# Access Normalization: Loop Restructuring for NUMA Computers

Wei Li
Keshav Pingali
*Department of Computer Science*
*Cornell University*
*Ithaca, New York 14853*

**Abstract:** In scalable parallel machines, processors can make local memory accesses much faster than they can make remote memory accesses. In addition, when a number of remote accesses must be made, it is usually more efficient to use block transfers of data rather than to use many small messages. To run well on such machines, software must exploit these features. We believe it is too onerous for a programmer to do this by hand, so we have been exploring the use of restructuring compiler technology for this purpose. In this paper, we start with a language like HPF-FORTRAN with user-specified data distribution and develop a systematic loop transformation strategy called *access normalization* that restructures loop nests to exploit locality and block transfers. We demonstrate the power of our techniques using routines from the BLAS (Basic Linear Algebra Subprograms) library. An important feature of our approach is that we model loop transformations using *invertible* matrices and integer lattice theory.

## 1 Introduction

Scalable parallel machines are usually organized as networks of processor-memory pairs in which a processor can access local data much faster than it can access remote data. For example, in the BBN Butterfly, accesses to local memory take 0.6 microseconds while accesses to remote memory take about 6.6 microseconds [8]. Distributed memory machines like the Intel iPSC/i860 have even greater non-uniformity in memory access times because access to remote data must be orchestrated through the exchange of messages. If non-local accesses are on the critical path through a program, making these accesses local through proper data management will speed up program execution.

A second feature of such architectures is that block transfer of data between processors is more efficient than sending this data using many small messages. Data transfer between processors can be viewed as a pipeline with a large setup time compared to the time per stage. For example, on the Intel iPSC/i860, it takes 70 microseconds to start up communication, but it takes only 1 microsecond to transfer a double precision floating point number between nearest neighbors once the communication has been setup. Therefore, when a number of data items must be sent from one processor to another, it is preferable to use a single long message to amortize startup time.

Contention in the network has the effect of increasing the expected latency of non-local references; therefore, data management to avoid non-local references has the added benefit of reducing contention, thereby improving performance. Interestingly, some analytical studies show that long messages can increase the latency of non-local accesses [1]. This is an argument against long messages, but on current machines, this effect seems to be of secondary importance compared to the benefits of amortizing start-up time, as we show in Section 8.

For the software writer, the implication of these features of non-uniform memory access (NUMA) [2] machines is that programs must not only exploit parallelism but must also manage data to eliminate non-local references wherever possible; where non-local references are necessary, they should be grouped together for block transfers. We believe that it is too onerous for the programmer to accomplish this by hand, so we have been exploring the use of restructuring compilers for this purpose. Existing compiler technology is oriented mostly towards *uniform* memory access machines in which the only concern is exploitation of parallelism. Parallel code is generated by distributing iterations of the outermost loop in a loop nest among the processors, with synchronization

[1] This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and by grants from the Hewlett-Packard Corporation and the Cornell Theory Center.
1

[2] We use this term in a broad sense to include distributed memory machines.

instructions being inserted to take care of dependences carried by this loop. To reduce synchronization, transformations like loop interchange are carried out to move parallel loops outermost wherever possible [3, 7, 26, 37]. This approach does not perform any data management, so it is not suitable for generating good code on NUMA architectures.

An alternative approach, implemented by the Id Nouveau [32] and FORTRAN-D systems [14], among others, is to give the programmer control over how data structures are distributed across the processors. The compiler uses this *data decomposition* information to determine how to assign work to processors. One simple way to do this is to use the so-called *ownership* rule — a processor executes an assignment statement if the left hand side variable of the statement is mapped to the local memory of that processor. A processor executes a loop iteration if it has any work to do in the body for that iteration. Although this strategy takes data mappings into account, it can generate inefficient code, in which all processors execute all iterations 'looking for work to do' if the structure of the loop nest does not match the data distribution [39]. In many of these cases, loop restructuring can improve code quality, but no general approach to loop transformation has been available in this context [14].

In this paper, we present a systematic approach to loop restructuring for parallel machines with a memory hierarchy. As in the ownership approach, our starting point is a language like HPF-FORTRAN with user-specified data decomposition. However, rather than use this information directly to generate code, we use the data distribution information to drive *access normalization*, which transforms loop nests so that code can be generated by distributing iterations of the outermost loop among the processors without compromising locality. The structure of inner loops is chosen so that data can be transferred using block transfers wherever possible.

Our work makes two contributions.

- We describe a new loop transformation strategy called *access normalization* that is useful for compiling programs for parallel machines with non-uniform memory access. It has applications in other areas such as the generation of vector code.
- Our loop transformations are expressed in the framework of *invertible* matrices and integer lattice theory, which is an important generalization of existing frameworks that use unimodular matrices.

The rest of the paper is organized as follows. In Section 2, we discuss a simple example that gives an overview of our compiling strategy. We also introduce the *data access matrix*, which plays a key role in the development. In Section 3, we discuss the framework of *invertible* matrices as a foundation for loop transformations. For some programs, the data access matrix is invertible and can be used directly to transform the loop nest, as we show in Section 4. In general, however, this matrix may not be invertible, and the techniques of Section 5 must be used to produce an invertible matrix for the transformation. The final problem is guaranteeing that the transformation respects program dependences; this is done in Section 6. In Section 7, we discuss how code can be generated after loops have been restructured according to our methods. We present experimental results in Section 8 that demonstrate that our methods work well on programs of practical interest such as routines from the BLAS (Basic Linear Algebra Subroutines) library [10]. Finally, we discuss related work in Section 9.

# 2 Overview of NUMA Compilation

In this section, we give an overview of our compilation strategy for NUMA architectures. We also introduce a key data structure called the *data access matrix*.

## 2.1 NUMA Compilation

Our compiler accepts programs written in FORTRAN-77 extended with data distribution declarations that specify how arrays are to be distributed across the local memories of the machine. We support most of the data distributions commonly used by programmers of NUMA machines, such as *wrapped* and *blocked* column and row distributions. In a wrapped column distribution, the columns of an array are distributed in a round-robin manner to the processors: if *P* is the number of processors, then processor 0 gets columns *0, P, 2P* and so on, while processor 1 gets columns *1, P+1, 2P+1,* etc. Most of the examples in this paper use a *wrapped column* distribution. Blocked column distribution is similar, except that a processor gets a contiguous set of columns. We also support so-called 2-D blocks in which rectangular *subblocks* of the array are distributed to the processors [14], but for lack of space, we will not consider them any further in this paper.

Data distributions can be specified precisely using a distribution function.

**Definition 2.1** A *distribution function* is a function from array indices to integers between 0 and P-1, where P is the number of processors in the machine. An array dimension is a *distribution dimension*, if that dimension is used in the distribution function for the array.

For example, the distribution function for the wrapped column distribution of a two dimensional array is $W_2(i, j) = j \ mod \ P$, and the second dimension of the array is a distribution dimension.

To understand the need for loop restructuring, consider the program in Figure 1(a), which is a simplified version of the

```
for i = 0, N_1 - 1
  for j = i, i+b-1
    for k = 0, N_2 - 1
      B[i, j-i] = B[i, j-i] + A[i, j+k]
```
(a)

```
for i = p, N_1 - 1, step P
  for j = i, i+b-1
    for k = 0, N_2 - 1
      B[i, j-i] = B[i, j-i] + A[i, j+k]
```
(b)

```
for u = 0, b-1
  for v = u, u + N_1 + N_2 - 2
    for w = 0, N_1 - 1
      B[w, u] = B[w, u] + A[w, v]
```
(c)

```
for u = p, b-1, step P
  for v = u, u + N_1 + N_2 - 2
    read A[*,v];
    for w = 0, N_1 - 1
      B[w, u] = B[w, u] + A[w, v]
```
(d)

Figure 1: Transformation and Code Generation for a Simple Example

SYR2K code discussed in Section 8. Assume that both $A$ and $B$ have a wrapped column distribution. Distributing iterations of the outer loop among the processors (Figure 1(b)) results in processor $p$ executing iterations $p$, $p + P$, etc. Consider accesses to elements of array $B$. Each iteration of the outer loop makes $N_2(b - b/P)$ non-local accesses, and the total number of non-local accesses is $N_1N_2b(1 - 1/P)$.

The ownership rule uses data decomposition information to generate code. A processor is involved in the execution of an iteration $(i, j, k)$ if it owns any of the elements referenced in the body of the loop in that iteration. Therefore, processor $p$ has work to do in iteration $(i, j, k)$ if $(j - i)\ mod\ P\ =\ p$ (it must update an element of $B$) or if $(j + k)\ mod\ P\ =\ p$ (it must send an element of $A$ to whichever processor is updating $B$ in that iteration). This is accomplished by placing these conditional tests in front of the statement, and having all the processors execute all iterations 'looking for work to do' [32, 9]. In simple programs, these conditional tests can be optimized away, but in general they must be executed at runtime, which is inefficient. Moreover, in our program, the code cannot make use of block transfers of elements of $A$ since the elements of $A$ referenced during one iteration of the $j$ loop are mapped to different processors.

Now, consider the program of Figure 1(c). This program computes the same function as Figure 1(a), but if we distribute the outermost loop among the processors as before (Figure 1(d)), there are no non-local accesses to $B$. There are non-local accesses to $A$ but these can be performed using block transfers since the subscript in the distribution dimension of $A$ is invariant in the innermost loop. The loop transformations described in this paper transform the program of Figure 1(a) to that of Figure 1(c). Given the transformed program, the code generation techniques described in Section 7 generate the parallel code shown in Figure 1(d).

## 2.2 Data Access Matrix

Since the transformations are driven by the data access patterns, it is convenient to define a data structure to represent array subscripts in a loop nest in a convenient way. This data structure is called the *data access matrix*. It is used by our loop restructuring system as the starting point for determining what transformations to apply to the loop nest. For the loop nest in Figure 1(a), the data access matrix is

$$\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

This matrix represents the subscripts in the sense that the product of the data access matrix with the column vector $[i, j, k]^T$ yields a column vector in which each element is a subscript from the program. For our example, this product is the column vector $[j - i, j + k, i]^T$ which corresponds to the three subscripts of the program. Constants in a subscript are omitted from the corresponding entry in the data access matrix.

The order in which these subscripts are represented in the data access matrix is important and corresponds to an estimate of their relative importance for achieving good performance. A reasonable heuristic is to give highest importance to subscripts in the distribution dimension(s) of arrays; in our example, the subscripts $j - i$ and $j + k$ dominate the subscript $i$ since they occur in the distribution dimensions of arrays $B$ and $A$. Notice that $j - i$ occurs twice, but $j + k$ occurs only once. Therefore, we let $j - i$ dominate $j + k$. This yields the data access matrix shown above.

The technical development in the rest of the paper is independent of how subscripts were ordered to obtain the data access matrix. In addition, a subscript that is 'overly complex' for any reason (such as a non-linear function of loop indices) may be omitted from the data access matrix without affecting correctness.

## 3 Loop Transformations and Invertible Matrices

In this section, we show how invertible matrices can be used to model the loop transformations of interest in the NUMA context. Consider a simple loop nest

```
for i = 1, 3
  for j = 1, 3
    A[2i+4j, i+5j] = j;
```

It is to be restructured to the form

```
for u = 6, 18 step 2
  for v = u/2 + max( 3⌈(u-6)/4⌉, 3 ),
          u/2 + min(3⌊(u-2)/4⌋, 9 )
          step 3
    A[u, v] = (2v-u)/6;
```

To determine how to perform the transformation, consider the iteration spaces of the two loops, shown in Figure 2. Since the bodies of both loops have the same statement, we must ensure that the work done in any iteration of the original loop nest is done in exactly one iteration of the new loop nest. Therefore, we must construct a one-to-one mapping from the old iteration space to the new one. Moreover, every iteration of the new loop nest must correspond to some point in the old iteration space, so the mapping must be an onto mapping. In other words, we must construct an invertible mapping between the two iteration spaces. One such mapping can be described concisely by the following set of equations, written in matrix form:

$$\begin{pmatrix} 2 & 4 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}.$$

This mapping can be represented using an invertible integer matrix because it is a *linear, integral, invertible mapping* between the two iteration spaces.

The use of invertible matrices to model loop transformations is a generalization of the *unimodular* approach which can be used to model loop interchange, skewing and reversal [7, 37]. Invertible matrices include unimodular matrices as a special case, and permit us to model *loop scaling* as well. An example of this transformation, which replaces a loop index with an integer multiple of the loop index, is shown below.

```
for i = 1, 3          for u = 2,6,2
  A[2*i] = i            A[u] = u/2
```

(a) original code (b) loop scaling

Loop scaling may introduce integer divisions, as is shown in the example, but these operations can be strength reduced and replaced with additions. Like skewing or reversal, loop scaling is not particularly interesting in isolation, but combined with the other transformations, it lets us do wholesale loop restructuring for NUMA architectures.

The algorithm for generating a restructured program starting from a loop nest and an invertible mapping is given in the technical report[24]. This algorithm is non-trivial since the new loop nest must traverse points in the new iteration space in lexicographic order, and the starting point, ending point and step size of a loop in the restructured loop nest can depend on only the loop indices of outer loops (for instance, these values for the outermost loop must be constant). It is not immediately obvious that this can be done for any invertible matrix $T$. Fortunately, the iteration space of a loop nest forms what is called an *integer lattice*; by applying some results from integer lattice theory, we can easily construct the required loop nest.

Since invertible matrices are closed under matrix product, it follows that any sequence of these loop transformations (permutation, reversal, skewing and scaling) can also be modeled as an invertible matrix. This means that the problem of performing the right sequence of loop transformations now reduces to that of finding an appropriate invertible matrix that models the desired sequence of transformations. We show how to do this next.

## 4 Invertible Data Access Matrices

In this section, we consider the simple case where the data access matrix is invertible. Consider the program of Figure 1 again. The data access matrix for the program is $X$.

$$X = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

It is easy to verify that $X$ is invertible; the result of transforming the source program using $X$ as the transformation matrix was shown in Figure 1(c).

Consider what happens when code is generated for the new loop nest by distributing iterations of the outermost loop among the processors in a round-robin manner. Since the outermost loop index is also the the subscript of the distribution

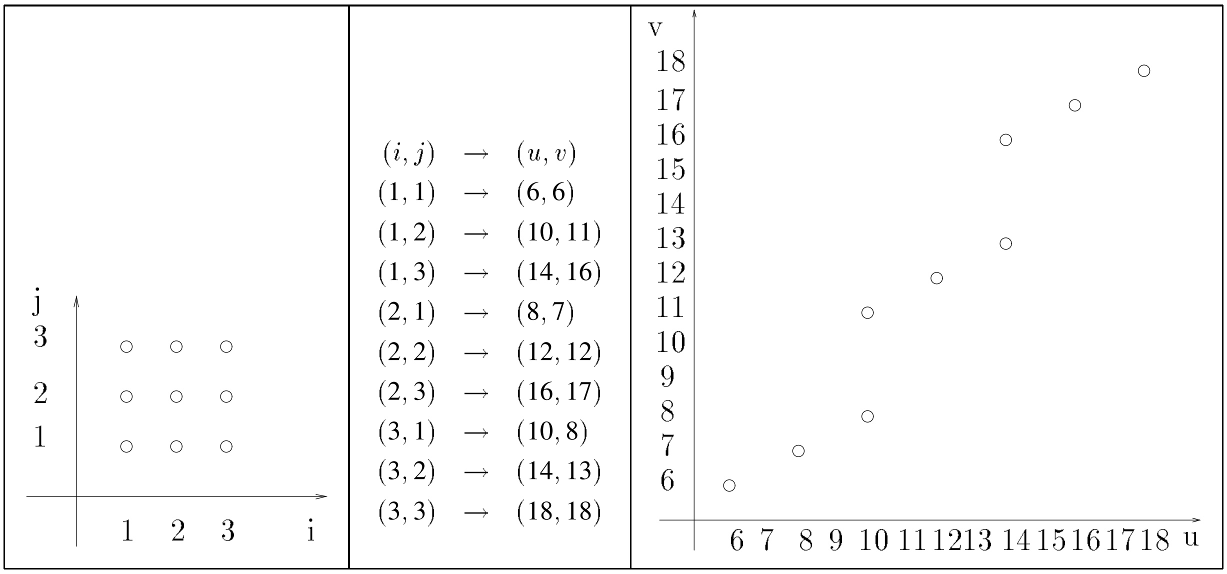



Figure 2: Mapping between Iteration Spaces

dimension of array $B$, all references to $B$ will be purely local. We cannot accomplish this for both $A$ and $B$ simultaneously since the subscripts in the distribution dimensions of $A$ and $B$ are different; therefore, there will be non-local accesses to $A$. However, since the subscript in the distribution dimension of the reference to $A$ was placed second in the data access matrix, this subscript in the new loop nest corresponds to the second loop index and we can perform block transfers for accesses to $A$, as was shown in Figure 1(d).

For future reference, we define the following notion.

**Definition 4.1** Given an array reference, an array subscript is *normal* with respect to loop $i$, if it is equal to the loop index variable $i$.

In this example, the data access matrix yielded the transformation without any complications. This is not the case in general. First, the data access matrix may not be invertible. We handle this case in Section 5. Second, the transformation suggested by the data access matrix may violate one or more data dependences. We take care of this problem in Section 6. In both cases, the goal is to produce an invertible matrix that retains as many rows of the data access matrix as possible.

# 5 Non-invertible Data Access Matrices

In general, the data access matrix is not invertible, so it cannot be used directly to transform the loop nest. The techniques in this section convert such a matrix into an invertible matrix that retains as many rows (subscripts) of the data access matrix as possible. This is done in two stages — first, we eliminate linearly dependent rows from the data access matrix using Algorithm *BasisMatrix*, and second, we pad this reduced matrix with additional rows using Algorithm *Padding*, to get a matrix that is invertible. The details of these algorithms can be found in the associated technical report; here we outline what these algorithms do.

## 5.1 Basis Matrix

It is easy to design an inefficient algorithm that takes a data access matrix and selects as many linearly independent rows as possible: we simply go down the rows of the matrix in sequence, discarding a row if it is linearly dependent on the rows before it, and keeping it otherwise. It is important to traverse the rows in sequence since it ensures that less important rows are discarded in favor of more important ones. For future reference, let us call the resulting matrix the *basis matrix* corresponding to the data access matrix.

**Definition 5.1** The *basis matrix* of a data access matrix *A* is the first row basis of *A*.

The algorithm described informally above is simple, but it is expensive to keep checking rows for independence. A more efficient algorithm is obtained by using a variation of computing the Hermite normal form[23]. A detailed understanding of this algorithm is not important for reading the rest of the paper, so we give an informal description of what it does. Given a data access matrix, Algorithm *BasisMatrix* in Figure 3 returns a permutation matrix $P$, and the rank $d$ of the data access matrix (the number of linearly independent rows). The first $d$ rows of the permutation matrix $P$ tell us which rows of the data access matrix are in the basis matrix. The following example should make this clear.

**Input:** *An $m \times n$ data access matrix A.*
**Output:** *An $m \times m$ permutation matrix and the rank of A.*

```
Algorithm BasisMatrix(A) : (PermMatrix, Rank)

begin
  P = I, where I is the m × m identity matrix.
  done = false;
  i = 1;
  While not done do
    /* Consider the submatrix A[i:m, i:n] */
    Search for the first j ≥ i such that A[j, i:n] ≠ 0⃗;
    If no such j exists Then
      done = true;
    Else
      If j ≠ i then
        Exchange A[i, 1:n] with A[j, 1:n]
        Exchange P[i, 1:m] with P[j, 1:m]
      End-If
      Apply the elementary column operations to make
        A[i, i] nonzero and A[i, i+1:n] zero.
      i = i + 1;
    End-If
  End-While
  return (P, i-1);
end
```

Figure 3: Computing a Basis Matrix

Consider the data access matrix $X = \begin{pmatrix} 1 & 1 & -1 & 0 \\ 2 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$. This data access matrix can arise from the following program

```
for i = ...
  for j = ...
    for k = ...
      for l = ...
        R[i+j-k, 2i+2j-2k, k-l] = ...
```

Algorithm *BasisMatrix(X)* returns the permutation matrix $P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ and rank $d = 2$. The first two rows of the permutation matrix tell us which rows of $A$ form a linearly independent basis: the position of the non-zero entry in these rows of $P$ indicates which row of $A$ is in the basis. In this example, the first and third rows form the basis matrix $B = \begin{pmatrix} 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$. The significance of this in terms of transformations is that only the first and third subscripts can be normalized. This is reasonable because the subscript $2i + 2j - 2k$ is just a multiple of the subscript $i + j - k$.

## 5.2 Padding Matrix

To extend the basis matrix to an invertible matrix, we need to add additional mutually independent rows which are also independent of the rows of the basis matrix. There is some flexibility in the choice of the padding matrix, and we will use this flexibility to our advantage in the next section when we discuss dependences.

**Input:** *An $m \times n$ basis matrix B.*
**Output:** *An $(n - m) \times n$ padding matrix H.*

```
Algorithm Padding(B) : PadMatrix

begin
  H = I, where I is an n × n identity matrix.
  For i = 1, m do
    /* Consider the submatrix B[i:m, i:n] */
    apply the elementary column operations to make
      B[i, i] nonzero and B[i, i+1:n] zero.
    If columns i and j have been exchanged Then
      exchange rows i and j of H
    End-If
  End-For
  return (H[m+1:n, 1:n]);
end
```

Figure 4: Computing a Padding Matrix

Algorithm *Padding* in Figure 4 constructs one possible padding matrix as follows. It is well known that for a full row rank matrix, there exist $m$ columns that are linearly independent. We simply need to pad these columns with $\vec{0}$ and the rest of the columns with columns from the $(n - m) \times (n - m)$ identity matrix $I$. For the above program, since the first column and the third column are linearly independent, the padding matrix is $H = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. The mapping between the old and new iteration spaces is

$$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ l \end{pmatrix} = \begin{pmatrix} u \\ v \\ w \\ z \end{pmatrix}$$

In the transformed program shown below, the reference becomes *R[u, 2u, v]*, and second index is not normalized.

*for u = ...*
  *for v = ...*
    *for w = ...*
      *for z = ...*
        *R[u, 2u, v]*

# 6 Data Dependences

The results of Section 5 showed that a basis matrix can always be padded to yield an integer, invertible matrix. However, there is no guarantee that the transformation corresponding to this final matrix is legal, because this transformation may violate data dependences.

There are three kinds of data dependences between statements. A *data flow-dependence* occurs when a value computed in one statement is used in another statement. A *data anti-dependence* occurs when a variable used in one statement before being reassigned by another statement. A *data output-dependence* occurs when a variable is computed before being recomputed by another statement. A data dependence in the loop nest can be represented by *distance vector* or *direction vector*. For example, the distance vector $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ tell us that the dependence is between successive iterations of the innermost loop. A dependence vector has the property that its leading non-zero is always positive; a legal transformation must preserve this property for each dependence, since the source of the dependence must be executed before its destination. More information on data dependences and techniques of dependence analysis can be found in [6].

To understand the legality of transformations, consider $A = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}$, a basis matrix, and $D_A = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, the dependence matrix. Each column of the dependence matrix represents the distance vector of a dependence in the loop nest. In our example, there is just one dependence. If $T$ is an invertible matrix representing a loop transformation, it is easily shown that $TD$ is the dependence matrix of the restructured loop nest; therefore, the leading non-zero element in each column of $TD$ must be positive. By looking at the product $AD_A = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, we can see at once that $A$ cannot be padded to give us a transformation that respects data dependences. The intuition is that the first two rows of $A$ determine the two outermost loops of the transformed loop nest. In the original program, the dependence was carried by the innermost loop, but in the new program, the dependence is 'carried' by the second loop. Unfortunately, the negative value of the second dimension of $AD_A$ means that the source of the dependence will be executed after the sink. Clearly, there is nothing we can do in the *inner* loops that would remedy this situation, so it is impossible to pad $A$ to yield a legal transformation.

**Input:** *An $m \times n$ basis matrix B*
  *and a dependence matrix D.*
**Output:** *A legal basis Matrix.*

*Algorithm LegalBasis (B, D) : BasisMatrix*

*begin*
  *Let $B_i$ be the ith row of B*
    *and $d_i$ be the ith column of D.*
  *For i = 1, m*
    *$f^T = B_i\, D$*
    *If each element of f is non-negative then*
      *D = D - $d_j$, where f[j] > 0*
    *Elseif each element of f is non-positive then*
      *$B_i$ = (-1) $B_i$;*
      *D = D - $d_j$, where f[j] < 0*
    *Else*
      *B = B - $B_i$;*
    *End-If*
  *End-For*
  *return B;*
*end*

Figure 5: Computing a Legal Basis Matrix

To get around this problem, we proceed in two steps. We start with the basis matrix and use Algorithm *LegalBasis* to produce a new basis matrix that does not violate dependences. Then, we pad this matrix using Algorithm *LegalInvt* to yield the final transformation. In this paper, we discuss only the case when dependences are represented by distances; it is straight-forward to extend these results to dependence *directions*[23].

## 6.1 Generating a Legal Basis

Algorithm *LegalBasis* , shown in Figure 5, takes a basis matrix and checks each row against the dependences. For example, consider the product of the first row and $D_A$. This gives us a row vector in which entries can be positive, zero or negative. If an entry is positive, it means that the corresponding dependence will be carried by the new outermost loop. Therefore, the structure of the inner loops does not matter as far as this dependence is concerned, and we may delete it from the $D_A$ matrix for the rest of the algorithm. If the entry is zero, then the dependence will not be carried by the potential outermost loop, so we leave the dependence in the $D_A$ matrix. However, if we have a negative entry, the dependence is 'carried' by the potential outer loop, but the order of

**Input:** *An $m \times n$ legal basis matrix B*
*and a dependence matrix D.*
**Output:** *An $n \times n$ legal invertible matrix T.*

*Algorithm LegalInvt(B, D) : Matrix*

*begin*
*/* Let $B_i$ be row $i$ of $B$, and $d_i$ be column $i$ of $D$ */*
*For i = 1, m*
*$f^T = B_i\ D$*
*D = D - $d_j$, where $f[j] > 0$*
*End-For*
*r = m + 1;*
*While D is not empty do*
*$Z^T$ = the basis matrix of $D^T$;*
*find the first $e_k$ that is not orthogonal to $D$;*
*$x = cZ(Z^T Z)^{-1} Z^T e_k$;*
*where c is a positive integer that makes x*
*an integer vector.*
*$f^T = x^T\ D$ ; /* $f[j] \geq 0$ */*
*D = D - $d_j$, where $f[j] > 0$*
*$B_r = x^T$;*
*r = r + 1;*
*End-While*

*H = Padding(B);*
*return(append(B, H));*
*end*

Figure 6: Computing a Legal Invertible Matrix

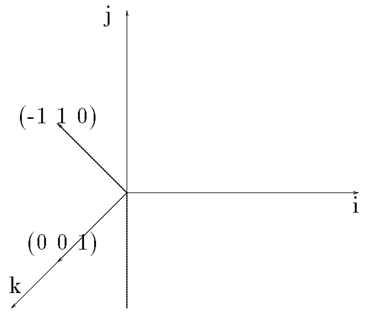
the iterations is wrong. Notice that if all of the entries of the row vector are 0 or negative (intuitively, for all dependences, the potential outer loop either does not carry the dependence or the source of the dependence is executed after the sink), we can simply reverse the direction of the loop. Problems arise only if some entries are positive and others negative — in that case, we cannot keep that row of the basis matrix, and we delete it from the basis matrix. For the above example, *LegalBasis (A)* generates the basis $A_1 = \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$.

## 6.2 Legal Padding Matrix

To pad a legal basis matrix, we need to satisfy two constraints. First, any row added must be linearly independent of other rows, so that the final matrix is invertible. Second, the row must not violate dependence constraints. Once a new row has been added during padding, all dependences carried by the loop corresponding to this row may be dropped from consideration when filling in the rest of the matrix. When there are no further dependences to be satisfied, we can apply Algorithm *Padding* of Section 5.2 to complete the generation of a legal, invertible matrix.

As an example, consider the basis matrix $B = \begin{pmatrix} -1 & 1 & 0 \end{pmatrix}$ which is legal with respect to the dependence matrix $D = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$. The first dependence is carried by the new outermost loop represented by the first row of $B$, and can be dropped from consideration for the rest of the procedure. The inner product of the first row with the second dependence is 0, meaning that this dependence is not carried by the new outermost loop; therefore, it must be taken into account when padding the matrix. To pad $B$, we need to find a row whose inner product with the second dependence vector is non-negative. In the geometric sense, the angle between the two vectors must be less than or equal to 90 degrees. Thus, the general problem can be stated succinctly as that of finding a vector that is linearly independent of the existing row vectors in the basis matrix and within 90 degrees of each dependence vector.

It is not immediately clear that such a vector exists; fortunately, Algorithm *LegalInvt* in Figure 6 gives a positive answer by computing such a vector using a standard result about projections. This vector can be written as $x = cZ(Z^T Z)^{-1} Z^T e_k$ for some positive scaling integer $c$ that makes all of the entries integers, where $e_i^T = [0, 0, .., 1, .., 0]$, with the 1 in the $i$th position, and Z is a column basis from $D$.




For our example, the remaining dependence to be satisfied is $e_3$. The new row vector for the padding is $x = e_3$. Since the dependence is carried by the loop corresponding to this new row vector, we can drop the dependence from consideration now. The dependence matrix is empty at this point. The new legal basis matrix is $B_1 = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Then we can use the Algorithm *Padding* to produce an invertible matrix. The final matrix $T = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ is a linear, invertible matrix and the corresponding transformation satisfies all of the dependences.

The correctness of Algorithm *LegalInvt* follows from the following theorem.

**Theorem 6.1** *The invertible matrix returned by Algorithm LegalInvt is consistent with program dependences.*

**Proof:** Notice that the dependence vectors that need to be satisfied are orthogonal to the row vectors of the basis matrix. If we can find a vector from the subspace spanned by the dependence vectors, then this vector must be orthogonal to the basis rows, therefore linearly independent of the existing row vectors. The invariants of the while-loop are that $AD = 0$; the rows of $A$ are linear independent; and for every column $d_i$ of $D$, $e_k^T d_i \geq 0$. Let $d_i = Zy$ for some $y$ since $Z$ is a basis of the columns of $D$. Since x = $cZ(Z^T Z)^{-1} Z^T e_k$ and $AZ = 0$, $Ax = 0$. Since $e_k^T d_i \geq 0$, we have that $x^T d_i \geq 0$. After each step, the rank of the column space of $D$ decreases at least by one, so the size of $D$ is decreasing and the algorithm will terminate. □

As a final remark, we note that the choice of the padding matrix in this paper is quite arbitrary. For a machine in which processors have a first-level cache, there is the obvious possibility of selecting the padding to improve cache performance by incorporating results on blocking of nested loops[12, 33]. We leave this for future work.

### 6.3 Direction Vectors

Direction vectors provide a conservative approximation when the distance of dependences can not be detected at compile time. They can be represented by signs "$<$", "$>$" ,"=" and "$*$". "$<$" means that the distance is positive; "$>$" negative, and "$*$" unknown. A direction vector can be ($< >$ =) or (= $< *$), as long as the leading nonzero is positive.

The algorithms in this paper can be extended to handle direction vectors. For lack of space, details can be found in the associated technical report [23].

## 7 NUMA Code Generation

Once the program has been transformed by access normalization, we must generate the code that will run on each processor. We generate the same code for each processor, but this code is parameterized by the processor number so that each processor does only the work for which it is responsible.

The general technique for partitioning the iteration space of the loop nest among the processors is called *tiling*. Here, we will restrict ourselves to the special case of wrapped and blocked distributions introduced in Section 2. For these distributions, it is sufficient to distribute the iterations of the outermost loop of the transformed loop nest among the processors. Consider the first row of the transformation matrix: one of the following cases must be true.

- The row was present in the data access matrix, so it corresponds to a subscript in the original program, and this subscript is in a distribution dimension.

- The row was present in the data access matrix, but it is not a distribution dimension.

- The row was introduced by padding.

In cases (ii) and (iii), access normalization cannot exploit locality, and we generate code simply by assigning iterations to processors in a round-robin manner. This code can still exploit block transfers. For case (i), an iteration should be executed by a processor if the corresponding data element is mapped to its local memory.

First, consider the case when the step size is 1. For a wrapped distribution, processor $p$ owns the *data segments* $p$, $p + P$, $p + 2P$, .. etc, where a data segment is a column in the wrapped column distribution or a row in the wrapped row distribution. Since the iterations that access the data segments on processor $p$ are assigned to processor $p$, it is easy to verify that the iterations executed by processor $p$ are the ones shown in Figure 7(b). The lower bound $\lceil \frac{l-p}{P} \rceil * P + p$ is the first iteration between $l$ and $u$ that belongs to process $p$.

When the step size is not 1 (Figure 7)(d), we must solve a linear congruence for the wrapped distribution. Assume that the step size is positive, since the solution can be easily extended to handle the case when the step size is negative. The iterations can be represented by $i = l + n * s$ where $n$ is a parameter with integer values. The iterations that belong to process $p$ are these satisfying the equation $l + n * s = p$ (mod $P$). Using results from number theory, we know that the when the g.c.d of $s$ and $P$ ( written as $(s, P)$) divides $(l - p)$, there is an infinite number of solutions in the form of $n = n_0 + t * P/(s, P)$ for some integer solution $0 \leq n_0 < P/(s, P)$ and integer free variable $t$. However, only certain $t$'s are solutions for iterations within the loop bounds. Since $l \leq i \leq u$ and $i = l + (n_0 + t * P/(s, P)) * s$, the range of $t$ is $\lceil \frac{-n_0}{P/(s,P)} \rceil = 0 \leq t \leq \lfloor \frac{u-l-n_0*s}{P/(s,P)*s} \rfloor$. Therefore the loop for processor $p$ is in Figure 7(e).

Given this assignment of iterations to processors, we must generate synchronization instructions to take care of dependences carried by the outermost loop, and insert block transfers wherever possible. Inserting block transfers is similar to message vectorization in distributed memory machines or block-invalidates for software cache coherent schemes. These steps are routine [13, 26, 31], and are omitted from this paper.

| | | |
|---|---|---|
| *for i = l, u* | *for i =* $\lceil \frac{l-p}{P} \rceil * P + p$,<br>*u,*<br>*step P* | *for i = max(l,* $p * S$*),*<br>*min(u,* $(p+1) * S - 1$*)* |
| (a) unit step | (b) task $p$ for wrapped distribution | (c) task $p$ for blocked distribution |
| *for i = l, u, step s* | *for i =* $l + n_0 * s$,<br>*u,*<br>*step* $P/(P,s) * s$ | *for i = max(l,* $p * S$*),*<br>*min(u,* $(p+1) * S - 1$*),*<br>*step s* |
| (d) non-unit step | (e) task $p$ for wrapped distribution | (f) task $p$ for blocked distribution |

Figure 7: Distributing loops among processors

# 8 Empirical Results and Performance Analysis

In this section, we report the performance of our techniques on routines from the BLAS (Basic Linear Algebra Subprograms) library. The target machine is a BBN Butterfly GP-1000. On this machine, a processor can access its local memory in about 0.6 microsecond, but a non-local access takes about 6.6 microseconds even in the absence of contention in the network. For block transfers, the startup time is about 8 microseconds, and after that, a byte is transferred every 0.31 microseconds [8]. Our compiler takes as input FORTRAN-77 programs with data distribution information, and it generates C code for each processor; this node program is compiled into native code using the Green Hills C compiler (Release 1.8.4). The C compiler performs only conventional code optimizations, so our experimental results are not skewed by any restructuring performed by this compiler. We will use pseudo-code in discussing examples.

For the GEMM code, our techniques are successful in eliminating non-local accesses significantly, so block transfers contribute just a small amount to overall performance. In the SYR2K code, the reduction of non-local accesses is less significant, so block transfers of non-local data are important for good performance.

## 8.1 GEMM

General matrix multiplication (GEMM) is one of the central subroutines in BLAS.

```
for i = 1, N
  for j = 1, N
    for k = 1, N
      C[i, j] = C[i, j] + A[i, k] * B[k, j]
```

All arrays are of size 400 by 400 and are distributed in wrapped column manner. By distributing the outermost loop among the processors without doing any transformations, we obtain the graph labeled *gemm* in Figure 8(a).

The data access matrix is $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$, and dependence matrix is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. The invertible matrix for the transformation is $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$.

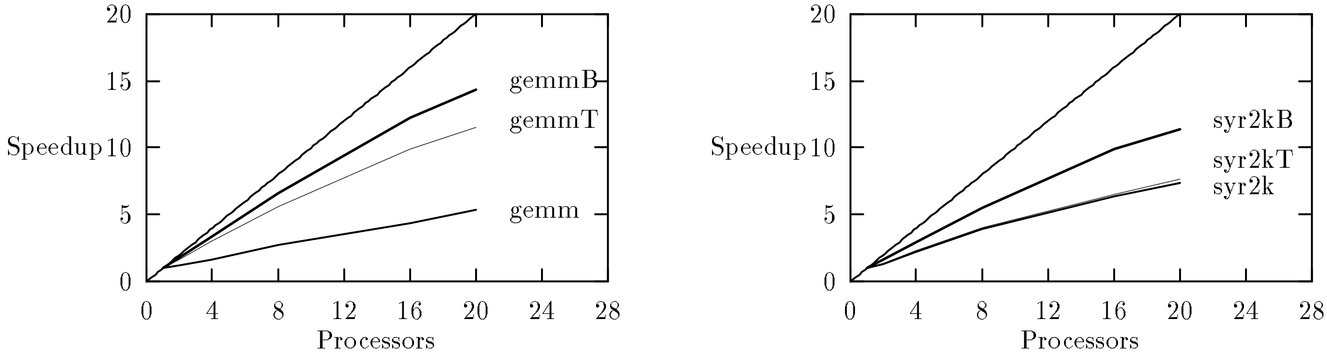
The transformed loop nest yields the following parallel code with the performance labeled *gemmB*(Figure 8(a)). The curve labeled *gemmT* is the speedup without block transfers.

```
for u = p, N, step P
  for v = 1, N
    read A[*, v];
    for w = 1, N
      C[w, u] = C[w, u] + A[w, v] * B[v, u]
```

After access normalization, accesses to *C* and *B* are local, but there are non-local accesses to *A*. Since three out of four data structure accesses in each iteration have become local, the effect of block transfers is relatively small.

## 8.2 SYR2K

When remote accesses are necessary due to the problem structure, it is beneficial to use block data transfers to amortize the cost of the startup time. Consider the rank *2k* update SYR2K from BLAS (Basic Linear Algebra Subroutines) [10]. The subroutine computes $C = \alpha A^T B + \alpha B^T A + C$. Suppose *A* and *B* are banded matrices with band width $b$, then *C* is symmetric and banded with band width $2b - 1$. The banded matrices *A, B* are stored in $n \times 2b - 1$ arrays $A_b, B_b$ such




(a) GEMM

(b) banded SYR2K

Figure 8: Speedups of GEMM and banded SYR2K

that the elements $A[i, j]$, $B[i, j]$ are in $A_b[i, j - i + b - 1]$ and $B_b[i, j - i + b - 1]$. $C$ is symmetric so only the upper triangular matrix is stored in an $n \times (2b - 1)$ array $C_b$ such that $C[i, j]$ is in $C_b[i, j - i]$. The program is shown below.

```
for i = 1, N
  for j = i, min(i+2b-2, N)
    for k = max(i-b+1, j-b+1, 1),
          min(i+b-1, j+b-1, N)
      Cb[i,j-i+1] = Cb[i,j-i+1]
        + αAb[k,i-k+b]*Bb[k,j-k+b]
        + αAb[k,j-k+b]*Bb[k,i-k+b]
```

Assume that we are given a wrapped-column mapping for each array. The data access matrix is $\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}$. If we apply Algorithm *BasisMatrix*, we get a base matrix $B$ consisting of the first three rows. However, the dependence matrix is $[0, 0, 1]^T$. The legal base mapping is $B_{legal} = \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, which is $B$ with the second row negated. This matrix is invertible. Using $B_{legal}$ as the transformation matrix and parallelizing the new nest, we get the parallel code

```
for u = p, 2b-2, step P
  for v = 1-b, b-u
    read Ab[*,-u-v+b]; read Ab[*,-v+b];
    read Bb[*,-v+b]; read Bb[*,-u-v+b];
    for w = max(1, u+v), min(N, N+v)
      Cb[-u-v+w+1, u] = Cb[-u-v+w, u]
        + αAb[w, -u-v+b]*Bb[w, -v+b]
        + αAb[w, -v+b]*Bb[w, -u-v+b]
```

The experimental results are shown in Figure 8(b). The problem size is 500 with a band size of 200. Block transfers are relatively important in this example, since there are many non-local accesses left in the transformed code.

A simple performance model explaining these results can be found in the associated technical report.

# 9 Summary and Related Work

This paper is a contribution to the state of the art of compiling programs in languages like HPF-FORTRAN that permit user-defined data decomposition for parallel machines with a memory hierarchy, which is the goal of a number of projects including FORTRAN-D, Id Nouveau, Superb and Crystal [32, 9, 14, 19, 22, 27, 34, 39]. The emphasis in these projects has been on code generation mechanisms (such as the ownership rule discussed in Section 2) and on recognizing and exploiting special patterns of computation and communication such as reductions. Although it is well-known that loop restructuring before code generation can improve performance, no systematic loop restructuring mechanism using a general loop transformation framework has been available.

We have attempted to exploit locality by matching code to the data distribution across the machine. This is a *static* notion of locality, and must be differentiated from the *dynamic* locality that must be exploited on parallel machines with coherent caches [17]. On such machines, the key to high performance is *data reuse*, and the code must be restructured to allow reuse of cached data wherever possible. Restructuring techniques for doing this have been explored by Wolf and Lam [36]. Their approach is complementary to the one described here. It is likely that scalable parallel architectures will be organized as networks of processor-memory pairs in which processors have an on-chip cache and perhaps a second level cache between the processor and its local memory. The techniques in this paper can be used to partition work and data among the processors; techniques to enhance data reuse can be used to optimize uniprocessor cache performance.

Our use of matrix techniques generalizes the unimodular matrix approach [7, 37]. Unimodular matrices were used by Kumar, Kulkarni and Basu [20] to eliminate outermost loop-carried dependences in generating code for distributed memory machines. In our work, we use invertible matrices, which include unimodular matrices as a special case. This lets us model loop scaling as well, which is important in the NUMA context. In general, it is easier to work with invertible matrices since there are fewer constraints to be satisfied in generating invertible matrices, as opposed to unimodular matrices. There are a number of other loop transformations like distribution, jamming and alignment that are useful in generating code for parallel machines [38]. It would be useful to extend the matrix framework to incorporate these transformations. Related work on loop transformations can be found in [4, 3, 12, 16, 21, 25, 28, 29, 33, 35, 37, 38].

The data access matrix is a new concept introduced in this paper, and access normalization is useful in other contexts such as code generation for vector machines. On many vector machines such as the CRAY-1 and CRAY-2, vector loads and stores must have constant stride. Even on machines such as the Fujitsu FACOM that support scatter and gather operations, it is more efficient to use constant stride accesses wherever possible since address generation for vector elements is faster. The techniques in this paper can be used to accomplish this[23].

We require the programmer to specify data distributions. Automatic deduction of this information for special programs has been investigated by Balasundaram and others [5], by Gannon *et al* [12] on CEDAR-like architectures, by Hudak and Abraham [15] for sequentially iterated parallel loops, by Knobe *et al* [18] for SIMD machines, by Li and Chen [22] for index domain alignment and by Ramanujam and Sadayappan[30] who find communication-free partitioning of arrays in fully parallel loops. These efforts focus on deducing good data distributions for particular kinds of programs such as fully parallel loops, and no general solution to this problem is known. We speculate that it might be possible to start with the dependence matrix and use our techniques in reverse, so to speak, to determine what a good data distribution should be. The main difficulty in doing this is to ensure that the resulting parallel code is load balanced.

# 10 Acknowledgments

We thank Radha Jagadeesan and Danny Ralph for useful discussions, Richard Huff for proof reading the draft, and Mark Charney, Richard Johnson, Mayan Moudgill and Paul Stodghill for comments. The anonymous referees from both ASPLOS '92 and TOCS have provided us many helpful comments. In particular, we are grateful to Mary Lou Soffa for shepherding the ASPLOS version of this paper.

# References

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] F. Allen, J. Cocke, and K. Kennedy. *Reduction of Operator Strength*, pages 79–101. Prentice-Hall, 1981.

[3] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Progamming Languages and Systems*, 9(4):491–542, October 1987.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[5] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proc. 5th Distributed Memory Comput. Conf.*, April 1990.

[6] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[7] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[8] BBN Advanced Computers Inc. *Butterfly GP1000 Switch Tutorial*, 1989.

[9] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2), October 1988.

[10] T. Coleman and C. Van Loan. *Handbook for Matrix Computations*. SIAM Publication, Phil, 1988.

[11] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[12] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformaions. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[13] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, Bonn University, FRG, 1989.

[14] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for FORTRAN-D on MIMD distributed-memory machines. Technical Report TR91-156, Rice University, April 1991.

[15] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proc. ACM Int. Conf. Supercomputing*, June 1990.

[16] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, January 1988.

[17] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, Ma 02154. *Parallel Programming Manual*, 1991.

[18] K. Knobe, J. Lukas, and G. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, February 1990.

[19] C. Koelbel and P. Mehrotra. Compiling global namespace parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2, October 1991.

[20] K. G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. Technical Report FG-TR-014, Center for Development of Advanced Computing, Bangalore, INDIA, January 1991.

[21] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, pages 83–93, February 1974.

[22] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical report, Yale University, 1989.

[23] W. Li and K. Pingali. Access normalization: loop restructuring for NUMA compilers. Technical Report 92-1278, Department of Computer Science, Cornell University, 1992.

[24] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. Technical Report 92-1294, Department of Computer Science, Cornell University, July 1992.

[25] L. Lu. A unified framework for systematic loop transformations. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, April 1991.

[26] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on computers*, C-36:1485–1495, December 1987.

[27] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proc. of the 2nd Int. Conf. on Supercomputing*, July 1988.

[28] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of ACM*, 29(12):1184–1201, December 1986.

[29] A. Porterfield. *Software Methords for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[30] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2, October 1991.

[31] A. Rogers. *Compiling for Locality of Reference*. PhD thesis, Cornell University, 1990.

[32] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[33] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, NASA RIACS, May 1990.

[34] P. Tseng. *A Parallelizing Compiler For Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, 1989.

[35] D. Whitfield and M. L. Soffa. Automatic generation of global optimizers. In *Proc. of the SIGPLAN '91 Conf. on Programming Language Design and Implementation, SIGPLAN Notices*, June 1991.

[36] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[37] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[38] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.

[39] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series, New York, New York, 1990.