Transformations Techniques for extracting Parallelism in Non-Uniform Nested Loops

FAWZY A. TORKEY¹, AFAF A. SALAH², NAHED M. EL DESOUKY² and SAHAR A. GOMAA²

¹)Kaferelsheikh University, Kaferelsheikh, EGYPT
²)Mathematics Department, Faculty of Science, Al Azhar University, Nasr City, EGYPT
nahedmgd@yahoo.com

Abstract: - Executing a program in parallel machines needs not only to find sufficient parallelism in a program, but it is also important that we minimize the synchronization and communication overheads in the parallelized program. This yields to improve the performance by reducing the time needed for executing the program. Parallelizing and partitioning of nested loops requires efficient iteration dependence analysis. Although many loop transformations techniques exist for nested loop partitioning, most of these transformation techniques perform poorly when parallelizing nested loops with non-uniform (irregular) dependences. In this paper the affine and unimodular transformations are applied to solve the problem of parallelism in nested loops with non-uniform dependence vectors. To solve these problem few researchers converted the non-uniform nested loops to uniform nested loops and then find the parallelism. We propose applying directly the two approaches affine and unimodular transformations to extract and improve the parallelism in nested loops with non-uniform dependences. The study shows that unimodular transformation is better than affine transformation when the dependences in nested loops exist only in one statement. While affine transformation is more effective when the nested loops have a sequence of statements and the dependence exists between these different statements.

Keywords: - Unimodular transformation, Affine transformation, Parallelism, Uniform dependence, Non-uniform dependence, Distance vector, Distance matrix.

1 Introduction
To get high performance on a multiprocessor, it is important to find parallelism that does not incur high synchronization cost between the processors. Loop transformations have been shown to be useful for extracting parallelism from nested loops for a large class of machines.

A loop transformation of nested loops is a mechanism that changes the (sequential) execution order of the statement instances without changing the set of statement instances. Before the loop transformation can be performed, the data dependence between the individual statements must be determined. There are several well-known data dependence algorithms that have been developed to test data dependence, for example [16], [20], [21], [24], [28], [29], [31], [33-35] and [36].

Loops with data dependences can be divided into two groups; loops with regular (uniform) dependences and loops with irregular (non-uniform) dependences. The dependences are uniform only when the dependence vectors are uniform. In other words, the dependence vectors can be expressed by constants, distance vectors. Figure 1 shows an example of nested loop with uniform dependences. Dependence vectors in Figure 1 are (0,1) and (1,0) for all iterations in the given loops. In the same fashion, we call some dependences non-uniform when dependence vectors are in irregular patterns which cannot be expressed by constants, distance vectors. Figure 2 shows an example of nested loop with non-uniform dependences.

![Figure 1 An example of nested loop with uniform dependences.](image-url)
Many loop transformations have been developed to improve parallelism and data locality in nested loops with uniform dependences [7-9], [14], [18], [23], [25]. Examples that include unimodular loops with uniform dependences [7-9], [14], [18], [23], [25]. But few techniques have been developed for nested loops with non-uniform dependences [2-5], [22], [23], [25].

Definitions and algorithms about the affine transformation are introduced. The study shows that the unimodular transformation and the affine transformation can be applied to programs with perfect (sequence) nested loops. The dependences in the loop are abstracted by a set of distance and direction vectors. By applying a unimodular transformation to the original iteration space to produce a new iteration space, a sequential loop is transformed into another loop. A transformation is legal as long as the dependence vectors in the original iteration space remains lexicographically positive in the transformed space.

Consider a model program is a perfect nested loop $L$ of $m$ FOR loops:

$$L_i: \text{FOR } I_1 = p_i, q_i \rightarrow I_2 = p_2(I_1), q_2(I_1) \rightarrow \cdots \rightarrow L_m: \text{FOR } I_m = p_m(I_1, I_2, \ldots, I_{m-1}), q_m(I_1, I_2, \ldots, I_{m-1})$$

It is a perfect nested loop, which has no statements between loops. The perfect nested loop $L$ may be expressed by the formula $L = (L_1, L_2, \ldots, L_m)$, where $m$ is the number of FOR loops inside the nested loop $L$. When $m$ has the value 1, 2, or 3, the perfect nested loop is a single, double, or triple loop, respectively. The index variables of the individual loops are $I_1, I_2, \ldots, I_m$ and they form the index vector $I = (I_1, I_2, \ldots, I_m)$ of the nested loop $L$. The lower and upper limits of the FOR loop $L_i$ are the integers $p_i$ and $q_i$, $1 \leq r \leq m$, respectively. The stride of each index variable is 1.

The set of all index variables is called the index space of $L$; it can be expressed as a set of integer $m$-vectors $I = (I_1, I_2, \ldots, I_m)$ such that

$$p_1 \leq I_1 \leq q_1 , p_2(I_1) \leq I_2 \leq q_2(I_1) $$

$$\vdots$$

$$p_m(I_1, I_2, \ldots, I_{m-1}) \leq I_m \leq q_m(I_1, I_2, \ldots, I_{m-1})$$

Based on the approach of translation of sequential nested loop into semantically equivalent sequential nested loop. The translation of the nested loop into an equivalent another form which can be parallelized needs to loop transformation. Many loop transformations have been developed to improve parallelism in programs, for instance a unimodular transformation.

A unimodular transformation is a loop transformation defined by unimodular matrix (A square integer matrix, which may be unimodular if its determinant is equal to $\pm 1$). The unimodular transformation can be applied to programs with perfect (sequence) nested loops. The dependences in the loop are abstracted by a set of distance and direction vectors. By applying a unimodular transformation to the original iteration space to produce a new iteration space, a sequential loop is transformed into another loop. A transformation is legal as long as the dependence vectors in the original iteration space remains lexicographically positive in the transformed space.

2 Unimodular Transformation

The Whole idea of transforming the nested loop $L$ is to find a suitable new execution order for its iterations such that independent iterations can be executed in parallel. After determining all dependence information in a given nested loop, the independent iterations can be executed in parallel. So, a given nested loop needs to be rewritten in a form that allows parallel execution of independent iterations. The loop parallelization algorithms are

$$\text{For } I_1 = 1, 5$$

$$\text{For } I_2 = 1, 5$$

$$S_1: A[I_1+I_2, 3I_1+I_2+3] = A[I_1+I_2+1, I_1+2I_2+4]$$

Figure 2 An example of nested loop with non-uniform dependences.
where \( p_i(I_1, I_2, \ldots, I_{m}) \) and \( q_i(I_1, I_2, \ldots, I_{m}) \), \( 1 \leq r \leq m \) are the lower and upper limits of \( L_i \), respectively.

Consider a given perfect nested loop \( L = (L_1, L_2, \ldots, L_m) \) with a distance matrix \( D \) (each row is a dependence vector in nested loop \( L \)), we want to find a valid unimodular transformation \( L \rightarrow L_U \) such that one or more innermost loops of \( L_U \) can be executed in parallel. This means that finding an \( m \times m \) unimodular matrix \( U \) such that all elements of the first column in \( DU \) are positive. If this is satisfied, then the loops \( L_{12}, L_{13}, \ldots, L_{um} \) of the transformed nested loop can be executed in parallel. Note that, the notation \( x \rightarrow y \) is a function that maps an element \( x \) of its domain to an element \( y \) of its range.

The algorithm, which uses to find the matrix \( U \), is described in [30]. Also it presents the theorems of necessary and sufficient conditions, which make a loop \( L_U \) executed in parallel.

After determining the form of a unimodular transformation by using the distance matrix of a given nested loop, the limits of index variables must be determined. Let us consider the transformation of \( L \) into the nested loop \( L_U \) with an \( m \times m \) unimodular matrix \( U \), the index vectors \( I = (I_1, I_2, \ldots, I_m) \) of \( L \) and \( K = (K_1, K_2, \ldots, K_m) \) of \( L_U \). The index vectors \( I \) and \( K \) are connected by the equation

\[
K = IU,
\]

so that

\[
I = KU^T.
\]

To find the limits of new loops \( K_1, K_2, \ldots, K_m \) the Fourier’s elimination method is used [29]. The new limits of loops are described by the set of inequalities:

\[
\begin{align*}
\alpha_1 &\leq K_1 \leq \beta_1 \\
\alpha_2(K_1) &\leq K_2 \leq \beta_2(K_1) \\
& \vdots \\
\alpha_m(K_1, K_2, \ldots, K_{m-1}) &\leq K_m \leq \beta_m(K_1, K_2, \ldots, K_{m-1})
\end{align*}
\]

3 Affine Transformation

The basic concept of the affine transformation is to separate the program into as many independent partitions as possible and then these independent partitions can be executed in parallel. The affine transformation can be applied to programs with perfect and arbitrary nested loops. The dependences in the program are abstracted by a set of affine functions, which contains all pairs of data dependent access functions in this program.

Consider a program \( P = < S, \delta, \eta, D_0, F, \omega > \), where

- \( S \) is an ordered list of statements.
- \( \delta \) is the depth of statement \( s \), defined to be the number of loops surrounding statement \( s \).
- \( \eta_{s,i} \) is the number of loops surrounding both statements \( s \) and \( s' \).
- \( D_0(i) \geq 0 \) is a system of linear constraints that describes the valid loop iterations for statement \( s \).
- \( F_{sr}(i) \) is the affine array access function for the \( r \) th array reference to array \( z \) in statement \( s \).
- \( \omega_{sr} \) is a Boolean value describing whether the \( r \) th array reference to array \( z \) in statement \( s \) is a write operation.

The data dependence set of a program \( P = < S, \delta, \eta, D_0, F, \omega > \) is

\[
\Re = \left\{ < F_{sr}, F_{sr}' > \mid \left( \omega_{sr} \land \omega_{sr'} \right) \land \left( \exists i \in Z^{\delta_s}, \tilde{i} \in Z^{\delta_{s'}} \mid i \prec_{ss'} \tilde{i} \right) \land \left( F_{sr}(i) - F_{sr'}(\tilde{i}) = 0 \right) \land \left( D_0(i) \geq 0 \land D_0(\tilde{i}) \geq 0 \right) \right\}
\]

where \( \prec \) is the “lexicographically less than” operator such that \( i \prec_{ss'} \tilde{i} \) if and only if iteration \( i \) of statement \( s \) is executed before iteration \( \tilde{i} \) of statement \( s' \) in the program \( P \).

The space partition constraint, as defined below, captures the necessary and sufficient constraint for an affine mapping to divide the computation into independent partitions.

**Definition 1** (space partition constraints) Let \( \Re \) be the data dependence set for program \( P = < S, \delta, \eta, D_0, F, \omega > \). An affine space partition mapping \( \Phi \) for \( P \) is independent if and only if

\[
\forall < F_{sr}, F_{sr'} > \in \Re, i \in Z^{\delta_s}, \tilde{i} \in Z^{\delta_{s'}} \text{ s.t.}
\]

\[
D_s(i) \geq 0 \land D_{s'}(\tilde{i}) \geq 0 \land F_{sr}(i) - F_{sr'}(\tilde{i}) = 0, (2) \quad \Phi_{s'}(\tilde{i}) - \Phi_s(i) = 0.
\]

This constraint simply states that data dependent instances must be placed in the same partition. That is if iteration \( i \) of statement \( s \) and iteration \( \tilde{i} \) of statement \( s' \) access the same data, then their mappings \( \Phi_s \) and \( \Phi_{s'} \) are constrained to assign these iterations to the same space partition, i.e. \( \Phi_{s'}(\tilde{i}) - \Phi_s(i) = 0 \).
Let $D_s(\tilde{i}) = D_s \tilde{i} + \bar{d}_s$, $F_{\tilde{s}\tilde{r}}(\tilde{i}) = F_{\tilde{s}\tilde{r}} \tilde{i} + \bar{f}_{\tilde{s}\tilde{r}}$ and $\Phi_s(\tilde{i}) = C_s \tilde{i} + c_s$. Each data dependence, $<F_{\tilde{s}\tilde{r}}F_{\tilde{s}'\tilde{r}'}> \in \mathbb{R}$, imposes the following constraints on each row of an independent mapping for statements $s$ and $s'$, $X = [-C_s, C_s]$: 

\[
\forall \bar{y} \text{ s.t. } G \bar{y} \geq 0 \land H \bar{y} = A \bar{y} = 0 \quad (3)
\]

where 

\[
\bar{y} = [\tilde{i}, \tilde{r}], G = \begin{bmatrix} D_s & 0 & \bar{d}_s \\ 0 & D_{\tilde{s}} & \bar{d}_{\tilde{s}} \end{bmatrix}, \\
H = \begin{bmatrix} F_{\tilde{s}\tilde{r}} & F_{\tilde{s}'\tilde{r}'} & (\bar{f}_{\tilde{s}\tilde{r}} - \bar{f}_{\tilde{s}'\tilde{r}'}) \end{bmatrix}
\]

In the space partition constraints, the coefficients of the affine mappings ($X$) and the loop index values correspond to the data dependent instances ($\bar{y}$) are both unknowns. Since $X$ and $\bar{y}$ are coupled in (3), the constraints are non-linear.

We can find the affine partition that maximizes the degree of independent partitions in a program by simply finding the maximal set of independent solutions to the corresponding space partition constraint. The algorithms which solve the constraints and find a partition mapping have been described in [4] and [5]. The algorithm of solving the constraints used the affine form of the Farkas lemma [6], [17] to transform the non-linear constraints into system linear inequalities. The algorithm of the affine partition mapping based on Fourier–Motzkin elimination [1] that can be used to generate the desired SPMD (Single Program Multiple Data) code.

4 Case Study

The following case study is introduced to show how the affine transformation and the unimodular transformation can be used to extract the parallelism from nested loops which have non-uniform dependences. If we have the following code

$L_1$: For $I_1 = 1$ to $N_1$
$L_2$: For $I_2 = 1$ to $N_2$
$S_1[I_1, I_2]:$ $A[I_1+I_2, 3I_1+I_2+3] = B[I_1, I_2]$
$S_2[I_1, I_2]:$ $C[I_1, I_2] = A[I_1+I_2+1, I_1+2I_2+4]$

Figure 3 shows the iteration space and hierarchical program structure for the above code. The figure shows a subset of the dynamic instances, 4 iterations in the $I_1$ loop and 6 iterations in the $I_2$ loop. Data dependences between the instances are shown as arrows in the iteration space representation.

Notice that, the number of iterations for the outer loop $I_1 = N_1 - 1 + 1 = N_1$, and for the inner loop $I_2 = N_2 - 1 + 1 = N_2$. So, the number of all iterations is $N_1 N_2$, and the number of all instances is $2 N_1 N_2$, where 2 is the number of statements.

4.1 Parallelism with Affine Transformation

We will show how the affine transformation can be used to find the independent partitions. The above code is represented as $P = <S, \delta, \eta, D_s, F, \omega>$, where
- $S = \{s_1, s_2\}$, is an ordered list containing the two statements in the code.
- $\delta_1 = 2$ is the number of loops surrounding statement $s_1$.
- $\delta_2 = 2$ is the number of loops surrounding statement $s_2$.
- $\eta_1, 2 = 2$ is the number of loops surrounding both statements $s_1$ and $s_2$.
- $D_s(\tilde{i}) \geq 0$ is a system of linear constraints that describes the valid loop iterations for statement $s$. So from the loop bounds of the loops surrounding
\[ D_1(I_1, I_2) = D_2(I_1, I_2) = \begin{bmatrix} 1 & 0 & -1 & 0 & I_1 & N_1 \\ 0 & 1 & 0 & -1 & I_2 & N_2 \end{bmatrix} \]

- \( F_{A_r}(i) \) is the affine array access function for the \( r \) th array reference to array \( z \) in statement \( s \). There are four \( F \) and four \( \omega \) in the representation. They correspond to the four access functions in the code, one write and one read in each statement.

\[ F_{AI}(I_1, I_2) = \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} I_1 + \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \]

\[ F_{BI}(I_1, I_2) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} I_2, \]

\[ F_{CI}(I_1, I_2) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} I_1 + \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \]

\[ F_{AI}(I_1, I_2) = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} I_1 + \begin{bmatrix} 1 \\ 4 \end{bmatrix}. \]

- \( \omega_{r_s} \) is a Boolean value describing whether the \( r \) th array reference to array \( z \) in statement \( s \) is a write operation.

\[ \omega_{1AI} = \text{true}, \omega_{2BI} = \text{false}, \omega_{3CI} = \text{true}, \omega_{2AI} = \text{false}. \]

The data dependence set for the above code is the 1-element set \( \{F_{AI}, F_{AI}\} \). From this set the dependences are between the different statements \( S_1 \) and \( S_2 \), but not at the same statement. Figure 3 shows that the dependence vectors are non-uniform.

After using the algorithm in [4], we get the space partition mapping \( \Phi = [F_1, F_2] \) with \( F_1 = I_1 + I_2 \) and \( F_2 = I_1 + I_2 + 1 \). The SPMD code for the previous code after the space partition mapping \( \Phi \) will be:

For \( I_1 = 1 \) to \( N_1 \)

For \( I_2 = 1 \) to \( N_2 \)

If (\( \Phi = I_1 + I_2 \)) then

\[ S_1[I_1, I_2]: A[I_1+I_2, 3I_1+I_2+3] = B[I_1, I_2] \]

If (\( \Phi = I_1 + I_2 + 1 \)) then

\[ S_2[I_1, I_2]: C[I_1, I_2] = A[I_1+I_2+1, 1+2I_2+4] \]

Now the new loop bounds are computed for the two statements \( S_1 \) and \( S_2 \) separately. The Fourier’s elimination method [29] is used to compute these new loop bounds. The new loop bounds for statement \( S_1 \) will be as follows:

\[ \max(1, \Phi - N_2) \leq I_1 \leq \min(N_1, \Phi - 1) \quad (1) \]

\[ 2 \leq \Phi \leq N_1 + N_2 \quad (2) \]

The new loop bounds for statement \( S_2 \) is as follows:

\[ \max(1, \Phi - N_2 - 1) \leq I_1 \leq \min(N_1, \Phi - 2) \quad (3) \]

\[ 3 \leq \Phi \leq N_1 + N_2 + 1 \quad (4) \]

From (2) and (4), the range of \( \Phi \) will be as follows:

\[ 2 \leq \Phi \leq N_1 + N_2 + 1 \]

The number of partitions \( \Phi = N_1 + N_2 + 1 \) can be executed at the same time and no synchronization is needed between the processors. While the instances of statements in the same partition need to be executed dependently. The time diagram with the affine transformation will be discussed, assuming that each statement will take 1 time clock (1 time unit).

Figure 4 give the time diagram. It shows that, if we have number of processors \( = N_1 \) (the number of

\[ \begin{array}{c|c|c|c|c|c} \Phi & S_1(1,1) & S_2(1,1) & S_1(1,2) & S_2(1,2) & S_1(1,3) \\
\hline 2 & S_2(1,3) & S_1(1,4) & S_2(1,4) & S_1(2,2) & S_2(2,2) \\
\hline 3 & S_1(2,3) & S_2(2,3) & S_1(3,1) & S_2(3,1) & S_1(3,2) \\
\hline 4 & S_2(3,2) & S_1(3,3) & S_2(3,3) & S_1(4,1) & S_2(4,1) \\
\hline 5 & S_1(4,2) & S_2(4,2) & & & \\
\end{array} \]

The table shows that different partitions \( \Phi = 2 \) to \( \Phi = 11 \) can be executed at the same time and no synchronization is needed between the processors. While the instances of statements in the same partition need to be executed dependently. The time diagram with the affine transformation will be discussed, assuming that each statement will take 1 time clock (1 time unit).

Figure 4 give the time diagram. It shows that, if we have number of processors = \( N_1 \) (the number of
iterations in the outer loop), then each processor will execute number of instances = number of statements*\(N_2\)(the number of iterations in the inner loop), where in this case \(N_1=4\) and \(N_2=6\).

\[
\begin{array}{c}
\text{PE 1} \\
| \Phi =2 | \Phi =6 | \Phi =10 |
\end{array}
\begin{array}{c}
\text{PE 2} \\
| \Phi =3 | \Phi =7 | \Phi =11 |
\end{array}
\begin{array}{c}
\text{PE 3} \\
| \Phi =4 | \Phi =8 |
\end{array}
\begin{array}{c}
\text{PE 4} \\
\Phi =5 \rightarrow \Phi =9 \rightarrow \text{Time units = 12 units}
\end{array}
\]

Figure 4 Time diagram with affine transformation

Notice that all processors will start at the same time because the partitions are independent and can be executed in parallel without need of synchronization. The total time units equal the number of statements * number of iterations in inner loop.

\[D = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix}\]

The outer and inner loops still cannot be executed in parallel. This is because of the dependence still at level 1 and level 2. We will use the following unimodular matrix

\[U = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}\]

to transform loop \(L\) into \(L_U\) whose inner loop can be executed in parallel.

So, \(K = IU \Rightarrow (K_1, K_2) = (I_1, I_2) \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow (K_1, K_2) = (3I_1 + I_2, I_1),\)

and \(I = KU^{-1} \Rightarrow (I_1, I_2) = (K_1, K_2) \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix} \Rightarrow (I_1, I_2) = (K_2, K_1 - 3K_2)\).

The constraints on \(I_1\) and \(I_2\) give the inequalities:

\[1 \leq K_2 \leq N_1\]
\[1 \leq K_1 - 3K_2 \leq N_2\]

By using Fourier’s elimination [29] we find

\[4 \leq K_1 \leq 3N_1+N_2\]

\[\max \left\{ 1, \left\lfloor \frac{K_1-N_2}{3} \right\rfloor \right\} \leq K_2 \leq \min \left\{ N_1, \left\lfloor \frac{K_1-1}{3} \right\rfloor \right\} .\]

The loop \(L\) becomes after transformation in the form \(L_U\) as follows, where in this case \(N_1=4\) and \(N_2=6\)

\[L_{U1}: \text{ For } K_1 = 4, 18\]

\[L_{U2}: \text{ For } K_2 = \max \left\{ 1, \left\lfloor \frac{K_1-6}{3} \right\rfloor \right\} , \min \left\{ 4, \left\lfloor \frac{K_1-1}{3} \right\rfloor \right\} \]

\[S_1(K_1, K_2): A(K_1, 2K_2, K_1+3) = B(K_2, K_1 + 3K_2) \]

\[S_2(K_1, K_2): C(K_2, K_1 + 3K_2) = A(K_1, 2K_2+1, 2K_1-5K_2+4)\]

After transformation the distance matrix has the form

\[DU = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} .\]
After transformation the dependence exists at level 1 but not at level 2. Thus, the inner loop can be executed in parallel and inner loop parallelism is performed. Table 2 shows the data array allocation to the four processors after parallelism using the unimodular transformation. The table shows that the unimodular transformation requires synchronization between the processors. The time diagram with the unimodular transformation is discussed, assuming that each statement will take 1 time clock (1 time unit).

From the table we can notice that processor 2 will remain idle till \( k_2 = 2 \) (when \( \left\lfloor 4 \left( \frac{K_1 - 1}{3} \right) \right\rfloor = 2 \) i.e \( K_1 = 7 \)). Also processor 3 will remain idle till \( k_2 = 3 \) (when \( \left\lfloor 4 \left( \frac{K_1 - 1}{3} \right) \right\rfloor = 3 \) i.e \( K_1 = 10 \)) and processor 4 will remain idle till \( k_2 = 4 \) (when \( \left\lfloor 4 \left( \frac{K_1 - 1}{3} \right) \right\rfloor = 4 \) i.e \( K_1 = 13 \)).

Table 2 Allocated data to 4 processors with unimodular transformation

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1(4,1) ): A(2,7)=B(1,1)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(4,1) ): C(1,1)=A(3,7)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(5,1) ): A(3,8)=B(1,2)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(5,1) ): C(1,2)=A(4,9)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(6,1) ): A(4,9)=B(1,3)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(6,1) ): C(1,3)=A(5,11)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(7,1) ): A(5,10)=B(1,4)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(7,1) ): C(1,4)=A(6,13)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(8,1) ): A(6,11)=B(1,5)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(8,1) ): C(1,5)=A(7,15)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(9,1) ): A(7,12)=B(1,6)</td>
<td>Idle</td>
</tr>
<tr>
<td>( S_1(9,1) ): C(1,6)=A(8,17)</td>
<td>Idle</td>
</tr>
</tbody>
</table>

Figure 5 give the time diagram. If we have number of processors = \( N_1 \) (the number of iterations in the outer loop), then each processor will execute number of instances = number of statements * \( N_2 \) (the number of iterations in inner loop), where in this case \( N_1 = 4 \) and \( N_2 = 6 \). So the time units equal to the number of iterations at the outer loop after transformation * number of statements which usually greater than the number of statements * number of iterations in the inner loop before transformation.
can be applied.

- Unimodular transformation can be applied when synchronization is needed between processors.

instances in different statements the affine transformation

- In nested loops with non-uniform dependences, the results show that

- In the affine transformation we find a valid unimodular matrix, such that inner loop can be executed in parallel, it requires synchronization between processors. While in the affine transformation we divide the instances of statements into partitions, such that dependent instances are placed in the same partition, by assigning each partition to a different processor, no synchronization is needed between processors.

- Unimodular transformation can be applied when dependence exists between the instances at the same statement, when the dependence exists between the instances in different statements the affine transformation can be applied.

5 Conclusion

In this paper we apply the unimodular transformation and the affine transformation to improve the parallelism in nested loops with non-uniform dependences, the results show that

- In nested loops with non-uniform dependences, the uniformization technique must be used to get nested loops with uniform dependences, so the unimodular transformation can be used. But the affine transformation can be used directly.

- In the unimodular transformation we find a valid unimodular matrix, such that inner loop can be executed in parallel, it requires synchronization between processors. While in the affine transformation we divide the instances of statements into partitions, such that dependent instances are placed in the same partition, by assigning each partition to a different processor, no synchronization is needed between processors.

- Unimodular transformation can be applied when dependence exists between the instances at the same statement, when the dependence exists between the instances in different statements the affine transformation can be applied.

References:

[10] D. K. Chen and P. C. Yew, On Effective Execution of Non-Uniform DOACROSS Loops, IEEE Transactions on Parallel and


