

Aspect-Oriented Programming of Sparse Matrix Code

John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar,
Tatiana Shpeisman

Published in proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA. December 1997. Springer-Verlag LNCS 1343.

© Springer-Verlag Berlin Heidelberg 1997.

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Aspect-Oriented Programming of Sparse Matrix Code

John Irwin, Jean-Marc Loingtier,
John R. Gilbert[^], Gregor Kiczales, John Lamping,
Anurag Mendhekar, Tatiana Shpeisman
©Copyright 1997, XEROX Corporation. All rights reserved.
Xerox Palo Alto Research Center*

The expressiveness conferred by high-level and object-oriented languages is often impaired by concerns that cross-cut a program's basic functionality. Execution time, data representation, and numerical stability are three such concerns that are of great interest to numerical analysts. Using aspect-oriented programming we have created AML, a system for sparse matrix computation that deals with these concerns separately and explicitly while preserving the expressiveness of the original functional language. The resulting code maintains the efficiency of highly tuned low-level code, yet is ten times shorter.

Introduction

Traditionally there are two ways of writing sparse matrix code. One is to write in a high-level matrix language that offers fast prototyping and easily readable code. Unfortunately the result is usually too slow to use for real problems. The problem is that efficient sparse matrix code requires careful choice of data structures depending on the structure of the computation. The data structures must store only the non-zeros, and do so in a way that fits well with the computation's needs. Further, the access patterns of the computation must be exploited to achieve efficient access and update while allowing the computation to avoid operations on zeros.

The other approach is to write in a low-level language such as C or Fortran, directly implementing and exploiting the most appropriate data structures. While

[^] The work of this author was partially supported by DARPA under contract DABT63-95-C0087.

* 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. gregor@parc.xerox.com,
lamping@parc.xerox.com

this approach can give good performance, such a program generally takes much longer to write, and is less readable and less maintainable. Neither alternative is attractive.

We used the aspect-oriented programming (AOP) approach [1] [2] to resolve this dilemma. We built a language environment called AML which lets the user write high-level sparse matrix code along with annotations describing an efficient implementation. We picked a set of representative sparse matrix algorithms to use as test cases; one such algorithm is described in detail in this paper. As efficiency is a quantitative result, we measured the speed of our code compared to standard versions of the same algorithm. The result was quite satisfactory – our code matches the speed of the standard version based on the same algorithm, yet is considerably shorter and less complex.

An example: sparse LU factorization

We will demonstrate our approach on a typical numerical analysis algorithm: LU factorization by Gaussian elimination with partial pivoting (GEPP). An engineer would normally invoke sparse GEPP from a library rather than coding it by hand; in that sense this example is artificial. However, GEPP is representative of a large and important class of algorithms, namely incomplete factorization preconditioners [3], that are often coded by hand for domain-specific numerical modeling applications.

The basic algorithm for Gaussian elimination without partial pivoting can be written in Matlab [4] as:

```
function [L,U] = lu(A);
[m,n] = size(A);
L = zeros(n,n);
U = zeros(n,n);
for j = 1:n
    t = A(:,j);
    for k = 1:j-1
        if t(k) ~= 0
            t = t - t(k) * L(:,k);
        end;
    end;
    U(1:j,j) = t(1:j) ;
    L(j+1:n,j) = t(j+1:n)/t(j);
end;
```

This code will be our standard of elegance. Our goal is to let the programmer describe how to add partial pivoting and describe how to implement the algorithm efficiently, while keeping as much of the elegance of the above code as possible.

An optimized version of GEPP, written to a custom designed C++ sparse matrix library, is too long to include here, but the inner loop gives the feel of the whole:

```
for (SpaNzInorderIterator<Permutation,Range>
     iter(t, p, r1); iter; ++iter) {
    int k = iter.get_row();
    SparseVector L_col_k(L, k);
    spaxpy(t, p, NoRange, iter, L_col_k, p, NoRange,
          -(iter.get_value()), NoTrans);
}
```

This algorithm is as efficient as we could make it while still using library routines rather than open coding the entire algorithm. The code is a tangled mess. It is difficult to see the base algorithm in among all the other issues.

Among the additional issues addressed by this code are: The use of partial pivoting (all the 'p' arguments in the code), the merging of several logically distinct operations into a single spaxpy call, the use of special data structures, such as sparse accumulators [5]. Our goal is to address these issues in a way that avoids the tangling above.

An AOP solution

In AOP, a system is broken down into *components* and *aspects*. Components correspond to units of functionality; aspects are issues that cross-cut component boundaries. Stated another way, the component program describes the functionality of the system, that part normally thought of as the functional intent of the programmer, as separated from the less algorithmic concerns. These latter concerns are contained in aspect programs.

A complete AML program consists of the basic algorithm written in the component language, and annotations describing an efficient implementation of the component program. These annotations are written in several different aspect languages. Finally, there is a tool called an Aspect Weaver that takes all this code and produces an executable program. The remainder of this section describes first the component language, then each aspect language in turn along with the issue it addresses from our LU example. The last section describes the aspect weaver and performance.

The component language

We chose a component language that approximates the familiar Matlab language [4] for expressing the basic algorithm. Starting with Matlab, we added an additional iteration construct, `for nzs`, to explicitly express iteration through

the non-zeros of a vector. We also imposed some restrictions to simplify compilation, such as not allowing different types of values to be assigned to the same variable.

Data representation aspect

The data representation aspect allows declarations along 5 different axes of data representation decisions:

Axis	Range
Element Type	integer, real, complex
Dimension	scalar, vector, row-vector, matrix
Representation	full, sparse, spa
Ordering	ordered, unordered
Orientation	by-rows, by-columns

The first two entries, element type and dimension, are the components of what is traditionally thought of as the type of the object. They describe what kind of data is visible to the base program.

The three other entries are additional representation information that is not visible to the base program. These other components determine the data structure that the implementation will use. They do not affect the behavior as seen from the base program, but they can dramatically affect performance. Representation specifies either full, sparse, or sparse accumulator (SPA). Ordering applies only to sparse or SPA representations, and indicates whether the non-zero elements should be maintained in order. Orientation applies only to sparse matrices, and indicates whether they should be stored by rows or columns.

Implicit permutation aspect

We found a need for an additional aspect to handle the implicit permutation of vectors and matrix rows and columns, which occur frequently in numerical code, to address issues like: ordering for sparsity [6], parallel partitioning [7], block triangular form [8], and effective preconditioning [9]. An implicit permutation declaration, written with a `view` annotation, like

```
view A through (p, :)
```

allows implicit permutations to be dealt with in one place, rather than being smeared throughout the code. It says that all references to the specified arrays should act as if the arrays were permuted as specified. Put another way, all indexing first goes through the permutation vectors. Changing the contents of the permutation vectors changes the effective permutation of the array.

Operator fusion aspect

The operator fusion aspect is responsible for indicating patterns of operations that can be efficiently performed as a unit by library routines. Unlike the other aspects, code for this aspect describes how to utilize the library in general, rather than how to implement a part of a particular application program. Since it is the library writer rather than the application programmer who writes the code for this aspect, we don't show it in this paper.

AML code for LU

The complete code for LU, using these languages, is:

```
function [L,U,p] = lu(A);
|declare real sparse matrix A, L, U;
|declare real          scalar v;
|declare int          scalar m, n, j;
[m,n] = size(A);
L = zeros(n,n);
U = zeros(n,n);
||declare permutation p;
||p=[1:n];
||view A,L,U through (p,:)
|   for j = 1:n
|       declare SPA t;
||       view t through p
|           t = A(:,j);
|           for nzs k in order in t(1:j-1)
|               t = t - t(k)*L(:,k);
|           end;
||           [v,piv] = max(abs(t(j:n)));
||           piv = piv+j-1;
||           p([j,piv]) = p([piv,j]);
|           U(1:j,j) = t(1:j);
|           L(j+1:n,j) = t(j+1:n)/t(j);
||       end view;
|   end;
||end view;
```

The data representation declarations have been marked with a single bar to the left. Double bars mark code that deals with implicit permutations, including code in the component language that adjusts the permutation.

To our eye, this code does a good job of preserving the structure of the basic algorithm while expressing the desired efficient implementation.

The weaver

In most respects the weaver is a very simple compiler. Rather than trying to make smart inferences, the power of our weaver comes from the crucial domain information given by the most knowledgeable person: the user.

The weaver is built of several passes that can be seen as successive transformations applied to an Abstract Syntax Tree (AST) produced by the parser. Each pass consists of rewriting rules which are supported by a walker and a pattern matcher. These rules translate the AST into a new language. The language at each pass is nearly a proper superset of the language at the next pass (and thus of all later passes also). Therefore a particular pass can choose to rewrite part of the tree, or leave it for a later pass to deal with.

The first passes deal with information dissemination. The parser gives us a single AST that contains the component program, the implicit permutation aspect program, and the data representation aspect program. The weaver then propagates the permutation and representation aspects throughout the tree, according to their respective scopes. Since permutations stay in force across function calls, the weaver must note any component functions that are called with permuted arguments so that they can be automatically instantiated for the kinds of arguments passed at each point in the tree.

The latter passes of the weaving process can be roughly described as code generation. First there is a canonicalization pass, where the large number of possible AST shapes, many of which represent the exact same computation, are reduced to a smaller set of regularized forms. The weaver then applies the operator fusion aspect program. The operation fusion code consists of a set of translation rules, ordered by decreasing level of fusion. In a single top-down walk of the tree, each rule in turn is applied to the subtree at that point. If the rule matches, it completely translates that subtree into the output language, possibly recursing if it contains unfusible children. In addition to translating fused operations, simple operations are also translated by the same process. The whole code generator is run over the tree until the tree stops changing. At this point the whole tree has been translated into the final language, which is a transparent representation of C/C++. A post processor handles the conversion to C++, low-level compiling, and linking.

Results

The goal of AML is to allow sparse matrix code that is both easy to understand and modify, and efficient. This section measures the expression of LU in AML against those goals. We compare sparse LU written in AML with two other implementations: GPLU [10], a well-known Fortran version of the algorithm, also implemented (in C) in the Matlab version 4 internal library [5]; and SuperLU [11], the best performing research code, which uses much more

complex algorithms to optimize cache behavior and to exploit structural symmetry.

Complexity

Quantifying the comprehensibility and modifiability of code is a notoriously difficult problem. There are many measures of code complexity [12], none of them entirely satisfactory. Using just the very simple measure of source lines of code, AML takes about 30 lines of code, GPLU takes 300 lines, and SuperLU takes 3000 lines. In other words, the AML code is an order of magnitude shorter than the GPLU implementation. The size comparison for the SuperLU code is not as pertinent, since it is performing a much more complicated algorithm.

Efficiency

Since the time to execute a sparse matrix code can depend on details of the actual matrix being manipulated, we measured the efficiency of the three implementations against a suite of standard test matrices from the Harwell-Boeing sparse matrix library [13]. The measurements were run on a SPARC 20, using the GNU compilers, running under SunOS 4.1.3.

The results, summarized in the table below, show that the AML implementation ranges from slightly slower than GPLU for easier matrices to essentially the same speed for hard matrices. Both are about 3 times slower than SuperLU. We include the comparison to SuperLU for completeness, even though it is not the kind of algorithm that the users of AML are likely to write. SuperLU deals with a complicated low-level issue that neither AML nor GPLU does, namely blocking data to exploit the memory hierarchy. One area of future work in AML would be to design an aspect to address this issue, hopefully yielding simpler code while retaining the performance of SuperLU.

#	name	size	nonzeros	AML (sec)	GPLU (sec)	SuperLU (sec)
1	gemat11	4929	33185	0.7	0.5	0.4
2	memplus	17758	99147	1.2	0.7	1.0
3	mcfe	765	24382	1.4	1.0	0.4
4	rdist1	4134	9408	4.1	3.3	2.1
5	orani678	2529	90158	4.2	3.8	0.6
6	jpwh991	991	6027	3.7	3.9	1.5
7	sherman5	3312	20793	6.2	6.0	2.1
8	linsp3837	3937	25407	8.6	7.9	3.4
9	lins3937	3937	25407	9.3	9.0	4.0
10	orsreg1	2205	14133	13.2	12.9	4.3
11	sherman3	5005	20033	12.9	13.9	4.5
12	saylr4	3564	22316	23.1	24.8	7.5
13	goodwin	7320	324772	150.2	161.6	41.1

References

1. Kiczales, G., *et al.*, *Aspect Oriented Programming*. 1996, Xerox PARC: <http://www.parc.xerox.com/spl/projects/aop/position.htm>.
2. Kiczales, G., *et al.* *Aspect-Oriented Programming*. in *European Conference on Object-Oriented Programming*. 1997. Finland: Springer-Verlag.
3. Saad, Y., *Iterative Methods for Sparse Linear Systems*. 1996, Boston: PWS Publishing Company.
4. Mathworks, *MATLAB User's Guide*. 1992, The Mathworks Inc.
5. Gilbert, J.R., C. Moler, and R. Schreiber, *Sparse Matrices in MATLAB: Design and Implementation*. *SIAM J. Matrix Anal. Appl.*, 1992. **13**: p. 333-356.
6. George, A. and J.W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. 1981: Prentice-Hall.
7. Shewchuk, J.R. and D.R. O'Hallaron, *Archimedes*. 1996: <http://www.cs.cmu.edu/~quake/archimedes.html>.
8. Pothen, A. and C.-J. Fan, *Computing the block triangular form of a sparse matrix*. *toms*, 1990. **16**: p. 303--324.
9. Duff, I.S. and G. Meurant, *The effect of ordering on preconditioned conjugate gradients*. *BIT*, 1989. **29**: p. 685--657.
10. Gilbert, J.R. and T. Peierls, *Sparse Partial Pivoting in Time Proportional to Arithmetic Operations*. *SIAM J. Sci. Statist. Comput.*, 1988. **9**: p. 862-874.
11. Demmel, J.W., *et al.* *A Supernodal Approach to Sparse Partial Pivoting*. in *ILAY Workshop on Direct Methods*. 1995. Toulouse, France.
12. Henry, S. and D. Kafura, *Software Structure Metrics Based on Information Flow*. *IEEE Transactions on Software Engineering*, 1981. **SE-7**: p. 509--518.
13. Duff, I.S., R.D. Grimes, and J.G. Lewis, *Sparse Matrix Test Problems*. *ACM Transactions on Mathematical Software*, 1989. **15**: p. 1-14.