EIGENVECTORS-BASED PARALLELISATION OF NESTED LOOPS WITH AFFINE DEPENDENCES

Patrick Lenders and Jingling Xue
School of Mathematical and Computer Science
University of New England, Armidale, NSW 2351, Australia

This paper is concerned with parallelising a special class of nested loops with affine dependences. The data dependences of the program are captured in a so-called dependence matrix. Based on the eigenvalues and eigenvectors of this matrix, the proposed approach can generate a greater degree of DOALL parallelism than traditional unimodular transformations.

1 Introduction

Although many compiler techniques (including the well-known unimodular transformations) have been developed[1,2,3,4,5,6,7,8,9,10,11], the DOALL parallelisation for nested loops with affine dependences remains a challenging task. This paper is concerned with parallelising nested loops with affine dependences. We restrict ourselves to doubly nested loops, although our method can be generalised to any n-dimensional nested loops. Our method is based on the concept of eigenvalues and eigenvectors that are derived from the dependence structures of the program. In comparison with the unimodular approach, our method can find a greater degree of parallelism for a special class of nested loops. However, our method may fail where the unimodular approach succeeds (Section 3).

Section 2 describes the program model and our formalism for characterising the data dependences in a program. Section 3 presents our method for parallelising loops with affine dependences. Section 4 concludes the paper.

2 Program Model

This paper considers doubly nested loops of the form (in the style of C):

$$\begin{array}{l} \mathsf{for}(j_1 = \ell_1;\ j_1 \leqslant u_1;\ j_1{+}{+}) \\ \quad \mathsf{for}(j_2 = \ell_2;\ j_2 \leqslant u_2;\ j_2{+}{+}) \\ \quad\quad A(\mu(j)) = f(A(\nu(j)), \cdots) \end{array}$$

where $j = (j_1, j_2)$ is a point in the *iteration space*, which is the set of points satisfying $\mathcal{J} = \ell_1 \leqslant j_1 \leqslant u_1 \wedge \ell_2 \leqslant j_2 \leqslant u_2$, and μ and ν are affine functions of loop variables: $\mu(j) = Uj + u$ and $\nu(j) = Vj + v$, where $U, V \in \mathbb{Z}^{2\times 2}$ and $u, v \in \mathbb{Z}^2$.

The two array references can access the same element of A at two different iterations. Thus, there is a dependence between point j and point $\delta(j)$ (if such a point is in the iteration space) such that $\delta(j) = \nu^{-1}(\mu(j))$. In the case of affine dependences, we have the following *dependence function*:

$$\delta(j) \ = \ V^{-1}Uj + V^{-1}(u - v) \ = \ Dj + d$$

where the matrix D is called the *dependence matrix*. This dependence is represented by the difference between the following *relative dependence function*:

$$\rho(j) \ = \ \delta(j) - j \ = \ (D - I)j + d$$

Let us relate our dependence abstraction with the usual dependence vector abstraction[8], $\rho(j)$ is a *flow* or *true dependence* originating at j and sinking at $\delta(j)$ if $\rho(j) \succ 0$, and an *anti-dependence* originating at $\delta(j)$ and sinking at j if $\rho(j) \prec 0$. In the case when $\rho(j) \succ 0$, $\delta(j)$ is said to *depend* on j, and this dependence is characterised by the dependence vector $\rho(j)$. When $\rho(j) \prec 0$, j is said to *(anti-) depend* on $\delta(j)$, and this dependence is characterised by the dependence vector $-\rho(j)$. When representing dependences in the iteration space, a flow dependence is drawn with a solid arrow and an anti-dependence with a dashed arrow. In both cases, the point at the arrow head always depends on the point at the arrow tail.

Thus, all dependence vectors in the iteration space are lexicographically positive.

According to the lexicographic sign of $\rho(j)$, the iteration space is divided into:

$$\mathcal{J}_+ \;=\; \mathcal{J} \wedge \rho(j) \succ 0 \;=\; \mathcal{J} \wedge (\rho_1(j) > 0 \vee \rho_1(j) = 0 \wedge \rho_2(j) > 0)$$
$$\mathcal{J}_- \;=\; \mathcal{J} \wedge \rho(j) \prec 0 \;=\; \mathcal{J} \wedge (\rho_1(j) < 0 \vee \rho_1(j) = 0 \wedge \rho_2(j) < 0)$$
$$\mathcal{J}_0 \;=\; \mathcal{J} \wedge \rho(j) = 0 \;=\; \mathcal{J} \wedge \;\rho_1(j) = 0 \wedge \rho_2(j) = 0$$

where the dependence vectors contained in $\mathcal{J}_+$ ($\mathcal{J}_-$) are all flow dependences (anti-dependences), and the points in $\mathcal{J}_0$ are all mutually independent of each other.

The following lemma identifies the nature of dependences between the three regions $\mathcal{J}_+$, $\mathcal{J}_-$ and $\mathcal{J}_0$ and provides an ordering to schedule them for execution.

Lemma 1 *(a) All (flow or anti-) dependences spanning across $\mathcal{J}_+$ and $\mathcal{J}_-$ originate from $\mathcal{J}_+$ and terminate at $\mathcal{J}_-$. (b) All points of $\mathcal{J}_0$ are independent of each other and, if D is nonsingular, these points are also independent of those in $\mathcal{J}_+$ and $\mathcal{J}_-$.*

Let us use an example to illustrate the concepts introduced so far.

Example 1 Consider a double loop:

```
for ($j_1 = -N$; $j_1 \leqslant N$; $j_1$ ++)
    for ($j_2 = -N$; $j_2 \leqslant N$; $j_2$ ++)
        $A(4j_1 + 4j_2 - 3, j_1 + 3j_2) = 2 * A(j_1 + j_2, j_2 + 1)$
```

The array subscript functions μ and ν are:

$$\mu(j) \;=\; Uj + u \;=\; \begin{bmatrix} 4 & 4 \\ 1 & 3 \end{bmatrix} j \;+\; \begin{bmatrix} -3 \\ 0 \end{bmatrix}, \quad \nu(j) \;=\; Vj + v \;=\; \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} j \;+\; \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The dependence function δ is:

$$\delta(j) \;=\; Dj + d \;=\; \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} j \;+\; \begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

The eigenvalues of D are 2 and 4, corresponding to the eigenvectors $(1, -1)$ and $(1, 1)$, respectively. The relative dependence function ρ is:

$$\rho(j) \;=\; (D - I)j + d \;=\; \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} j \;+\; \begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

The two lines $\rho_1(j) = 0$ and $\rho_2(j) = 0$ separate the iteration space into the three regions $\mathcal{J}_+$, $\mathcal{J}_-$ and $\mathcal{J}_0$, as depicted in Fig. 1. $\mathcal{J}_0 = \{(1, 0)\}$, and $\mathcal{J}_+$ has both flow dependences and anti-dependences originating from it and terminating at $\mathcal{J}_-$.

3 Eigenvectors-Based DOALL Parallelisation

Two assumptions are made about the dependence matrix D.

1 *D must have rational eigenvalues*, ensuring that the corresponding eigenvectors are integer vectors and can be used to generate DOALL parallelism.

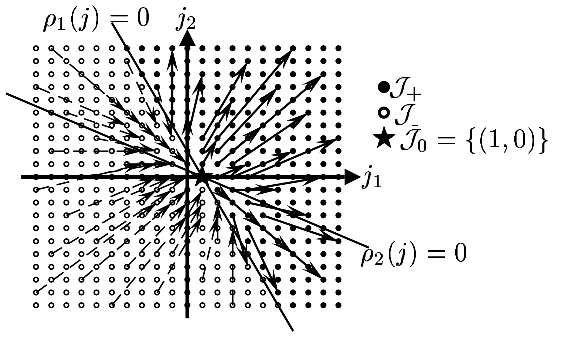


Figure 1: The iteration space of Example 1 ($N = 10$).

2 *D must be nonsingular*, implying that both U and V are nonsingular. In this case, the program has a single assignment semantics. That is, every array element is written (or read) at most once. Thus, the dependence function $\delta(j) = Dj + d$ captures all dependences involving both $A(\mu(j))$ and $A(\nu(j))$.

Next, we provide a quick outline of our method. An *execution ordering* $\lessdot$ is a partial order on the iteration space $\mathcal{J}$, where $p \lessdot q$ means that iteration p is executed before iteration q. An execution ordering for a program is *legal* if it preserves the data dependences of the program. In other words, any ordering that does not schedule an iteration before all its dependent iterations have been executed is legal. In our formalism, this idea translates into the following definition.

Definition 1 (**Legality of Execution Ordering**) An execution ordering $\lessdot$ on $\mathcal{J}$ is legal if $\forall\, j \in \mathcal{J}_+ : j \lessdot \delta(j)$ and $\forall\, j \in \mathcal{J}_- : \delta(j) \lessdot j$.

Note that by Lemma 1, the points of $\mathcal{J}_0$ can be executed in any order.

To generate DOALL parallelism from a program, we proceed as follows.

1 We apply a unimodular transformation to the iteration space (Section 3.1):

$$T \;:\; \mathcal{J} \longrightarrow \mathcal{J}', \text{ where } j' = Tj$$

such that in the *transformed space* $\mathcal{J}'$, the dependence function becomes:

$$\delta'(j') \;=\; D'j' + d'$$

where D' is lower triangular. *Unlike the unimodular approach, T usually does not preserve the data dependences of the program in the traditional sense*[8] *. In other words, $T\rho(j)$ $(-T\rho(j))$ for a dependence vector $\rho(j)$ $(-\rho(j))$ such that $j \in \mathcal{J}_+$ $(j \in \mathcal{J}_-)$ can be either lexicographically positive or negative.*

2 We analyse the data dependences in the transformed space and detect the parallelism inherent in the original program (Section 3.2).

3 We generate the parallel code of the form (Section 3.3):

```
PAR
    code($\mathcal{J}'_0$)
    SEQ
        code($\mathcal{J}'_+$)
        code($\mathcal{J}'_-$)
```
(1)

where the two constructs PAR and SEQ are borrowed from Occam[12]. The order for executing the three regions follows from Lemma 1. The points of $\mathcal{J}_0$ are independent, so code($\mathcal{J}_0$) will not be discussed any further. In Section 3.3, we describe how to construct code($\mathcal{J}'_+$) and code($\mathcal{J}'_-$) with explicit DOALL parallelism while preserving the data dependences of the original program.

Theorem 1 *The execution ordering $\lessdot$ induced by the code in (1) is legal if $\forall\ j' \in \mathcal{J}'_+ : j' \lessdot \delta'(j')$ and $\forall\ j' \in \mathcal{J}'_- : \delta'(j') \lessdot j'$.*

3.1 Loop Transformation: Triangularisation

In this section, we present the technique used to transform the iteration space in order to have a lower triangular dependence matrix. Let T be a unimodular transformation from the original iteration space to the transformed space. Thus the point j' in the transformed space is $j' = Tj$. The new array subscript functions μ' and ν' are $\mu'(j') = \mu(T^{-1}j')$ and $\nu'(j') = \nu(T^{-1}j')$. Thus the new dependence function is $\delta'(j') = \nu'^{-1}(\mu'(j')) = T(\nu^{-1}(\mu(T^{-1}j'))) = T(\delta(T^{-1}j'))$. In the case of affine dependencies, we have:

$$\delta'(j') \quad = \quad D'j' + d' \ = \ TDT^{-1}j' + Td$$

and the relative dependence function becomes:

$$\rho'(j') \quad = \quad (D' - I)j' + d' \ = \ (TDT^{-1} - I)j' + Td$$

In the transformed space, the two lines separating it into the three regions $\mathcal{J}'_+$, $\mathcal{J}'_-$ and $\mathcal{J}_0$ of different lexicographic signs become: $\rho_1(T^{-1}j') = 0$ and $\rho_2(T^{-1}j') = 0$.

Based on the eigenvectors of D, we provide a constructive approach to finding a unimodular transformation T such that $D' = TDT^{-1}$ is a lower triangular matrix.

Theorem 2 *Let λ_1 and λ_2 be the two rational eigenvalues of D, and v_1 and v_2 be the two corresponding (normalised) integer eigenvectors. Let $a, b \in \mathbb{Z}$ be such that $T^{-1} = \left[\begin{smallmatrix} a & v_{2,1} \\ b & v_{2,2} \end{smallmatrix}\right]$ is unimodular. Then $D' = TDT^{-1} = \left[\begin{smallmatrix} \lambda_1 & 0 \\ c & \lambda_2 \end{smallmatrix}\right]$, where $c \in \mathbb{Z}$.*

It can be shown that D' has the same eigenvalues λ_1 and λ_2 as D, which correspond to the two eigenvectors Tv_1 and $Tv_2 = (0, 1)$, respectively.

The eigenvector v_2 in this theorem is called the *parallelising eigenvector*. Our intention is to execute in the iteration space concurrently all points in a line parallel to v_2. In the transformed space, v_2 becomes $Tv_2 = (0, 1)$. Therefore, we are essentially attempting to execute in the transformed space all points in a vertical line in parallel. This corresponds to wavefronting the transformed space along directions $\pm(1, 0)$.

Example 2 In Example 1, we choose $(1, -1)$ as the parallelising eigenvector:

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

The new dependence function is calculated to be as follows:

$$\delta'(j') \ = \ \begin{bmatrix} 4 & 0 \\ -1 & 2 \end{bmatrix} j' + \begin{bmatrix} -3 \\ 1 \end{bmatrix}$$

Fig. 2 depicts the transformed space of the original iteration space in Fig. 1. Note that some dependences are lexicographically positive and some negative.

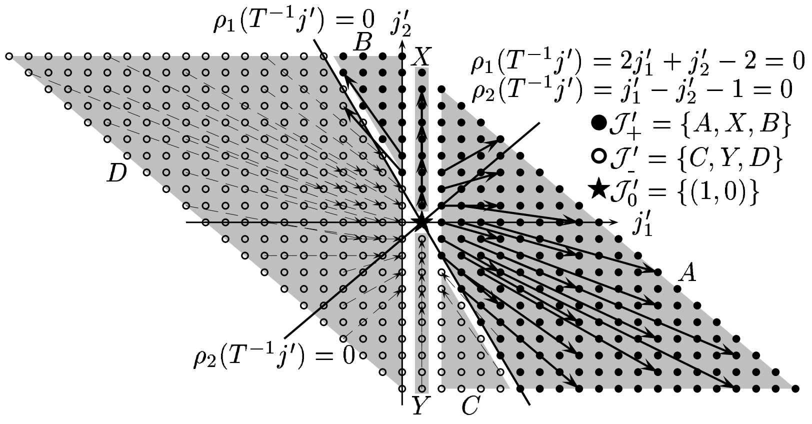

$\rho_1(T^{-1}j') = 0$
j'_2
B
X
$\rho_1(T^{-1}j') = 2j'_1 + j'_2 - 2 = 0$
$\rho_2(T^{-1}j') = j'_1 - j'_2 - 1 = 0$
$\bullet \mathcal{J}'_+ = \{A, X, B\}$
$\circ \mathcal{J}'_- = \{C, Y, D\}$
$\bigstar \mathcal{J}'_0 = \{(1,0)\}$
D
j'_1
A
$\rho_2(T^{-1}j') = 0$
Y
C

Figure 2: Transformed space of Fig. 1 ($N = 10$). The division of the transformed space into six regions A, B, X, C, D and Y as highlighted will be explained in the case when $\lambda_1 > 1$.

3.2 Parallelism Detection

In this section, we analyze the cross-iteration dependencies in the transformed space and detect the parallelism inherent in the program. We shall make use of the dependence function and relative dependence function in the transformed space:

$$\delta'(j') = \begin{bmatrix}\delta'_1(j')\\ \delta'_2(j')\end{bmatrix} = \begin{bmatrix}\lambda_1 & 0\\ c & \lambda_2\end{bmatrix} j' + \begin{bmatrix}d'_1\\ d'_2\end{bmatrix},\ \rho'(j') = \begin{bmatrix}\rho'_1(j')\\ \rho'_2(j')\end{bmatrix} = \begin{bmatrix}\lambda_1 - 1 & 0\\ c & \lambda_2 - 1\end{bmatrix} j' + \begin{bmatrix}d'_1\\ d'_2\end{bmatrix}$$

Theorem 3 *Let L_1 be a line parallel to an eigenvector v of D'. The set of points depending on any point of L_1 is on a line, L_2, parallel to L_1.*

This theorem suggests the following parallelization of the transformed space. All points on a line parallel to an eigenvector are executed concurrently, then all points on the next line parallel to the same eigenvector are executed concurrently, and so on. This eigenvector is referred to as the parallelising eigenvector in Section 3.1. In Theorem 2, by using T to transform the iteration space, we have implicitly assumed to use v_2 as the parallelising eigenvector. In the transformed space. v_2 becomes $Tv_2 = (0, 1)$. Thus, our intention is to execute all points in a vertical line in parallel.

A vertical line is called a *self-dependent* (*vertical*) *line* if all points on the line depend only on the points on the same line.

The following theorem characterises all self-dependent vertical lines in the transformed space. This is a special case of Theorem 3 when L_2 and L_1 are co-linear.

Theorem 4 *Consider the transformed space. If $\lambda_1 \neq 0$, $j'_1 = -d'_1/(\lambda_1 - 1)$ is the only self-dependent vertical line. If $\lambda_1 = 1$ and $d'_1 = 0$, all vertical lines are self-dependent. If $\lambda_1 = 1$ and $d'_1 \neq 0$, there are no self-dependent vertical lines.*

When $\lambda_1 \neq 0$, there is only one self-dependent vertical line $j'_1 = -d'_1/(\lambda_1 - 1)$. All dependences originating at one side of this line always sink at the same (opposite) side of the line if $\lambda_1 > 0$ ($\lambda_1 < 0$).

Theorem 5 *Assume $\lambda_1 \neq 0$. Let j' be a point in the transformed space not on the* *self-dependent line $j'_1 = -d'_1/(\lambda_1 - 1)$. Then, j' and $\delta'(j')$ are on the same side of the line $j'_1 = -d'_1/(\lambda_1 - 1)$ if $\lambda_1 > 0$ and on the opposite side if $\lambda_1 < 0$.*

We do not consider the case $\lambda_1 = 0$ because it implies D is singular. In this case, all dependences originating at one side of the line $j'_1 = -d'_1/(\lambda_1 - 1)$ sink on the line.

3.3 Loop Transformation: Parallelisation

In the last section we introduced parallelism detection techniques to identify sets (or lines) of independent iterations. In this section we use these techniques to generate code with explicit DOALL parallelism to execute the transformed space. Specifically, we discuss how to construct code$(\mathcal{J}'_+)$ and code$(\mathcal{J}'_-)$ as given in (1).

Our objective is to execute in the transformed space as many vertical lines in a single step as possible. We distinguish a total of six cases: (1) $\lambda_1 > 1$, (2) $0 < \lambda_1 < 1$, (3) $\lambda_1 = 1$, (4) $\lambda_1 < -1$, (5) $-1 < \lambda_1 < 0$, and (6) $\lambda_1 = -1$.

$\boxed{\boldsymbol{\lambda_1 > 1}}$ An example is used to illustrate the basic idea only. Consider the transformed space depicted in Fig. 2 from Example 2. The dependence function in the transformed space is given in Example 2. When $\lambda_1 > 1$, Theorem 4 suggests that there is only one self-dependent line among all vertical lines. In the current example, the self-dependent line is $j'_1 = 1$. Depending on whether a point is in the half space $j'_1 > 1$, on the line $j'_1 = 1$, or in the half space $j'_1 < 1$, we divide $\mathcal{J}'_+$ ($\mathcal{J}'_-$) into the three regions A, B and X (C, D and Y) as illustrated in Fig. 2.

The parallel code for both $\mathcal{J}'_+$ and $\mathcal{J}'_-$ is as follows:

```
code($\mathcal{J}'_+$): PAR
          code($A$) :   for($t = 2;\ t \leqslant 2N;\ t = 4t - 3$)
                         forall($j'_1 = t;\ j'_1 \leqslant \min(4t - 4, 2N);\ j'_1 ++$)
                           forall($j'_2 = \max(-2j'_1 + 2, -N);\ j'_2 \leqslant -j'_1 + N;\ j'_2 ++$)
                             $A(4j'_1 - 3, j'_1 - 2j'_2) = 2 * A(j'_1, -j'_2 + 1)$
          code($B$) :   for($t = 0;\ t \geqslant \lceil \frac{3-N}{2} \rceil;\ t = 4t - 3$)
                         forall($j'_1 = t;\ j'_1 \geqslant \max(4t - 2, \lceil \frac{3-N}{2} \rceil);\ j'_1 --$)
                           forall($j'_2 = -2j'_1 + 3;\ j'_2 \leqslant N;\ j'_2 ++$)
                             $A(4j'_1 - 3, j'_1 - 2j'_2) = 2 * A(j'_1, -j'_2 + 1)$
          code($X$) :   for($t = 1;\ t \leqslant N;\ t = 2t$)    /* $j'_1 = 1$ */
                         forall($j'_2 = t;\ j'_2 \leqslant \min(2t - 1, N);\ j'_2 ++$)
                           $A(1, 1 - 2j'_2) = 2 * A(1, -j'_2 + 1)$  code($\mathcal{J}'_-$) : PAR
          code($C$) :   for($t = \lfloor \frac{N+1}{2} \rfloor;\ t \geqslant 2;\ t = \lfloor \frac{t+3}{4} \rfloor$)
                          forall($j'_1 = t;\ j'_1 \geqslant \max(\lfloor \frac{t+3}{4} \rfloor + 1, 2);\ j'_1 --$)
                           forall($j'_2 = -N;\ j'_2 \leqslant 1 - 2j'_1;\ j'_2 ++$)
                             $A(4j'_1 - 3, j'_1 - 2j'_2) = 2 * A(j'_1, -j'_2 + 1)$
          code($D$) :   for($t = -2N;\ t \leqslant 0;\ t = \lceil \frac{t+3}{4} \rceil$)
                          forall($j'_1 = t;\ j'_1 \leqslant \min(\lceil \frac{t+3}{4} \rceil - 1, 0);\ j'_1 ++$)
                           forall($j'_2 = -j'_1 - N;\ j'_2 \leqslant \min(-2j'_1 + 2, N);\ j'_2 ++$)
                             $A(4j'_1 - 3, j'_1 - 2j'_2) = 2 * A(j'_1, -j'_2 + 1)$
          code($Y$) :   for($t = -N;\ t \leqslant -1;\ t = \lceil \frac{t}{2} \rceil$)    /* $j'_1 = 1$ */
                          forall($j'_2 = t;\ j'_2 \leqslant \min(\lceil \frac{t}{2} \rceil - 1, -1);\ j'_2 ++$)
                           $A(1, 1 - 2j'_2) = 2 * A(1, -j'_2 + 1)$
```

Unlike the existing unimodular approach [13,7,8], our eigenvector-based method schedules far more hyperplanes of iterations of loop j_1' for concurrent execution.

$\boxed{\mathbf{0 < \lambda_1 < 1}}$ This is the opposite of the case when $\lambda_1 > 1$. By Theorem 4, all flow dependences in $\mathcal{J}_+'$ are pointing toward the self-dependent line $j_1' = -d_1'/(\lambda_1 - 1)$ while all anti-dependences in $\mathcal{J}_-'$ are pointing away from this line. Thus, $\mathcal{J}_+'$ ($\mathcal{J}_-'$) in this case can be parallelised in the same way as $\mathcal{J}_-'$ ($\mathcal{J}_+'$) in the case when $\lambda_1 > 1$.

As an example, consider a double loop derived from Example 1 with the two references swapped. The dependence function is the inverse of the one in Example 1:

$$\delta(j) \;=\; Dj + d \;=\; \begin{bmatrix} 3/8 & -1/8 \\ -1/8 & 3/8 \end{bmatrix} j + \begin{bmatrix} 5/8 \\ 1/8 \end{bmatrix}$$

D has the eigenvalues $2/8$ and $4/8$, corresponding to the eigenvectors $(1,1)$ and $(1,-1)$, respectively. With the T in Example 2, we obtain the dependence function:

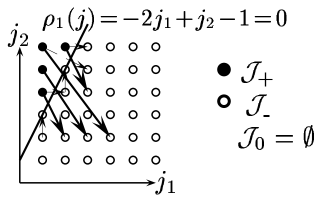
$$\delta'(j') \;=\; D'j' + d' \;=\; \begin{bmatrix} 2/8 & 0 \\ 1/8 & 4/8 \end{bmatrix} j' + \begin{bmatrix} 6/8 \\ -1/8 \end{bmatrix}$$

in the transformed space as in Fig. 2. So $\mathsf{code}(\mathcal{J}_+')$ and $\mathsf{code}(\mathcal{J}_-')$ are the same as their counterparts in Example 1 but with loop t reversed and loop j_1' modified accordingly.

$\boxed{\mathbf{\lambda_1 = 1}}$ We have $\rho_1'(j') = (\lambda_1 - 1)j_1' + d_1' = d_1'$. We distinguish two cases by generating outer DOALL parallelism if $d_1' = 0$ and inner DOALL parallelism if $d_1' \neq 0$.

The analysis of both cases are based on Theorem 4. If $d_1' = 0$, all dependences in the transformed space are of the form $(0,*)$. Thus, all vertical lines in $\mathcal{J}_+'$ ($\mathcal{J}_-'$) can be executed independently of each other. A vertical line can be further parallelised just like X and Y in Fig. 2 was parallelised.

If $d_1' \neq 0$, all (flow) dependences in $\mathcal{J}_+'$ have the form $(d_1',*)$ while all (anti) dependences in $\mathcal{J}_-'$ have the form $(-d_1',*)$. We can generate inner DOALL parallelism by wavefronting $\mathcal{J}_+'$ ($\mathcal{J}_-'$) along direction $(d_1',0)$ ($(-d_1',0)$) with $|d_1'|$ consecutive lines (or waves) being executed in parallel.



$\rho_1(j) = -2j_1 + j_2 - 1 = 0$
j_2
j_1
● $\mathcal{J}_+$
○ $\mathcal{J}_-$
$\mathcal{J}_0 = \emptyset$

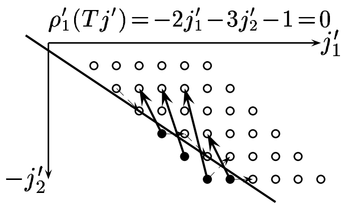


Fig. 3. The iteration space of Example 3. **Fig. 4.** The transformed space of Fig. 4 using T in Example 2.

Example 3 Consider the following example from [14], where N was set to 1000:

```
for ($j_1 = 1;\ j_1 \leqslant N;\ j_1{+}{+}$)
  for ($j_2 = 1;\ j_2 \leqslant N;\ j_2{+}{+}$)
    $A(j_1 + j_2, 3j_1 + j_2 + 3) = 2 * A(j_1 + j_2 + 1, j_1 + 2j_2 + 4)$
```

The dependence function is:

$$\delta(j) \;=\; \begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix} j \;+\; \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

The eigenvalues of D are 1 and -2, corresponding to the eigenvectors $(1,-1)$ and $(1,2)$, respectively. Since $D-I$ is singular and $\text{rank}(D-I,d] = 2$, we have $\mathcal{J}_0 = \emptyset$. $\mathcal{J}_+$ contains all iterations in the half space $\rho_1(j) = -2j_1 + j_2 - 1 > 0$, and $\mathcal{J}_-$ contains all iterations in the opposite half space $\rho_1(j) = -2j_1 + j_2 - 1 \leqslant 0$ (Fig. 3).

With $(1,-1)$ chosen as the parallelising eigenvector, we apply the T given in Example 2 to transform the iteration space. The dependence function in the transformed space as shown Fig. 4 is:

$$\delta'(j') = \begin{bmatrix} 1 & 0 \\ -2 & -2 \end{bmatrix} j' + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

We wavefront $\mathcal{J}'_+$ and $\mathcal{J}'_-$ along $(-1,0)$ and $(1,0)$, respectively:

```
code($\mathcal{J}'_+$) :for($j'_1 = \lfloor\frac{3N-2}{2}\rfloor$; $j'_1 \geqslant 5$; $j'_1$ --)
                     forall($j'_2 = \max(1-j'_1, -N)$; $j'_2 \leqslant \lfloor\frac{-2i-2}{3}\rfloor$; $j'_2$ ++)
                        $A(j'_1, 3j'_1 + 2j'_2 + 3) = 2 * A(j'_1 + 1, j'_1 - j'_2 + 4)$
code($\mathcal{J}'_-$) : for($j'_1 = 2$; $j'_1 \geqslant 2N$; $j'_1$ ++)
                     forall($j'_2 = \max(1-j'_1, \lceil\frac{-2j'_1-1}{3}\rceil, -N)$; $j'_2 \leqslant \min(-1, N-j'_1)$; $j'_2$++)
                        $A(j'_1, 3j'_1 + 2j'_2 + 3) = 2 * A(j'_1 + 1, j'_1 - j'_2 + 4)$
```

$\boxed{\boldsymbol{\lambda_1 < -1}}$ By Theorem 4, $j'_1 = -d'_1/(\lambda_1 - 1)$ is the only self-dependent vertical line. Let $j'_1 = a$ be the vertical line closest to $j'_1 = -d'_1/(\lambda_1 - 1)$ such that $|a - d'_1/(\lambda_1 - 1)|$ is an integer.

To parallelise $\mathcal{J}'_+$, we divide it into two regions: A contains the points on $j'_1 = -d'_1/(\lambda_1 - 1)$ and B contains the points not on $j'_1 = -d'_1/(\lambda_1 - 1)$. By Theorem 4, A and B are independent. By Theorem 5, all dependences in B cross the self-dependent line $j'_1 = -d'_1/(\lambda_1 - 1)$. The parallelisation of B is the following. At any single step we execute in parallel two strips of vertical lines that are symmetrical with the line $j'_1 = -d'_1/(\lambda_1 - 1)$:

$$\begin{array}{lll}
\text{Step 1:} & (a\lambda_1 - \frac{d'_1}{\lambda_1-1}, -a - \frac{d'_1}{\lambda_1-1}] & [a - \frac{d'_1}{\lambda_1-1}, -a\lambda_1 - \frac{d'_1}{\lambda_1-1}) \\
\text{Step 2:} & (-a\lambda_1^2 - \frac{d'_1}{\lambda_1-1}, a\lambda_1 - \frac{d'_1}{\lambda_1-1}] & [-a\lambda_1 - \frac{d'_1}{\lambda_1-1}, a\lambda_1^2 - \frac{d'_1}{\lambda_1-1}) \\
\text{Step 3:} & (a\lambda_1^3 - \frac{d'_1}{\lambda_1-1}, -a\lambda_1^2 - \frac{d'_1}{\lambda_1-1}] & [a\lambda_1^2 - \frac{d'_1}{\lambda_1-1}, -a\lambda_1^3 - \frac{d'_1}{\lambda_1-1}) \\
 & \cdots &
\end{array}$$

where $(a,b]$ ($[a,b)$) denote all lines $j'_1 = x$ such that x is an integer within the range.

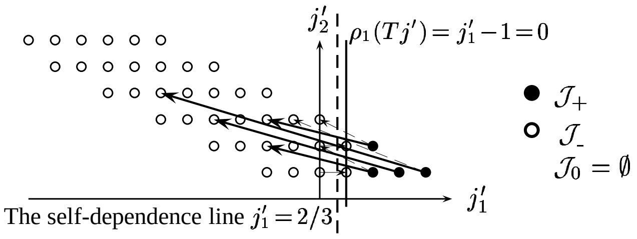
To understand this, consider an example where $\lambda_1 = -3$ and $d'_1 = 0$. The self-dependent line is $j'_1 = 0$. We shall parallelise B by executing in parallel, first lines $j'_1 = -2, -1, 1, 2$, then lines $j'_1 = -8, \cdots, -3, 3, \cdots, 8$, and so on.

$\mathcal{J}'_-$ is parallelised in the same way except the order for executing the above strips is reversed.

Example 4 Consider Example 3 again. We can execute the program in three steps if $(1,2)$ is selected as the the parallelising eigenvector. The matrix T is:

$$T = \begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}, \quad T^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

The dependence function in the transformed space as shown in Fig. 5 is:



j'_2
$\rho_1(Tj') = j'_1 - 1 = 0$
$\mathcal{J}_+$
$\mathcal{J}_-$
$\mathcal{J}_0 = \emptyset$
j'_1
The self-dependence line $j'_1 = 2/3$

Figure 5: The transformed space of Fig. 3 using T in Example 4.

$$\delta'(j') = \begin{bmatrix} -2 & 0 \\ 1 & 1 \end{bmatrix} j' + \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

The self-dependent line is $j'_1 = 2/3$, which does not contain any iterations.

$\mathcal{J}'_+$ is contained in the half space $j'_1 > 2/3$. All its points are independent, since any dependent points on $\mathcal{J}'_+$ must be on the opposite half space $j'_1 < 2/3$ by Theorem 5. The self-dependence line $j'_1 = 2/3$ divides $\mathcal{J}'_-$ into two regions: the points on the line $j'_1 = 1$ and the remaining points of $\mathcal{J}'_-$. We can execute $\mathcal{J}'_-$ in two steps, with those not on $j'_1 = 1$ in the first step and those on it in the second step.

In the original iteration space (Fig. 4), this corresponds to executing the points in the half space $-2i + j - 1 > 0$ in the first step, the points in the opposite half space $-2i + j - 1 < 0$ in the second step, and the points on the line $-2i + j - 1 = 0$ in the third step. This is considerably better than the scheme suggested in[14].

$\boxed{\mathbf{-1 < \lambda_1 < 0}}$ This is opposite of the case when $\lambda_1 < -1$. So $\mathcal{J}'_+$ ($\mathcal{J}'_-$) in this case can be parallelised in the same way as $\mathcal{J}'_-$ ($\mathcal{J}'_+$) in the case when $\lambda_1 < -1$.

$\boxed{\mathbf{\lambda_1 = -1}}$ By Theorem 4, the only self-dependence line is $j'_1 = d'_1/2$. Every line $j'_1 = a$ is inter-dependent only on the line $j'_1 = d'_1 - a$ symmetric about the line $j'_1 = d'_1/2$ (a corollary of Theorem 5 when $\lambda_1 = -1$). We can generate outer DOALL parallelism by executing every such a pair in sequence and all these pairs in parallel.

4 Conclusion

This paper discussed how to detect and exploit the parallelism inherent in nested loops with affine dependencies. We showed how to generate coarse-grain and fine-grain parallelism based on the eigenvectors derived from the dependence matrix D of the program. If D has an eigenvalue ± 1, the outer DOALL parallelisation is possible, making this technique appropriate for MIMD machines. If D does not have an eigenvalue ± 1, the inner loop can always be a DOALL loop. Thus this technique is appropriate for VLIW or superscalar machines. For a special class of nested loops, our method discovers far more parallelism than traditional unimodular transformations.

5 Acknowledgements

Thanks to the referees for their comments and suggestions. This work is supported by the Australian Research Council grants (A49330441 and A49600987).

References

1. U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.
2. A. Darte and F. Vivien. A comparison of nested loops parallelization algorithms. Technical Report 95–11, Ecole Normale Supérieure de Lyon, May. 1995.
3. P. Feautrier. Some efficient solutions to the affine scheduling problem, Part I, one-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, Oct. 1992.
4. P. Feautrier. Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.
5. W. Kelly and W. Pugh. A framework for iteration reordering transformations. In *1st Int. Conf. on Algorithms and Architectures for Parallel Processing (ICA^{3}PP)*, pages 153–612. IEEE Computer Society Press, Apr. 1995.
6. W. Shang, E. Hodzic, and Z. Chen. On uniformization of affine dependence algorithms. *IEEE Trans. on Computers*, 45(7):827–839, Jul. 1996.
7. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.
8. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
9. J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.
10. J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.
11. J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.
12. A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177ff., 1988.
13. L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, Feb. 1974.
14. Ten H. Tzen and Lionel M. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Trans. on Parallel and Distributed Systems*, 4(5):547–558, May 1993.