Access Normalization: Loop Restructuring for NUMA Compilers

Wei Li
Keshav Pingali

Department of Computer Science
Cornell University
Ithaca, New York 14853

CTC92TR99
July 1992

Abstract

A common feature of many scalable parallel machines is non-uniform memory access — a processor can access data in its local memory ten to a thousand times faster than it can access non-local data. 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 FORTRAN-D with user-specified data distributions and develop a systematic loop transformation strategy called *access normalization* that restructures loop nests to exploit both locality and block transfers wherever possible. We demonstrate the power of our techniques using routines from the BLAS (Basic Linear Algebra Subprograms) library. Our loop transformation strategy is expressed in the framework of *invertible* matrices and integer lattice theory, and it is an important generalization of Banerjee's framework of unimodular matrices.

[1]This research was supported by an NSF Presidential Young Investigator award (NSF grant #CCR-8958543), by NSF grant #CCR-9008526, and by a grant from the Hewlett-Packard Corporation. Wei Li was supported by the Cornell Theory Center.

1 Introduction

Scalable parallel machines are often organized as networks of processor-memory pairs; examples of such machines are the BBN TC 2000, the Kendall Square Research 'all-cache' machine, and multi-computers like the Intel iPSC/i860. These machines are called *non-uniform memory access* (NUMA) machines because a processor can access data in its local memory ten to a thousand times faster than it can access non-local data. For example, in the Kendall Square Research 'all-cache' machine, accesses to local memory take 18 cycles while accesses to non-local memory take 175 cycles [Ken91]. Distributed memory machines like the Intel iPSC/i860 have even greater non-uniformity in access times because access to non-local data must be orchestrated through the exchange of messages [Int91]. 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 most NUMA 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 [Rue]. 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, analytical studies show that long messages can increase the latency of non-local accesses [Aga91]. 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 NUMA 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, along with inserting synchronization instructions 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 [AK87, MP87, Ban90, WL91b]. This approach does not perform any data management; it is not suitable for generating good code on NUMA architectures.

An alternative approach, implemented by the FORTRAN-D language [HKT91], 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 [ZC90]. In many of these cases, loop restructuring can improve code quality, but no general approach to loop transformation has been available in this context [HKT91].

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 FORTRAN-D 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 is a loop restructuring technique that subsumes loop interchange, loop skewing, loop reversal and loop scaling [Wol89]. The objective of the restructuring is to transform loop nests so that code can be generated by distributing iterations of the outermost loops among the processors without compromising locality. The structure of inner loops is chosen so that data can be transferred using block transfers wherever possible.

This paper makes two contributions.

- We describe a new technique called *access normalization* for compiling programs for parallel machines with non-uniform memory access.
- Our loop transformations are expressed in the framework of *invertible* matrices and integer lattice theory, which is an important generalization of Banerjee's framework of unimodular matrices [Ban90]. This generalization is of interest in its own right and it has applications in other areas.

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 [CL88]. 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 [HKT91].

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 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' [CK88, ZC90]. 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

```
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

loop are referenced by 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. 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. The use of matrix methods for loop transformations was pioneered by Banerjee who showed how common loop transformations such as loop interchange could be modeled using *unimodular* matrices [Ban90]. Unimodular matrices are not sufficient for our purpose. In this section, we present an important generalization of Banerjee's technique that uses *invertible* matrices; unimodular matrices are a special case of invertible matrices.

3.1 Invertible Mapping

Consider the simple loop nest in Figure 2(a), which is to be restructured to the form shown in Figure 2(b). To determine how to perform the transformation, consider the iteration spaces of the two loops. 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. Figure 2 shows one such mapping which can be described concisely by the following set of equations, written in matrix form:

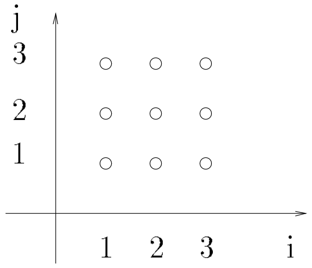
$$\begin{pmatrix} 2 & 4 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}.$$

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

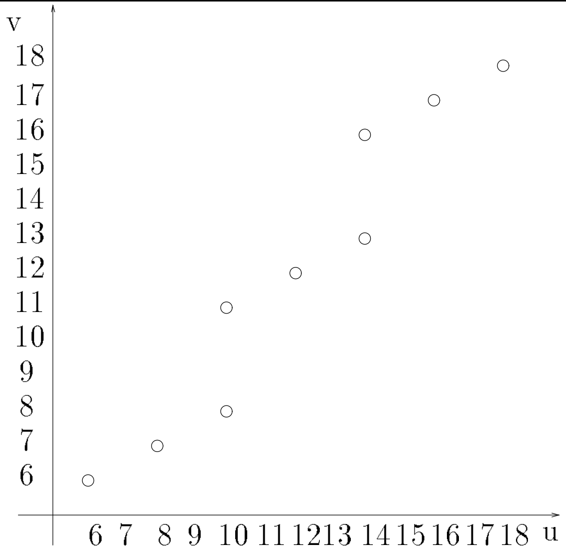
(a) original code

```
for u = lb1, ub1 step s1
  for v = lb2, ub2 step s2
    A[u, v] = new-expression;
```

(b) desired code



(i,j)	$\rightarrow$	(u,v)
$(1,1)$	$\rightarrow$	$(6,6)$
$(1,2)$	$\rightarrow$	$(10,11)$
$(1,3)$	$\rightarrow$	$(14,16)$
$(2,1)$	$\rightarrow$	$(8,7)$
$(2,2)$	$\rightarrow$	$(12,12)$
$(2,3)$	$\rightarrow$	$(16,17)$
$(3,1)$	$\rightarrow$	$(10,8)$
$(3,2)$	$\rightarrow$	$(14,13)$
$(3,3)$	$\rightarrow$	$(18,18)$



Mapping between iteration spaces

```
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;
```

(c) The transformed program

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

(d) original code

```
for u = 2,6,2
  A[u] = u/2
```

(e) loop scaling

Figure 2: An Example of Loop Restructuring

This mapping can be represented using an invertible integer matrix because it is a *linear, integral, invertible mapping* between the two iteration spaces. For this mapping, the reader can verify that the transformation results in the program shown in Figure 2(c).

The use of invertible matrices is a generalization of the results of Banerjee who showed that *unimodular* matrices can be used to model loop interchange, skewing and reversal [Ban90]. 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 in Figures 2(d) and (e). This transformation may introduce integer divisions, as is shown in the example, but these operations can be strength reduced and replaced with comparisons and additions [ACK81]. 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.

Section 3.2 gives the algorithm for generating a restructured program starting from a loop nest and an invertible mapping. 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 [Sch86], we can easily construct the required loop nest.

3.2 A Lattice-based Transformation Framework

Given a loop nest with loop indices $[i_1, i_2, ..., i_n]$ and an invertible mapping T, let the new loop indices be $[j_1, j_2, ..., j_n]$. It is convenient to refer to the old and new loop indices as the column vectors I and J respectively. Transforming the code involves the following:

- generating a new loop body by replacing occurrences of the old subscripts with the new subscripts;
- replacing the old loop bounds by new loop bounds;
- changing the step size.

In order to carry out the transformations, we formulate the loop iteration space using an *integer lattice*, which precisely characterizes the loop nest.

Definition 3.1 Let $a_1, a_2, ..., a_m$ be a set of linearly independent integer vectors. The set $\Lambda = \{\lambda_1 a_1 + \lambda_2 a_2 + ... + \lambda_m a_m \mid \lambda_1, ..., \lambda_m \in \mathbb{Z}\}$ is called an *integer lattice* generated by the *basis* $a_1, a_2, ..., a_m$.

For example, the iteration points in Figure 2(a) form an integer lattice generated by two integer vectors $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ within the area bounded by the loop bounds. The integer lattice for the loop nest in Figure 2(c) is generated by the vectors $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 4 \\ 5 \end{pmatrix}$. The invertible mapping between these two loop nests defines a mapping between these two integer lattices.

Now we consider the three steps to construct a new loop nest given an invertible mapping. In the rest of the section, we use I and J to represent both loop nest and the integer lattice defined by the loop nest. The first step is straightforward. To transform the loop body, notice that $I = T^{-1}J$. This is just a set of equations expressing the old subscripts in terms of the new ones, and it can be used to eliminate occurrences of the old subscripts in the body of the loop in favor of the new ones. Note that $T^{-1}J$ will always be an integer point even though T^{-1} may be a rational matrix. Therefore, this transformation may introduce opportunities for performing strength reduction, as discussed earlier.

To generate the new loop bounds, notice that the original loop nest has $2n$ inequalities of the form $lb_i \leq i \leq ub_i$, where lb_i and ub_i are the lower and upper bounds for loop i. We can replace I by $T^{-1}J$ to get a new set of inequalities; the solution to the system will give the new lower and upper bounds. This simple approach works when the transformation matrix is unimodular. A general solution can be found in [LP92].

Another non-trivial problem is that of determining the step size. The difficulty is that to generate code for a loop nest, the step size of any loop can depend only on the values of outer loop indices, and it is not immediately obvious that we can generate code of this form to traverse the new iteration space in lexicographic order. Algorithm *LoopSteps* applies *elementary column operations* to T to reduce it to a triangular matrix with a positive diagonal, and the step sizes can be read off from the diagonal of this reduced matrix. Elementary column operations are the following:

- exchanging two columns, and
- adding an integral multiple of one column to another column, and
- multiplying a column by -1.

The first positive integer on the diagonal will be the loop step for the outermost loop, the second positive integer on the diagonal will be the loop step for the second outermost loop and so on. For the loop nest in Figure 2, the invertible mapping is T, and its reduced matrix is R.

$$T = \begin{pmatrix} 2 & 4 \\ 1 & 5 \end{pmatrix} \qquad R = \begin{pmatrix} 2 & 0 \\ 1 & 3 \end{pmatrix}$$

The reduced matrix R has the diagonal $[2, 3]$, which means that the loop step is 2 for the outer loop, and 3 for the inner loop, as shown in the restructured loop nest.

It is easy to extend Algorithm *LoopSteps* to handle loop nests with non-unit step, as is shown in the following proof of correctness for the algorithm.

Input: *An $n \times n$ invertible mapping T.*
Output: *A vector of new loop steps.*

Algorithm LoopSteps(T : Mapping) : StepVector

```
begin
  For i = 1, n do
    /* Consider the submatrix T[i:n, i:n] */
    Apply elementary column operations to make T[i, i] positive
      and T[i, i+1:n] zero.
  End-For
  return(diag(T));
end
```

Figure 3: Computing Loop Steps

Theorem 3.1 *Algorithm LoopSteps computes the loop step sizes for the restructured loop nest.*

Proof:

(1) Consider a loop nest with unit loop step sizes. Since T is an integer mapping from the whole integer space with the basis I, the loop points in the new iteration space form an integer lattice with the basis T. Let B be the result of elementary column operations on T, which forms another basis for the same lattice. Every loop point is in the form of $\sum_{i=1}^{n} \alpha_i B_i$, where B_i is the ith column of B. When $i = 1$, B_{11} is the loop step size for the outer most loop. Similarly, fixing the outer most loop, B_{22} is the step size of the second loop. In general, fixing loops 1 to $i-1$, B_{ii} is the loop step for loop i.

(2) The original loop nest has non-unit loop step sizes. Without loss of generality, the loop nest can be *shifted* so that $\vec{0}$ is a loop point.

for $i_1 = 0$ *to* ub_1 *step* s_1
  for $i_2 = a_{21}i_1$ *to* ub_2 *step* s_2
    ...
      for $i_n = a_{n1}i_1 + a_{n2}i_2 + ... + a_{n(n-1)}i_{n-1}$ *to* ub_n *step* s_n

The triangular matrix S whose jth row comes from the coefficients of the lower bound and the loop step of the jth loop forms a basis for the lattice of the loop points.

$$S = \begin{pmatrix} s_1 & 0 & . & 0 \\ a_{21} & s_2 & . & 0 \\ . & . & . & . \\ a_{n1} & a_{n2} & . & s_n \end{pmatrix}$$

TS is a basis for the new iteration space. Applying Algorithm *LoopSteps* gives the loop step vector for the new iteration space. □

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, which is reproduced in Figure 4(a,b) for convenience.

```
for i = 0, N1 − 1
  for j = i, i+b-1
    for k = 0, N2 − 1
      B[i, j-i] = B[i, j-i] + A[i, j+k]
```

(a) Source Program

```
for u = 0, b-1
  for v = u, u + N1 + N2 − 2
    for w = 0, N1 − 1
      B[w, u] = B[w, u] + A[w, v]
```

(b) Restructured Program

Figure 4: The running example

The data access matrix for the program is

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

It is easy to verify that X is invertible, and that the program shown in Figure 4(b) is the result of transforming the source program using X as the transformation matrix as described in section 3.2. 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 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.

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

(a) Original Program

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

(b) Transformed Program

$$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 2 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$$

(c) X

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

(d) B

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

(e) H

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

(f) Invertible mapping

Figure 5: A loop nest with a non-invertible data access matrix

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, which makes use of a variation of computing the Hermite normal form, is given in Figure 6. Given a data access matrix, Algorithm *BasisMatrix* 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.

Consider the data access matrix X shown in Figure 5(c). This data access matrix can arise from the program shown in Figure 5(a). 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 in Figure 5. 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 are many possible choices for these rows, and we will use this flexibility when we take care of dependences in Section 6. Algorithm *Padding*, described in Figure 7, shows one simple way to pad the basis matrix. For an $m \times n$ full row rank matrix, we need to pad with an $(n-m) \times n$ matrix to form an invertible matrix. 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 program in Figure 5, since the first column and the third column are linearly independent, the padding matrix is H in Figure 5(e). The mapping between the old and new iteration spaces is in Figure 5(f). In the transformed program, shown in Figure 5(b), the reference becomes *R[u, 2u, v]*, and second index is not normalized.

The correctness of Algorithm *BasisMatrix* and Algorithm *Padding* follows from the fol-

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

```
Algorithm BasisMatrix(A : AccessMatrix) : (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:n] with P[j, 1:n]
      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 6: Computing a Basis Matrix

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

```
Algorithm Padding(B : BasisMatrix) : 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 7: Computing a Padding Matrix

lowing result.

Theorem 5.1 *Let A be an $m \times n$ integer matrix with rank d. There exists an $m \times m$ permutation matrix P and an $n \times n$ invertible matrix Q such that $A = P\begin{pmatrix} B & 0 \\ D & 0 \end{pmatrix}Q$, where B is $d \times d$ lower triangular and nonsingular.*

Proof: Suppose at step k, by elementary column operations and permutation row operations, $P_k A Q_k = \begin{pmatrix} B_k & 0 \\ D_k & E_k \end{pmatrix}$ where B_k is $k \times k$ lower triangular.

- case 1: E_k is a zero matrix, then done.
- case 2: the first row of E_k is $\vec{0}$, then by a row permutation on E_k, the first row can be made non-zero, which falls into case 3.
- case 3: the first row of E_k is not $\vec{0}$. By a sequence of elementary column operations, The top-left position of the first row can be made positive and others zero.

After step d, where the matrix E_d is zero, $P_d A Q_d = \begin{pmatrix} B_d & 0 \\ D_d & 0 \end{pmatrix}$. Let $P = P_d^{-1}$ and $Q = Q_d^{-1}$, since both are invertible matrices. Then $A = P \begin{pmatrix} B & 0 \\ D & 0 \end{pmatrix} Q$, where $B = B_d$ and $D = D_d$. $\square$

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. To understand the problem, consider Figure 9 in which A is a basis matrix and D_A is the dependence matrix. Each column of the dependence matrix represents the *distance vector* of a dependence in the loop nest [Ban90]. In our example, there is just one dependence, and the distance values tell us that the dependence is between successive iterations of the innermost loop. A distance 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. 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.

In Figure 9, A is the basis matrix; 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.

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. We first discuss only the case when dependences are represented by distances; the more general case of dependence directions is discussed in section 6.3.

6.1 Generating a Legal Basis

Algorithm *LegalBasis* , shown in Figure 8, takes a basis matrix and checks each row against the dependences. For example, consider the product of the first row and D. 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 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 matrix. However, if we have a negative entry, the dependence is 'carried' by the potential outer loop, but the order of 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[2] For the example in Figure 9, *LegalBasis (A)* generates the basis A_1 shown in Figure 9.

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 which is legal with respect to the dependence

[2]A better scheme, which is more complex to implement, is to maintain a list of rows that have been removed from the data access matrix, and reconsider these rows whenever dependencies are deleted from the dependence matrix.

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

Algorithm LegalBasis (B : BasisMatrix, D : DepMatrix) : 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 8: Algorithm LegalBasis: Computing a Legal Basis Matrix

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

A

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

D_A

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

A_1

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

B

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

B_1

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

D

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

T

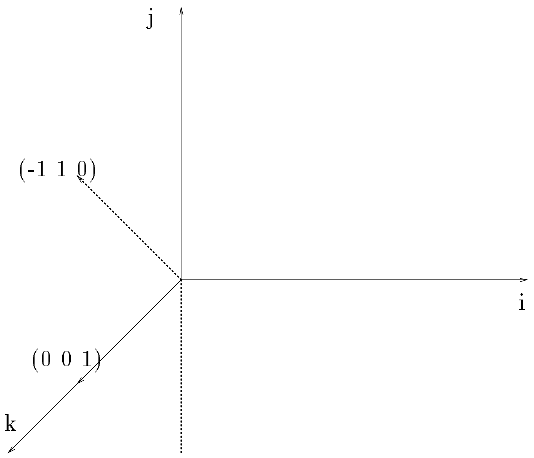


Figure 9: Legal basis and padding matrices

matrix D in Figure 9. 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 (Figure 9). 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 10 gives a positive answer by computing such a vector using a standard result about projections [Sch86]. This vector can be written as $x = cZ(Z^TZ)^{-1}Z^Te_k$ 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 ith 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 in Figure 9. Then we can use the Algorithm *Padding* to produce a linear, invertible matrix. The final matrix T in Figure 9 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, $e_k^Td_i \geq 0$. Let $d_i = Zy$ for some y since Z is a basis of the columns of D. Since x $= cZ(Z^TZ)^{-1}Z^Te_k$ and $AZ = 0$, $Ax = 0$. Since $e_k^Td_i \geq 0$, we have that $x^Td_i \geq 0$. 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. □

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.

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 : BasisMatrix, D : DepMatrix) : 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] ≥ 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 10: Computing a Legal Invertible Matrix

Table for Addition									
$+$	0	p	n	$<$	$\leq$	$>$	$\geq$	$\neq$	$*$
a	a	$a+c$	$a+c$						
$<$	$<$	$<$	$*$	$<$					
$\leq$	$\leq$	$<$	$*$	$\leq$	$\leq$				
$>$	$>$	$*$	$>$	$*$	$*$	$>$			
$\geq$	$\geq$	$*$	$>$	$*$	$*$	$\geq$	$\geq$		
$\neq$	$\neq$	$*$	$*$	$\neq$	$*$	$\neq$	$*$	$\neq$	
$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$

Table for Multiplication			
$\times$	0	p	n
c	0	$c \times p$	$c \times n$
$<$	0	$<$	$>$
$\leq$	0	$\leq$	$\geq$
$>$	0	$>$	$<$
$\geq$	0	$\geq$	$\leq$
$\neq$	0	$\neq$	$\neq$
$*$	0	$*$	$*$

Figure 11: Operators for Directions

In order to describe the computation over these signs, we define operations similar to addition and multiplication on integers. In Figure 11, let c and a be any constant, p be a positive and n be a negative.

Algorithm *LegalBasis* remains the same except that the addition and multiplication of integers are replaced by the addition and multiplication operators defined on directions. Direction vectors also have the property that the leading nonzero must be positive, i.e. either a positive integer or "<". A legal transformation must preserve this property for each dependence.

Consider the following example in which B is a basis matrix and D is the dependence matrix.

Example 6.1 Suppose $B = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}$ and $D = \begin{pmatrix} 0 \\ 0 \\ < \end{pmatrix}$. Then any extension of B to an invertible matrix will be illegal, since $BD = \begin{pmatrix} 0 \\ > \end{pmatrix}$.

The padding algorithm for direction vectors will be different from Algorithm *LegalPadding* in Section 6.2. Given a legal basis matrix, we still have two sets of constraints for a padding matrix, namely linearly independence with the basis matrix and satisfying the data dependences. In the case of the distance dependences, we solve the problem by choosing a vector from the subspace spanned by the columns of D. Let k be the first row in D with nonzeros. The inner product of e_k and each dependence vector is non-negative. We can not compute a basis for the subspace spanned by D, since there may be signs in D. So we take the projection of e_k to the null space of the basis matrix B. The projected vector is $x = c(I - B^T(BB^T)^{-1}B)e_k$, after being scaled to an integer vector by a constant c. Readers can verify that x is within 90 degrees of each dependence vector, and orthogonal to B.

7 NUMA Code Generation

After the program is 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*. First we discuss the special case of wrapped and blocked distributions introduced in Section 2. For these distributions, it can be shown that 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 12(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. For a blocked mapping, the corresponding iterations are shown in Figure 12(c).

When the step size is not 1 (Figure 12)(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 $l + n * s = p \pmod{P}$. 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 $0 \le n_0 < P/(s, P)$ and integer free variable t. However, only certain t's are solutions for iterations within the loop bounds. Since $l \le i \le u$ and $i = l + (n_0 + t * P/(s, P)) * s$, the range of t is $\lceil \frac{-n_0}{P/(s,P)} \rceil = 0 \le t \le \lfloor \frac{u-l-n_0*s}{P/(s,P)*s} \rfloor$. Therefore the loop for processor p is in Figure 12(e). The remaining question is how to compute the special solution n_0. This can be done by

$$\textit{for } i = l,\ u$$

(a) unit step

$$\textit{for } i = \lceil \tfrac{l-p}{P} \rceil * P + p,\quad u,\quad \textit{step } P$$

(b) task p for wrapped distribution

$$\textit{for } i = max(l,\ p * S),\quad min(u,\ (p+1) * S - 1)$$

(c) task p for blocked distribution

$$\textit{for } i = l,\ u,\ \textit{step } s$$

(d) non-unit step

$$\textit{for } i = l + n_0 * s,\quad u,\quad \textit{step } P/(P,s) * s$$

(e) task p for wrapped distribution

$$\textit{for } i = max(l,\ p * S),\quad min(u,\ (p+1) * S - 1),\quad \textit{step } s$$

(f) task p for blocked distribution

```
for i = l_i, u_i
  for j = l_j, u_j
    A[i, j] = 0;
```

(h) the loop nest

```
for i = max(l_i, p * b), min(u_i, (p + 1) * b - 1)
  for j = max(l_j, q * b), min(u_j, (q + 1) * b - 1)
    A[i, j] = 0;
```

(i) code for process (p, q)

Figure 12: Distributing loops among processors

Input: *An equation* $l + n * s = p \pmod{P}$.
Output: *A special solution* $0 \leq n_0 < P$.

Algorithm Asolution(l, s, p, P : Integer) : Integer

begin
if gcd(s, P) doesn't divide (p-l) then no solution;

$s_0 = s \bmod P$;
$c_0 = (p - l) \bmod P$;
/* *solve* $n * s_0 = c_0 \pmod{P}$ */
$c = c_0$;
while (s_0 *doesn't divide c)*
$c = c + P$;
$n_0 = c/s_0$;
return(n_0);
end

Figure 13: Computing a special solution

the algorithm in Figure 13. The equation $s_0 n = c_0 \pmod{P}$ has the same solutions as $l + s * n = p \pmod{P}$. If there is any solution to the equation $s_0 n = c_0 \pmod{P}$ then there is unique solution within $[0, P/(s, P))$ [HW79]. The algorithm finds that solution.

For multi-dimensional block distribution, we need to distribute multiple loops (loop *tiling*). For example, an $n \times n$ array A has a subblock distribution where each subblock is of size $b \times b$. Imagine that the processor names are also two dimensional. Then the element $A[i, j]$ is owned by processor $(i/b, j/b)$. If i and j are the two outermost loops as shown in Figure 12(h), then processor (p, q) has iterations in Figure 12(i). If A has more complicated subscripts or the subscripts are not the loop indices of the two outermost loops, access normalization will attempt to normalize the reference to $A[u, v]$ with u and v as the outermost loops. Therefore the code generation checks the rows in the transformation matrix from top down, and distributes each loop if these rows are from the same array reference.

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. These steps are routine [ZC90], and are omitted from this paper.

8 Empirical Results and Performance Analysis

In this section, we demonstrate the power of our techniques using 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 [BBN89]. 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. We develop a very simple performance model to explain these results.

8.1 GEMM

General matrix multiplication (GEMM) is one of the central subroutines in BLAS. The sequential code for this routine is shown in Figure 14(a). 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 14(i).

The data access matrix and dependence matrix produced by our system are shown in Figure 14(e) and (f). The invertible matrix for the transformation is shown in Figure 14(g). The transformed loop nest of Figure 14(b) yields the code of Figure 14(c), and the corresponding execution times and speed-ups are labeled *gemmT* in Figure 14(h,i). Inserting block transfers for accesses to A, we get the code shown in Figure 14(d), and the performance of this program is labeled *gemmB* in Figure 14(h,i).

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. To understand the behavior of this program, a simple performance model can be used. Since the outer loop of GEMM does not carry any dependences, the time to execute GEMM can be expressed as follows:

$$T_P(\alpha) = \frac{n_f t_f}{P} + \frac{n_m(\alpha t_l + (1-\alpha)t_r(P))}{P} + o(P)$$

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

(a) The original code

```
for u = 1, N
  for v = 1, N
    for w = 1, N
      C[w, u] = C[w, u] + A[w, v] * B[v, u]
```

(b) The transformed code

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

(c) The parallel code for node p

```
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]
```

(d) The parallel code with block transfer

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

(e) Access Matrix

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

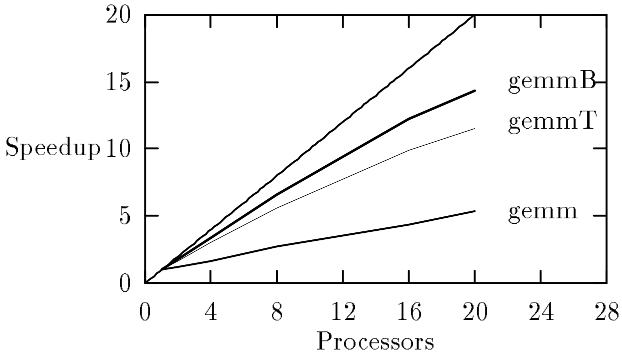
(f) Dependences

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

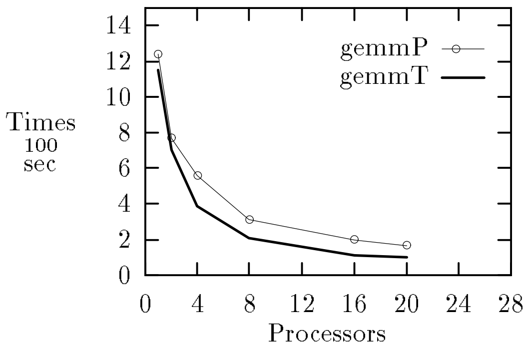
(g) Transformation

Nodes/Prog	gemm	gemmT	gemmB
1	1,245.2	1,150.0	1,117.8
2	1,072.5	704.3	630.6
4	784.4	387.5	328.1
8	462.5	206.2	169.3
16	283.1	116.8	92.5
20	230.6	100.0	78.7

(h) Execution Times (sec) of 400×400 GEMM



(i) Speedup



(j) Projected times

Figure 14: GEMM

where

- P: number of processors.
- α: the proportion of local memory accesses.
- $T_P(\alpha)$: time to execute program — a function of P and α
- n_f: number of arithmetic operations.
- t_f: effective time to execute an arithmetic operation.
- n_m: number of memory accesses.
- t_l: time for a local access.
- $t_r(P)$: time for a remote access. For a given program, it is an increasing function of P because of network contention.
- $o(P)$: overhead to spawn processes etc.

Access normalization increases the proportion of local accesses. Let α_0 and α be the proportions of local accesses before and after access normalization. The reduction in execution time due to access normalization is given by this equation:

$$T_{nb} = T_P(\alpha_0) - T_P(\alpha) = (\alpha - \alpha_0)(t_r(P) - t_l)\frac{n_m}{P}$$

GEMM has $4N^3$ memory accesses. The proportion of local accesses in the original version is $\alpha_0 = \frac{1}{P}$, and in the transformed version, it is $\alpha = \frac{1}{4}(3 + \frac{1}{P})$. Substituting these values, and assuming a non-local access time of 6 microseconds, we would expect that the time to execute the program is

$$T_P(\alpha_0) - T_{nb} = T_P(\alpha_0) - \frac{1200}{P}(1 - \frac{1}{P})$$

Since access normalization reduces network contention, and we have ignored network contention in these calculations, the actual execution times will be less than the prediction. The projected and realized times shown in Figure 14(j) bear these calculations out.

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) [CL88]. 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 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 in Figure 15(a).

Assume that we are given a wrapped-column mapping for each array. The data access matrix is matrix X shown in Figure 15(e). If we apply Algorithm *BasisMatrix*, we get a base matrix B consisting of the first three rows of X. However, the dependence matrix is $[0,0,1]^T$. The legal base mapping is B_{legal}, which is B with the second row negated, as is shown in Figure 15(f). This matrix is invertible. Using B_{legal} as the transformation matrix gives the transformed code in Figure 15(b). Distributing the outermost loop iterations and generating block transfers, we get the parallel code in Figure 15(d) for processor p. Figure 15(g) shows the experimental results. Block transfers are relatively important in this example, since there are many non-local accesses left in the transformed code.

To understand the experimental results, let us include block transfers into our model. When block transfers are done, a block of remote data is transferred to a piece of local memory, and the accesses to the remote location are replaced by accesses to the local copy. Let β to be the proportion of remote data that are transferred using block transfer and t_b be the amortized access time for remote references. The formula for the execution time of the program is refined as follows:

$$T_P(\alpha, \beta) = \frac{n_f t_f}{P} + \frac{n_m}{P}(\alpha t_l + \beta(1-\alpha)t_l + \beta(1-\alpha)t_b(P) + (1-\beta)(1-\alpha)t_r(P)) + o(P)$$

The amortized remote access time can be estimated as follows. Let the cost function of a block transfer be $a + bB$, where a is the startup time, b is the time between successive byte transfers, and B is the block size. Let n_b be the number of block transfers and $s_1, s_2, ...s_{n_b}$ be the sizes of blocks. The amortized access time for data transferred through block transfers can be computed using the following formula:

$$t_b = \frac{sizeof(float) \sum_{i=1}^{n_b} (a + b * s_i)}{\sum_{i=1}^{n_b} s_i} + t_l$$

SYR2K has $12Nb^2$ number of memory references, in which $\alpha = \frac{1}{3}(1 + \frac{2}{P})$ of them are local after transformation. The rest of the remote references are performed using block transfers. The number of block transfers of size N is $8b^2(1 - \frac{1}{P})$. The problem size is 500 with a band size of 200. Thus the amortized remote access time is $t_b \simeq 2\mu s$ per floating point. Therefore the time saved due to block transfers is T_b:

$$T_b = T_P(\alpha, 0) - T_P(\alpha, 1) = \frac{640}{P}(1 - \frac{1}{P})$$

The projected times and actual running times are shown in Figure 15(i). They are very close since contention plays less of a role in this program.

```
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)
      C_b[i,j-i+1] = C_b[i,j-i+1]
        + αA_b[k,i-k+b]*B_b[k,j-k+b]
        + αA_b[k,j-k+b]*B_b[k,i-k+b]
```

(a) The original code

```
for u = 1, 2b-1
  for v = 1-b, b-u
    for w = max(1, u+v), min(N, N+v)
      C_b[-u-v+w+1, u] = C_b[-u-v+w, u]
        + αA_b[w, -u-v+b]*B_b[w, -v+b]
        + αA_b[w, -v+b]*B_b[w, -u-v+b]
```

(b) The transformed code

```
for u = p, 2b-1, step P
  for v = 1-b, b-u
    for w = max(1, u+v), min(N, N+v)
      C_b[-u-v+w+1, u] = C_b[-u-v+w, u]
        + αA_b[w, -u-v+b]*B_b[w, -v+b]
        + αA_b[w, -v+b]*B_b[w, -u-v+b]
```

(c) The parallel code for node p

```
for u = p, 2b-2, step P
  for v = 1-b, b-u
    read A_b[*,-u-v+b]; read A_b[*,-v+b];
    read B_b[*,-v+b]; read B_b[*,-u-v+b];
    for w = max(1, u+v), min(N, N+v)
      C_b[-u-v+w+1, u] = C_b[-u-v+w, u]
        + αA_b[w, -u-v+b]*B_b[w, -v+b]
        + αA_b[w, -v+b]*B_b[w, -u-v+b]
```

(d) The parallel code with block transfer

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

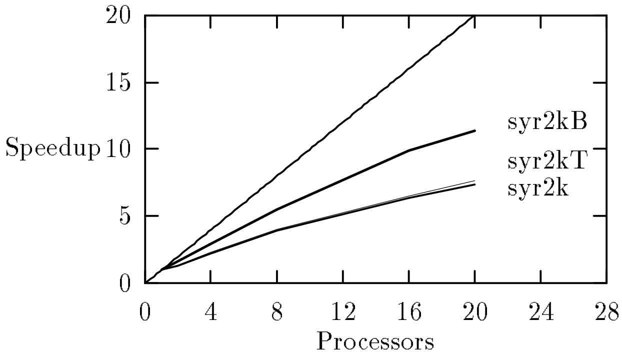
(e) Access Matrix

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

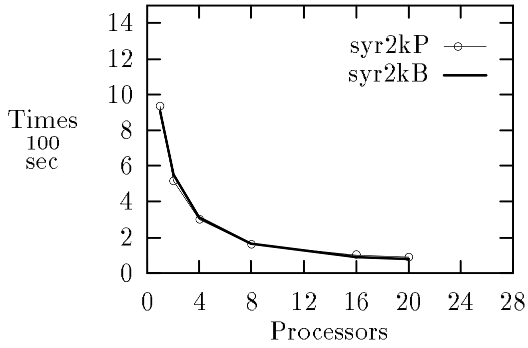
(f) Transformation

Nodes/Prog	syr2k	syr2kT	syr2kB
1	1,289.3	933.5	904.3
2	961.8	675.6	554.3
4	581.8	415.8	310.0
8	334.3	241.2	166.8
16	200.0	143.1	91.8
20	173.8	121.8	79.3

(g) Times (sec) for SYR2K



(h) Speedup



(i) Projected times

Figure 15: SYR2K

9 Discussion and Related Work

This paper is a contribution to the state of the art of compiling programs in languages like FORTRAN-D that permit user-defined data decomposition for parallel machines with a memory hierarchy, which is the goal of a number of projects including Parascope, Superb, Id Nouveau, Crystal, Kali, PARTI and ASPAR projects [CK88, HKT91, ZC90, RP89, Tse89, LC91, KM91, MSS$^+$88, IFKF90]. 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 general loop restructuring mechanism has been available until now.

We require the programmer to specify data distributions. Automatic deduction of this information for special programs has been investigated by Balasundaram and others [BFKK90], by Gannon *et al* [GJG88] on CEDAR-like architectures, by Hudak and Abramham [HA90] for sequentially iterated parallel loops, by Knobe *et al* [KLS90] for SIMD machines, by Li and Chen [LC89] for *index domain alignment* and by Ramanujam and Sadayappan[RS91] 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 have opted to generate code by distributing outer loop iterations among the processor. However, access normalization can also be integrated with the ownership rule. Rather than construct the data access matrix by choosing the dominant subscripts from *all* references as described in Section 2, we can choose the dominant subscripts in the distribution dimension(s) of the array references on the *left* hand side of the assignments only, since the processor that owns the array element being defined executes the assignment statement. Loop nests with multiple assignment statements present other opportunities for generating better code. In this paper, loop restructuring works on entire loop nests and every statement in the loop body undergoes the same transformation. A possible extension is to have different transformations for different statements. For example, statement S_1 has transformation T_1 and S_2 has T_2. Let the new iteration spaces for S_1 and S_2 be J_1 and J_2 respectively. The new loop nest is an union of J_1 and J_2 with appropriate guards (conditionals) for S_1 and S_2 to insure that they only get executed in their subspace. A special case is when the data dependences allow J_1 and J_2 to be two separate iteration spaces; in that case, we can construct different loop nests for S_1 and S_2 respectively — this is the same as *loop distribution*[Wol89]. Further experimentation is required to determine whether this is useful in practice.

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 and a single shared global memory. 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 [WL91a]. Their approach is complementary to the one described here. It is likely that scalable parallel architectures will

```
for i = 1, N
  for j = 1, N
    for k = 1, N
      A[i, j+k, k] = 0;
```

(a)

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

(b)

```
for u = 1, N
  for v = u+1, u+N
    for w = 1, N
      A[w, v, u] = 0;
```

(c)

Figure 16: Restructuring for Constant Stride Accesses

be organized as networks of processor-memory pairs with an on-chip cache and perhaps a second level cache between the processor and its local memory. The techniques in this paper restructure the outer loops of a loop nest for locality and block transfers; however, there is considerable flexibility in padding the basis matrix and Algorithm LegalInvt in Section 6 shows one of many different ways to pad the basis matrix. We conjecture that techniques to enhance data reuse can be used in padding the basis matrix, thereby optimizing inner loops for cache behavior. Note that this style of code generation is recommended for the Kendall Square Research 'all-cache' parallel processor [Ken91].

The data access matrix is a new concept introduced in this paper, and access normalization is useful in other contexts. 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.

As an example, assume that the array in the program in Figure 16(a) is stored in column major order as is usual in FORTRAN. The data access matrix is shown in Figure 16(b). The program in Figure 16(c) is the transformed code using the data access matrix for the transformation. In the new program, accesses to A in the innermost loop have constant stride, and can be performed used a vector store operation with a fixed stride.

Our use of matrix techniques follows the ground-breaking work of Banerjee who showed that unimodular matrices model loop interchange, skewing and reversal. Unimodular matrices were used by Kumar, Kulkarni and Basu [KKB91] 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 this paper, we have required the programmer to specify data distribution, and we have used this to determine how to restructure loops. 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. Once this is automated, it may become feasible to distribute an array differently in different parts of the program.

10 Acknowledgments

We would like to thank Radha Jagadeesan and Danny Ralph for discussions, Richard Huff for proof reading the draft, and other members of the Typhoon project Mark Charney, Richard Johnson, Mayan Moudgill and Paul Stodghill for comments.

References

[ACK81] F. Allen, J. Cocke, and K. Kennedy. *Reduction of Operator Strength*, pages 79–101. Prentice-Hall, 1981.

[Aga91] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[AK87] 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.

[Ban90] 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.

[BBN89] BBN Advanced Computers Inc. *Butterfly GP1000 Switch Tutorial*, 1989.

[BFKK90] 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.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2), October 1988.

[CL88] T. Coleman and C. Van Loan. *Handbook for Matrix Computations.* SIAM Publication, Phil, 1988.

[GJG88] 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.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proc. ACM Int. Conf. Supercomputing*, June 1990.

[HKT91] 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.

[HW79] G. H. Hardy and E. W. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proc. of the 5th Distributed Memory Comp. Conf.*, April 1990.

[Int91] Intel Corporation. *iPSC/i860 System Overview*, 1991.

[Ken91] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, Ma 02154. *Parallel Programming Manual*, 1991.

[KKB91] 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.

[KLS90] 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.

[KM91] C. Koelbel and P. Mehrotra. Compiling global namespace parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2, October 1991.

[LC89] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical report, Yale University, 1989.

[LC91] J. Li and M. Chen. Compiling communication efficient program for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:361–376, July 1991.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Annual Workshop on Languages and Compilers for Parallelism*, August 1992.

[MP87] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on computers*, C-36:1485–1495, December 1987.

[MSS+88] 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.

[RP89] 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.

[RS91] 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.

[Rue] Roland Ruehl. personal communication.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[Tse89] P. Tseng. *A Parallelizing Compiler For Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, 1989.

[WL91a] 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.

[WL91b] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, New York, New York, 1990.