Transformations Techniques for extracting Parallelism in Non-Uniform Nested Loops

FAWZY A. TORKEY[1], AFAF A. SALAH[2], NAHED M. EL DESOUKY[2] and SAHAR A. GOMAA[2]
[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.

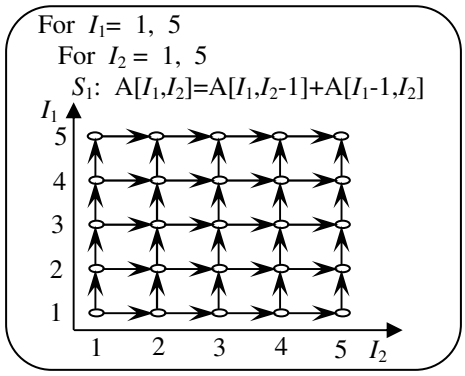

Figure 1 An example of nested loop with uniform dependences.

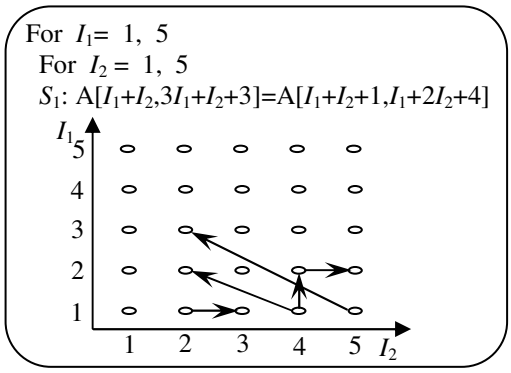

Figure 2 An example of nested loop with non- uniform dependences

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 transformations can be found at [19], [13], [29], [30], [12], [15] and affine partitioning at [2-5], [22], [32]. But few techniques have been developed for nested loops with non-uniform dependences [10], [11], [26], [27].

This paper presents two techniques for loop transformations, which can be used to extract and improve parallelism in nested loops with non-uniform dependences. These two techniques are the unimodular transformation and the affine transformation. The study shows that the unimodular transformation requires synchronization between the processors, while in the affine transformation no synchronization is needed between the processors.

The reset of this paper is organized as follows. In section 2 the legality and properties of the unimodular transformation are introduced. Definitions and algorithms about the affine transformation are described in section 3. A case study for double nested loop with non-uniform dependence is explained in section 4. Finally, conclusion is given in section 5.

2 Unimodular Transformation

The Whole idea of transforming the nested loop $\boldsymbol{L}$ 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 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 ±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.

Consider a model program is a perfect nested loop $\boldsymbol{L}$ of m FOR loops:

$$
\begin{aligned}
&L_1\text{: FOR } I_1 = p_1, q_1\\
&L_2\text{: FOR } I_2 = p_2(I_1),\ q_2(I_1)\\
&\qquad\vdots \qquad\qquad \vdots\\
&L_m\text{: FOR } I_m = p_m(I_1, I_2, \ldots, I_{m-1}),\ q_m(I_1, I_2, \ldots, I_{m-1})\\
&\qquad H(I_1, I_2, \cdots, I_m)
\end{aligned}
$$

It is a perfect nested loop, which has no statements between loops. The perfect nested loop $\boldsymbol{L}$ may be expressed by the formula $\boldsymbol{L}$=(L_1, L_2,…, L_m), where m is the number of FOR loops inside the nested loop $\boldsymbol{L}$. 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,…, I_m, and they form the *index vector* $\boldsymbol{I}$ = (I_1, I_2,…, I_m) of the nested loop $\boldsymbol{L}$. The *lower* and *upper limits* of the FOR loop L_r are the integers p_r and q_r, $1 \le r \le m$, respectively. The stride of each index variable is 1.

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

$$
\left.
\begin{aligned}
p_1 \le I_1 \le q_1\\
p_2(I_1) \le I_2 \le q_2(I_1)\\
\vdots\\
p_m(I_1, I_2, \ldots, I_{m-1}) \le I_m \le q_m(I_1, I_2, \ldots, I_{m-1}).
\end{aligned}
\right\},
$$

where $p_r(I_1,I_2,\ldots, I_{r-1})$ and $q_r(I_1,I_2,\ldots, I_{r-1})$, $1 \le r \le m$ are the lower and upper limits of $\boldsymbol{L}$, respectively.

Consider a given perfect nested loop $\boldsymbol{L}=(L_1, L_2,\ldots, L_m)$ with a distance matrix $\boldsymbol{D}$ (each row is a dependence vector in nested loop $\boldsymbol{L}$), we want to find a valid unimodular transformation $\boldsymbol{L} \mapsto \boldsymbol{L}_U$ such that one or more innermost loops of $\boldsymbol{L}_U$ can be executed in parallel. This means that finding an $m \times m$ unimodular matrix $\boldsymbol{U}$ such that all elements of the first column in $\boldsymbol{DU}$ are positive. If this satisfied, then the loops $L_{U2}, L_{U3}, \ldots, L_{Um}$ of the transformed nested loop can be executed in parallel. Note that, the notation $x \mapsto y$ 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 $\boldsymbol{U}$, is described in [30]. Also it presents the theorems of necessary and sufficient conditions, which make a loop $\boldsymbol{L}_U$ 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 $\boldsymbol{L}$ into the nested loop $\boldsymbol{L}_U$ with an $m \times m$ unimodular matrix $\boldsymbol{U}$, the index vectors $\boldsymbol{I}=(I_1, I_2,\ldots, I_m)$ of $\boldsymbol{L}$ and $\boldsymbol{K}=(K_1, K_2,\ldots, K_m)$ of $\boldsymbol{L}_U$. The index vectors $\boldsymbol{I}$ and $\boldsymbol{K}$ are connected by the equation

$$\boldsymbol{K} = \boldsymbol{IU},$$

so that

$$\boldsymbol{I} = \boldsymbol{KU}^{-1}.$$

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:

$$\left.\begin{array}{c} \alpha_1 \le K_1 \le \beta_1 \\ \alpha_2(K_1) \le K_2 \le \beta_2(K_1) \\ \vdots \\ \alpha_m(K_1,K_2,\ldots,K_{m-1}) \le K_m \le \beta_m(K_1,K_2,\ldots,K_{m-1}). \end{array}\right\}$$

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 $\boldsymbol{P} = < S,\delta,\eta,D_s,F,\omega >$, where

- S is an ordered list of statements.
- δ_s is the depth of statement s, defined to be the number of loops surrounding statement s.
- $\eta_{s,s'}$ is the number of loops surrounding both statements s and s'
- $D_s(\vec{i}) \ge \vec{0}$ is a system of linear constraints that describes the valid loop iterations for statement s
- $F_{srz}(\vec{i})$ is the affine array access function for the r th array reference to array z in statement s.
- ω_{srz} 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 $\boldsymbol{P} = < S,\delta,\eta,D_s,F,\omega >$ is

$$\Re = \left\{ < F_{szr}, F_{s'zr'} > \left| \begin{array}{l} (\omega_{szr} \vee \omega_{s'zr'}) \wedge \\ (\exists \vec{i} \in Z^{\delta_s}, \vec{i'} \in Z^{\delta_{s'}} | (\vec{i} \prec_{ss'} \vec{i'}) \\ \wedge (F_{szr}(\vec{i}) - F_{s'zr'}(\vec{i'}) = \vec{0}) \wedge \\ (D_s(\vec{i}) \ge \vec{0} \wedge D_{s'}(\vec{i'}) \ge \vec{0})) \end{array} \right. \right\} \quad (1)$$

where $\prec$ is the "lexicographically less than" operator such that $\vec{i} \prec_{ss'} \vec{i'}$ if and only if iteration $\vec{i}$ of statement s is executed before iteration $\vec{i'}$ of s' in the program $\boldsymbol{P}$.

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 $\boldsymbol{P} = < S,\delta,\eta,D_s,F,\omega >$. An affine space partition mapping Φ for $\boldsymbol{P}$ is independent if and only if

$$\begin{array}{l} \forall < F_{szr}, F_{s'zr'} > \in \Re, \vec{i} \in Z^{\delta_s}, \vec{i'} \in Z^{\delta_{s'}} \text{ s.t} \\ D_S(\vec{i}) \ge \vec{0} \wedge D_{s'}(\vec{i'}) \ge \vec{0} \wedge F_{szr}(\vec{i}) - F_{s'zr'}(\vec{i'}) = \vec{0}, \quad (2) \\ \Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) = 0. \end{array}$$

This constraint simply states that data dependent instances must be placed in the same partition. That is if iteration $\vec{i}$ of statement s and iteration $\vec{i'}$ of statement s' access the same data, then their mappings Φ_s and $\Phi_{s'}$ are constrained to assign these iterations to the same space partition, i.e. $\Phi_{s'}(\vec{i'}) - \Phi_s(\vec{i}) = 0$.

Let $D_s(\vec{i}) = D_s\vec{i} + \vec{d}_s$, $F_{szr}(\vec{i}) = F_{szr}\vec{i} + \vec{f}_{szr}$ and $\Phi_s(\vec{i}) = C_s\vec{i} + c_s$. Each data dependence, $<F_{szr}, F_{s'zr'}> \in \Re$, imposes the following constraints on each row of an independent mapping for statements s and s', X = $\left[-C_s \quad C_{s'} \quad (c_{s'} - c_s)\right]$:

$$\forall \vec{y} \text{ s.t. } G\begin{bmatrix}\vec{y}\\1\end{bmatrix} \geq \vec{0} \wedge \mathcal{H}\begin{bmatrix}\vec{y}\\1\end{bmatrix} = \vec{0}, \mathrm{X}\begin{bmatrix}\vec{y}\\1\end{bmatrix} = 0 \qquad (3)$$

where

$$\vec{y} = \begin{bmatrix}\vec{i}\\ \vec{i'}\end{bmatrix}, G = \begin{bmatrix} D_s & 0 & \vec{d}_s \\ 0 & D_{s'} & \vec{d}_{s'}\end{bmatrix},$$

$$\mathcal{H} = \begin{bmatrix} F_{szr} & F_{s'zr'} & (\vec{f}_{szr} - \vec{\bar{f}}_{s'zr'})\end{bmatrix}$$

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

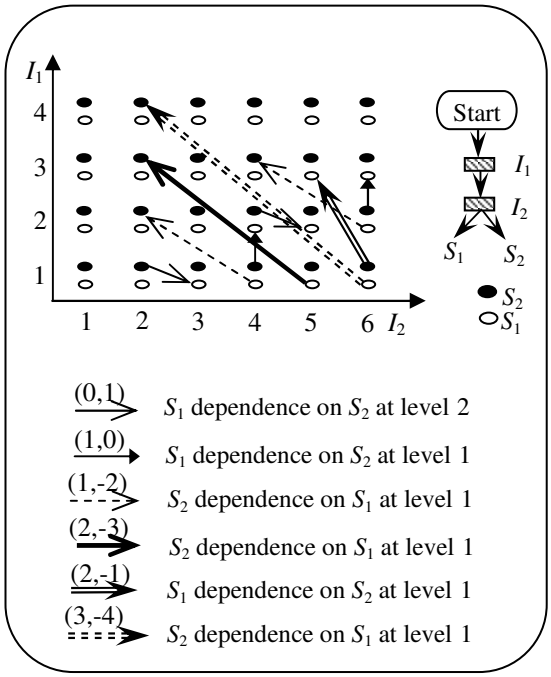

Figure 3 iteration space, hierarchical program structure and data dependence.

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 $\boldsymbol{P} = < S, \delta, \eta, D_s, F, \omega >$, 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.
- $D_s(\vec{i}) \geq \vec{0}$ is a system of linear constraints that describes the valid loop iterations for statement s. So from the loop bounds of the loops surrounding

s_1 and s_2 we can find $D_1(\vec{i})$ and $D_2(\vec{i})$. They will have the following same form

$$D_1(I_1,I_2)=D_2(I_1,I_2)=\begin{bmatrix}1&0\\-1&0\\0&1\\0&-1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}-1\\N_1\\-1\\N_2\end{bmatrix}.$$

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

$$F_{1A1}(I_1,I_2)=\begin{bmatrix}1&1\\3&1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}0\\3\end{bmatrix},$$

$$F_{1B1}(I_1,I_2)=\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix},$$

$$F_{2C1}(I_1,I_2)=\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix},$$

$$F_{2A1}(I_1,I_2)=\begin{bmatrix}1&1\\1&2\end{bmatrix}\begin{bmatrix}I_1\\I_2\end{bmatrix}+\begin{bmatrix}1\\4\end{bmatrix}.$$

- ω_{srz} is a Boolean value describing whether the r th array reference to array z in statement s is a write operation.

ω_{1A1} = true, ω_{1B1} = false, ω_{2C1} = true, ω_{2A1} = false.

The data dependence set for the above code is the 1-element set $\{< F_{1A1}, F_{2A1} >\}$. 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 $\Phi = [\Phi_1, \Phi_2]$ with $\Phi_1 = I_1 + I_2$ and $\Phi_2 = I_1 + I_2 + 1$. The SPMD code for the previous code after the space partition mapping Φ 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, I_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:

$$\max(1,\Phi-N_2)\le I_1\le\min(N_1,\Phi-1) \quad (1)$$

$$2 \le \Phi \le N_1+N_2 \quad (2)$$

The new loop bounds for statement S_2 is as follows:

$$\max(1,\Phi-N_2-1)\le I_1\le\min(N_1,\Phi-2) \quad (3)$$

$$3 \le \Phi \le N_1+N_2+1 \quad (4)$$

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

$$2 \le \Phi \le N_1+N_2+1$$

The number of partitions (Φ)=N_1+N_2+1-2+1=N_1+N_2. The table 1 shows all independent partitions, which can be executed in parallel. In this case N_1 = 4 and N_2 = 6.

Table 1 Independent partitions with affine transformation

Φ =2	Φ =3	Φ =4	Φ =5	Φ =6
S_1(1,1)	S_2(1,1)	S_2(1,2)	S_2(1,3)	S_2(1,4)
	S_1(1,2)	S_1(1,3)	S_1(1,4)	S_1(1,5)
	S_1(2,1)	S_2(2,1)	S_2(2,2)	S_2(2,3)
		S_1(2,2)	S_1(2,3)	S_1(2,4)
		S_1(3,1)	S_2(3,1)	S_2(3,2)
			S_1(3,2)	S_1(3,3)
			S_1(4,1)	S_2(4,1)
				S_1(4,2)

(Cont.) Table 1

Φ =7	Φ =8	Φ =9	Φ =10	Φ =11
S_2(1,5)	S_2(1,6)	S_2(2,6)	S_2(3,6)	S_2(4,6)
S_1(1,6)	S_2(2,5)	S_2(3,5)	S_2(4,5)	
S_2(2,4)	S_1(2,6)	S_1(3,6)	S_1(4,6)	
S_1(2,5)	S_2(3,4)	S_2(4,4)		
S_2(3,3)	S_1(3,5)	S_1(4,5)		
S_1(3,4)	S_2(4,3)			
S_2(4,2)	S_1(4,4)			
S_1(4,3)				

The table shows that different partitions (Φ =2 to Φ = 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.

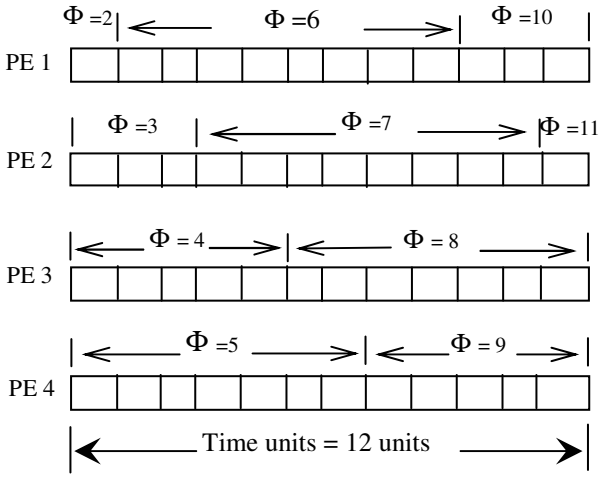

Figure 4 Time diagram with affine transformation

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.

4.2 Parallelism with Unimodular Transformation

For the previous code we will show how we can use the unimodular transformation to extract the parallelism. Here the dependence exists in outer and inner loop (at levels 1, and 2). The outer and inner loops cannot be executed in parallel. The unimodular transformation cannot be used directly because the data dependence is non-uniform dependence. So the distance matrix $\boldsymbol{D}$ cannot be performed. The dependence uniformization technique in [27] can be used to uniformize the data dependences. By applying the idea of the dependence uniformization technique, a set of basic dependences is chosen to replace all original dependence constraints in every iteration so that the dependence pattern becomes uniform. Then, this uniformization helps in applying the unimodular transformation. The distance matrix $\boldsymbol{D}$ after uniformization is as follows:

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

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

$$\boldsymbol{U} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}$$

to transform loop $\boldsymbol{L}$ into $\boldsymbol{L}_U$ whose inner loop can be executed in parallel.

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

$(K_1, K_2) = (3I_1 + I_2, I_1)$,

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

$(I_1, I_2) = (K_2, K_1 - 3K_2)$.

The constraints on I_1 and I_2 give the inequalities:

$$1 \le K_2 \le N_1$$

$$1 \le K_1 - 3K_2 \le N_2$$

By using Fourier's elimination [29] we find

$$4 \le K_1 \le 3N_1+N_2$$

$$\max\left(1, \left\lceil \frac{K_1 - N_2}{3} \right\rceil\right) \le K_2 \le \min\left(N_1, \left\lfloor \frac{K_1 - 1}{3} \right\rfloor\right).$$

The loop $\boldsymbol{L}$ becomes after transformation in the form $\boldsymbol{L}_U$ as follows, where in this case N_1= 4 and N_2= 6

L_{U1}: For $K_1 = 4, 18$

L_{U2}: For $K_2 = \max\left(1, \left\lceil \frac{K_1 - 6}{3} \right\rceil\right), \min\left(4, \left\lfloor \frac{K_1 - 1}{3} \right\rfloor\right)$

$S_1(K_1,K_2)$: 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

$$\boldsymbol{DU} = \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

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 min $\left(4,\left\lfloor\frac{K_1-1}{3}\right\rfloor\right)$= 2 i.e K_1= 7). Also processor 3 will remain idle till k_2=3 (when min $\left(4,\left\lfloor\frac{K_1-1}{3}\right\rfloor\right)$= 3 i.e K_1= 10) and processor 4 will remain idle till k_2=4 (when min $\left(4,\left\lfloor\frac{K_1-1}{3}\right\rfloor\right)$= 4 i.e K_1= 13).

Table 2 Allocation of the data to 4 processors with unimodular transformation

Processor 1	Processor 2
S_1(4,1): A(2,7)=B(1,1)	Idle
S_2(4,1): C(1,1)=A(3,7)	Idle
S_1(5,1): A(3,8)=B(1,2)	Idle
S_2(5,1): C(1,2)=A(4,9)	Idle
S_1(6,1): A(4,9)=B(1,3)	Idle
S_2(6,1): C(1,3)=A(5,11)	Idle
S_1(7,1): A(5,10)=B(1,4)	S_1(7,2): A(3,10)=B(2,1)
S_2(7,1): C(1,4)=A(6,13)	S_2(7,2): C(2,1)=A(4,8)
S_1(8,1): A(6,11)=B(1,5)	S_1(8,2): A(4,11)=B(2,2)
S_2(8,1): C(1,5)=A(7,15)	S_2(8,2): C(2,2)=A(5,10)
S_1(9,1): A(7,12)=B(1,6)	S_1(9,2): A(5,12)=B(2,3)
S_2(9,1): C(1,6)=A(8,17)	S_2(9,2): C(2,3)=A(6,12)
	S_1(10,2): A(6,13)=B(2,4)
	S_2(10,2): C(2,4)=A(7,14)
	S_1(11,2): A(7,14)=B(2,5)
	S_2(11,2): C(2,5)=A(8,16)
	S_1(12,2): A(8,15)=B(2,6)
	S_2(12,2): C(2,6)=A(9,18)

(Cont.) Table 2

Processor 3	Processor 4
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
Idle	Idle
S_1(10,3): A(4,13)=B(3,1)	Idle
S_2(10,3): C(3,1)=A(5,9)	Idle
S_1(11,3): A(5,14)=B(3,2)	Idle
S_2(11,3): C(3,2)=A(6,11)	Idle
S_1(12,3): A(6,15)=B(3,3)	Idle
S_2(12,3): C(3,3)=A(7,13)	Idle
S_1(13,3): A(7,16)=B(3,4)	Idle
S_2(13,3): C(3,4)=A(8,15)	Idle
S_1(14,3): A(8,17)=B(3,5)	S_1(13,4):A(5,16)=B(4,1)
S_2(14,3): C(3,5)=A(9,17)	S_2(13,4):C(4,1)=A(6,10)
S_1(15,3): A(9,18)=B(3,6)	S_1(14,4):A(6,17)=B(4,2)
S_2(15,3):C(3,6)=A(10,19)	S_2(14,4):C(4,2)=A(7,12)
	S_1(15,4):A(7,18)=B(4,3)
	S_2(15,4):C(4,3)=A(8,14)
	S_1(16,4):A(8,19)=B(4,4)
	S_2(16,4):C(4,4)=A(9,16)
	S_1(17,4):A(9,20)=B(4,5)
	S_2(17,4):C(4,5)=A(10,18)
	S_1(18,4):A(10,21)=B(4,6)
	S_2(18,4):C(4,6)=A(11,20)

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.

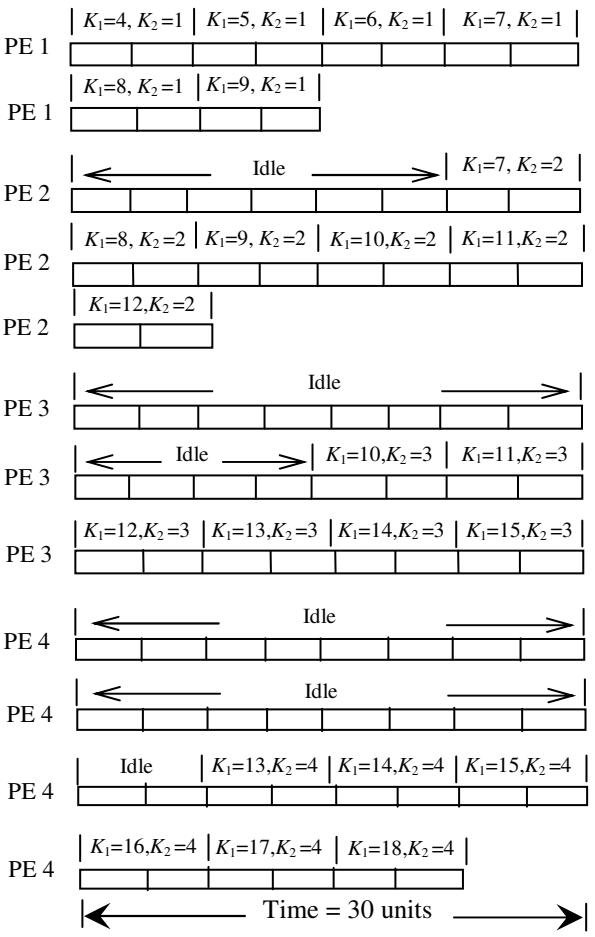

Figure 5 Time diagram with unimodular transformation

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.
- Unimodular transformation needs not only to determine the variables that have dependence but it needs to determine the distance and direction of dependence vectors. In affine transformation it is needed only to determine the variables that have dependence. Time of execution with the unimodular transformation is longer than time of execution with the affine transformation, when the dependence is between different statements.

References:

[1] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, Chichester, 1986.

[2] A. W. Lim, G. I. Cheong, and M. S. Lam, An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication, *In Proceeding of the 13th ACM SIGARCH international Conference on Supercomputing,* Rhodes, Greece, June 1999.

[3] A. W. Lim and M. S. Lam, Cache Optimizations with Affine Partitioning, *In Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[4] A. W. Lim and M. S. Lam, Maximizing Parallelism and Minimizing Synchronization with affine partitions, *Parallel Computing* vol. 24, no. 3-4, pp. 445-475, May 1998.

[5] A. W. Lim and M. S. Lam, Maximizing Parallelism and Minimizing Synchronization with Affine Transforms, *In Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[6] C. Bastoul, Improving Data Locality in Static Control Programs, PhD thesis, Univ. Paris 6, Pierre et Marie Curie, December 2004.

[7] C. Gong, R. Melhem and R. Gupta, Loop Transformations for Fault Detection in Regular Loops on Massively Parallel Systems, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1238-1249, December 1996.

[8] C. T. Yang, S. S. Tseng, S. H. Kao, M. H. Hsieh and M. F. Jiang, Run-time Parallelization for Partially Parallel Loops, *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pp. 308-313, 1997.

[9] C. Xu, Effects of Parallelism Degree on Run-Time Parallelization of Loops, *Proceedings of the 31st Hawaii International Conference on System Sciences* (*HICSS'98*).

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

Distributed System, vol. 7, pp. 463-476, May 1996.

[11] D. L. Pean and C. Chen, An Optimized Three Region Partitioning Technique to Maximize Parallelism of Nested Loops with Non-uniform Dependences, *Journal of Information Science and Engineering*, vol. 17, pp. 463-489, 2001.

[12] D. R. Chesney and B. H. Cheng, Generalizing the Unimodular Approach, *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pp. 398-403, 1994.

[13] E. D'Hollander, Partitioning and Labeling of Loops by Unimodular Transformations, *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 4, pp. 465-476, July 1992.

[14] G. Goumas, M. Athanasaki and N. Koziris, An Efficient Code Generation Technique for Tiled Iteration Spaces, *IEEE Tran. On Parallel and Distrib. Syst.,* vol. 14, no. 10, pp. 1021-1034, 2003.

[15] J. Xue, Unimodular Transformations of Non-Perfectly Nested Loops, *Parallel Computing*, vol. 22, no. 12, pp. 1621-1645, 1997.

[16] K. Psarris, X. Kong and D. Klappholz, The Direction Vector *I* Test, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 11, pp. 1280-1290, November 1993.

[17] L. N. Pouchet, When Iterative Optimization Meets the Polyhedral Model: One-dimensional Date, Master thesis (Univ. of Paris-Sud XI), Orsay, France, October 2006.

[18] M. Rim and R. Jain, Valid Transformations: A New Class of Loop Transformations for High-Level Synthesis and Pipelined Scheduling Applications, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 4, April 1996.

[19] M. wolf and M. Lam, A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[20] M. Wolfe, *High Performance Compilers for Parallel Computing*, California, Addison-Wesley publishing Company, Inc, 1996.

[21] M. wolfe and C. Tseng, The Power Test for Data Dependence, *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 5, pp. 591-601, September 1992.

[22] N. Likhoded, S. Bakhanovich and A. Zherelo, Obtaining Affine Transformations to Improve Locality of Loop Nests, *Programming and Computer Software*, vol. 31, no. 5, pp 270-281, 2005.

[23] N. Manjikian and T. S. Abdelrahman, Fusion of Loops for Parallelism and Locality, *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 2, pp. 193-209, February 1997.

[24] P. M. Petersen and D. A. Padua, Static and Dynamic Evaluation of Data Dependence Analysis Techniques, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 11, pp. 1121-1132, November 1996.

[25] Q. Yi and K. Kennedy, Transforming Complex Loop Nests for Locality, Technical Report TR02-386, Computer Science Dep., Rice Univ., Feb. 2002.

[26] S. Punyamurtula, V. Chaudhary, J. Ju and S. Roy, Compile Time Partitioning of Nested Loop Iteration Spaces with Non-uniform Dependences, *Journal of Parallel Algorithms and Applications*, vol. 12, no. 1-3, pp. 113-141, 1997.

[27] T. Tzen and Lionel M. Ni, Dependence Uniformization: A Loop Parallelization Technique, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 5, pp. 547-558, May 1993.

[28] U. Banerjee, An Introduction to a Formal Theory of Dependence Analysis, *Journal of Supercomputing*, 2, pp. 133-149, 1988.

[29] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Boston: Kluwer Academic Publishers, 1993.

[30] U. Banerjee, *Loop Parallelization*, Boston: Kluwer Academic Publishers, 1994.

[31] U. Banerjee, *Dependence Analysis*, Boston: Kluwer Academic Publishers, 1997.

[32] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev and P. Sadayappan, Affine Transformations for Communication Minimal Parallelization and Locality Optimization of Arbitrarily Nested Loop Sequences, Technical Report OSU-CISRC-5/07-TR43, Computer Science and Engineering Dep., Ohio State Univ., May 2007.

[33] V. Maslov, Lazy array data flow dependence analysis, *Conference record of the annual ACM symposium on principles of programming languages*, pp. 311-325 1994.

[34] W. Pugh, A Practical Algorithm for Exact Array Dependence Analysis, *Communications of The ACM*, vol. 35, no. 8, pp. 102-114, August 1992.

[35] X. Kong, D. Klappholz and K. Psarris, The *I* Test: An Improved Dependence Test for Automatic Parallelization and Vectorization, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 342-349, July 1991.

[36] Z. Li, P. Yew and C. Zhu, An Efficient Data Dependence Analysis for Parallelizing Compilers, *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 26-34, January 1990.