Left-looking to Right-looking and vice versa: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring

Nikolay Mateev, Vijay Menon, and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. We have recently developed a new program analysis strategy called *fractal symbolic analysis* that addresses some of limitations of techniques such as dependence analysis. In this paper, we show how fractal symbolic analysis can be used to convert between left-looking and right-looking versions of three kernels of central importance in computational science: Cholesky factorization, LU factorization with pivoting, and triangular solve.

1 Introduction

Most computational science codes require the solution of linear systems of equations. These systems can be written as $Ax = b$ where A is a matrix, b is a vector of known values, and x is the vector of unknowns. Direct methods for solving linear systems factorize the matrix A into the product of an upper triangular matrix and a lower triangular matrix, and then find x by solving the two triangular systems. If the matrix is symmetric and positive-definite, Cholesky factorization is usually used to find the two triangular factors; otherwise, LU with partial pivoting is used.

Substantial effort has been invested by the numerical analysis community in implementing high-performance versions of these algorithms. For example, the LAPACK library contains blocked implementations of these algorithms, optimized to perform well on a memory hierarchy [2]; the SCALAPACK library contains parallel implementations of these algorithms for distributed-memory machines [3].

In the compiler community, researchers have developed techniques to synthesize blocked and parallel implementations of these algorithms from high-level algorithmic formulations. These *restructuring* techniques perform source-to-source transformations to improve parallelism and locality of reference. A significant challenge for compiler optimization is the fact that there are many variations in how these algorithms can be expressed. The two most important variations are called *right-looking* or *eager*, and *left-looking* or *lazy*.

[0] This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.

– *Eager*: In matrix factorization codes, the matrix is walked by column from left to right. After the current column has been computed, updates to columns to the right of the current column are performed immediately. Similarly in triangular solves, the current unknown is computed, and its contribution is immediately subtracted from the remaining equations. In the numerical analysis community, these are referred to as *right-looking* formulations.
– *Lazy*: Updates to the current element/column from earlier elements/columns are performed as late as possible, in a lazy manner. These are also referred to as *left-looking* formulations.

The effectiveness of different compiler optimizations can be sensitive to the original formulation. The storage layout of a matrix in memory or across multiple processors may lead to a preference for one or the other of these formulations. Thus, it is important for compilers to transform one form to the other.

The most commonly used technique for proving legality of transformations is *dependence analysis* [8], which computes and enforces a partial order between the statements based upon data dependences. A more powerful technique that subsumes dependence analysis is *symbolic analysis*, which compares symbolically two programs for equality. Both approaches have their shortcomings. The constraints imposed by dependence analysis are sufficient but not necessary, and fail to prove equality of right- and left-looking LU. Symbolic analysis, on the other hand, is precise but intractable for all but the simplest programs.

To bridge this gap between dependence analysis and symbolic analysis, we developed *fractal symbolic analysis* [6]. In this paper, we show how this new analysis technique can be used to convert between left- and right-looking versions of triangular solve, Cholesky factorization, and LU factorization with partial pivoting. In Section 2, we abstract the transformation required to convert between left- and right-looking formulations, and show that dependence analysis is too weak to prove the equality of left- and right-looking versions of LU factorization with pivoting. In Section 3, we summarize fractal symbolic analysis. In Section 4, we demonstrate its effectiveness in verifying the legality of these transformations on LU with pivoting (for lack of space, we do not discuss triangular solve and Cholesky, but both dependence analysis and fractal symbolic analysis are adequate for these programs. These and other details can be found in an expanded version of this paper [7]). Finally, we conclude with future directions.

2 Factorizations and Triangular Solve

In this section, we discuss right- and left-looking formulations of three important numerical kernels: Cholesky factorization, LU factorization with pivoting, and lower triangular solve. Neither right- nor left-looking forms should be viewed as canonical in general. For example, in [4], a standard text on matrix computations, Cholesky and lower triangular solve are introduced in a left-looking or lazy manner, while LU is introduced in a right-looking or eager manner. It should

```
do k = 1,n
  B1(k);
  do j = k+1,n
    B2(k,j);
```

(a) Eager/Right-looking Code

```
do j = 1,n
  do k = 1,j-1
    B2(k,j);
  B1(j);
```

(b) Lazy/Left-looking Code

Fig. 1. Equivalent Right-looking and Left-looking Codes

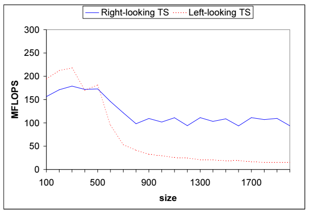

Fig. 2. Triangular Solve

Fig. 3. Cholesky

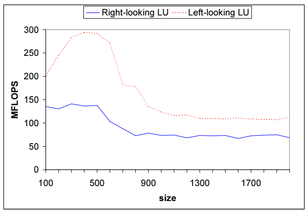

Fig. 4. LU with Pivoting

be noted that this has no correlation with performance. To illustrate this, we present the performance of both forms on an SGI Octane[1].

At a high-level, the transformation between left- and right-looking versions can be viewed as a transformation we call *right-left loop interchange*, illustrated in Figure 1. In each of the codes discussed below, right-looking formulations correspond to Figure 1a, and left-looking formulations correspond to Figure 1b. The underlying operations, denoted by `B1` and `B2`, are the same in both cases. We show that conversion between right- and left-looking formulations may be accomplished by one or more applications of right-left loop interchange.

2.1 Lower Triangular Solve

Triangular solve, shown in Figure 5, maps directly to the template of Figure 1. Both `B1` and `B2` are represented by a single statement. `B1` corresponds to the final scaling step of solving a single equation with one unknown, and `B2` corresponds to the substitution of a solved unknown (`x(k)`) to compute an unsolved unknown (`x(j)`). Although triangular solve is often introduced in its left-looking form (as in [4]), the right-looking form can sometimes be desirable for performance. When compiled in Fortran on the SGI Octane, the right-looking form considerably outperforms the left-looking form as shown in Figure 2. Here, the right-looking code has better spatial locality as `A` is stored in column-major order.

2.2 Cholesky Factorization

Our second example is Cholesky factorization, a key computational kernel for factoring symmetric, positive-definite matrices. Figure 6 presents both right- and

[1] This 300MHz machine has a 2MB L2 cache, and an R12K processor. All compiled code was generated using the SGI MIPSpro f77 compiler with flags: -O3 -n32 -mips4.

```
      do k = 1,n
        // Compute current unknown
B1(k) : x(k) = x(k)/A(k,k)

        // Update from current unknown
        // to later unknowns
        do j = k+1,n
B2(k,j) : x(j) = x(j)-A(j,k)*x(k)
```

(a) Right-looking Triangular Solve

```
      do j = 1,n
        // Update from earlier unknowns
        // to current unknown
        do k = 1,j-1
B2(k,j) : x(j) = x(j)-A(j,k)*x(k)

        // Compute current unknown
B1(j) : x(j) = x(j)/A(j,j)
```

(b) Left-looking Triangular Solve

Fig. 5. Lower Triangular Solve

```
      do k = 1,N
        // Scale current column
B1(k) : A(k,k) = sqrt(A(k,k))
        do i = k+1,N
          A(i,k) = A(i,k)/A(k,k)

        // Update from current column
        // to columns to right
        do j = k+1,N
B2(k,j) : do i = j,N
            A(i,j) = A(i,j)-A(i,k)*A(j,k)
```

(a) Right-looking Cholesky

```
      do j = 1,N
        // Update from columns
        // to left to current column
        do k = 1,j-1
B2(k,j) : do i = j,N
            A(i,j) = A(i,j)-A(i,k)*A(j,k)

        // Scale current column
B1(j) : A(j,j) = sqrt(A(j,j))
        do i = j+1,N
          A(i,j) = A(i,j)/A(j,j)
```

(b) Left-looking Cholesky

Fig. 6. Cholesky Factorization

left-looking versions of this operation. As in the case of triangular solve, Cholesky maps directly to the template suggested in Figure 1. In this case, `B1` and `B2` are represented by small blocks of code. `B1` corresponds to the computation in the current column, and `B2` corresponds to updates from earlier columns to the left to later columns to the right. As before, performance is sensitive to the formulation that is used. However, in this case, as shown in Figure 3, the left-looking formulation results in better performance.

2.3 LU Factorization with Partial Pivoting

Our last example is LU factorization with partial pivoting, which is used for factoring general unsymmetric matrices. Without pivoting, LU factorization is quite similar to Cholesky in the previous section, but suffers from instability due to accumulating floating point error. In practice, partial pivoting provides a solution to this problem.

For this example, converting between a right-looking formulation (as in Figure 7a) and a left-looking formulation (as in Figure 7b) is a more involved process. The pivot operation performed in each column requires a corresponding swap of elements in every other column. This swap can be viewed as a second 'update' between columns. Conversion between right- and left forms may be accomplished by two applications of right-left loop interchange. In the right-looking code in Figure 7a, the update alone may be converted to left-looking form, as in Figure 7c. Converting the swap is slightly more complicated, as the swap is

```
do k = 1, N
  // Pick the pivot
B1.a(k) :
  p(k) = k
B1.b(k) :
  do i = k+1, N
    if abs(A(i,k)) > abs(A(p(k),k))
      p(k) = i

  // Swap rows
B1.c(k) :
  do j = 1, N
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Scale current column
B1.d(k) :
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k)

  // Update from current column
  // to columns to right
  do j = k+1, N
B2(k,j) :
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
```

(a) Right-looking LU

```
do j = 1, N
  // Swap rows from left
  do k = 1, j-1
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Update from columns to left
  // to current column
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
```

(b) Left-looking LU

```
do j = 1, N
  // Update to current column
  // from columns to left
  do k = 1, j-1
B2(k,j) :
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
B1.a(j) :
  p(j) = j
B1.b(j) :
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows
B1.c(j) :
  do k = 1, N
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
B1.d(j) :
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
```

(c) Hybrid Right-Left LU #1

```
do j = 1, N
  // Update to current column
  // from columns to left
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)

  // Swap rows to the right
  do k = j+1, N
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp
```

(d) Hybrid Right-Left LU #2

Fig. 7. LU Factorization with Partial Pivoting

```
        do i = 1,N
          do j = 1,N
S(i,j) :    k = k + A(i,j)
```

Fig. 8. Reduction

$$A(\boldsymbol{k}) = \begin{cases} guard_1(\boldsymbol{k}) \rightarrow expression_1(\boldsymbol{k}) \\ guard_2(\boldsymbol{k}) \rightarrow expression_2(\boldsymbol{k}) \\ \vdots \\ guard_n(\boldsymbol{k}) \rightarrow expression_n(\boldsymbol{k}) \end{cases}$$

Fig. 9. Guarded Symbolic Expression

never purely right-looking since pivoting requires swaps in earlier columns as well as latter columns. Nevertheless, the right-looking portion of the swap may be isolated, via index-set-splitting and statement reordering [8], as in Figure 7d. A second application of right-left loop interchange produces the left looking code in Figure 7b.

Figure 4 shows the performance of these codes on the SGI Octane. Although the right-looking version is simpler, the left-looking version has notably better cache performance. One key obstacle to automatic conversion between right- and left-looking forms is the inability of dependence analysis to establish their equivalence. At the level of matrix operations, the pivot swaps may be viewed as row permutations and the updates as matrix multiplications. In converting between right- and left-looking forms, these operations are interchanged, but they are not independent since they modify certain common storage locations. Hence, a compiler that relies on dependence analysis will not be able to prove the equivalence of these versions. In the next section, we present a more powerful analysis tool that can establish the legality of this transformation.

3 Fractal Symbolic Analysis

In this section, we give a brief overview of *fractal symbolic analysis*, a technique we proposed in [6] to establish legality of program transformations. As mentioned earlier, dependence analysis is too conservative to handle a code such as LU factorization with pivoting. Traditional symbolic analysis, on the other hand, is generally impractical. Fractal symbolic analysis provides an accurate and tractable means of analyzing many codes.

To illustrate the basic idea, consider the simple example in Figure 8. For a number of reasons, a compiler may desire to interchange the `i` and `j` loops in this code. However, every instance of the statement `S` writes to the variable `k`. As a result, dependence analysis enforces a total ordering between all instances of `S` and, therefore forbids loop interchange. Nevertheless, if commutativity and associativity of addition is allowed, this interchange produces equivalent results. Most modern compilers would use pattern recognition to figure out that the interchange is legal. However, pattern recognition is notoriously fragile, so a more robust test is desirable. Symbolic analysis is one option, but direct symbolic comparison of programs is usually intractable.

Transformation	Legality Condition

Loop Interchange
`do i = 1,n              do j = 1,m`
`  do j = 1,m      <->     do i = 1,n`
`    S(i,j);                  S(i,j);`
	$commute(\langle S(p,q), S(r,s)\rangle :$
	$1 <= p < r <= n \wedge 1 <= s < q <= m)$

Right/Left-looking Interchange
`do k = 1,n              do j = 1,n`
`  S1(k);                   do k = 1,j-1`
`  do j = k+1,n    <->         S2(k,j);`
`    S2(k,j);               S1(j);`
	$commute(\langle S1(t), S2(r,s)\rangle :$
	$1 <= r < t < s <= n) \wedge$
	$commute(\langle S2(p,q), S2(r,s)\rangle :$
	$1 <= p < r < s < q <= n)$

Fig. 10. Legality Conditions for Program Transformations

Commute Condition	Recursive Condition
Statement Sequence $commute($⟨ `S1; S2;...; SN,B2` ⟩ $: cond)$	$commute(\langle S1, B2\rangle : cond) \wedge$ $commute(\langle S2, B2\rangle : cond) \wedge$ $\ldots$ $commute(\langle SN, B2\rangle : cond)$
Loop $commute($⟨ `do i = l,u` `  S1(i);` `,B2`⟩ $: cond)$	$commute(\langle S1(i), B2\rangle : cond \wedge l <= i <= u)$

Conditional Statement
$commute($⟨ `if (pred) then`
`  S1;`
`else`
`  S2;` `,B2`⟩ $: cond)$
	$commute(\langle S1, B2\rangle : cond \wedge pred) \wedge$
	$commute(\langle S2, B2\rangle : cond \wedge \neg pred)$

Fig. 11. Recursive Simplification Rules

The key idea behind fractal symbolic analysis is the following. Loop interchange reorders particular instances of statements. This reordering may be viewed *incrementally* by interchanging instances *one pair at a time*. In the example above, the legality of loop interchange is established by symbolically demonstrating that two individual instances, `S(i,j)` and `S(i',j')`, commute (that is, that they can be done in any order). This only requires proving that $\mathtt{k}_{out}$ `=` $\mathtt{k}_{in}$ `+ A(i,j) + A(i',j')` and $\mathtt{k}_{out}$ `=` $\mathtt{k}_{in}$ `+ A(i',j') + A(i,j)` are equivalent, which may be verified by a relatively simple symbolic engine.

In general, there are two aspects to fractal symbolic analysis: (i) recursive simplification, and (ii) base symbolic comparison.

3.1 Recursive simplification

As discussed above, fractal symbolic analysis simplifies programs recursively till they are simple enough for the base symbolic comparison engine. There are three key ideas to this simplification. First, if the programs to be compared are too complex for symbolic comparison, simplified programs are generated such that equality of the simplified programs is a sufficient, but not in general necessary condition to establish the equality of the original codes. Second, for codes obtained by common program transformations, the appropriate simplification may be derived from the transformation as in the example above. Figure 10 provides the legality conditions for both the loop interchange performed above and the right-left loop interchange presented earlier in this paper. Finally, this simplification process may be applied recursively until tractable programs are obtained. Figure 11 provides rules for recursive simplification.

3.2 Base symbolic comparison

Although compared programs may be repeatedly simplified as needed, each simplification step results in a loss of accuracy as equality of the simplified programs is a sufficient but not necessary condition. Because of this, it is important to symbolically compare programs with as few simplification steps as possible. In [6], we describe a core symbolic comparison engine that is effective under the following constraints. Recursive simplification may be applied until these constraints are met.

- Programs consist of assignment statements, for-loops and conditionals.
- Loops do not carry dependences.
- Array indices and loop bounds are restricted to be affine functions of enclosing loop variables and symbolic constants, and predicates are restricted to be conjunctions and disjunctions of affine inequalities.

Under these conditions, we have shown that the effect of a program on each live, modified variable may be summarized as a *guarded symbolic expression*, as shown in Figure 9. Each guard describes a polyhedral region of array indices, and the corresponding expression describes the values of the array elements for those indices. Computation and comparison of guarded symbolic expressions is a straightforward process and is described in detail in [6].

4 LU with Pivoting

We now demonstrate how fractal symbolic analysis can be applied to establish the legality of the right-left transformation on LU factorization with pivoting. For conciseness, we will focus on the equivalence of the codes in Figure 7a and 7c. As discussed earlier, these codes differ by a single application of right-left loop interchange. Since reordered operations are not independent, dependence analysis is insufficient to establish legality. On the other hand, our implementation of fractal symbolic analysis, described in the last section and in greater detail in [6], is able to automatically verify the legality of this transformation. In this section, we describe this process.

Recall that dependence analysis cannot prove Figure 7a and 7c equivalent due to dependences between reordered swaps ($B1.c$) and updates ($B2$). However, given the fact that $k \leq p(k)$,[2] the two codes still produce the same results.

[2] The predicate $k \leq p(k)$ is easily inferred from the code using techniques such as array value propagation [5].

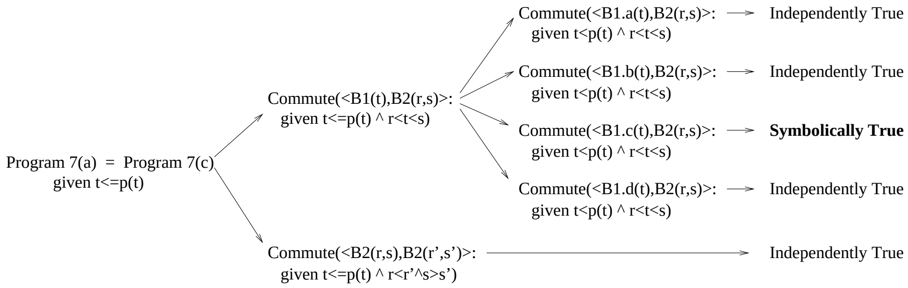

Fig. 12. Fractal Symbolic Analysis of LU

$B2(m,n)$:
```
do i = m+1, N
   A(i,n) = A(i,n) - A(i,m)*A(m,n)
```

$B1.c(l)$:
```
do k = 1, N
   tmp = A(l,k)
   A(l,k) = A(p(l),k)
   A(p(l),k) = tmp
```

(a) $B2(m,n); B1.c(l)$

$B1.c(l)$:
```
do k = 1, N
   tmp = A(l,k)
   A(l,k) = A(p(l),k)
   A(p(l),k) = tmp
```

$B2(m,n)$:
```
do i = m+1, N
   A(i,n) = A(i,n) - A(i,m)*A(m,n)
```

(b) $B1.c(l); B2(m,n)$

Fig. 13. Simplified Comparison

Fractal symbolic analysis deduces correctly that these codes are equal. Figure 12 illustrates this process on the two codes. Essentially, fractal symbolic analysis is able to reduce the legality of the right-left interchange to the symbolic legality of reordering swaps ($B1.c$) and updates ($B2$).

This simpler legality test is illustrated in in Figure 13. Dependences are still violated, but these programs are "simple enough" to be compared by direct symbolic analysis. The only live, altered variable in either program is the array `A`, and the core symbolic engine generates equivalent guarded symbolic expressions for `A` from each program:

$$A_{out}(i,j) = \begin{cases} i = l \wedge j = n & \rightarrow A_{in}(p(l),n) - A_{in}(p(l),m) * A_{in}(m,n) \\ i = p(l) \wedge j = n & \rightarrow A_{in}(l,n) - A_{in}(l,m) * A_{in}(m,n) \\ i = l \wedge j \neq n & \rightarrow A_{in}(p(l),j) \\ i = p(l) \wedge j \neq n & \rightarrow A_{in}(l,j) \\ i \neq l \wedge i \neq p(l) \wedge j = n & \rightarrow A_{in}(i,n) - A_{in}(i,m) * A_{in}(m,n) \\ i \neq l \wedge i \neq p(l) \wedge j \neq n & \rightarrow A_{in}(i,j) \end{cases}$$

At this point, the symbolic expressions corresponding to each guard are syntactically equivalent. This need not be the case in general. However, in this case, it demonstrates that the programs in Figure 13 (and, thus, the original codes in Figure 7a and 7c) are computationally equivalent. That is, fractal symbolic analysis is able to demonstrate that no floating point computation is reordered between right- and left-looking formulations of LU factorization with partial pivoting, therefore the transformation does not affect numerical stability.

5 Conclusions

In this paper, we have studied right and left formulations for three important linear algebra kernels and argued the importance of automatically converting between the two formulations. Furthermore, we have abstracted the high-level transformation that equates the two formulations of these codes. We have discussed how fractal symbolic analysis may be used to establish the legality of this transformation, and have demonstrated its applicability to LU factorization with pivoting, a case in which dependence analysis fails. As far as we are aware, fractal symbolic analysis is the only technique general enough to equate left and right formulations for all the examples mentioned in this paper.

As a future goal, we would like to *synthesize* transformation sequences using fractal symbolic analysis. Dependence information can be represented abstractly using dependence vectors or polyhedra, and these representations have been exploited to synthesize transformation sequences [1, 8]. At present, we do not know suitable representations for the results of fractal symbolic analysis, nor do we know how to synthesize transformation sequences from such information.

References

1. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia, 1995.
3. L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.
4. Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.
5. V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. International Conference on Supercomputing*, pages 265–269, July 1995.
6. Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis for program transformations. Technical Report TR2000-1781, Cornell University, Computer Science, January 2000.
7. Nikolay Mateev, Vijay Menon, and Keshav Pingali. Left-looking to right-looking and vice versa: An application of fractal symbolic analysis to linear algebra code restructuring. Technical Report TR2000-1797, Cornell University, Computer Science, June 2000.
8. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.