Torres, J. [et al.]. Align and distribute-based linear loop transformations. A: Workshop on Programming Languages and Compilers for Parallel Computing. "Languages and Compilers for Parallel Computing: 6th International Workshop Portland, Oregon, USA, August 12–14, 1993: proceedings". Springer, 1993, p. 321-339. ISBN 978-3-540-57659-4. The final authenticated version is available online at https://doi.org/10.1007/3-540-57659-2_19.

Align and Distribute-based Linear Loop Transformations*

Jordi Torres, Eduard Ayguadé, Jesús Labarta and Mateo Valero

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6, Gran Capità s/núm. 08071 - Barcelona, SPAIN

Abstract

In this paper we generalize the framework of linear loop transformations in the sense that loop alignment is considered as a new component in the transformation process. The aim is to match the structure of loop nests with the data distribution and alignment in order to eliminate non-local references whenever possible when compiling a sequential program for a distributed memory machine. The alignment and distribution functions are assumed to be user specified or automatically generated by the compiler. The transformation process is modelled with non-singular matrices and we use the ideas recently proposed in this field to find part of the transformation matrix and generate an efficient transformed code. However, additional aspects have to be studied when the alignment and distribution functions are considered, both in the obtaining of the transformation matrix and in the generation of code.

1 Introduction

Loop transformations have been recognized to be one of the most important components of the parallelizing and vectorizing technology for current supercomputers. The aim is to transform nested-loop structures in the source program into semantically equivalent versions with more opportunities to parallelize them [1, 2].

When distributed memory machines are considered to run scientific codes, data decomposition is needed. Data have to be decomposed into pieces and distributed among all processors. Optimizing locality is crucial for this kind of architectures and for non-uniform memory architectures (NUMA) in general. As a consequence, it is important to access local data whenever possible to avoid the access to remote data. When non-local references are necessary, and in order to amortize their remote access, block transfer of data and data reuse are additional aspects to be considered and optimized to improve the efficiency.

The programming model offered by Fortran-D [3] and HPF [4] gives the programmer control over how to align and distribute data structures across processors. The compiler has to be able to assign work to processors (the ownership rule is the simplest way to do this [5]) and restructure loop nests with the aim of avoiding non-local accesses as much as possible, and when necessary, optimize communication to transfer remote data [6]. The single-program multiple-data (SPMD) model [7] is used to generate code. Each processor runs the same program but accesses to different parts of the data.

Research in the past years has focussed at finding a matrix theory for program transformations

to reveal program parallelism [8, 9] or exploit data locality and block transfers [10, 11]. From the specification of the source loop nest and the transformation matrix, a target loop nest is generated with more opportunities to exploit parallelism or for data reuse. This step has been solved when unimodular matrices are used [8, 9] and in general, when non-unimodular matrices are considered. The key point in the solutions proposed in the last case is the use of the Fourier-Motzkin elimination method and the Hermite Normal Form decomposition [12, 13, 14].

In this paper we propose to consider the set of statements in the loop body as a new component in the framework of non-singular transformations. This allows to consider loop alignment [15, 16, 17] in a unified way with other loop transformations. We assume that data alignment and distribution is either user specified or automatically generated by the compiler. From the reaching alignment and distribution functions for each array, and array references in the statements, a different transformation for each statement of the loop is derived with the aim of reducing the number of non-local accesses. In the scope of this paper we consider that the same transformation matrix is used for all the statements and add a different alignment component to each of them.

The owner computes rule is the basic mechanism to associate loop iterations to processors. Sometimes the owner computes rule can be relaxed allowing processors to compute values for data they do not own. Once computed, these values have to be send to the owners. Deciding the alignment component for a statement can be done by analyzing its multiple right-hand side and the left-hand side references to distributed arrays. In order to reduce the number of non-local accesses, the owner computes rule can be broken.

Aspects dealing with the assignment of iterations to processors and generation of synchronization or communication instructions that take care of dependences are not considered in this paper.

The rest of the paper is organized as follows. In section 2 we present the terminology and assumptions used along this paper. In section 3 we outline some previous work on loop transformations, code generation for them and obtaining of the transformation matrix when NUMA architectures are considered. Section 4 presents the alignment component in the transformation framework and code generation. In section 5 we discuss some ideas and problems to obtain the alignment component that is added to each statement. Finally, we conclude the paper and present some future work.

2 Terminology and Assumptions

Through this paper we consider perfectly nested loops $\{L_1, ..., L_n\}$ where bounds for any loop L_k $(1 \le k \le n)$ are affine functions of indices of its outer loops $L_1, ..., L_{k-1}$, that is,

$$i_k \ge a_{k,0} + a_{k,1} \cdot i_1 + ... + a_{k,(k-1)} \cdot i_{k-1} = l_k$$
$$i_k \le b_{k,0} + b_{k,1} \cdot i_1 + ... + b_{k,(k-1)} \cdot i_{k-1} = u_k$$

The iteration space for this loop nest is defined as

$$IS = \{ (i_1, ..., i_n) \in Z^n, l_k \le i_k \le u_k, 1 \le k \le n \}$$

and can be written following the matrix notation used in the literature

$$\alpha \cdot I \le \beta$$

where

$$\alpha = \begin{bmatrix} L - ID \\ ID - U \end{bmatrix} \qquad \text{and} \qquad \beta = \begin{bmatrix} -l \\ u \end{bmatrix}$$

L is constructed from the coefficients $a_{k,i}$ ($1 \le k \le n$, $1 \le i \le k-1$) of the loop indices in the lower bound expressions, U is constructed from the coefficients $b_{k,i}$ ($1 \le k \le n$, $1 \le i \le k-1$) of the loop indices in the upper bound expressions, ID is the identity matrix, and l and u are constructed from the independent coefficients $a_{k,0}$ and $b_{k,0}$ ($1 \le k \le n$) of the lower and upper bounds respectively.

For example, for the loop nest in Figure 1.a, the IS of the loop can be defined by

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \le \begin{bmatrix} -1 \\ -1 \\ 10 \\ 8 \end{bmatrix}$$

Figure 1.b shows the aspect of the IS for this loop. Each point represents the execution of one iteration of the inner loop body.

In the scope of this paper we consider dense iteration spaces, i.e., spaces where all points correspond to iterations of the loop.

The loop body is composed of multiple assignment statements $\{S_1, ..., S_m\}$ that reference array variables whose subscripts are affine functions of loop indices $i_1, ..., i_n$. Let V be the set of statements in the loop body. The Statement per Iteration Space (SIS) of a loop nest is defined as the cartesian product

$$SIS = IS \times V$$

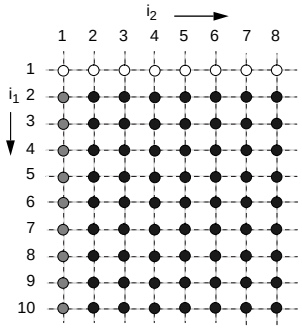
Each point in the SIS represents the execution of an iteration of a statement of the loop body.

Dependence relations between a pair of statements S_i and S_j (denoted $S_i \, \delta \, S_j$) appear when there is an execution ordering between them [18]. We do not distinguish between different kinds of data dependences because they all impose ordering constraints in the same way. When dependence relations are uniform (i.e., invariant through the SIS), they can be characterized by distance vectors $\bar{d}=(d_1, ..., d_n)$ expressing the number of iterations that the dependence extends across in each loop dimension. Dependences are lexicographically positive, that is, the leading non-zero component is always positive.

Arrays accessed during the execution of a loop are considered to be aligned among them and distributed across processors. The alignment and distribution of the arrays is assumed to be user specified or automatically generated by the compiler [19, 20]. In any case, we consider that each array accessed in the loop is affected by a set of reaching alignment and distribution functions. These functions are the standard supported by current data-partitioning languages such as Fortran-D. The ALIGN statement maps each array element onto an index domain or DECOMPOSITION. The DISTRIBUTE statement groups elements of the decomposition and maps them onto the parallel machine. Alignment can be either within or between dimensions and include offsets. Each dimension is localized or distributed in a block, cyclic or block-cyclic manner.

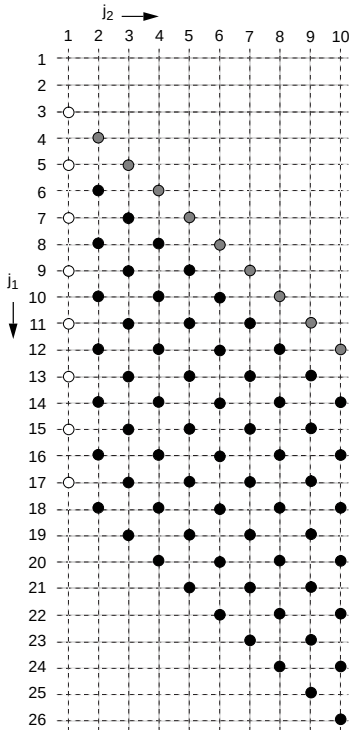
```
ALIGN A, C WITH E
ALIGN B (i, j) WITH E (i+1, j-1)
DISTRIBUTE E (CYCLIC, :)
DO i1=1, 10
    DO i2=1, 8
        A[i1+2·i2, i1] = f(A[i1+2·i2, i1], C[i1+1, i2], B[i1+2·i2-1, i1])
        B[i1+2·i2, i1] = g(A[i1+2·i2+1, i1], B[i1+2·i2, i1])
    ENDO
ENDO
```

(a)



(b)

$$T = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}$$



(c)

```
ALIGN A, C WITH E
ALIGN B (i, j) WITH E (i+1, j-1)
DISTRIBUTE E (CYCLIC, :)
DO j1=3, 26
    DO j2 = j1 + 2 · max (-8, ⌈(1-j1)/2⌉), j1 + 2 · min (-1, ⌊(10-j1)/2⌋), 2
        A[ j1,  j2] = f(A[ j1,  j2], C[  j2+1, ( j1- j2)/2], B[ j1-1,  j2])
        B[ j1,  j2] = g(A[ j1+1,  j2], B[ j1,  j2])
    ENDO
ENDO
```

(d)

Figure 1: Working example: (a) Source loop nest and (b) original IS. (c) Target IS when a non-singular transformation matrix T is used. (d) Transformed loop nest that scans the points in the transformed IS skipping over points that do not have to be executed.

The sequential code in Figure 1.a shows the reaching alignment and distribution functions for the arrays referenced inside the loop. Notice that just one dimension is distributed, as specified by the : attribute in the DISTRIBUTE statement which denotes that the dimension is assigned locally. In the example, if P is the number of processors, processor p (p=1, 2, ..., P) owns rows p, p+P, p+2·P, ... of matrices A and C and p-1, p+P-1, p+2·P-1, ... of matrix B.

3 Linear Loop Transformations and Data Access Matrix

A loop transformation is a mapping between two iteration spaces (named original and target IS). In this paper we consider linear transformations, which can be modelled using non-singular integer matrices. Unimodular matrices (i.e., matrices whose determinant is ±1) are a particular case and can be used to model some basic transformations such as permutations, skewing and reversal [8, 9]. Non-unimodular matrices can be used to model other basic transformations such as scaling [12], but in general, any linear transformation represented by a non-singular integer matrix can be viewed as a composition of these four basic transformations [12].

Let I be a point of the original IS, J a point of the target IS and T the transformation matrix. The relationship among them is

$$J = T \cdot I$$

The target IS can be dense or sparse depending on the unimodularity of the transformation matrix.

In our model, the first p rows of the transformation matrix T define the spatial component of the transformation and the last n-p rows the temporal component [21]. Iterations in the outermost loops of the transformed nest are distributed among the processors while iterations the innermost loops are executed sequentially within each processor.

Let $\bar{d}$ be a distance vector in the original IS. Due to the fact that T is a linear transformation, $T \cdot \bar{d}$ is the transformed distance vector in the target IS. A transformation T is legal if

$$T \cdot \bar{d} > \bar{0}$$

for all dependence relations $\bar{d}$ in the loop. This means that each transformed dependence has to be lexicographically positive in the target IS.

3.1 Data Access Matrix

A linear transformation has to be found in order to match the structure of the loop nests in a program with the reaching data decomposition functions. [11] proposes a representation for array subscripts named Data Access Matrix and its use as starting point to obtain the transformation matrix. The data access matrix A is a n·n matrix such that the product

$$A \cdot I$$

yields a vector of n subscripts from array references in the loop. The subscripts and the order they appear in A correspond to an estimate of their relative importance. For instance, [11] propose an heuristic that gives more importance to subscripts in the distributed dimensions of the arrays and, among them, to those that appear more times.

If a data access matrix A has to be used as a transformation matrix, two conditions have to be imposed:

- matrix A must be invertible.
- matrix A must be legal, that is, it must not violate dependences in the loop.

When matrix A is not invertible, linearly dependent rows have to be eliminated yielding a basis matrix. This basis matrix has to be legal so additional rows must be deleted if they violate dependences. Once a legal basis matrix is obtained, it is padded to an invertible matrix by adding rows which are independent of the rows in the basis matrix and which do not violate dependence constraints. The transformation matrix obtained using this approach retains as many rows of the original data access matrix as possible.

Figure 1 shows the example that is used as working example along the paper. Figure 1.b shows the original IS and Figure 1.c shows the target IS when the following non-singular transformation matrix is used

$$T = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}$$

Different shades are used in the points to help the reader to establish the relationship between points in the original and transformed IS. This transformation matrix corresponds to the data access matrix for the array subscripts in the distributed array dimensions shown in Figure 1.a.

3.2 Code Generation

Once a legal transformation T has been defined, we have to generate a loop nest that appropriately scans the points in the target IS. In order to do that, we have to:

- generate the bounds of the target DO loops and the stride for each loop index in order to skip over points of the target IS that do not have to be executed. As we have seen, the original loop nest is defined by $\alpha \cdot I \leq \beta$. Therefore, if a transformation matrix T is applied, the target IS can be obtained from the inverse matrix T^{-1}

  $$\alpha \cdot T^{-1} \cdot J \leq \beta$$

  However, the bounds obtained by direct elimination from the above inequality might not have the structure assumed at the beginning of the previous section.
- replace the subscripts in the array references that appear in the loop body such that they are affine functions of the new loop indices (i.e., substitute I with $T^{-1} \cdot J$ in all subscript functions).

Some previous works [8, 9] have addressed the problem of code generation when unimodular matrices are used to transform the original IS. [22] includes conditional statements in order to deal with the sparseness that is introduced in the target IS when a non-unimodular matrix is used. Other authors [12, 13, 14] have made proposals to avoid these conditionals and, as a consequence, reduce the overhead introduced by them. The key point in all of them is the use of the Fourier-Motzkin elimination method and the Hermite Normal Form decomposition [23] to obtain the target loop nest.

Figure 2.a summarizes the procedure when the source IS is dense and the transformation matrix is unimodular. With these conditions, the target IS is also dense. If the transformation matrix is non-unimodular, then a sparse IS is obtained. Figure 2.b shows the procedure applied to obtain the target code. The basic idea, as proposed in [12] is to decompose the matrix T into the product of a lower triangular matrix H with positive diagonal elements (Hermite matrix of T) and a unimodular matrix U (reflecting column transformations performed on T to obtain H) such that

$$T = H \cdot U$$

Applying U to the original IS, and using Fourier-Motzkin elimination, the bounds of a dense auxiliary IS are obtained. Because of the sparseness of the target IS, two things have to be done: adjust loop bounds and skip over points that do not have to be executed. Matrix H is used to obtain the bounds of the sparse target loop nest by direct elimination. The diagonal elements of matrix H are the strides of each loop index variable in the target loop nest.

Figure 1.d shows how the original loop nest shown in Figure 1.a is transformed applying the procedure outlined in this section.

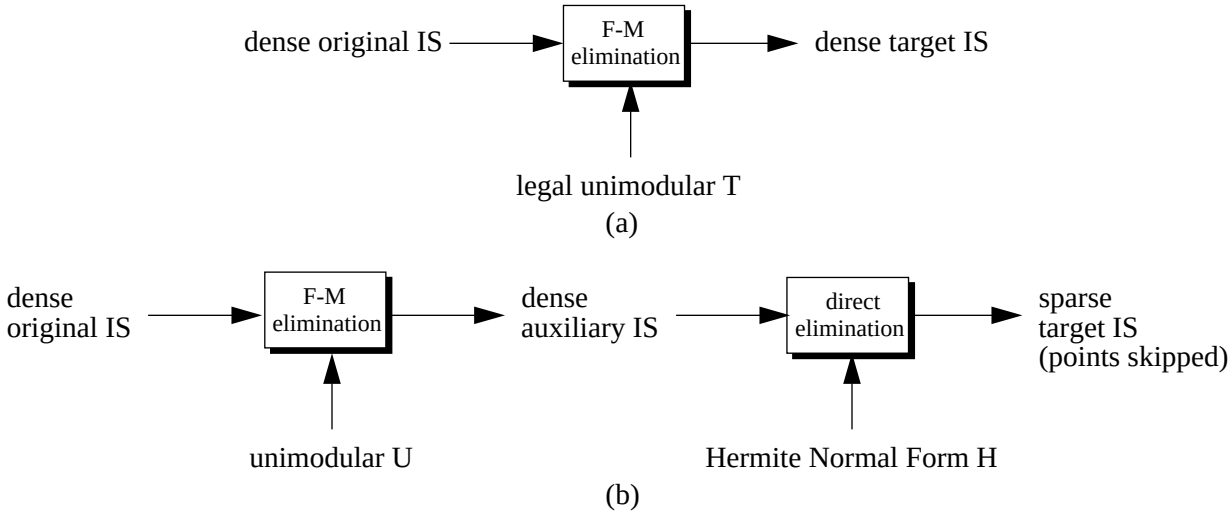


Figure 2: Transforming a dense IS using Fourier-Motzkin elimination. (a) When a unimodular matrix T is used and (b) when a non-unimodular matrix T is used.

4 The Alignment Component

Before presenting the general framework we present the underlying idea in our working example. Assume that iterations of the outermost loop in Figure 1.d are distributed among P processors in such a way that processor p (p=1, 2, ..., P) is involved in the execution of an iteration j_1 if (j_1-1) mod P = (p-1). With this assignment, the accesses to matrices A and B are local in the execution of statement S_1 but non-local in the execution of statement S_2. For instance and assuming P=4, processor p=1 executes iterations j_1=5, 9, ..., 25 and accesses

$$A(5, j_2), A(9, j_2), ..., A(25, j_2)$$

$$B(4, j_2), B(8, j_2), ..., B(24, j_2)$$

in the execution of S_1 which are stored in its local memory and accesses

$$A(6, j_2), A(10, j_2), ..., A(26, j_2)$$

$$B(5, j_2), B(9, j_2), ..., B(25, j_2)$$

in the execution of S_2 which are stored in the local memory of processor p=2. In fact, to make local the accesses to matrices A and B in statement S_2, we would have had to distribute iterations in such a way that j_1 mod P = (p-1). In order to make local all the accesses to A and B, it is necessary to align the execution of statements S_1 and S_2, as shown below.

Consider that each statement in the loop is represented with a hyperplane in the SIS and that we apply a different transformation to each statement in such a way that the resulting target IS is the one shown in Figure 3. Different shades are used to identify the different hyperplanes. If processor p executes iterations $j_1$so that (j_1-1) mod P = (p-1), then all the accesses to matrices A and B are local. For instance, processor p=1 executes j_1=5, 9, ..., 25 and accesses

$$A(5, j_2), A(9, j_2), ..., A(25, j_2)$$

$$B(4, j_2), B(8, j_2), ..., B(24, j_2)$$

in the execution of both S_1 and S_2.

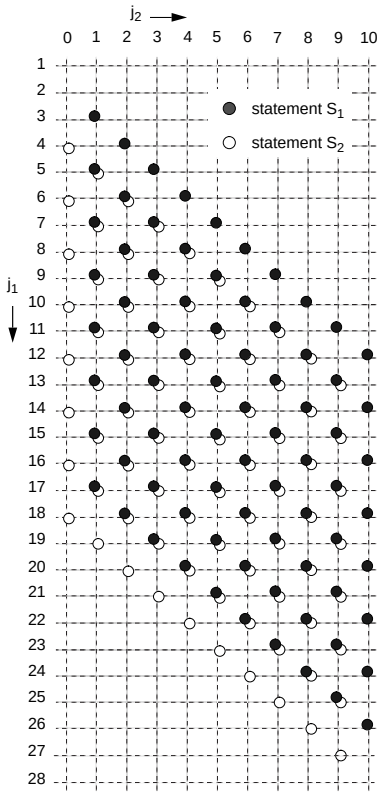


Figure 3: Target SIS assuming that each statement of the loop is transformed with a different transformation.

Due to the alignment of the hyperplanes, most of the iterations in the target loop execute both statements but others just execute one of them. The new bounds of the target loop nest have to be the union of the bounds for each statement. The body has to include the appropriate conditional statements to ensure that each statement is executed within its bounds.

In this section we generalize the framework of linear transformations to include loop alignment. Some additional aspects have to be considered in the generation of the target loop nest, such as the generation of conditional guards to preserve the semantics of the original loop nest and the optimization of these conditional guards to reduce as much as possible the execution overhead introduced.

4.1 Bounds for each statement

We can consider that the SIS is the composition of several hyperplanes, one for each statement in the loop body. Consider that the hyperplane associated to a statement S_i is transformed using the following transformation

$$J^i = T \cdot \left(I + D^i \right)$$

where D^i is the displacement applied to the S_i hyperplane. Notice that all the statements are transformed using the same transformation matrix T.

Let T=H·U be the decomposition of matrix T into a unimodular matrix U and the Hermite upper triangular matrix H. Therefore, the transformed IS for statement S_i is

$$J^i = H \cdot U \cdot \left(I + D^i \right) = H \cdot K^i$$

being K^i the auxiliary IS of the statement S_i

$$K^i = U \cdot \left(I + D^i \right)$$

The bounds of the auxiliary space for statement S_i can be derived from the bounds of the original IS

$$\alpha \cdot I \leq \beta$$

and using U^{-1}, the inverse matrix of the unimodular transformation,

$$I = U^{-1} \cdot K^i - D^i$$

So the bounds of the auxiliary IS of S_i can be obtained applying the Fourier-Motzkin elimination method to

$$\left(\alpha \cdot U^{-1} \cdot K^i \right) - \left(\alpha \cdot D^i \right) \leq \beta$$

or more clearly to

$$\alpha' \cdot K^i \leq \beta^i$$

where

$$\alpha' = \alpha \cdot U^{-1} \qquad \text{and} \qquad \beta^i = \beta + \alpha \cdot D^i$$

Using the matrix H, we can obtain the bounds of the target IS for each statement S_i by direct elimination.

For instance, consider that we transform each statement hyperplane in the working example in the following way:

$$J^1 = T \cdot I$$

$$J^2 = T \cdot \left(I + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right)$$

The decomposition of T in the Hermite Normal Form is given by

$$T = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$$

The bounds of the auxiliary space of each statement can be obtained from

$$\begin{bmatrix} -1 & -2 \\ 0 & 1 \\ 1 & 2 \\ 0 & -1 \end{bmatrix} \cdot K^1 \leq \begin{bmatrix} -1 \\ -1 \\ 10 \\ 8 \end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix} -1 & -2 \\ 0 & 1 \\ 1 & 2 \\ 0 & -1 \end{bmatrix} \cdot K^2 \leq \begin{bmatrix} 0 \\ -2 \\ 9 \\ 9 \end{bmatrix}$$

and are,

DO k_1^1 = 3, 26

  DO k_2^1 = max (-8, $\lceil (1-k_1^1)/2 \rceil$), min (-1, $\lfloor (10-k_1^1)/2 \rfloor$)

for statement S_1, and

DO k_1^2 = 4, 27

  DO k_2^2 = max (-9, $\lceil -k_1^2/2 \rceil$), min (-2, $\lfloor (9 -k_1^2)/2 \rfloor$)

for statement S_2. Due to the sparseness of the target SIS, we have to modify or adjust these bounds using matrix H. Since H is lower triangular, we can derive the relationship between both spaces

$$j_1^i = k_1^i \qquad j_2^i = k_1^i + 2 \cdot k_2^i$$

and finally obtain the bounds

DO j_1 = 3, 26

  DO $j_2 = j_1 + 2 \cdot$ max (-8, $\lceil (1-j_1)/2 \rceil$) , $j_1 + 2 \cdot$ min (-1, $\lfloor (10-j_1)/2 \rfloor$), 2

for statement S_1, and

DO j_1 = 4, 27

  DO $j_2 = j_1 + 2 \cdot$ max (-9, $\lceil -j_1/2 \rceil$) , $j_1 + 2 \cdot$ min (-2, $\lfloor (9 -j_1)/2 \rfloor$) , 2

for statement S_2. Notice that the elements in the diagonal of matrix H are the strides in each loop dimension.

4.2 Cover Bounds

Now we have to obtain the bounds of the union of the two statement hyperplanes in order to obtain the target loop nest. We define the cover bounds as the bounds of a space that includes all the statement hyperplanes. The cover bounds can be obtained by elimination from

$$\alpha' \cdot K^{cover} \leq \max_{\forall i} \{\beta^i\}$$

where the maximum function is applied to each component of the vectors β^i. In our working example we have to solve

$$\begin{bmatrix} -1 & -2 \\ 0 & 1 \\ 1 & 2 \\ 0 & -1 \end{bmatrix} \cdot K^{cover} \leq \begin{bmatrix} 0 \\ -1 \\ 10 \\ 9 \end{bmatrix}$$

obtaining the following target loop nest

DO j_1 = 2, 28

DO $j_2 = j_1 + 2 \cdot \max(-9, \lceil -j_1/2 \rceil), j_1 + 2 \cdot \min(-1, \lfloor (10-j_1)/2 \rfloor), 2$

From now on, and for reasons of clarity, we denote with lower_i^k and upper_i^k the lower and upper bounds for each hyperplane S_k in loop dimension L_i, with step_i the stride for loop dimension L_i, and with lower_i^{cover} and upper_i^{cover} the lower and upper bounds of the cover in loop dimension L_i.

With this notation, the basic structure of the target loop nest and body can be written as shown in Figure 4. The key point to note here is that this code is expensive in terms of run-time overhead. In each iteration of the loop, the guard conditionals that control the execution of each statement have to be evaluated. In general, there is a part of the SIS where all the statements of the loop body are executed and it encompasses a large part of the cover.

```
DO j1 = lower1^cover , upper1^cover , step1
    ...
    DO jk = lowerk^cover , upperk^cover , stepk
    ...
        DO jn = lowern^cover , uppern^cover , stepn
            ...
            IF (lower1^k ≤ j1 ≤upper1^k ) and (...) and (lowern^k ≤ jn ≤uppern^k ) THEN {statement Sk}
            ...
        ENDDO
    ...
    ENDDO
    ...
ENDDO
```

Figure 4: Basic Structure of the code with guard conditionals.

4.3 Core Bounds

We define the core bounds as the bounds of a space where all the statements of the loop body are executed. The core bounds can be obtained by elimination from

$$\alpha' \cdot K^{core} \leq \min_{\forall i} \{\beta^i\}$$

where the minimum function is applied to each component of the vectors β^i. In our working example we have to solve

$$\begin{bmatrix} -1 & -2 \\ 0 & 1 \\ 1 & 2 \\ 0 & -1 \end{bmatrix} \cdot K^{core} \leq \begin{bmatrix} -1 \\ -2 \\ 9 \\ 8 \end{bmatrix}$$

obtaining the following loop nest

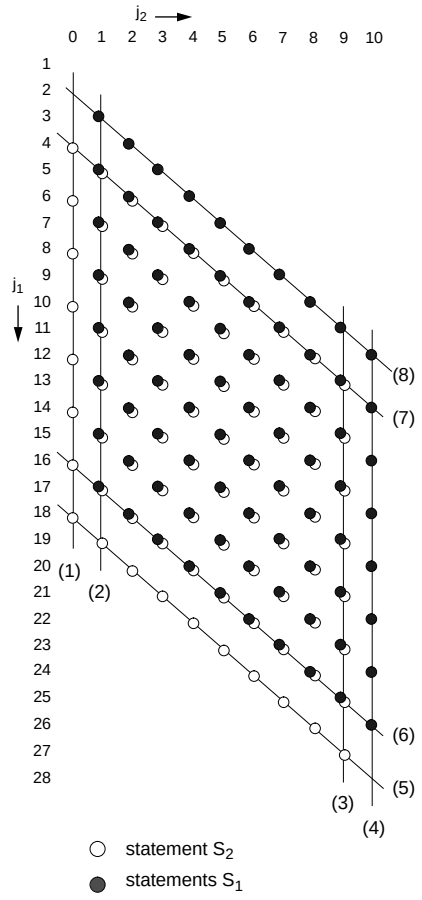
DO j_1 = 5, 25

DO $j_2 = j_1 + 2 \cdot \max(-8, \lceil (1-j_1)/2 \rceil), j_1 + 2 \cdot \min(-2, \lfloor (9-j_1)/2 \rfloor), 2$

Again we denote with $lower_i^{core}$ and $upper_i^{core}$ the lower and upper bounds of the core in each loop dimension L_i. Notice that this core part can be executed without guard conditionals at the statement level, reducing the run-time overhead introduced by them.

4.4 Code Generation

Let us consider again our working example. Figure 5 shows the skeleton of the target SIS and the expressions of the boundaries for each of the spaces considered.



Expressions of the lines that determine the bounds:

(1) : $j_1 + 2 \cdot \lceil -j_1/2 \rceil$

(2) : $j_1 + 2 \cdot \lceil (1-j_1)/2 \rceil$

(3) : $j_1 + 2 \cdot \lfloor (9-j_1)/2 \rfloor$

(4) : $j_1 + 2 \cdot \lfloor (10-j_1)/2 \rfloor$

(5) : $j_1 + 2 \cdot (-9)$

(6) : $j_1 + 2 \cdot (-8)$

(7) : $j_1 + 2 \cdot (-2)$

(8) : $j_1 + 2 \cdot (-1)$

Upper and lower bounds for each hyperplane, cover and core:

lower S_1: $j_1 \geq 3$, $j_2 \geq \max\{(2), (6)\}$

upper S_1: $j_1 \leq 26$, $j_2 \leq \min\{(4), (8)\}$

lower S_2: $j_1 \geq 4$, $j_2 \geq \max\{(1), (5)\}$

upper S_2: $j_1 \leq 27$, $j_2 \leq \min\{(3), (7)\}$

lower cover: $j_1 \geq 2$, $j_2 \geq \max\{(1), (5)\}$

upper cover: $j_1 \leq 28$, $j_2 \leq \min\{(4), (8)\}$

lower core: $j_1 \geq 5$, $j_2 \geq \max\{(2), (6)\}$

upper core: $j_1 \leq 25$, $j_2 \leq \min\{(3), (7)\}$

Figure 5: Basic skeleton of the SIS for our working example and expressions for the lines that determine each bound.

In general, the code has three parts, namely prolog, core and epilog parts. The prolog and epilog parts are defined as the parts of the cover not included in the core that are executed before or after the core part, respectively. We have to include guard conditionals in these two parts in order to control the execution of each statement.

For instance consider what happens when j_1=10 in Figure 5. In this case, first we have to execute the iteration j_2=0 of S_2 (prolog part), then the iteration j_2=2, 4 and 6 of both statements (core part) and finally the iteration j_2=8 of S_1 (epilog part). Not always these three parts are executed. For example, observe that for j_1=15 only points in the core part have to be executed, and that for j_1=4 no points have to be executed in the core part.

The structure of the code can be written using our notation as shown in Figure 6. In this code, and depending on the original loop bounds, transformation and alignment components, the compiler has a lot of opportunities to simplify the conditionals generated and make the code less run-time expensive.

```
                DO j1 = lower1^cover , upper1^cover , step1
-------------------------------------------------------------------------------
                    DO j2 = lower2^cover , lower2^core - step2, step2
                        ...
prolog part             IF (lower1^k ≤ j1 ≤upper1^k ) and (lower2^k ≤ j2≤upper2^k ) THEN {statement Sk}
                        ...
                    ENDDO

                    IF (lower1^core ≤ j1≤upper1^core)  THEN
-------------------------------------------------------------------------------
                        DO j2 = lower2^core , upper2^core , step2
                            ...
core part                   {statement Sk}
                            ...
                        ENDDO
-------------------------------------------------------------------------------
                        low = upper2^core + step2
                    ELSE
                        low = lower2^core
                    ENDIF

epilog part         DO j2 = low, upper2^cover , step2
                        ...
                        IF (lower1^k ≤ j1 ≤upper1^k ) and (lower2^k ≤ j2 ≤upper2^k ) THEN {statement Sk}
                        ...
                    ENDDO
-------------------------------------------------------------------------------
                ENDDO
```

Figure 6: Target code generated with prolog, core and epilog parts (for two-dimensional loops)

In particular, the code that is obtained for our working example is shown in Figure 7. For instance, the compiler can recognize that just one iteration is executed in the prolog and epilog parts due to the values of the alignment components. Therefore, loops in these parts can be substituted by conditional statements. The compiler can also recognize that statement S_1 is never executed in the prolog part and statement S_2 is never executed in the epilog part.

4.5 Dependence Relations in the Target Space

Let d_{ij} be a dependence relation between two different statements S_i and S_j (i.e., $S_i \delta S_j$). The distance vector $\overline{d}_{ij}$ is the number of iterations the dependence extends across in each loop dimension. So

$$\overline{d_{ij}} = I_2 - I_1$$

where I_1 and I_2 are two points in the original IS directly dependent because of the dependence d_{ij}.

```
DO j1 = 2, 28
    lcover =  j1 + 2 · max (-9,⌈(-j1) /2⌉)
    ucover = j1 + 2 · min (-1, ⌊(10-j1)/2⌋)
    lcore =  j1 + 2 · max (-8,⌈(1-j1) /2⌉)
    ucore = j1 + 2 · min (-2, ⌊(9-j1)/2⌋)
    IF lcover < lcore THEN B[ j1-1,  lcover+1] = g(A[ j1,  lcover+1], B[ j1-1,  lcover+1])
    DO j2 = lcore, ucore, 2
            A[ j1,  j2] = f(A[ j1,  j2], C[  j2+1, ( j1- j2)/2], B[ j1-1,  j2])
            B[ j1-1,  j2+1] = g(A[ j1,  j2+1], B[ j1-1,  j2+1])
    ENDDO
    IF ucore < ucover THEN A[ j1,  ucover] = f(A[ j1,  ucover], C[  ucover+1, ( j1- ucover)/2], B[ j1-1,  ucover])
ENDDO
```

Figure 7: Target loop nest obtained for the example in Figure 1.a when a different transformation is applied to each statement of the loop body.

If T is the transformation matrix applied,

$$T \cdot \left(I_1 + D^i \right) \qquad \text{and} \qquad T \cdot \left(I_2 + D^j \right)$$

are the corresponding points in the target SIS. Therefore, due to the fact that T is a linear transformation, the transformed distance is

$$T \cdot \left(\overline{d_{ij}} + D^j - D^i \right)$$

The transformation matrix and alignment components must be legal in the sense that they have to respect all data dependences d_{ij} in the loop.

5 Obtaining the Alignment Components

In this section we present some ideas about how to obtain the alignment component for each statement in the loop body. The aim of the process is to reduce interprocessor communication when parallelizing for a coarse-grain MIMD system. With the alignment components we try to minimize the number of non-local data references in the execution of the code assigned to each processor.

In the same way [11] constructs the Data Access Matrix from the subscripts in array references, we obtain the alignment components for each statement in the loop body. Constants in the set of subscripts that make up the Data Access Matrix and the alignment functions for the arrays subscripted by them contribute to find the alignment components. In fact, just those subscripts that correspond to the spatial part of the transformation matrix have to be analyzed.

Assume that we are analyzing the reference to a matrix M in a statement of the loop. Let σ be a vector with the independent coefficients of the subscripts in the spatial part of the transformation matrix used in the reference to M. For instance, consider the reference $B[i_1+2 \cdot i_2-1, i_1]$ in statement S_1 in the example shown in Figure 8.a. Assume the same transformation matrix T used along the paper in which the first row corresponds to the spatial part of the transformation. In this case, the subscript term is $i_1+2 \cdot i_2$ and the independent coefficient is $\sigma=[-1]$.

Let ϕ be a vector whose components are the offsets of the ALIGN function for array M in the dimensions where the correspondent subscripts are used. In the same example, the align function for matrix B is

ALIGN B(i, j) WITH E(i+1, j-1)

and therefore ϕ=[1] as indicated in the first component of the decomposition.

If we denote with T_s the spatial part of the transformation, the alignment component D for the reference analyzed can be obtained from

$$T_s \cdot D = \sigma + \phi$$

For the reference we are analyzing, we have to solve

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} D_1 \\ D_2 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix} \qquad \rightarrow \qquad d_1 + 2 \cdot d_2 = 0$$

Due to the fact that this equation has several integer solutions, we select the one that maximizes the size of the core part in the resulting code. In this case, this means that the minimum value for D_2 is chosen, so [0, 0] is the alignment vector obtained.

The alignment component D^i for a statement S_i is the alignment component that appears more times in all the array references in S_i.

Applying this procedure to all the references in the loop shown in Figure 8.a, we obtain $D^1=[0, 0]^t$ for statement S_1 and $D^2=[-1, 1]^t$ for statement S_2. The code obtained in this case is shown in Figure 8.b. Now references $B[j_1-1, j_2+1]$ and $A[j_1, j_2+1]$ in statement S_2 are local and references $B[j_1-2, j_2+1]$ and $C[j_2+2, (j_1-j_2-2)/2])$ are non-local, reducing the number of remote accesses.

If no alignment components had been added to the transformation, the code shown in Figure 8.c would have been obtained. Analyzing the data accesses performed, one can see that references $A[j_1, j_2]$ and $B[j_1-1, j_2]$ in the statement S_1 and reference $B[j_1-1, j_2]$ in statement S_2 are local. Other references are remote. However, observe that the access to $C[i_1+1, i_2]$ in statement S_2 does not require a remote access because it is reusing the same value that has been accessed in statement S_1. So in this case just accesses to $B[j_1, j_2]$ and $A[j_1+1, j_2]$ are non-local. In conclusion, the same number of remote accesses are performed.

In general it is necessary to consider the reuse of the array elements in order to find the alignment components.

6 Conclusions and Future Work

In this paper we have proposed the inclusion of loop alignment as a new component in the framework of non-singular transformations. This allows the compiler to match the structure of loop nests according to data alignment and distribution specifications. The aim is to reduce the number of remote data accesses in distributed-memory machines.

```
ALIGN A, C WITH E
ALIGN B (i, j) WITH E (i+1, j-1)
DISTRIBUTE E (CYCLIC, :)
DO i1=1, 10
    DO i2=1, 8
        A[i1+2·i2, i1] = f(A[i1+2·i2, i1], B[i1+2·i2-1, i1], C[i1+1, i2])
        B[i1+2·i2, i1] = g(A[i1+2·i2+1, i1], B[i1+2·i2-1, i1], C[i1+1, i2])
    ENDO
ENDO
```

(a)

```
DO j1=2, 28
    ...
    DO j2 = lcore,ucore, 2
        A[ j1,  j2] = f(A[ j1,  j2], B[ j1-1,  j2], C[  j2+1, ( j1- j2)/2] )
        B[ j1-1,  j2+1] = g(A[ j1,  j2+1], B[ j1-2,  j2+1], C[  j2+2, ( j1- j2-2)/2])
    ENDO
    ...
ENDO
```

(b)

```
DO j1=3, 26
    DO j2 = j1 + 2 · max (-8, ⌈(1-j1)/2⌉), j1 + 2 · min (-1, ⌊(10-j1)/2⌋), 2
        A[ j1,  j2] = f(A[ j1,  j2], B[ j1-1,  j2], C[  j2+1, ( j1- j2)/2])
        B[ j1,  j2] = g(A[ j1+1,  j2], B[ j1-1,  j2],C[  j2+1, ( j1- j2)/2] )
    ENDO
ENDO
```

(c)

Figure 8: (a) Original loop nest. Two different options for the alignment components: (b) $D^1=[0, 0]^t$ and $D^2=[-1, 1]^t$ and (c) $D^1=D^2=[0, 0]^t$.

We have used some ideas recently published in the literature to generate code that controls the correct execution of each statement in each iteration of the transformed loop nest and body. In this case it is more difficult to generate code and we have shown how to reduce the overhead due to conditionals that appear in the loop body [24].

We have presented a simple method to obtain the alignment component for each statement of the loop body. It is based on the analysis of constants in the set of subscripts that make up the data access matrix (in the spatial part of the transformation) and the alignment functions for the arrays subscripted by them. Data reuse has not been considered in the method we have proposed. A more precise formulation of the problem of finding the alignment components where data reuse is considered is part of our future work.

A possible extension to the work presented in this paper is the use of a different transformation for each statement. From array references in each loop statement, a different data access matrix or transformation matrix T and alignment component can be obtained to increase locality. The target loop nest obtained with this approach has the same structure as the one we have presented in this paper. In general, the transformed dependence distances become non-uniform and as a consequence, it is more difficult to generate the associated synchronization/communication instructions. The study of the legality of all the transformation matrices is more complex because of this non-uniform characteristic of dependences in the target space.

Acknowledgments

This work has been supported by the Ministry of Education of Spain under contract TIC-880/92, by the ESPRIT Basic Research Action 6634 APPARC and by the CEPBA (European Center for Parallelism of Barcelona).

References

[1] Polychronopoulos C., *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[2] Wolfe M., *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1989.

[3] Fox G. et al., Fortran-D Language Specification, Technical Report TR90-140, Dept. of Computer Science, Rice University, revised January 1992.

[4] David Loveman (ed.), Draft High Performance Fortran Language Specification Version 1.0, Technical Report TR92-225, CRPC, Rice University, January 1993.

[5] Callahan D. and Kennedy K., Compiling Programs for Distributed-Memory Multiprocessors, *Journal of Supercomputing*, vol. 2, no. 2, October 1988.

[6] Hiranandani S., Kennedy K. and Tseng C., Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, in *Proceedings of the 1992 ACM International Conference on Supercomputing*, July 1992.

[7] Karp A.H., Programming for Parallelism, *Computer*, vol. 20, no. 5, May 1987.

[8] Banerjee U., Unimodular Transformations of Double Loops, chapter 10 of *Advances in Languages and Compilers for Parallel Processing*, The MIT Press, 1991.

[9] Wolf M.E. and Lam M.S., A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, October 1991.

[10] Wolf M.E. and Lam M., A Data Locality Optimizing Algorithm, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[11] Li W. and Pingali K., Access Normalization: Loop Restructuring for NUMA Compilers, in *Proceedings of the Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[12] Li W. and Pingali K., A Singular Loop Transformation Framework Based on Non-Singular Matrices, in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computers*, August 1992.

[13] Fernández A., Systematic Transformation of Systolic Algorithms for Programming Distributed Memory Multiprocessors, Ph.D. Thesis, Department of Computer Architecture, Polytechnic University of Catalunya (Spain), November 1992.

[14] Ramanujam J., Non-unimodular Transformations of Nested Loops, in *Proceedings of the Supercomputing'92*, November 1992.

[15] Padua D.A., Multiprocessors: Discussions of some theoretical and practical problems, Technical Report DCS UIUCDCS-R-79-990, Ph.D. dissertation, University of Illinois at Urbana-Champaign, November 1979.

[16] Peir J-K., Program Partitioning and Synchronization on Multiprocessor Systems, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1986.

[17] Allen R., Callahan D. and Kennedy K., Automatic Decomposition of Scientific Programs for Parallel Execution, in *Proceedings of the 14th ACM Symposium Principles of Programming Languages*, January 1987.

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

[19] Kennedy K. and Kremer U., Automatic Data Alignment and Distribution for Loosely Synchronous Problems in an Interactive Programming Environment, Technical Report TR91-155, Dept. of Computer Science, Rice University, April 1991.

[20] Li J., Compiling Crystal for Distributed-Memory Machines, Ph.D. Thesis, Dep. of Computer Science, Yale University, December 1991.

[21] Moldovan D.I. and Fortes J.A.B., Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays, *IEEE Transactions on Computers*, vol. 35, no.1, January 1986.

[22] Lu L. and Chen M., New Loop Transformation Techniques for Massive Parallelism, Research Report TR-833, Department of Computer Science, Yale University, October 1990.

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

[24] Ayguadé E. and Torres J., Partitioning the Statement per Iteration Space Using Non-singular Matrices, in *Proceedings of the 1993 ACM International Conference on Supercomputing*, July 1993.