Applicability of Program Comprehension to Sparse Matrix Computations*

Christoph W. Keßler

FB 4 – Informatik, Universität Trier, D-54286 Trier, Germany
e-mail: `kessler@psi.uni-trier.de`

Abstract. Space–efficient data structures for sparse matrices typically yield programs in which not all data dependencies can be determined at compile time. Automatic parallelization of such codes is usually done at run time, e.g. by applying the inspector–executor technique, incurring tremendous overhead. — Program comprehension techniques have been shown to improve automatic parallelization of dense matrix computations. We investigate how this approach can be generalized to sparse matrix codes. We propose a speculative program comprehension and parallelization method. Placement of parallelized run–time tests is supported by a static data flow analysis framework.

Data structures for sparse matrices storing only the nonzero elements save 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. Irregular sparsity patterns are usually defined by run-time data. Typical data structures in Fortran77 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. C implementations may also use linked lists which enhance dynamic insertion and deletion of elements.

While the problems of automatic parallelization for *dense* matrix computations are, meanwhile, well understood and sufficiently solved, these problems are, for *sparse* matrix computations, solved in a rather conservative way, e.g. by run–time parallelization techniques such as the inspector–executor method [6] or run–time analysis of sparsity patterns for load–balanced array distribution [9], because indirect array indexing or pointer dereferencing makes exact static access analysis impossible. Run-time analysis, though, usually incurs tremendous overhead which is not always likely to be weighed out by parallel speedup, as the good volume–to–surface ratio of computation to communication typical for dense matrix computations is usually not present in sparse ones.

For automatic parallelization of dense matrix computations and other completely statically analyzable codes, program comprehension techniques appeared to be useful [4]. After suitable preprocessing, the intermediate program representation — abstract syntax tree and/or program dependence graph — is submitted to a concept recognition tool which locally identifies code fragments for which there exist special code transformations or particular parallel algorithms tailored to the target machine. In the back–end phase, these code pieces can be replaced by suitable parallel implementations. The information derived in the recognition phase also supports automatic data layout and performance prediction.

We are interested in how far this approach can be extended to sparse matrix codes. One problem we are faced with is that there is no standard data structure to store a sparse matrix, compared to the two–dimensional array which is the "natural" storage scheme for the dense matrix. A survey of sparse storage formats is given e.g. in [1] and [11]. We have examined several representative source

* For the full version of this paper see `http://www.informatik.uni-trier.de/~kessler/sparamat`

codes for implementations of basic linear algebra operations like dot product, matrix–vector multiplication, matrix–matrix multiplication, or LU factorization for sparse matrices [2, 3, 5] and recorded a preliminary list of basic computational kernels (also often called "templates" or "concepts") for sparse matrix computations, together with their various syntactical appearances ("idioms").

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

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

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.

Static and dynamic matching of sparse matrix concepts. Safe identification of a sparse matrix operation consists of (1) a test for the syntactical properties of this operation, which can be performed by concept recognition at compile time, and (2) a test for the semantic properties which may partially have to be performed at run time.

Testing the static part is, in principle, not hard. Just in the same way as we did for dense matrix operations [4], we work bottom–up on the abstract syntax tree, incrementally compute dataflow relations ("cross edges"), and deterministically identify concepts based on already matched subconcepts. For ef-

Fig. 1. Some common concepts in sparse matrix computations

This survey is far from being exhaustive. It is rather an illustration of our intention. DO *rng*(i) stands for a common for resp. DO loop where loop variable i iterates over a regular range *rng*(i). The example computations given in the column "could look like" serve only as an illustration; there may be many more possibilities to express that concept. It is the job of program comprehension to abstract from such syntactical variations.

Concepts for some elementwise indirect vector operations		
name	parameters	could look like
VGATHER	*rng*(i), x, y, ix	DO i=*rng*(i) x[i]=y[ix[i]]
VSCATTER	*rng*(i), x, ix, y	DO i=*rng*(i) x[ix[i]]=y[i]
VXAADDSV	*rng*(i), x, ix, x, u, v	DO i=*rng*(i) x[ix[i]]=x[ix[i]]±u*v[i]
Concepts for some indirect 1D reductions		
VXSUM	*rng*(i), s, y, ix, c	s=c; DO i=*rng*(i) s=s±y[ix[i]]
VXDOTV	*rng*(i), s, y, z, iz, c	s=c; DO i=*rng*(i) s=s±y[i]*z[iz[i]]
VXMAXVAL	*rng*(i), s, x, ix, c	s=c; DO i=*rng*(i) if (s>x[ix[i]]) s=x[ix[i]]
VXMAXLOC	*rng*(i), k, x, ix, t	k=t; DO i=*rng*(i) if (x[ix[k]]<x[ix[i]]) k=i

s, c, k, t are scalar (not indexed by loop variable i). Initializers s=c, k=t are optional.

Concepts for some kinds of sparse matrix-vector multiplication		
MXFV	*rng*(i), j, first, a, b, col, C, init	DO i=*rng*(i) a[i] = init[i]; DO i=*rng*(i) DO j=first[i],first[i+1]-1 a[i]=a[i]±b[col[j]]*C[j]
VMXF	*rng*(i), j, first, a, b, col, C, init	DO i=*rng*(i) a[i] = init[i]; DO i=*rng*(i) DO j=first[i],first[i+1]-1 a[col[j]]=a[col[j]]±b[i]*C[j]

Explicit names like 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 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 concepts for them. — The initializing loops are optional.

ficiency reasons, the matching rules are hierarchically organized ("concept hierarchy graph"), following the natural hierarchical composition of computations by applying loops and sequencing to subcomputations. Normalizing transformations, such as rerolling of unrolled loops, are done whenever applicable.

Speculative concept matching and run time tests. 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 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 implicit row–compressed format (MXFV resp. VMXF, see Fig. 1) we have to check for the following conditions on the concept parameters

1. first[i] $\leq$ first[$i+1$] $\leq n$ for all $i \in$ *rng*(i)

2. array `col[]` is injective over the index domain [`first[`i`]:first[`$i+1$`]`-1] for all $i \in rng(\mathtt{i})$

These conditions may be checked for separately. Here we consider the verification of injectivity; the other conditions can be checked in a straightforward way. We use a dataflow framework to minimize the number of generated injectivity tests.

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. [10]), 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 $\pi_1, \pi_2, ..., \pi_k$ [2]. Then we call u *use–safe* with respect to x if for all $i = 1, 2, ..., k$, definition d_i is def–safe or there is a run–time test *test(x)* for injectivity of 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)* 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. [10]. 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 u is not a definition of x, q inherits the value $SAFEUSE(u,x)$ from u. Moreover, $TEST(q,x)$ and $SAFEDEF(u,x)$ influence $SAFEUSE(q,x)$. (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

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

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

We must eliminate all ambiguous uses for array x in whose injectivity we are interested in. This is described by the following simple nondeterministic

```
Algorithm: Elimination of ambiguous uses
for all $q$ do $TEST$(q,x)=0 od
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 test(x) appropriately od
```

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

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, where the weights are given by the statement frequencies computed from (estimated or profiled) loop iteration counts, true ratios etc. for the parallelized program. This constitutes an interesting static optimization problem.

Parallelized injectivity test. For a shared memory parallel target machine we apply an algorithm similar to bucket sort[3] to test injectivity of an integer array in parallel. — On a distributed memory system, an existing parallel sorting algorithm can be extended for our purposes (for details, see the full paper).

Future research[4] will address several directions: The list of concepts for the various data structures will be completed. The syntactic matching rules will be implemented, e.g. on top of the existing PARAMAT concept 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. Finally, the concepts 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 [9] 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.

References

1. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.
2. I. S. Duff. `MA28` – a set of Fortran subroutines for sparse unsymmetric linear equations. Tech. rept. AERE R8730, HMSO, London. Sources at netlib [7], 1977.
3. R. Grimes. `SPARSE-BLAS` basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [7], 1984.
4. C. W. Keßler. Pattern-driven Automatic Parallelization. *Scientific Programming*, 5:251–274, 1996.
5. K. Kundert. `SPARSE 1.3` package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [7], 1988.
6. 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*, pages 140–152. ACM Press, July 1988.
7. 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`.
8. L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. 8th ACM Int. Conf. on Supercomputing*, pages 33–43. ACM Press, July 1994.
9. M. Ujaldon, E. Zapata, S. Sharma, and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *J. of Parallel and Distr. Computing*, 38(2), 1996.
10. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison–Wesley, 1990.
11. Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer, 1991.

[3] A similar test on independent array accesses was suggested in [8].

[4] The new research project SPARAMAT funded by DFG will begin in 1997.