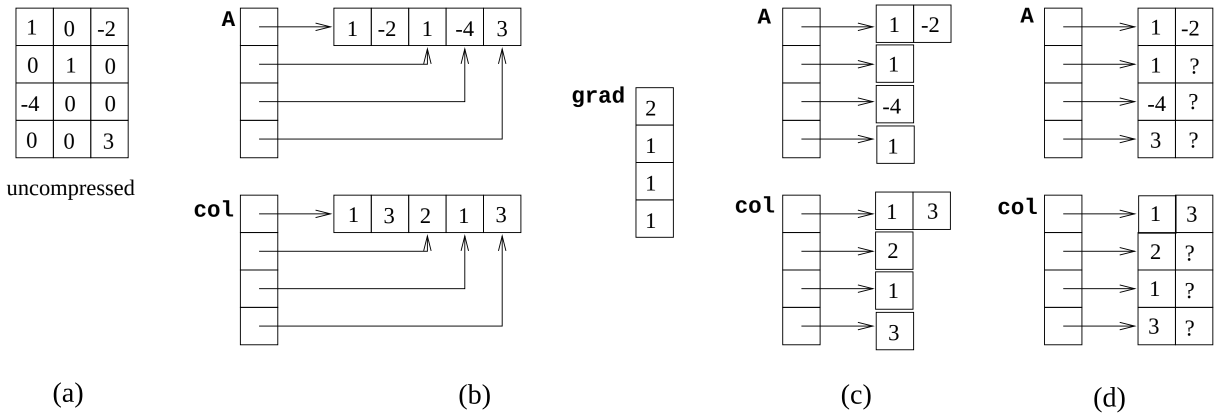

Figure 2: Explicit row–compressed format. The three storage variants for the dense matrix (a) are shown in (b), (c) and (d). The `grad` array remains the same as in (b).

2 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.

The top–most classification divides such data structures into those that use indirect array indexing into a linear work array, and those which implement linked lists using structures and pointers. The former category is the only one when considering Fortran 77 codes (as Fortran 77 offers neither pointers nor structs), the latter one is more common (more natural?) in codes written in C (or similar programming languages supporting pointers and structs). In the following description we use C notation, as C covers both categories.

2.1 Index vectors and workspace arrays

We summarize several data structures based on indirect access on a linear workspace array, at the example of a sparse matrix–vector multiplication. Only the first two variants may require also pointers. — As there is no systematic naming convention for these data structures, we partially use our own terminology in order to avoid misunderstandings.

A main difference that arises from these data structures is that the comfortable symmetry present in the two–dimensional arrays implementing dense matrices 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.

1. *explicit row–compressed format*, storing a pointer `A[i]` to the first nonzero element and the number `grad[i]` of nonzero entries for each row `i`, and the value `A[i][j]` and the column index `col[i][j]` for the `j`th nonzero element of row `i`, in row–wise storage. Note that array `A` may be allocated as (1) a one–dimensional array containing alias addresses for consecutive entries in the linear workspace array `A[0]`, or as (2) a one–dimensional array of one–dimensional arrays `A[i]` which may not necessarily be allocated consecutively in main memory, or as (3) a two–dimensional array with a fixed number of entries allocated along the compressed dimension (here, rows). See Figure 2 for an illustration.

   Matrix–vector multiplication may look like

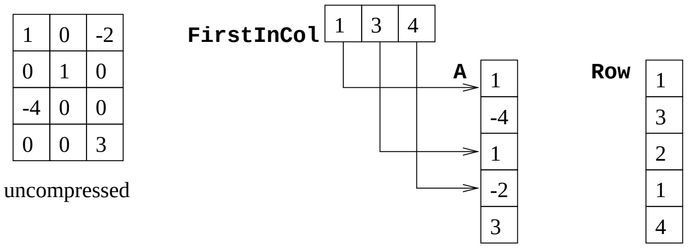

Figure 3: Implicit column–compressed format.

```
for (i=0; i<n; i++)
  for (j=0;j<grad[i];j++)
    b[i] = b[i] + A[i][j] * x[col[i][j]]
```

2. *explicit column–compressed format.* Matrix–vector multiplication may look like

```
for (j=0; j<n; j++)
  for (i=0;i<grad[j];i++)
    b[row[i]] = b[row[i]] + A[i][j] * x[j]
```

3. *implicit column–compressed format*, storing the workarray index `FirstInCol[i]` of the first nonzero element of column `i`, and the value `A[k]` and the row index `Row[k]` for the `k`th nonzero element, in column–wise storage. See Figure 3 for an illustration.

   Matrix–vector multiplication may look like

```
for (i=0; i<n; i++)
  for (k=FirstInCol[i]; k<FirstInCol[i+1]; k++)
    b[i] = b[i] + A[k] * x[Row[k]]
```

   This format is described e.g. in [Ben86] and used in the Harwell `MA28` package [NAG88].

4. *implicit row–compressed format*,

5. *diagonal–extended implicit column–compressed format.* The sparse matrix routines of the *Numerical Recipes* [PTVF92] mainly follow the implicit format but first store all diagonal elements in the workarray, regardless of whether they are zero or not. This is motivated by the fact that, often, most of the diagonal elements are nonzero, and are accessed more frequently than others. The `FirstInCol` information above is stored in the first n elements of the `ija` array. The routine `sprsax()` looks then as follows:

```
for (i=1; i<=n; i++) {
   b[i] = A[i] * x[i];                 /* diagonal elements */
   for (k=ija[i]; k<=ija[i+1]-1; k++)
      b[i] = b[i] + A[k] * x[ija[k]]
}
```

   For a transposed access to matrix `A`, the corresponding routine `sprstx()` is

```
for (i=1; i<=n; i++)
   b[i] = A[i] * x[i];     /* diagonal elements */
for (i=1; i<n; i++)
   for (k=ija[i]; k<=ija[i+1]-1; k++)
      b[ija[k]] = b[ija[k]] + A[k] * x[i]
```

6. *diagonal–extended implicit row–compressed format*

While matrix–vector multiplication codes for a sparse matrix looks quite simple and seems to be somehow identifiable by pattern 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 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 workarray and the index vectors. This is e.g. applied in `SPARSE` [Kun88]. More flexible data structures that allow fast insertion of new nonzero elements are the following:

- the *unsorted Gustavson data structure* (cf. [SBV93]) uses two offset arrays whose entries contain the beginning of each new row resp. column in the workarray. For each nonzero element in the workarray, the row and column index are explicitly given.
- the *sorted Gustavson data structure* keeps workarray entries sorted in increasing order of row and column indices.

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

2.2 Linked lists

Linked list data structures for sparse matrices support efficient fill–in. The following four types are common [SBV93], and are also often varied by additional organizational information such as e.g. an array of pointers to the diagonal elements.

1. the unsorted singly–linked two–dimensional list: Beyond its row and column index, each nonzero matrix element holds two pointers, to its successor in the same row resp. in the same column. Two arrays of pointers point to the first nonzero elements of each row resp. of each column.
2. the sorted singly–linked two–dimensional list, in increasing order of row and column indices. This variant is used in `SPARSE` [Kun88].
3. the unsorted doubly–linked two–dimensional list: each nonzero matrix element additionally holds a pointer to its predecessor in the same row resp. in the same column.
4. the sorted doubly–linked two–dimensional list.

2.3 Consequences for Program Comprehension

Regarding program comprehension, we find 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 pattern 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.

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 which is executed if the proof returns TRUE, and (2) a conservative parallelization, maybe using the inspector–executor technique, which is executed otherwise.

3 A first list of common patterns in sparse matrix computations

Within the next two sections, 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 extend our syntactic pattern matcher from our earlier PARAMAT project [Keß96] more straightforwardly for indirect array accesses than for pointer accesses. Later on, we will consider linked list implementations for sparse matrices.

The following survey of patterns is far from being exhaustive. It should be regarded rather as an illustration of what we are planning to do. We abbreviate by `DO` *range*(`i`) a common `for` resp. `DO` loop over a regular range *range*(`i`) where loop variable `i` iterates from some lower bound to some upper bound, potentially with a non–unit stride. The example computations given in the column "could look like" serve only as an illustration; there may be many more possibilities to express that pattern. E.g., an operand may also be negated, inverted, or its absolute value may be taken; loops may be reversed, unrolled, blocked, distributed or interchanged, etc. It is the job of program comprehension to abstract from such syntactical variations.

3.1 Elementwise indirect vector operations

Elementwise indirect vector operations		
name	parameters	could look like
VGATHER	*range*(i), x, y, ix	DO i=*range*(i) x[i] = y[ix[i]]
VSCATTER	*range*(i), x, ix, y	DO i=*range*(i) x[ix[i]] = y[i]
VXASSIGN	*range*(i), x, ix, v	DO i=*range*(i) x[ix[i]] = v
VXINIT	*range*(i), x, ix, c	DO i=*range*(i) x[ix[i]] = c
VXAADDSV	*range*(i), x, ix, x, u, v	DO i=*range*(i) x[ix[i]] = x[ix[i]] ± u * v[i]

Further order 1 operations based on indirect array references can be composed from these.

3.2 Indirect 1D reductions

Indirect 1D reductions		
name	parameters	could look like
`VXSUM`	*range*(`i`), `s`, `y`, `ix`, `c`	`s=c;  DO i=`*range*`(i)  s = s ± y[ix[i]]`
`VXPROD`	*range*(`i`), `s`, `y`, `ix`, `c`	`s=c;  DO i=`*range*`(i)  s = s * y[ix[i]]`
`VXDOTV`	*range*(`i`), `s`, `y`, `z`, `iz`, `c`	`s=c;  DO i=`*range*`(i)  s = s ± y[i]*z[iz[i]]`
`VXDOTVX`	*range*(`i`), `s`, `y`, `iy`, `z`, `iz`, `c`	`s=c;  DO i=`*range*`(i)  s = s ± y[iy[i]]*z[iz[i]]`
`VXQSUM`	*range*(`i`), `s`, `y`, `iy`, `c`	`s=c;  DO i=`*range*`(i)  s = s ± y[iy[i]]*y[iy[i]]`
`VXMAXVAL`	*range*(`i`), `s`, `x`, `ix`, `c`	`s=c;  DO i=`*range*`(i)  if (s>x[ix[i]])  s=x[ix[i]]`
`VXMINVAL`	*range*(`i`), `s`, `x`, `ix`, `c`	`s=c;  DO i=`*range*`(i)  if (s<x[ix[i]])  s=x[ix[i]]`
`VXMAXLOC`	*range*(`i`), `k`, `x`, `ix`, `t`	`k=t;  DO i=`*range*`(i)  if (x[ix[k]]<x[ix[i]])  k=i`
`VXMINLOC`	*range*(`i`), `k`, `x`, `ix`, `t`	`k=t;  DO i=`*range*`(i)  if (x[ix[k]]>x[ix[i]])  k=i`

`s`, `c`, `k` and `t` are scalar variables (i.e., they are not indexed by loop variable `i`). — The initializers `s=c` or `k=t` are optional.

3.3 Sparse matrix-vector multiplication using index vectors

Sparse matrix-vector multiplication		
name	parameters	could look like

`MXGV`	*range*(`i`), `j`, `grad`, `a`, `b`,
	`col`, `C`, `init`
		`DO i=`*range*`(i)  a[i] = init[i];`
		`DO i=`*range*`(i)  DO j=0,grad[i]-1`
		`  a[i] = a[i] ± b[col[i][j]] * C[i][j]`

`VMXG`	*range*(`i`), `j`, `grad`, `a`, `b`,
	`col`, `C`, `init`
		`DO i=`*range*`(i)  a[i] = init[i];`
		`DO i=`*range*`(i)  DO j=0,grad[i]-1`
		`  a[col[i][j]] = a[col[i][j]] ± b[i] * C[i][j]`

`MXFV`	*range*(`i`), `j`, `first`, `a`, `b`,
	`col`, `C`, `init`
		`DO i=`*range*`(i)  a[i] = init[i];`
		`DO i=`*range*`(i)  DO j=first[i],first[i+1]-1`
		`  a[i] = a[i] ± b[col[j]] * C[j]`

`VMXF`	*range*(`i`), `j`, `first`, `a`, `b`,
	`col`, `C`, `init`
		`DO i=`*range*`(i)  a[i] = init[i];`
		`DO i=`*range*`(i)  DO j=first[i],first[i+1]-1`
		`  a[col[j]] = a[col[j]] ± b[i] * C[j]`

Explicit names like `grad`, `col`, `first` etc. for the variable symbols are used here only for better readability. Although taken from the row–compressed storage format, the column–compressed storage format shows, apart from variable names, exactly the same syntactical appearance: Matrix vector multiplication (`MV`) for row–compressed format looks like transposed matrix vector multiplication (`VM`) for column–compressed format, and vice versa. Thus we need not define two distinct sets of patterns for them. — The initializing loops are optional.

Table 1 lists the routines from `SPARSE-BLAS` [Gri84] and the patterns that could be identified there.

4 Static and dynamic matching of sparse matrix patterns

Safe identification of a sparse matrix operation consists of (1) a test for the syntactical properties of this operation, which can be performed by pattern recognition at compile time, and (2) a test

SPARSE-BLAS routine	pattern
DAXPYI	VXAADDSV$^{(1)}$
DDOTI	VXDOTV$^{(1)}$
DGTHR	VGATHER$^{(1)}$
DGTHRZ	VGATHER$^{(1)}$ + VXINIT$^{(1)}$(after speculative loop distribution)
DROTI	—
DSCT	VSCATTER$^{(1)}$

Table 1: The SPARSE-BLAS routines and the patterns corresponding to them.

for the semantic 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.

Testing the static part is, in principle, not hard. Just in the same way as we did for dense matrix operations [Keß96], we work bottom–up on the abstract syntax tree, incrementally compute dataflow relations ("cross edges"), and deterministically identify patterns based on already matched subpatterns. For efficiency reasons, the matching rules are hierarchically organized ("pattern 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 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 pattern P is annotated by a summary node ("pattern 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:

```
    for (i=0; i<n; i++) {
S1:    b[i] = 0.0;
       for (j=0;j<grad[i];j++)
S2:       b[i] = b[i] + A[i][j] * x[col[i][j]]
    }
```

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 `AADDMUL( b[i], A[i][j], x[col[i][j], b[i] )`. Now the loop around `S2` is considered. The second index expressions of arrays `A` and `col` are bound by the loop variable `j` which ranges from 0 to the loop–invariant value `grad[i]`-1. Thus the accesses to arrays `A` and `col` during the `j` loop can be summarized as vector accesses `A[i][0:grad[i]-1]` and `col[i][0:grad[i]-1]`. The entire `j` loop is matched as an indexed dot product $\mathsf{VXDOTV}^{(1)}$, and the unparsed program now looks as follows:

```
    for (i=0; i<n; i++) {
S1':   SINIT( b[i], 0.0 );
S2':   VXDOTV( j=[0:grad[i]-1], b[i],  A[i][0:grad[i]-1],
               x[col[i][0:grad[i]-1]], b[i] );
    }
```

The initialization slot for $\mathsf{VXDOTV}^{(1)}$ has been filled by the (old) value of `b[i]` in order to reflect all dataflow properties of the summary node. 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'` which is described in the next subsection.

4.2 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 pattern 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 `VXDOTV` 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:

```
    for (i=0; i<n; i++) {
S'':    VXDOTV( j=[0:grad[i]-1], b[i],  A[i][0:grad[i]-1],
                x[col[i][0:grad[i]-1]], col[i][0:grad[i]-1], 0.0 );
    }
```

which can then, again applying vertical matching, be directly matched with the `i` loop. The accesses to `A` and `col` become matrix accesses as now also their first dimension is bound by another loop variable, `i`. The additional occurence of `grad[i]` in the limits of the second dimension of these matrix accesses must be treated as another parameter read by the summary node for the `i` loop:

```
S''':  MXGV( i=[0:n-1], j=[0:?], grad[0:n-1], b[0:n-1],
             A[0:n-1][0:?], col[0:n-1][0:?], x[?:?], 0.0 );
```

4.3 Speculative pattern matching and run time tests

In this framework, syntactic pattern matching creating summary nodes as the `MXGV` instance in the example above, is *speculative*, i.e. it assumes the run time tests are positive. We now look at these run time tests in more detail.

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, at q for all $i, j \in$`[L:U]` holds $i \neq j \Longrightarrow$ `iv[`i`]` $\neq$ `iv[`j`]`. 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 $o_1, o_2, ...$ 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 using explicit row–compressed format (`MXGV` resp. `VMXG`) we must check whether the following three conditions hold on the program objects corresponding to the organizational variables given by the pattern parameters `n`, *range*(`i`), `grad` and `col`:

1. `A[`i`]` $+$ `grad[`i`]` $=$ `A[`$i+1$`]` for all $i \in$ *range*(`i`) (for contiguous storage, see Fig. 2(c))
2. `col[`i`]` $+$ `grad[`i`]` $=$ `col[`$i+1$`]` for all $i \in$ *range*(`i`), $j \in [0:$`grad[`i`]`$-1]$ (for contiguous storage)
3. array `col[`i`][]` is injective over the index domain $[0:$`grad[`i`]`$-1]$ for all $i \in$ *range*(`i`)

For their equivalents in implicit row–compressed format (`MXFV` resp. `VMXF`) we have to check for the following conditions:

1. `first[`i`]`$\leq$`first[`$i+1$`]` $\leq n$ for all $i \in$ *range*(`i`)
2. array `col[]` is injective over the index domain [`first[`i`]`:`first[`$i+1$`]`-1] for all $i \in$ *range*(`i`)

These conditions may be checked for separately. As an illustration, we will now consider the verification of injectivity; the other conditions can be checked in a straightforward way.

To minimize the number of injectivity tests that must really be generated, we use the following dataflow framework.

For a program point d and an array x, we set $DEF(d,x) = 1$ if d is a definition of an element of x, and 0 otherwise. Similarly, we set $USE(d,x) = 1$ if d is a use of an element of x, and 0 otherwise.

A definition d of an element of an array x is called *def–safe* with respect to x if it can be statically shown for all execution paths containing d that x is injective immediately after execution of d.

For each program point q and each array x to be tested, we compute, based on reaching definitions (see e.g. [ZC90]), a boolean value $SAFEDEF(q,x)$ which equals 1 if only def–safe definitions of x reach q, and 0 otherwise.

Let u be any program point. u may contain a use of an element of x. Assume that definitions $d_1, d_2, ..., d_k$ of x reach u, via paths $\pi_1, \pi_2, ..., \pi_k$ [1]. Then we call u *use–safe* with respect to x if

[1] Note that these d_i are usually a conservative overestimation of the actual definition of the array element(s) used in u.

for all $i = 1, 2, ..., k$, definition d_i is def–safe or there is a run–time test *test(x)* placed on path π_i between d_i and u. Otherwise we call u *ambiguous* with respect to x.

Let $TEST(q, x)$ denote a function that returns 1 if there is a run–time test *test(x)* for injectivity of x placed immediately before program point q, and 0 otherwise.

Use–safety of any program point q with respect to x can be computed by standard dataflow analysis techniques, see e.g. [ZC90]. Let $SAFEUSE(q, x)$ denote the characteristic function of use–safety to be computed for each program point q with respect to each array x. Value 0 means ambiguous and 1 means use–safe. (1) If q is not a meeting point of joining control flow paths, let u denote its immediate predecessor in the control flow graph. If q is not a use of x and u is not a definition of x, q inherits the value $SAFEUSE(u, x)$ from u. If u is a definition of x, i.e. $DEF(u, x) = 1$, then also $SAFEDEF(q, x)$ influences $SAFEUSE(q, x)$. (2) At a meeting point q of joining control flow paths $\pi_1, \pi_2, ...$, with the direct predecessors of q being $u_1 \in \pi_1$, $u_2 \in \pi_2$ etc., we can assume use–safety only if for each u_i, def–safety holds for u_i or, (if u_i is not a definition of x) use–safety holds already for u_i. As general iteration equation for any program point q, we obtain

$$
\begin{aligned}
SAFEUSE(q, x) \quad = \quad & TEST(q, x) \\
& \vee \{ SAFEUSE(q, x) \wedge \\
& \quad \wedge_i (\; SAFEDEF(q_i, x) \; \vee \; (\neg DEF(q_i, x) \wedge SAFEUSE(q_i, x)) \;) \; \}
\end{aligned}
$$

provided that we have initialized $SAFEUSE(q, x) = 1$ for all q and x. The predicates *TEST* and *SAFEDEF* are constant during the computation of *SAFEUSE*. Since the sequence of $SAFEUSE(u, x)$ values over the iterations is, obviously, monotonically decreasing and bounded by 0, the iterative algorithm will converge.

Our goal is to eliminate all ambiguous uses for array x in whose injectivity we are interested in. A very general method to achieve this is given by the following simple nondeterministic algorithm:

Algorithm: *Elimination of ambiguous uses*
for all q **do** $TEST$(q,x)=0;
forever do
  (re)compute $SAFEUSE(q, x)$ for all program points q and arrays x
  if $USE(q, x) \leq SAFEUSE(q, x)$ for all q **then break**;
  place a run–time test appropriately
od

Obviously, a run–time test should be placed in such a way that at least one $SAFEUSE(q, x)$ value for a "real" use q changes from 0 to 1 when recomputing all *SAFEUSE* values in the next iteration. The goal is to place the run–time tests in such a way that the total run time overhead induced by them in the speculatively parallelized program is minimized, This constitutes an interesting static optimization problem. Clearly it requires static run time prediction techniques which compute statement frequencies from (estimated or profiled) loop counts, true ratios etc. for the parallelized program [Fah93, van96].

If there is no fill-in to be provided for in the program, we probably can completely eliminate all tests but the first one, which is located immediately after building the organizational data structures for the sparse matrix.

4.4 Symbolic data flow analysis for cross matching

Now let us return to the example of the subsection 4.2. There remains the problem to maintain the original dataflow properties also for the summary nodes, indicated by the question marks in the `MXGV` parameters.

For instance, since `grad[`i`]` is not statically known, it is not known which elements of `A` and `x` are accessed. Statically, we can only provide a conservative overestimation, by assuming that the entire second dimension of `A`, as allocated, will be accessed (assuming `grad[`i`]` will not be greater than the allocated size) and an underestimation which takes care of the possibility that `grad[`i`]` may be zero, thus there may be no access at all. — The same holds for the access to array `x`. A conservative overestimation assumes that *all* elements of `x` are accessed while an underestimation must assume an empty vector access in the worst case.

When working speculatively, we can nevertheless assume here that the *entire* sparse matrix is accessed in `MXGV`. Although this is no means to get *absolute* access and data flow information, it can support determining *relative* data flow between summary nodes by symbolic analysis. For instance, if an entire sparse matrix is written by a summarized computation, and read by a following one, and no organizational or data variables are killed in between, then a FLOW cross edge between these two summary nodes can be installed. Along this cross edge, the two summary nodes may be merged if an appropriate pattern covering them exists.

4.5 Speculative loop distribution

Consider the following code fragment taken from the `SPARSE-BLAS` [Gri84] 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 as for the speculative recognition of sparse matrix operations we can here, by a run–time test, check whether `INDX` is injective. Note that, if the computation considered is part of a larger sparse matrix computation, the result of the injectivity test may be reused in the run time test for that computation, verifying that `INDX` is an organizational variable of a sparse matrix data structure.

We thus assume that `INDX` is injective in the range `1:NZ`. As now there remain no loop–carried dependencies, we can apply loop distribution [ZC90] to the `I` loop:

```
   < assumes injectivity of INDX(1:NZ) >
   DO 10 I = 1, NZ
       X(I)        = Y(INDX(I))
10 CONTINUE
   DO 11 I = 1, NZ
       Y(INDX(I)) = 0.0D0
11 CONTINUE
```

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

```
< assumes injectivity of INDX(1:NZ) >
VGATHER(I=[1:NZ],X,Y,INDX)
VXINIT(I=[1:NZ],Y,INDX,0.0D0)
```

Speculative loop distribution saves the original program structure for the code generation phase, in order to handle the case that the run time test fails.

4.6 A parallel algorithm for the injectivity tests

We apply a parallel algorithm to speed up the time overhead due to the injectivity test.

For a shared memory parallel target machine we apply an algorithm similar to bucket sort 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, $[1 : m]$). 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[2] 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 which may be appropriately embedded into the present processor network 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 Extension for linked lists

Linked list data structures cause operations on them (traversal, insert/delete) to be inherently sequential in nature. 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 is clearly possible if all operations on objects of this type have been recognized.

5.1 Alias analysis

Pointer alias analysis addresses the problem of computing and representing the points–to relation, i.e. to determine for each program object v (variables and dynamically allocated blocks of memory) which pointer variable at which point of time definitely points to v (*must-alias*) or may point to v (*may-alias*). The term "point to" is used here in the sense of addressability, i.e. also

[2] This should be done by an atomic fetch&increment operation such as `mpadd(&(counter[a[j]]),1)` on the SB-PRAM, cf. [KS97].

relates to multiple dereferencing and structure component offsets. Most practically relevant alias problems are NP–hard [LR91] or even intractable. Generally, one falls back upon conservative approximations, e.g. by limiting the number of indirections considered (*k–limiting*).

Current alias analysis methods can be classified into two groups: First, dynamically allocated program objects can, following their type declarations, be considered as atomic units ("*store–less*" alias analysis, [HN90, CWZ90, LR92, CBC93, Deu94, EGH95]). This causes problems at naming of heap objects and expects a certain "benignity" of the program to be analyzed, e.g., that there is no access via pointer arithmetics beyond the (allocated) boundaries of such an object. On the other hand, it potentially allows for identification of pointers to tree or directed acyclic graph structures [HN90].

Second, not relying on any assumption of benignity, one could consider structure–free storage blocks where, in principle, each memory location may contain a pointer [CWZ90, EGH94, WL95, Ruf95]. The latter scenario is unavoidable for the correct handling of type casting, unions, pointer arithmetics and boundary violation of arrays. Nevertheless, this renders the identification of recursive data structures more difficult.

A different classification of current approaches considers whether analysis is done only once per procedure (context–insensitive, imprecise), as in [CWZ90, Deu94, Ruf95], or recomputed for each possibly occurring call context of a procedure (context–sensitive, precise) as in [CBC93, EGH94, WL95]. The latter category generally produces exact results but has exponential run time in the worst case. Compression techniques [WL95] may help here.

We are currently investigating which of these methods is most suitable for our purpose, i.e. gives most information for the reconstruction of sparse matrix data structures and computations. On the other hand, our program comprehension techniques may support more exact approximations for alias analysis.

5.2 Examples for patterns on linked lists

As an illustration, we extracted the following patterns on linked lists from `SPARSE` [Kun88].

Elementwise Order 1 operations on a linked list		
name	parameters	could look like
`LINIT`	`p`, `first`, `->next`, `->real`, `init`	`p=first;` `while (p!=NULL) { p->real=init; p=p->next}`
`LCNT`	`k`, `p`, `first`, `->next`	`p=first;` `while (p!=NULL) { k++; p=p->next}`
`LSRCHCOND`	`p`, `first`, `->next`, `cond`, `->key`	`p=first;` `while (p!=NULL && !(cond(p->key))) p=p->next`
`LSCATTER`	`p`, `first`, `->next`, `x`, `->row`, `->real`	`p=first;` `while (p!=NULL) { x[p->row]=p->real; p=p->next}`
`LGATHER`	`p`, `first`, `->next`, `->real`, `x`, `->row`	`p=first;` `while (p!=NULL) { p->real=x[p->row]; p=p->next}`

`VLAADDSL`	`p`, `first`, `->next`,
	`x`, `->row`, `s`, `->real`
		`p=first;`
		`while (p!=NULL)`
		`{ x[p->row]=x[p->row]±s*p->real; p=p->next}`

Frequently accessed list elements are temporarily stored in an array to eliminate the overhead of repeated list traversal. — The initializer `p=first` is optional.

Order 1 reductions on a linked list		
name	parameters	could look like
`LSUM`	`p`, `first`, `->next`, `s`, `->real`, `offset`	`p=first; s=offset;` `while (p!=NULL)` `{ s=s±p->real; p=p->next}`
`SLSVLP`	`p`, `first`, `->next`, `s`, `x`, `->row`, `->real`, `offset`	`p=first; s=offset;` `while (p!=NULL)` `{ s=s±x[p->row]*p->real; p=p->next}`

The initializers `p=first` and `s=offset` are optional.

Order 2 operations on a linked list		
name	parameters	could look like
`MLINIT`	*range*(`i`), `p`, `first[:]`, `->next`, `->real`, `init`	`DO i=`*range*(`i`) `LINIT(p,first[i],->next,->real,init);`
`ML`	*range*(`i`), `p`, `first[:]`, `->next`, `->real`, `sol`	`DO i=`*range*(`i`) `SLSVLP(p,first[i],->next,sol[i],->real,sol[i]);`

In principle it is possible to apply the same combination of compile-time pattern matching and run-time tests as shown above. One difference is that traversing a linked list to check for injectivity is strictly sequential in nature; thus the run time test is not parallelizable. Nevertheless, on a fine–grained synchronous parallel architecture such as the SB-PRAM, instruction level parallelism may be explored e.g. by software pipelining.

Furthermore, extensive alias analysis has to guarantee that there are no other pointer variables that may be used to access one of the linked list elements; otherwise a safe replacement of the data structure would be impossible.

6 Related Work

[RAP95] describes an improvement to the inspector–executor method by recognition of simple reductions and privatizable scalar variables. Their inspector uses parallelized tests on independent array accesses similar to our parallelized injectivity test.

7 Conclusion and future work

We have tried to give an impression of the problems and challenges in applying program comprehension techniques to sparse matrix computations with the goal of automatic parallelization.

We see that it is possible to perform speculative program comprehension where static analysis does not provide sufficient information, where the static tests on the syntactic properties and consistency of the organizational variables are complemented by suitably placed run–time tests. Only if both tests are passed, the target program may branch to an optimized parallel sparse matrix library routine. Otherwise, conservatively generated parallel code must be executed. Moreover, we have given simple parallel algorithms for verifying injectivity at run time.

These techniques are also applicable in a non–parallel environment; they could be used 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.

Future research[3] will address several directions: First, the list of patterns for the various data structures will be completed. Second, the syntactic matching rules will be implemented, e.g. on top of the existing PARAMAT pattern recognizer. More efficient parallel run time tests for distributed memory machines may be devised, as well as an algorithm for optimization of the placement of the run time tests. Ideally, we would finally be able to show (by running this prototype compiler on a set of benchmark programs) that, at least for the set of application programs considered, our system involves less run–time overhead than the classical inspector–executor variant.

Finally, the patterns and matching rules for linked list data structures have to be formally defined and implemented. The interdependence with pointer alias analysis has to be investigated.

Moreover, we will address automatic array distribution and redistribution. It seems to be a reasonable idea to integrate run–time distribution schemes for sparse matrices [RZ93] 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.

References

[Ben86] J. Bentley. *Programming Pearls.* Addison–Wesley, Reading, MA, 1986.

[BKK93] 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.

[BW96] 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.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Side effects. In *ACM SIGPLAN Principles of Programming Languages*, pages 232–245, 1993.

[CWZ90] D. Chase, M. Wegman, and F. Zadeck. Analysis of Pointers and Structures. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 296–310, 1990.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 230–241, 1994.

[DSv93] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing Analysis and Indirect Access to Distributed Arrays. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing.* Springer–Verlag, Aug. 1993.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 242–256, 1994.

[EGH95] M. Emami, R. Ghiya, and L. Hendren. Connection Analysis: A Practical Interprocedural Heap Analysis for C. In *Languages and Compilers for Parallel Computing*, pages 515–533, Aug. 1995. Springer LNCS 1033.

[ESH93] Kalluri Eswar, P. Sadayappan, and Chua-Huang Huang. Compile-Time Characterization of Recurrent Patterns in Irregular Computations. In *Proc. Int. Conf. on Parallel Processing*, pages II–148–155, 1993.

[Fah93] Thomas Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Universität Wien, 1993.

[3] These issues are the subject of a new research project called SPARAMAT which is funded by DFG and will begin in 1997.

[Gri84] R.G. Grimes. SPARSE-BLAS basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [NET], 1984.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[Keß96] Christoph W. Keßler. Pattern-driven Automatic Parallelization. *Scientific Programming*, 5:251–274, 1996.

[KK93] Ken Kennedy and Ulrich Kremer. Initial Framework for Automatic Data Layout in Fortran D: A Short Update on a Case Study. Technical Report CRPC-TR93324-S, Center for Research on Parallel Computation, Rice University, Houston, TX, July 1993.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[KMR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. In *ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 177–186, 1990.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD machines. In J. JaJa, editor, *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 416–423. Los Alamitos: IEEE Computer Society Press, 1990.

[KS97] 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.

[Kun88] K.S. Kundert. SPARSE 1.3 package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [NET], 1988.

[KW94] Peter M.W. Knijnenburg and Harry A.G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Dept. of Computer Science, Leiden University, 1994.

[LB96] Antonio Lain and Prithviraj Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers. In *Proc. 10th Int. Conf. on Supercomputing*, pages 1–9. New York: ACM Press, 1996.

[LC90] Jingke Li and Marina Chen. Index Domain Alignment: Minimizing Cost of Cross–referencing between Distributed Arrays. In J. JaJa, editor, *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 424–433. Los Alamitos: IEEE Computer Society Press, 1990.

[LR91] W. Landi and B. Ryder. Pointer-induced Aliasing: A Problem Classification. In *ACM SIGPLAN Principles of Programming Languages*, pages 91–103, Jan. 1991.

[LR92] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 235–248, 1992.

[NAG88] Numerical Algorithms Group NAG. MA28/MA30 sparse LU factorization routines from the Harwell Scientific Library, written in Fortran 77. Source code available via netlib [NET], 1988. http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html.

[NET] NETLIB. Collection of free scientific software. Accessible by anonymous ftp to netlib2.cs.utk.edu or netlib.no or send e-mail to netlib@netlib.no containing the message send index.

[PTVF92] 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.

[RAP95] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. In *Proc. 9th Int. Conf. on Supercomputing*, pages 137–146. New York: ACM Press, July 1995.

[Ruf95] E. Ruf. Context-Insensitive Alias Analysis Reconsidered. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 13–22, 1995.

[RZ93] L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. In H.J. Sips, editor, *Proc. 4th Workshop on Compilers for Parallel Computers*, pages 154–167. Delft University of Technology, Dec. 1993.

[SBV93] A. Frank van der Stappen, Rob H. Bisseling, and Johannes G.G. van de Vorst. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Runtime Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[van96] Arjan van Gemund. *Performance Modeling of Parallel Systems.* PhD thesis, Technical University of Delft (NL), 1996.

[WL95] R. Wilson and M. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *ACM SIGPLAN Programming Language Design and Implementation*, pages 1–12, 1995.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.