
ASPECT-ORIENTED PROGRAMMING OF SPARSE MATRIX CODE

JOHN IRWIN, JEAN-MARC LOINGTIER,
JOHN 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 the 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

1. INTRODUCTION

Traditionally there are two ways of writing sparse matrix code. You can 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 other approach is to write in a low-level language such as C or Fortran. Such programs generally take much longer to write, are less readable, and thus less maintainable. Of course the reason people bother is because the performance is acceptable using this approach. Often programmers prototype in high-level language then rewrite in efficient one. But what if you then need to make changes – both programs need to be rewritten, a major headache.

We used the aspect-oriented programming (AOP) [1, 2, 3] approach 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 commercial and advanced research versions of the same algorithm. The result was quite satisfactory – our code matches the speed of the commercial version, yet is considerably shorter and less complex.

In section 2 we describe the problems faced when optimizing sparse matrix code by examining a sample efficient low-level solution to see what issues complicate the code. Section 3 presents AML, the AOP solution we created, and shows how

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

* 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. {jirwin,loingtier}@parc.xerox.com (415) 812-4381 (415) 812-4334(fax)

AOP was used to address the issues from our example. Section 4 briefly explains how we generate a running program from the languages described in section 3. Finally section 5 discuss experimental results we obtained by comparing the run time efficiency of our solution with optimized C and Fortran codes.

2. PROBLEM DOMAIN, TRADITIONAL APPROACHES, AND THEIR DRAWBACKS

2.1. SPARSE MATRIX COMPUTATION

Matrix computation is central to numerical modeling in many domains of science and engineering. Examples include structural analysis, materials modeling, fluid mechanics, and more generally any situation in which a physical system is modeled by partial differential equations. These models often have thousands of degrees of freedom so the corresponding matrices are both large, typically thousands to hundreds of thousands of rows and columns, and sparse, where the great majority of the elements have the value zero. Generally fewer than 1% of the elements are nonzero; sometimes fewer than .1% are. Storing such a matrix as a full two-dimensional array would be prohibitively expensive in terms of both space and processing time. Thus various special-purpose data structures are used to store only the nonzero elements.

Algorithms that manipulate sparse matrices perform standard matrix, vector, and scalar arithmetic, matrix-vector products, transposes, vector norms, and so on. They may iterate over a matrix in several ways, such as through all nonzeros in a single column or through all rows in a matrix. They may also invoke more complex operations, such as linear system solution, either from libraries or in some high-level languages by built in operators. As in many domains of scientific computing, the scale of problems that can be solved is primarily determined by how quickly they can be run. Thus efficiency is nearly always a crucial consideration.

Unlike dense matrix code where there are usually standard algorithms for a given problem, efficient sparse matrix operations often require domain specific algorithms, tuned to a particular kind of matrix structure. Thus it is important for users to be able to quickly design new algorithms to fit the domain in question. Because the algorithms are designed primarily to achieve efficient solutions, testing must be done on realistic (large) matrices. To summarize, a key requirement in this domain is the ability to quickly iterate new versions while maintaining reasonable efficiency.

2.2. TRADITIONAL APPROACHES

Sparse matrix code is written in a large number of languages, but the most popular general purpose languages are Fortran and C. These are reasonably good choices for expressing the functional intent of the programmer, although neither language supports matrices directly. But as we will see later they are not so good at expressing the other concerns of interest. In particular, to get good efficiency, the resulting code tends to be long, convoluted, and difficult to maintain. Sparse matrix algorithms written in Fortran or C are best used unmodified from a library.

A popular special purpose language is Matlab. Matlab is a high-level interpreted language originally developed by Cleve Moler [4] at the University of New Mexico, and now sold commercially by *The MathWorks Inc.* It supports matrices as first class objects¹, so writing matrix code is extraordinarily straightforward compared with Fortran/C. For example a common numerical algorithm, a variant of sparse LU factorization we will study in detail later, is 15 to 30 lines in Matlab and 300 lines in Fortran. Of course there is the usual price paid for a high-level language: poor performance. The 30 line version runs about 100 times slower than the Fortran code. Partly this is due to Matlab being an interpreted environment – MathWorks does sell a compiler for Matlab however it does not support sparse matrix code. In fact efficient compilation of sparse matrix code is quite difficult as we will see.

Operations that work on entire matrices at a time (so-called BLAS-2 and BLAS-3 operations [5]) can be provided efficiently by a pre-built library, such as the one built into Matlab. However, sparse data structures provide poor support for ran-

¹ In fact, matrices are the **only** datatype in Matlab: vectors, strings, and even scalars are just special-purpose forms of matrices.

dom access. Thus algorithms that work on individual rows or columns (BLAS-1 operations) or elements (scalar operations) for a sparse matrix cannot be implemented efficiently by such a library.

Some common numerical kernels, such as iterative linear solvers, operate at the BLAS-2 level and above. Libraries such as SPARSLIB++ [6] and PetSC [7, 8] support these high-level operations well. However, many sparse matrix methods, including direct linear solvers and the preconditioners needed to make iterative solvers effective, require extensive BLAS-1 and scalar operations.

So what is a programmer to do? Often they prototype and develop their code in Matlab, since as a high-level language it provides a convenient and efficient prototyping environment. Then, if the code is to be delivered, it must be rewritten in a lower level language, usually Fortran or C/C++.

2.3. AN EXAMPLE: SPARSE LU FACTORIZATION

Next we will examine a sample highly optimized C++ version of a typical numerical analysis algorithm: LU factorization with partial pivoting. Sparse LU factorization (Gaussian elimination with partial pivoting or GEPP. See the side-bars on this page and page 8.) is a standard kernel in numerical libraries [9]. 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 [10], that are often coded by hand for domain-specific numerical modeling applications.

LU FACTORIZATION	
<p style="text-align: center;">Description</p> <p>One way to solve a system of linear equations $Ax=b$ is by use of LU factorization. LU factorization of matrix A finds the lower triangular matrix L and upper triangular matrix U such that $L*U=A$. After these matrices have been determined the problem of solving $Ax=b$ is reduced to the straightforward problem of solving two triangular systems: $L*y=b$ and $U*x=y$.</p>	<p style="text-align: center;">Basic Algorithm (<i>without pivoting</i>)</p> <pre>function [L,U] = lu(A); [m,n] = size(A); L = zeros(n,n); U = zeros(n,n); for j = 1:n t = A(:,j); k = first_nonzero_index_in t(1:j-1) while k ~= [] t = t - t(k) * L(:,k); ← A k = first_nonzero_index_in t(k+1:j-1) end; U(1:j,j) = t(1:j) ; ← B L(j+1:n,j) = t(j+1:n)/t(j); end;</pre>

Figure 1 shows an optimized version of LU written to a custom designed C++ sparse matrix library. This algorithm is as efficient as we could make it while still using library routines rather than open coding the entire algorithm inside the LU function itself. This code, while as efficient as the standard Fortran version and fairly short, is quite a tangled mess. It is difficult to see the base algorithm (from the sidebar on the previous page) in amongst all the other efficiency issues. The rest of this section examines the issues that make this code difficult to understand (and hence to modify and maintain).

Reading through the function, the first statement that doesn't have an analog in the basic functionality alone is the definition of variable t as a SPA at point C. A SPA [11], as we will see later, is an alternate representation of a sparse vector that trades off space for speed. Algorithmically we need only know that t is a vector, any non-algorithmic data structure proper-

ties are uninteresting under the component language abstraction. Stated differently, the representation of data cross-cuts the basic functionality of the component language.

```

void
lu(SparseMatrix& A, SparseMatrix& L, SparseMatrix& U, Permutation& p)
{
    // calculate L, U, p such that L*U = p*A

    double v;
    int m, n, piv, j;

    L.zeros(n, n);
    U.zeros(n, n);
    p.assign(n);

    for (j = 1; j <= n; j = (j + 1)) {
C → SPA t;

        SparseVector A_col_j(A, j);
        assign(t, p, NoRange, NoIter, A_col_j, p, NoRange, NoTrans);
        Range r1(1, j - 1);
        for (SpaNzInorderIterator<Permutation,Range> iter(t, p, r1);
             iter; ++iter) {
A → spaxpy(t, p, NoRange, iter, L_col_k, p, NoRange,
              -(iter.get_value()), NoTrans);
        }

        Range r2(j, n);
        max_abs(t, p, r2, v, piv);
        piv = ((piv + j) - 1);
        p.swap(j, piv);
        if (v == 0.0) {
            assign(t, NoIter, unview(j, p, NoRange), 1.0);
        }

        Range r3(1, j);
B → assign_column(U, j, p, r3, t, p, r3, NoTrans);
        Range r4(j + 1, n);
        assign_column(L, j, p, r4, t, p, r4, NoTrans);
        SparseVector L_col_j(L, j);
        divide(L_col_j, p, r4, t.get_value(unview(j, p, NoRange)));
        }
        L.apply_row_permutation(p);
        U.apply_row_permutation(p);
    }
}

```

Figure 1

Looking into the inner loop, we find the single statement calling library function `spaxpy` at point **A**. This is the second issue, and it is again performance related. We are computing the expression at the same point **A** from the side-bar version, inside the inner loop of the function. Suppose we computed the expression in the most obvious way: multiplying `vec` by the

scalar to give a temporary vector. This vector would then be added to t . Since this is the inner loop this code is in the critical timing path of the function, thus the straightforward approach is too slow. Instead we must fuse the two operations into one. This is what `spaxpy` does. Unfortunately the other thing it does it to make the code much less readable.²

Later in the function we see the statement at point **B**, which corresponds to the same labeled point in the side-bar. P is the permutation, or pivot, which in this case is the same on both sides of the assignment.

The user wants to program as if the matrix is permuted by the permutation vector, but for performance reasons does not actually want to physically permute the data structure, as changes to the permutation vector would require re-permuting the data. Because per-operation efficiency is so important, the permutation vector must be passed explicitly to each function that operates on an implicitly permuted data structure. Thus the P 's scattered through the code.³

These are just three of the issues complicating the code. In addition these are sometimes themselves tangled together, and there are others caused by the particular library implementation that we will not address here as alternative library implementations would have different ones. In the next section, we will discuss how we dealt with these issues in designing and implementing the AML system.

PARTIAL PIVOTING

Gaussian elimination is a standard algorithm for computing LU factorizations. At each step k of the elimination, the coefficient of the k^{th} variable in the k^{th} equation – the “pivot” – is used to eliminate that variable from other equations. If a pivot is zero, ordinary Gaussian elimination fails; if a pivot is very small, the factorization may be numerically unstable.

A more robust method is Gaussian elimination with partial pivoting (GEPP for short). Before step k of GEPP, row k is exchanged with a later row if necessary to ensure that the pivot will have the largest magnitude among the coefficients of the k^{th} variable in all the equations k through n . Formally, this produces a factorization $PA = LU$, where P is a permutation matrix that describes the sequence of exchanges of rows or equations.

3. AN AOP SOLUTION

Having these tangled issues to deal with, we decided to try to use AOP for this problem. In AOP [2] 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 WeaverTM 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 following section describes the aspect weaver.

² This is not specific only to sparse matrices. The non-sparse SAXPY operation was identified as fundamental in LINPACK(~1970) [12] and is now central to the design philosophy of every successful vector and superscalar processor architecture.

³ Actually it turns out that for the LU example, there are alternate library implementations using not yet widely available ANSI C++ template facilities that would fix the tangling because LU has such simple use of permutations. More complex examples involving multiple or nested permutations would have problems under any library implementation.

3.1. THE COMPONENT LANGUAGE

The choice of component language was a critical step in our application of AOP techniques to this domain. We wanted the language to be as familiar and natural as possible to programmers in the domain in question. We also wanted a language at a high level of abstraction so that the programmer can think directly in terms of the particular functional algorithms being written. In fact, the abstraction level is key to more than just ease of programming. As discussed in [2] it is essential to AOP that the component language make clear those distinctions the aspect programs need to discriminate on. This makes it easier to join the aspect languages to the component language, which makes it easier to design the aspect languages. For all these reasons we chose a subset of Matlab, with one key addition that is described below, as the component language for AML.

Matlab was created as a user-friendly language for scientific computation. It was never meant to be an input language for a compiler. As a result, compiling Matlab can be a real challenge [13, 14]. As our goal was to use Matlab as a starting point for AML rather than to create the best compiler for Matlab, we impose several restrictions on the language. Most of the features that we don't support are arguably not used in good Matlab programming style.

The most important restriction that we impose is that one variable cannot be assigned values of different types, compared to standard Matlab where variables are untyped. Although in some cases it may be possible to deduce a type and shape of the variable at compile time, in general this problem is very hard [13]. Requiring a variable to have constant type and shape allows us to avoid any kind of run-time type dispatching. We also introduce reasonable limitations on the context under which variables of certain type and shape can be used. For example, we require array subscripts to be of type integer.

Another feature of Matlab that we don't support is dynamic growth of matrices. In fact, we do not check matrix bounds at all. This is similar to what the Matlab compiler does if run with the `-i` option [15]. The rest of the restrictions are mostly just limitations of our current implementation. They include the following requirements: functions should have fixed number of input and output parameters, only real and integer types are supported, no string variables are supported, and no indexing a vector by another arbitrary vector is provided.

For our LU example, the component program looks very much like the abstract version from the LU side-bar:

```
function [L,U] = lu(A);           % calculate L, U such that L*U = A

[m,n] = size(A);
L = zeros(n,n);
U = zeros(n,n);

for j = 1:n
    t = A(:,j);
    for nzs k in order in t(1:j-1)
        t = t - t(k)*L(:,k);
    end;

    U(1:j,j) = t(1:j);
    L(j+1:n,j) = t(j+1:n)/t(j);
end;
```

Figure 2

Most of the code in figure 2 is standard Matlab. The colon `:` operator indicates a range of values. By itself it means all possible values. So `A(:,j)` means select elements from column `j` of matrix `A`. `t(1:j)` means select elements from vector `t` with indices from 1 to `j`.

The `for nzs` code is not part of the Matlab language. This is the one addition we needed to make to the component language to bring it to the proper abstraction level. In sparse matrix code a commonly required operation is to iterate through the nonzero elements of a vector or matrix column. Unfortunately Matlab provides an abstraction that is too high – deal with an entire vector at a time – and one that is too low – iterate through every element of a vector. In order to iterate through only the non-zero elements of a vector in unmodified Matlab, we’d have to write something like:

```
k = min(find(t));
while k ~= []
    t = t - t(k) * column;
    k = k + min(find(t(k+1:n)));
end;
```

Unfortunately overall this loop is $\Theta(n)$, which is very large compared with the actual number of nonzero elements in the sparse vector. There are better Matlab implementations, but they are not applicable to this example. Since new nonzero elements can appear in `t` during the iterations of the loop – a phenomenon called “fill”, which is common in direct sparse matrix methods, – the following clumsy code must be used:

```
for k = 1:n
    if (t(k) ~= 0)
        t = t - t(k) * column;
    end;
end;
```

Thus obscuring the actual intent of the loop: to compute the sum of the elements of the vector. Also this code, which can be implemented in $\Theta(\text{nonzeros})$ time in a low-level language, takes $\Theta(\text{nonzeros}^2)$ in Matlab. To fix this, we added the intermediate level construction `for nzs` to AML, which iterates through the nonzero elements of a sparse vector, either in an arbitrary order or in order from lowest to highest index. In AML our loop then becomes:

```
for nzs k in order in t
    t = t - t(k) * column;
end;
```

3.2. DATA REPRESENTATION ASPECT

As we saw when examining the tangled C++ LU code, a key issue in efficient code is data representation. This is not the same as the traditional notion of data type. There are several axes of information that combine to produce the complete description of a given object.

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

Element type and dimension, the darkly shaded entries, 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 lightly shaded entries are additional representation information that is not visible to the base program. These other components are needed to determine the data structure that the implementation will work with. (Traditional OOP systems conflate these into a single class.)

THE THREE ARRAY REPRESENTATIONS

The full representation allocates memory for each array element. This gives fast random access, but enumerating the nonzero elements is slow, because the entire vector must be searched.

The sparse representation allocates memory only for storing the non-zero elements and for storing the indices of the non-zeros. This requires much less memory, gives slow random access, since the indices must be searched, but allows the nonzeros to be found quickly.

A sparse accumulator, or SPA, which is used only on vectors, is a combination of a full representation of the values and a sparse list of the indices of the non-zeros. This lets it combine the fast random access of a full representation with the fast identification of all non-zeros of a sparse representation.

Representation	Size	Finding nonzeros	Random access
sparse	small	fast	slow
full	large	slow	fast
SPA	large	fast	fast

The representation entry is the key additional piece of information, specifying one of three different styles of representation: an ordinary full matrix, a sparse matrix, or a sparse accumulator (SPA). All have the same behavior as seen from the base program, but make different performance tradeoffs, described in the box above and in [11]. The other two representation axes modify sparse representations, indicating whether the non-zeros should be stored in order and whether a two-dimensional sparse representation should be organized along rows or columns.

Much of the data information comes from declarations, which function much like conventional type declarations. They apply over a given lexical scope, for example. There are some differences, however. Ordering can be requested by `in order` iteration statements. The data information also interacts with one more kind of information, permutations, which will be discussed shortly, to fully determine the representation. Finally, the implementation of an operation may depend not only on the data representations of its inputs, but also on the data representation desirable for the result.

Adding the data representation aspect code to our LU example gives us the following, with the data representation aspect in red:

```

function [L,U] = lu(A);           % calculate L, U such that L*U = 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);

for j = 1:n
  declare real SPA t;
  t = A(:,j);
  for nzs k in order in t(1:j-1)
    t = t - t(k)*L(:,k);
  end;

  U(1:j,j) = t(1:j);
  L(j+1:n,j) = t(j+1:n)/t(j);
end;

```

Figure 3

As mentioned above, the scoping rules are quite simple. Matlab has no local scoping constructs, and only top-level functions, so there are only two scopes: global and function. Another important point is that it is not feasible to merely require the user to specify that the vector is sparse and have the compiler choose a SPA representation based on the usage; the program analysis and understanding involved is so significant that compilers cannot be counted upon to handle this reliably.

3.3. OPERATION FUSION ASPECT

As in many other areas of scientific computing, eliminating temporary results, where possible, is key to achieving good performance. This is because the temporary results are quite large, generally larger than the size of the primary processor cache. The usual approach is to fuse a set of operations into a single more powerful operation. For sufficiently simple code, such as direct array manipulation, a compiler may be able to infer the high-level loop structure and automatically fuse looped operations. Doing this for sparse matrix code would be quite difficult, as the basic loop structure is less transparent.

Our approach was to define another aspect language to describe how operations may be fused. Since we made the decision to have the compiler emit code that is then linked against an existing C++ library, the operation fusion aspect language gives a mapping from abstract operations to concrete implementation operations. A set of these rules is applied over the input code, the set ordered by decreasing amount of fusion. Note that this works because the critical sections of code operate on whole vectors at a time.

As an example, here is the rule from our system that describes the fusion of `max(abs(vector))`, a common operation to find the largest absolute entry in a vector. In contrast to the other aspect languages, this code is not written by the user of the system, but rather by the library implementor. For this reason we have designed this language for ease of processing rather than ease of reading.⁴

⁴ As we gain more experience with this aspect we intend to both design a more declarative and perspicuous fusion aspect language and a method for defining both the fusion and the library implementation of the fused function.

```

(defrule maxabs (node)
  (if-match node
    ;; Matches: [v,piv] = max(abs(vec(j:n)))
    (**mv-assign (**ref ?max-val ?(type-is mvt :real :scalar))
      (*ref ?max-index ?(type-is mit :int :scalar)))
      (**lib-call (*lib "max")
        (**lib-call (*lib "abs")
          (**vr-ref ?vec ?vec-range
            ?(type-is vrt :real :vector))))))
    ;; Generates: max_abs(v, Perm(v), Range(j:n), &v, &piv)
    `(*lib-call (*lib "max_abs")
      ,vec ,(permutation vrt) ,(cg vec-range)
      (*ref ,max-val ,mvt) (*ref ,max-index ,mit))))

```

Figure 4

The corresponding concrete library implementation operation is `max_abs`. Without this fusion a temporary vector would be created for the result of `abs(vector)` before calling `max`. The fused route eliminates the temporary which saves a write/read of the vector's data.

3.4. IMPLICIT PERMUTATION ASPECT

The previous aspects dealt exclusively with performance; adding them to the component code did not change the algorithmic functionality of the code. An additional aspect we saw a need for was the implicit permutation of vectors and matrix rows & columns. For example, in LU it is important to choose the order in which the rows are processed so that there are no divisions by zero, retaining numerical stability. Note that permutations do not affect the basic algorithm – given a well behaved matrix the answers are the same. Permutations allow methods to get answers for some less behaved matrices also. Matrix permutations are important in many other contexts as well, including ordering for sparsity [16], parallel partitioning [17], block triangular form [18], and effective preconditioning [19].

Permutations could be implemented by actually reordering the rows of the matrix, but that would be rather expensive. The standard trick used is to access the vector or matrix column through a permutation vector, that for each index of the vector stores its actual location. The presence of implicit permutation vectors has a significant effect on the other parts of the system. For example, the library operations need to be able to operate on permuted objects. Even more, if an operation involves two objects having the same permutation, the library code can safely ignore the fact that they are permuted.

Adding the implicit permutation aspect code to our LU example finally gives us the complete LU code, with the implicit permutation aspect in green:

The `view through` statement establishes the given permutations, one for each dimension of the permuted objects, throughout the enclosing lexical scope. A lexical block with such a statement behaves the same way as it would have by replacing all occurrences of the variable with the same variable but with the permutations applied. While this statement has some resemblance to a declaration, it is not one, since it affects the results of the program. Further, it should be noted that this statement differs from replacing the array by its permutation in two ways: at the end of the lexical scope, the permutation is undone. If the permutation is updated while the view is in effect, the effect of the change is seen in future array accesses. In the LU example, the central block of green code updates the permutation.

```

function [L,U,p] = lu(A); % calculate L, U, p such that L*U = p*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]);
    if v == 0.0,
        t(j)=1.0;
    end;

    U(1:j,j) = t(1:j);
    L(j+1:n,j) = t(j+1:n)/t(j);
end view;
end;
end view;

```

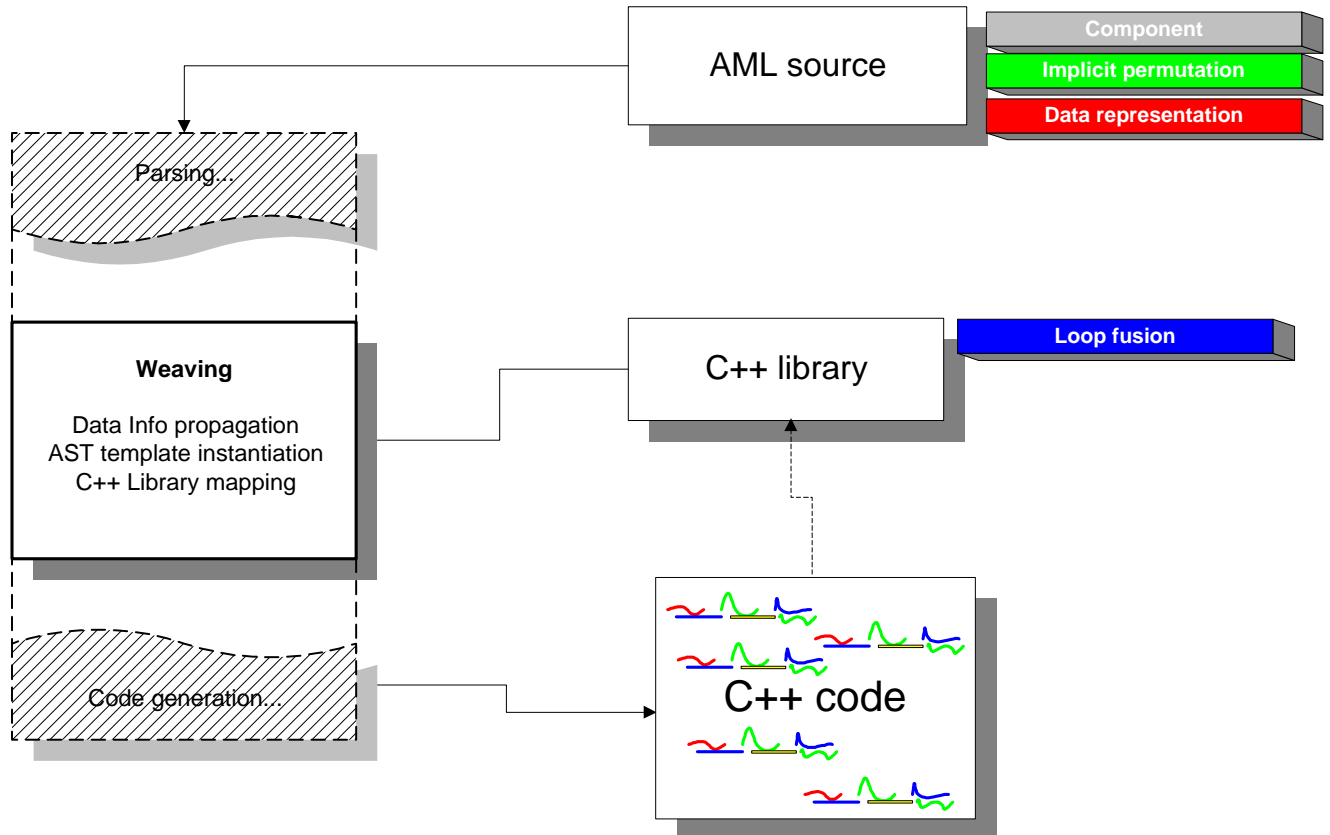
Figure 2

Another important thing to realize about implicit permutations is that they follow the dataflow of the variable. If a permuted object is passed to a subroutine, that function must be prepared to deal with the permutation. This is handled in two ways by our system. The C++ implementation of the library uses the template mechanism to define multiple versions of each function – the versions differing in whether the arguments are permuted or not. For functions that are defined by the user rather than built into the library, the weaver automatically generates multiple versions of the function.

So the complete AML system is made up of the component language and the three aspect languages: data representation, operation fusion, and implicit permutation. The user of AML writes a single program text that contains the component program, the data representation program, and the implicit permutation program. This works nicely for AML since it allows an easy and natural means of describing where the aspect programs tie into the component program. In the next section we will see how all these programs get turned into a running executable.

4. THE WEAVER

In most respects the weaver is a very simple compiler. Rather than trying to make very smart inferences, the power of our weaver relies upon 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 roughly be 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.

Having C++ as our library and back-end language allowed us to make extensive use of overloading and templating to deal with the variability of the fusion operations we needed to put in the library. For example there is only one function called `max`, and it can be applied to a wide variety of concrete argument types. The compile time nature of C++ allows this approach to generate efficient code.

The current weaver is complete enough to correctly compile many example programs of the complexity of the LU factorization example presented herein. Future weaver work includes extending the fusion & generation passes to handle more possible input codes. We also would like to add additional aspects to the system. For example we have looked at the possibility of a *shape* aspect which the user would use to tell the weaver about the particular distribution of nonzero elements in certain matrices. For some algorithms this is an additional issue that can be used to generate more optimal code.

Finally, to return to our LU example, you might ask what the output of the weaver looks like when applied to the complete example. The answer is simple, for the weaver is generating the **very same code** we exposed in section 2 as an example of optimized LU tangled code. Of course, we have yet to show that this code exhibits good performance. That is the aim of the next section.

5. EXPERIMENTAL RESULTS

The goal of AML is to allow sparse numerical 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[20], a well-known Fortran version of the algorithm implemented (in C) in the Matlab version 4 internal library; and SuperLU[9], the best performing research code, which uses much more complex algorithms to optimize cache behavior and to exploit structural symmetry.

5.1. COMPLEXITY

Quantifying the comprehensibility and modifiability of code is a notoriously difficult problem. There are many measures of code complexity [21, 22, 23], none of them entirely satisfactory. We take just the very simple measure of source lines of code. The results are shown in the following table:

Implementation	Lines of Code
LU (AML)	30
GPLU (Fortran)	300
SuperLU (C)	7000

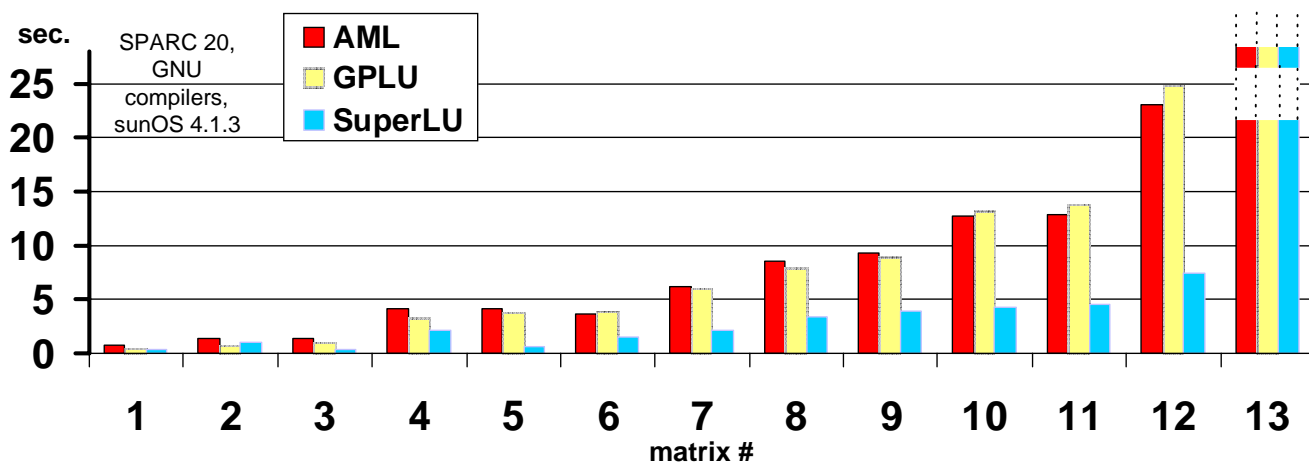
As this table shows, the AML code is an order of magnitude shorter than the Fortran implementation. The size comparison for the supernodal code is not as pertinent, since that is performing a much more complicated algorithm.

5.2. 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. The results are given in the table below.

matrix #	name	size	nonzeros	LU AML(sec.)	GPLU Fortran(sec.)	SuperLU C (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	lnsp3837	3937	25407	8.6	7.9	3.4
9	lns3937	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

Summarized graphically⁵ with matrices being ordered by their GPLU execution time, the results are:



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 supernodal LU for completeness, even though it is not the kind of algorithm that the users of AML are likely to write, as SuperLU deals with a complicated low-

⁵ Due to the scale limitations, we didn't completely represent the result for matrix #13.

level issue that neither AML nor GPLU does, namely exploiting 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.

AML LU – 30 source lines of code – is on average as fast as GPLU Fortran - 300 source lines of code. AML LU even appears to perform better than GPLU Fortran for the hardest matrices.

6. CONCLUSION

One of the primary goals of software research at the present time is to reduce or make manageable the complexity of real-world software systems. The essential premise of AOP is that this complexity is often caused by concerns that cross-cut the functionality of the system. The various concerns are tangled together in the actual code, which causes it to be longer, less readable, and less easy to maintain. As we have seen, efficient sparse matrix computation is one domain where this occurs. The complex versions of algorithms we have studied are ten to two hundred times as long as the description of the actual algorithm at a functional level. Of course length is not a sufficient condition to prove increasing complexity. But it is clearly highly correlated.

This work was originally motivated by a materials modeling application that required special-purpose sparse matrix code [24]. In that application, we prototyped preconditioning codes in Matlab, experimented on toy-sized problems, and then translated the most effective algorithms into Fortran by hand for efficiency of execution. AML was begun as an attempt to automate this laborious process.

We examined several of the core issues in this domain that cause complexity by cross-cutting the actual algorithm. Existing approaches that handle these issues result in complex, unreadable code. Isolating these issues into their own aspect programs, written in domain specific aspect languages, greatly reduces the complexity of the whole, both by perceived elegance and by measured length. Using an aspect weaver, it is straight-forward to produce a running executable from these simple, readable programs that maintains the efficiency of the complex tangled solution.

We have had good initial success in applying AOP techniques to this domain. We can code several representative programs from this domain using AOP. The resulting programs are simpler, easier to write, and easier to work with than those written using traditional techniques. Equally important, they are almost as efficient as optimized hand written versions of similar algorithms. We are optimistic that further development of our AOP-based sparse matrix system will provide a system that is useful to engineers writing sparse matrix code.

7. 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 *OOPSLA 97*. 1997. Submitted.
3. Kiczales, G., *et al.* *Aspect-Oriented Programming*. in *POPL Workshop on Domain Specific Languages*. 1997. Paris.
4. Moler, C.B., *MATLAB User's Guide*. 1980, Albuquerque: C.S. Dept. University of New Mexico.

5. Dongarra, J.J., *et al.*, *An extended set of basic linear algebra subroutines*. ACM Transac. on Math. Software, 1988. **14**: p. 1-17, 18-32.
6. Pozo, R., K. Remington, and A. Lumsdaine, *SPARSELIB++ V. 1.5 sparse matrix class library reference guide*. NISTIR 5861. 1996: National Institute of Standards and Technology.
7. Gropp, W.D. and B.F. Smith, *The design of data-structure neutral libraries for the iterative solution of sparse linear systems*. Scientific Programming, 1996. **5**: p. 329-336.
8. Balay, S., *et al.*, *PETSc 2.0 Users Manual*. 1996, Argonne: Argonne National Lab.
9. Demmel, J.W., *et al.* *A Supernodal Approach to Sparse Partial Pivoting*. in *ILAY Workshop on Direct Methods*. 1995. Toulouse, France.
10. Saad, Y., *Iterative Methods for Sparse Linear Systems*. 1996, Boston: PWS Publishing Company.
11. Gilbert, J.R., C. Moler, and R. Schreiber, *Sparse Matrices in MATLAB: Design and Implementation*. SLAM J. Matrix Anal. Appl., 1992. **13**: p. 333-356.
12. Dongarra, J.J., *et al.*, *LINPACK User's Guide*. 1979, Philadelphia: SIAM.
13. Johnson, S.C. and M. C. *Compiling Matlab*. in *Very High Level Languages Symposium (VHLL)*. 1994. Santa Fe, NM: USENIX.
14. DeRose, L., *Compiler techniques for Matlab programs.*, in *CS Dept*. 1996, University of Illinois: Urbana-Champaign.
15. MathWorks, *Matlab compiler, User's Guide*. 1995: The MathWorks Inc.
16. George, A. and J.W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. 1981: Prentice-Hall.
17. Shewchuk, J.R. and D.R. O'Hallaron, *Archimedes*, . 1996: <http://www.cs.cmu.edu/~quake/archimedes.html>.
18. Pothén, A. and C.-J. Fan, *Computing the block triangular form of a sparse matrix*. toms, 1990. **16**: p. 303--324.
19. Duff, I.S. and G. Meurant, *The effect of ordering on preconditioned conjugate gradients*. BIT, 1989. **29**: p. 685--657.
20. 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.
21. Henry, S. and D. Kafura, *Software Structure Metrics Based on Information Flow*. IEEE Transactions on Software Engineering, 1981. **SE-7**: p. 509--518.
22. McClure, C. *A Model for Program Complexity Analysis*. in *3rd International Conference on Software Engineering*. 1978. Los Alamitos, CA: IEEE Computer Society Press.
23. Yau, S. and J. Collofello, *Some Stability Measures for Software Maintenance*. tse, 1980. **SE-6**: p. 545--552.
24. Torres, F. and J.R. Gilbert. *Use of an iterative solver in Stokesian dynamics simulations*. in *AICHE Annual Meeting*. 1994: American Institute of Chemical Engineering.