Eigenvectors-Based Parallelisation of Nested Loops with Affine Dependences

Patrick Lenders
School of Mathematical and Computer Sciences
University of New England
Armidale, NSW 2351, Australia

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

Abstract

This paper presents a method for parallelising nested loops with affine dependences. The data dependences of a program are represented exactly using a dependence matrix rather than an imprecise dependence abstraction. By a careful analysis of the eigenvectors and eigenvalues of the dependence matrix, we detect the parallelism inherent in the program, partition the iteration space of the program into sequential and parallel regions and generate parallel code to execute these regions. For a class of programs considered in the paper, the proposed method can expose more coarse-grain and fine-grain parallelism than a hyperplane-based loop transformation.

1 Introduction

Over the past two decades, researchers have developed a variety of loop transformations to optimise loop-oriented programs [14]. However, the DOALL parallelisation for nested loops with affine dependences remains difficult. The unimodular approach [2, 13, 14] can generate both coarse-grain and fine-grain DOALL parallelism, but in its full generality, it is limited mainly to nested loops with constant (i.e., uniform) dependences [16]. In the case of nested loops with affine dependences, two approaches can be identified. The first approach [16] approximates the dependences of a program with constant dependences by means of a dependence cone — with artificial dependences added *implicitly* — and then generates DOALL parallelism using the unimodular approach. The second approach [3, 5, 9, 11, 12, 17] uses *explicitly* artificial (constant) dependences to uniformise a program so that the uniformised program can be expressed as a set of loops with constant dependences. In both approaches, the artificial dependences introduced often span the entire iteration space of the program, making it only possible to generate Lamport-style hyperplane-based coarse-grain DOALL parallelism [6], where the parallel code consists of one sequential loop with unity stride followed by a sequence of $n-1$ DOALL loops (n being the dimension of the iteration space).

This paper is concerned with parallelising nested loops with a special form of affine dependences. We shall restrict ourselves to doubly nested loops, although our method generalises to n-dimensional nested loops. Our method is centered around the concepts of eigenvalues and eigenvectors that are derived from the dependence structures of the program. For the class of programs considered in this paper, our method has two major advantages over the previous work discussed above. First, our method dispenses with an expensive uniformisation process and contains a procedure for generating parallel code with DOALL parallelism. Second, our method can find more parallelism than any existing method that first uniformises the dependences in the program and then executes the uniformed program by applying a hyperplane-based unimodular approach.

An example is used below to illustrate our method and compare it with the two methods presented in [11, 12].

Example 1 Consider the following double loop from [11] where N was set to 5:

$$\begin{array}{l} \mathsf{for}\ (j_1 = 1;\ j_1 \leqslant N;\ j_1 ++) \\ \quad \mathsf{for}\ (j_2 = 0;\ j_2 \leqslant N;\ j_2 ++) \\ \quad\quad A(3j_1, 3j_2) = 2 * A(j_1, j_2) \end{array}$$

Figure 1 shows the iteration space along with all data dependences between every pair of dependent points. In [12], this program is first uniformised and then parallelised using hyperplane-based loop transformations. The uniformised program has the same iteration space with two constant dependence vectors $(1, 0)$ and $(0, 1)$. The scheduling vector for optimal DOALL parallelism is (1,1), requiring $2N - 1$ steps. This corresponds to wavefronting the iteration space along $(1, 1)$. If the two dependence vectors $(2, 2)$ and $(2, 0)$ are used instead as suggested in [11], the optimal scheduling vector is $(2, 0)$, reducing the run time to $\lceil N/2 \rceil$ steps. This corresponds to wavefronting the iteration space along the direction $(2, 0)$, with two consecutive "waves" being executed in parallel.

Using the method of this paper, we build a matrix D from the array subscripts:

$$D \;=\; \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}$$

D has one eigenvalue 3 with two associated eigenvectors $(1, 0)$ and $(0, 1)$. In the iteration space, all points in a vertical line parallel to the eigenvector $(0, 1)$ are independent. Thus, the outer loop can be made a DOALL loop immediately. But our method can detect more parallelism and produce the following parallel code:

$$\boxed{\begin{array}{l} \mathsf{for}\ (t = 1;\ t \leqslant N;\ t = 3t) \\ \quad \mathsf{forall}\ (j_1' = 1;\ j_1' \leqslant \max(3t - 1, N);\ j_1' ++) \\ \quad\quad \mathsf{forall}\ (j_2' = 0;\ j_2' \leqslant N;\ j_2' ++) \\ \quad\quad\quad A(3j_1', 3j_2') = 2 * A(j_1', j_2') \end{array}}$$

As depicted in Figure 1, we execute in parallel, the two lines $j_1' = 1$ and $j_1' = 2$, then the lines $j_1' = 3$ to $j_1' = 8$, then the lines $j_1' = 9$ to $j_1' = 26$, and so on. In general, we execute in parallel at step $t = c$ all lines from $j_1' = 3^{c-1}$ to $j_1' = 3^c - 1$. The number of lines executed at one iteration of loop t triples at the next iteration. Thus, the scheduling latency of our parallel code is $\lfloor \log_3 N \rfloor + 1$.

(End of Example)

The rest of this paper is organised as follows. 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 and also states the major limitations of our method. Section 4 discusses the related work. Section 5 concludes the paper.

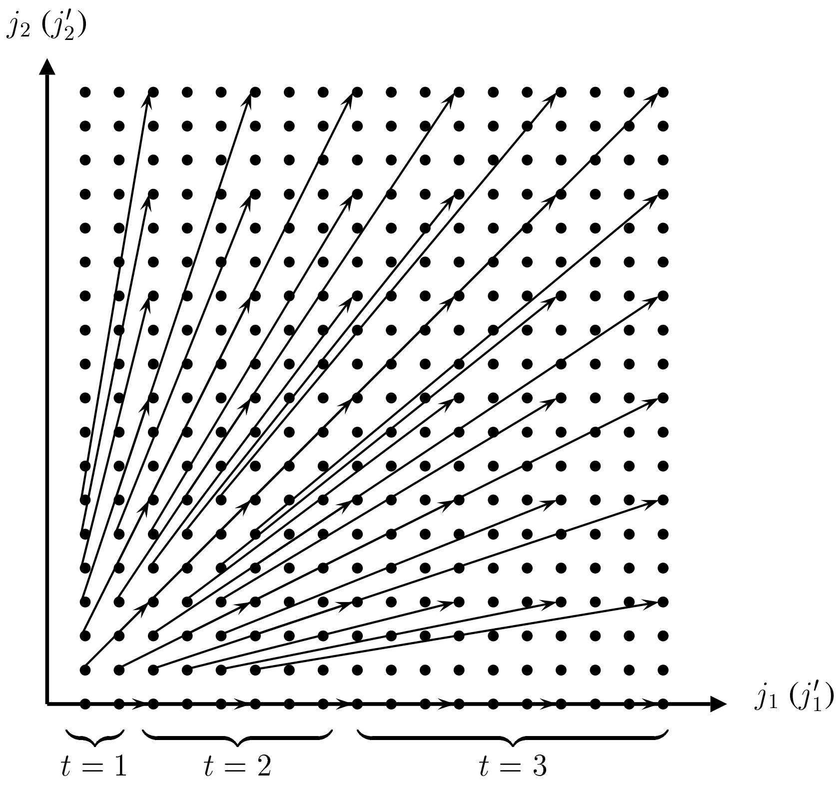

Figure 1: Eigenvectors-based parallel execution of Example 1 ($N = 18$).

2 Program Model

$\mathbb{Z}$ and $\mathbb{Q}$ denote the set of integers and rationals, respectively. We use $\succ$ and $\prec$ to represent the lexicographic order operators "larger than" and "less than", respectively. I denotes the identity matrix.

The following lemma states some useful facts about eigenvalues and eigenvectors.

Lemma 1 *Let $A \in \mathbb{Q}^{n\times n}$. Let $P \in \mathbb{Q}^{n\times n}$ be nonsingular. PAP^{-1} and A have the same set of eigenvalues. If v is an eigenvector of A associated with the eigenvalue λ, then Pv is an eigenvector of PAP^{-1} corresponding to the same eigenvalue λ.*

This paper considers doubly nested loops of the following form:

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

In the loop body, $j = (j_1, j_2)$ 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:

$$\mu(j) \quad = \quad Uj + u$$

$$\nu(j) \;=\; Vj + v$$

where $U, V \in \mathbb{Z}^{2\times 2}$ are nonsingular and $u, v \in \mathbb{Z}^2$ are two-dimensional integer vectors.

The two array references in the program 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 $\delta(j) = \nu^{-1}(\mu(j))$. Thus, we have the following *dependence function*:

$$\delta(j) \;=\; \begin{bmatrix} \delta_1(j) \\ \delta_2(j) \end{bmatrix} \;=\; V^{-1}Uj + V^{-1}(\mu - \nu) = Dj + d$$

where D is called the *dependence matrix*. Since U and V are nonsingular, D is nonsingular. The following *relative dependence function* describes both the distance and direction of the dependences between the two points j and $\delta(j)$:

$$\rho(j) \;=\; \begin{bmatrix} \rho_1(j) \\ \rho_2(j) \end{bmatrix} \;=\; \delta(j) - j \;=\; (D - I)j + d$$

Let us relate our dependence abstraction with the traditional dependence vector abstraction [14]: $\rho(j)$ 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 $\rho(j) \prec 0$. If $\rho(j) \succ 0$, $\delta(j)$ is said to *depend* on j and this dependence is characterised by the dependence vector $\rho(j)$. If $\rho(j) \prec 0$, j is said to *(anti-) depend* on $\delta(j)$ and this dependence is characterised by the dependence vector $-\rho(j)$. When dependences are drawn 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 drawn in the iteration space are lexicographically positive.

According to the lexicographic sign of $\rho(j)$, the iteration space can be divided into the following three regions (each of which may be empty):

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

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 parallel execution.

Lemma 2 *The following properties about $\mathcal{J}_+$, $\mathcal{J}_-$ and $\mathcal{J}_0$ can be observed:*

(a) Every dependence vector contained in $\mathcal{J}_+$ is a flow dependence.

(b) Every dependence vector contained in $\mathcal{J}_-$ is an anti-dependence.

(c) Every dependence vector (flow or anti) that connects $\mathcal{J}_+$ and $\mathcal{J}_-$ originates from $\mathcal{J}_+$ and sink at $\mathcal{J}_-$.

(d) All points of $\mathcal{J}_0$ are independent of each other and of the points in $\mathcal{J}_+$ and $\mathcal{J}_-$.

Proof. The first two properties follow directly from the definitions of $\mathcal{J}_+$ and $\mathcal{J}_-$, respectively. The third property follows from the definitions of $\mathcal{J}_+$, $\mathcal{J}_-$ and the dependence vectors introduced previously. For every $j \in \mathcal{J}_+$, we have $\delta(j) - j \succ 0$. This dependence originates from $\mathcal{J}_+$ and must sink at $\mathcal{J}_-$ if it spans the two regions. For every $j \in \mathcal{J}_-$, we have $\delta(j) - j \prec 0$. This dependence sinks at $\mathcal{J}_-$ and must originate from $\mathcal{J}_+$ if it spans the two regions. For the fourth property, let us prove that $\mathcal{J}_0$ is self-dependent. It is obvious that $\delta(\mathcal{J}_0) \subseteq \mathcal{J}_0$ since for all $j \in \mathcal{J}_0$ we have $\delta(j) = j$. Moreover for $j \in \mathcal{J}$ such that $\delta(j) \in \mathcal{J}_0$, we have $\rho(\delta(j)) = 0$, which is equivalent to $(D-I)(Dj+d)+d = D^2j - Dj + Dd - d + d = D^2j - Dj + Dd = 0$. Since D is nonsingular, we obtain $Dj - j + d = (D-I)j + d = \rho(j) = 0$. Hence, $j \in \mathcal{J}_0$, which implies $\delta^{-1}(\mathcal{J}_0) \subseteq \mathcal{J}_0$. Adding this result to the fact that, by construction, a point of $\mathcal{J}_0$ is only dependent on itself, the fourth property has thus been established. ∎

In fact, $\mathcal{J}_0$ contains all fixed points to the equation $\rho(j) = 0$. If $D - I$ is nonsingular, $\mathcal{J}_0$ contains the singleton $j = (D-I)^{-1}d$ if $j \in \mathcal{J}$ and is empty otherwise. If $D - I$ is singular, there are three cases. If $D = I$, $\mathcal{J}_0 = \mathcal{J}$ if $d = 0$ and is empty otherwise. If $D \neq I$ and d is linearly dependent on the two columns of $D - I$, then $\mathcal{J}_0$ contains all ponts on the line $\rho_1(j) = 0$ (or equivalently, $\rho_2(j) = 0$). Otherwise, $\mathcal{J}_0$ is empty.

Let us use an example to illustrate the concepts introduced so far. This example will also be used for illustrations in the remaining part of the paper.

Example 2 Consider a double loop:

$$
\begin{array}{l}
\textsf{for } (j_1 = -N;\ j_1 \leqslant N;\ j_1 ++) \\
\quad \textsf{for } (j_2 = -N;\ j_2 \leqslant N;\ j_2 ++) \\
\qquad A(4j_1 + 4j_2 - 3, j_1 + 3j_2) = 2 * A(j_1 + j_2, j_2 + 1)
\end{array}
$$

The array subscript functions μ and ν are:

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

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:

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

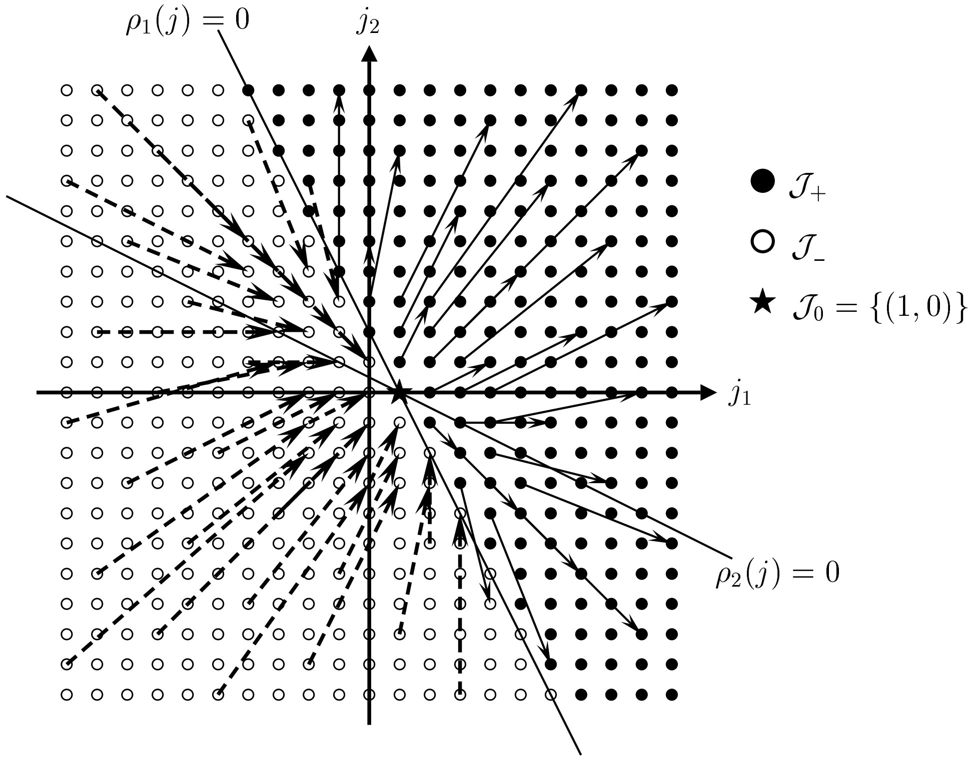

Figure 2: The iteration space of Example 2 ($N = 10$).

The two lines $\rho_1(j) = 0$ and $\rho_2(j) = 0$ separate the iteration space into the three non-empty regions $\mathcal{J}_+$, $\mathcal{J}_-$ and $\mathcal{J}_0$, as depicted in Figure 2. $\mathcal{J}_0 = \{(1, 0)\}$, and $\mathcal{J}_+$ has both flow dependences and anti-dependences originating from it and terminating at $\mathcal{J}_-$. **(End of Example)**

3 Eigenvectors-Based DOALL Parallelisation

First of all, we state the two major limitations of our method.

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

2. *D must be nonsingular*, implying that both U and V are nonsingular. In this case, every array element is read (or written) 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 an outline of our method and highlight one difference between our method and the unimodular approach. 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 $\forall\, j \in \mathcal{J}_- : \delta(j) \lessdot j$.

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 [14, p. 345]. 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):

   $$\boxed{\begin{array}{l}\mathsf{PAR}\\ \quad \mathsf{code}(\mathcal{J}'_0)\\ \quad \mathsf{SEQ}\\ \quad\quad \mathsf{code}(\mathcal{J}'_+)\\ \quad\quad \mathsf{code}(\mathcal{J}'_-)\end{array}} \tag{1}$$

   where the two constructs PAR and SEQ are borrowed from Occam [10]. The order for executing the three regions follows from Lemma 2. 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 $\forall\, j' \in \mathcal{J}'_- : \delta'(j') \lessdot j'$.*

Proof. Follows from the fact that T is nonsingular and Definition 1. ∎

Example 3 Consider Example 1 again. D is already lower triangular. With $T = I$, $\mathcal{J}'_+ = \mathcal{J}_+ = \mathcal{J}$, $\mathcal{J}'_- = \mathcal{J}_- = \emptyset$ and $\mathcal{J}'_0 = \mathcal{J}_0 = \emptyset$. By Theorem 1, any execution ordering on $\mathcal{J}'_+$ is legal as long as every iteration $j' = j$ is executed before $\delta'(j') = \delta(j)$. The parallel code in Example 1 can be generated using the method to be described in Section 3.3.1. **(End of Example)**

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:

$$\begin{aligned} \mu'(j') &= \mu(T^{-1}j') \\ \nu'(j') &= \nu(T^{-1}j') \end{aligned}$$

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:

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

and the relative dependence function becomes:

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

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:

$$\begin{aligned} \rho_1(T^{-1}j') &= 0 \\ \rho_2(T^{-1}j') &= 0 \end{aligned}$$

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

Let us recall the following result from linear algebra on which our method is based.

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 integer eigenvectors. Let $g = \gcd(v_{2,1}, v_{2,2})$. Let $T^{-1} = \begin{bmatrix} a & v_{2,1}/g \\ b & v_{2,2}/g \end{bmatrix}$ such that* $\det(T) = (av_{2,2} - bv_{2,1})/g = \pm 1$. *Then $D' = TDT^{-1} = \begin{bmatrix} \lambda_1 & 0 \\ c & \lambda_2 \end{bmatrix}$.*

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 the directions $\pm(1, 0)$.

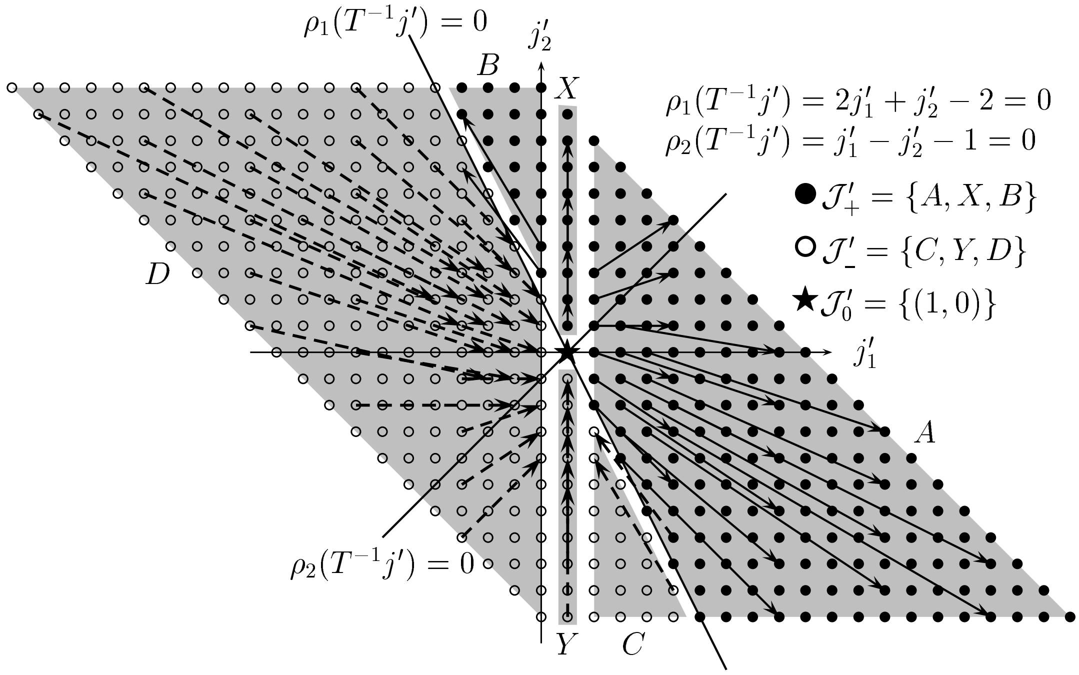

Figure 3: Transformed space of Figure 2 ($N = 10$). The division of the transformed space into six regions A, B, X, C, D and Y as highlighted will be explained in Section 3.3.1.

Example 4 Continuing from Example 2, we choose $(1, -1)$ as the parallelising eigenvector:

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

The new dependence function is calculated to be as follows:

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

The new relative dependence function becomes:

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

Figure 3 depicts the transformed space of the original iteration space in Figure 2. Unlike the traditional unimodular approach, some dependences in the transformed space are lexicographically positive and some can be negative. (**End of Example**)

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:

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

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.*

Proof. Let p and q be two points of the line L_1. Let the corresponding dependent points $\delta'(p) = D'p + d'$ and $\delta'(q) = D'q + d'$ be on a line L_2. Let L_1 be parallel to the eigenvector v of D', with eigenvalue λ. Thus we have: $p - q = av$, where $a \in \mathbb{Q}$, and $\delta'(p) - \delta'(q) = D'(p - q) = D'av = \lambda av$, implying that L_1 and L_2 are parallel. ∎

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. By using the T in Theorem 2 to transform the iteration space, we have implicitly assumed to use v_2 in that theorem 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 (i.e., happen to represent the same vertical line).

Theorem 4 *Consider the transformed space. If $\lambda_1 \neq 1$, $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ is the only self-dependent vertical line. If $\lambda_1 = 1$, all vertical lines are (not) self-dependent when $d'_1 = 0$ ($d'_1 \neq 0$).*

Proof. Follows from the fact that the first component of every dependence vector is $\rho'_1(j') = (\lambda_1 - 1)j'_1 + d'_1$ in $\mathcal{J}'_+$ and is $-\rho'_1(j') = -((\lambda_1 - 1)j'_1 + d'_1)$ in $\mathcal{J}'_-$. ∎

When $\lambda_1 \neq 1$, there is only one self-dependent vertical line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$. For the two dependent points j' and $\delta(j')$ that are not on the self-dependent line, the sign of λ_1 determines whether the two points are simultaneously on the same side of the self-dependent line and the magnitude of $|\lambda_1|$ determines which of the two points is further away from the self-dependent line. These two facts, summarised in Theorem 5, can be deduced from the following tautology:

$$(\lambda_1 j'_1 + d'_1) + \frac{d'_1}{\lambda_1 - 1} = \lambda_1(j'_1 + \frac{d'_1}{\lambda_1 - 1}) \qquad (5)$$

where the left-hand side represents the distance from $\delta_1(j') = \lambda_1 j'_1 + d'_1$ to the self-dependent line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ and the right-hand side represents the distance from j'_1 to the same self-dependent line.

Theorem 5 *Assume $\lambda_1 \neq 1$. Let j' be a point in the transformed space not on the self-dependent line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$. The following two statements are true:*

1. *j' and $\delta'(j')$ are on the same side of the line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ if $\lambda_1 > 0$ and on the opposite side of the line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ if $\lambda_1 < 0$.*
2. *$\delta(j')$ is further away from the self-dependent line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ than j' if and only if $|\lambda_1| > 1$.*

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

3.3 Loop Transformation: Parallelisation

In the previous 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 the parallel code template (1).

Our objective in DOALL parallelisation is to execute in the transformed space as many vertical lines in a single step as possible. Based on Theorem 5, 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) $\lambda_1 = -1$. The sign of λ_1 determines how the transformed iteration space is partitioned and the magnitude of $|\lambda_1|$ determines the order in which the vertical lines in a partition are executed.

3.3.1 $\lambda_1 > 1$

An example is used to illustrate the basic idea only. Consider the transformed space depicted in Figure 3 from Example 4. The dependence function in the transformed space is given in (3). 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 Figure 3.

By Theorem 5(a), the three regions A, B and X that partition $\mathcal{J}'_+$ are mutually independent. By Theorem 5(b), all (flow) dependences are pointing away from the self-dependent line. In our formalism, for every j' in A (B), the corresponding dependent $\delta'(j')$ is farther away from the self-dependent line. This suggests the following parallelisation scheme. We execute A by running in parallel, first lines from $j'_1 = t_1 = 2$ up to but excluding $j'_1 = t_1 * \lambda_1 - d'_1 = 2 * 4 - 3 = t_2 = 5$, then lines from $j_1'=t_2=5$ up to but excluding $j_1' = t_2 * \lambda_1 - d_1' = 5*4-3 = 17$, and so on. Similarly, we execute B by running in parallel, first lines from $j_1' = 0$ up to but excluding $j_1'=0*4-3=-3$, then lines from $j_1' = -3$ up to but excluding $j_1' = -3*4-3 = -15$, and so on. The idea of parallelising 2-D spaces A and B generalises to 1-D spaces such as X. We execute X by running in parallel, first point $(1,1)$, then points $(1,2)$ and $(1,3)$, and so on.

```
code(J'_+): PAR
        code(A) : for (t = 2; t ⩽ 2N; t = 4t − 3)
                    forall (j'_1 = t; j'_1 ⩽ min(4t − 4, 2N); j'_1 ++)
                      forall (j'_2 = max(−2j'_1 + 2, −N); j'_2 ⩽ −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 ⩾ ⌈(3−N)/2⌉; t = 4t − 3)
                    forall (j'_1 = t; j'_1 ⩾ max(4t − 2, ⌈(3−N)/2⌉); j'_1 −−)
                      forall (j'_2 = −2j'_1 + 3; j'_2 ⩽ N; j'_2 ++)
                        A(4j'_1 − 3, j'_1 − 2j'_2) = 2 * A(j'_1, −j'_2 + 1)
        code(X) : for (t = 1; t ⩽ N; t = 2t)   /* j'_1 = 1 */
                    forall (j'_2 = t; j'_2 ⩽ min(2t − 1, N); j'_2 ++)
                      A(1, 1 − 2j'_2) = 2 * A(1, −j'_2 + 1)
```

The step size of loop t is $\delta_1'(t)$ in both code(A) and code(B), and $\delta_2'(t)$ in code(X).

$\mathcal{J}'_-$ is parallelised in the same way as $\mathcal{J}'_+$, except that in the regions C and D, the vertical lines farther away from the self-dependent line $j_1' = 1$ are executed earlier, and in Y, the points farther away from $(1,0)$ are executed earlier.

```
code(J'_-) : PAR
        code(C) : for (t = ⌊(N+1)/2⌋; t ⩾ 2; t = ⌊(t+3)/4⌋)
                    forall (j'_1 = t; j'_1 ⩾ max(⌊(t+3)/4⌋ + 1, 2); j'_1 −−)
                      forall (j'_2 = −N; j'_2 ⩽ 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 ⩽ 0; t = ⌈(t+3)/4⌉)
                    forall (j'_1 = t; j'_1 ⩽ min(⌈(t+3)/4⌉ − 1, 0); j'_1 ++)
                      forall (j'_2 = −j'_1 − N; j'_2 ⩽ 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 ⩽ −1; t = ⌈t/2⌉)   /* j'_1 = 1 */
                    forall (j'_2 = t; j'_2 ⩽ min(⌈t/2⌉ − 1, −1); j'_2 ++)
                      A(1, 1 − 2j'_2) = 2 * A(1, −j'_2 + 1)
```

The step size of loop t is $\delta_1'^{-1}(t)$ in both code(C) and code(D), and $\delta_2'^{-1}(t)$ in code(Y).

Unlike the existing unimodular approach [6, 13, 14], our eigenvector-based method schedules far more hyperplanes of iterations of loop j_1' for concurrent execution.

3.3.2 $0 < \lambda_1 < 1$

This is the opposite of the case when $\lambda_1 > 1$ in the sense that, by Theorem 5(b), all flow dependences in $\mathcal{J}'_+$ are pointing toward the self-dependent line $j_1' = -\frac{d_1'}{\lambda_1 - 1}$ 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 $\lambda_1 > 1$.

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:

$$\delta(j) = Dj + d = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} j + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

D has one eigenvalue $1/3$ with two associated eigenvectors $(1, 0)$ and $(0, 1)$. We can calculate that $\mathcal{J}_+ = \emptyset$, $\mathcal{J}_- = \mathcal{J}$ and $\mathcal{J}_0 = \emptyset$. With $T = I$, $\mathcal{J}'_- = \mathcal{J}' = \mathcal{J}$. The parallelisation of $\mathcal{J}'_-$ in this case is the same as $\mathcal{J}'_+$ in the case when $\lambda_1 > 1$ as discussed in Example 1. For this particular example, the same code as in Example 1 is produced.

As a second example, consider a double loop derived from Example 2 with the two references swapped. The dependence function is the inverse of the one in Example 2:

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

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

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

in the transformed space as in Figure 3. So $\mathsf{code}(\mathcal{J}'_+)$ ($\mathsf{code}(\mathcal{J}'_-)$) in this case is the same as the $\mathsf{code}(\mathcal{J}'_+)$ ($\mathsf{code}(\mathcal{J}'_-)$) for Example 2 but with loop t reversed and loop j'_1 modified accordingly.

3.3.3 $\lambda_1 = 1$

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 how X and Y in Figure 3 are 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.

Example 5 Consider the following example from [12], where N was set to 1000:

```
for (j1 = 1; j1 <= N; j1 ++)
   for (j2 = 1; j2 <= N; j2 ++)
      A(j1 + j2, 3j1 + j2 + 3) = 2 * A(j1 + j2 + 1, j1 + 2j2 + 4)
```

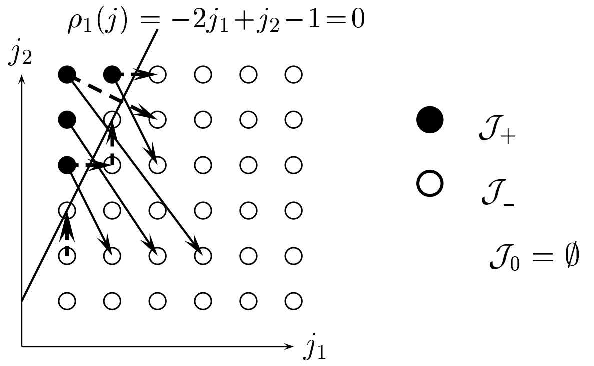

Figure 4: The iteration space of Example 5.

The dependence function is:

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

The two eigenvalues of D are 1 and -2, corresponding to the eigenvectors $(1,-1)$ and $(1,2)$, respectively. Since $D-I$ is singular and $\mathrm{rank}(D-I,d] = 2$, we have $\mathcal{J}_0 = \emptyset$. As shown in Figure 4, $\mathcal{J}_+$ 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 $\rho_1(j) = -2j_1+j_2-1 \leqslant 0$ (Figure 4).

With $(1,-1)$ chosen as the parallelising eigenvector, we apply the T given in (2) to transform the iteration space. The dependence function in the transformed space as shown Figure 5 is:

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

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

```
code(J'_+) :  for (j'_1 = ⌊(3N−2)/2⌋; j'_1 ⩾ 5; j'_1 −−)
                 forall (j'_2 = max(1 − j'_1, −N); j'_2 ⩽ ⌊(−2i−2)/3⌋; j'_2 ++)
                    A(j'_1, 3j'_1 + 2j'_2 + 3) = 2 ∗ A(j'_1 + 1, j'_1 − j'_2 + 4)
code(J'_-) :  for (j'_1 = 2; j'_1 ⩾ 2N; j'_1 ++)
                 forall (j'_2 = max(1 − j'_1, ⌈(−2j'_1−1)/3⌉, −N); j'_2 ⩽ 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)
```

(**End of Example**)

3.3.4 $\lambda_1 < -1$

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

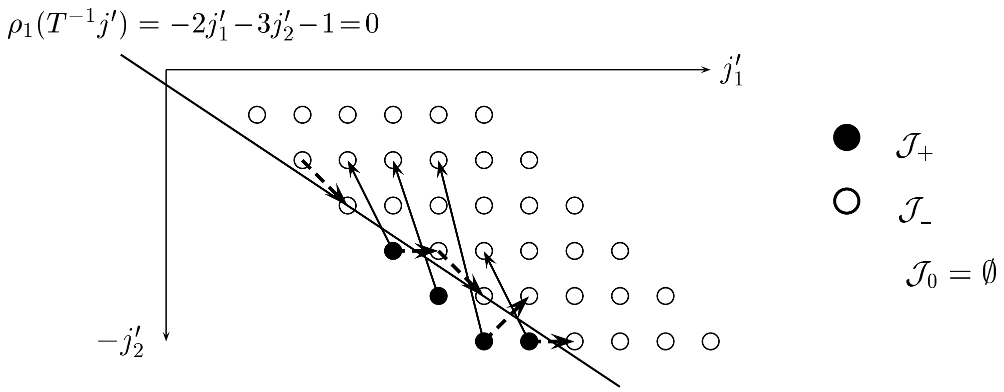

Figure 5: The transformed space of Figure 4 using (2).

To parallelise $\mathcal{J}'_+$, we divide it into two regions: A contains the points on $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$ and B contains the points not on $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$. By Theorem 4, A and B are independent. By Theorem 5(a), all dependences in B cross the self-dependent line $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$. By Theorem 5(b), all (flow) dependences are pointing away from the self-dependent line. 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 $j'_1 = -\frac{d'_1}{\lambda_1 - 1}$:

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

where $(a, b]$ ($[a, b)$) denote all vertical 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 that the order for executing the above strips is reversed.

Example 6 Consider Example 5 again. We can execute the program in three steps if $(1, 2)$ is selected as the the parallelising eigenvector. The matrix T is:

$$T = \begin{bmatrix} -2 & 1 \\ 1 & 0 \end{bmatrix}, \quad T^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \tag{6}$$

The dependence function in the transformed space as shown in Figure 6 is:

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

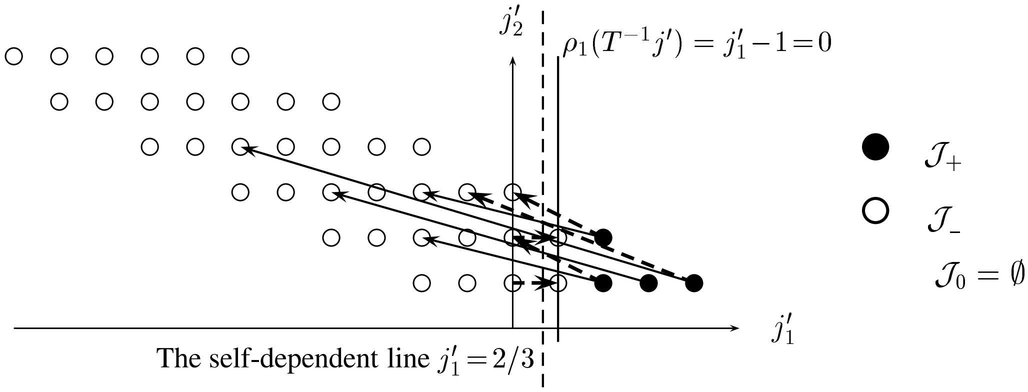

Figure 6: The transformed space of Figure 4 using (6).

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(a). The self-dependent 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 the line $j_1' = 1$ executed in the first step and those on the line in the second step.

In the original iteration space (Figure 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 [12]. **(End of Example)**

3.3.5 $-1 < \lambda_1 < 0$

This is opposite of the case when $\lambda_1 < -1$ in the sense of Theorem 5(b). So $\mathcal{J}_+'$ ($\mathcal{J}_-'$) in this case can be parallelised in the same way as $\mathcal{J}_-'$ ($\mathcal{J}_+'$) in the case when $\lambda_1 < -1$.

3.3.6 $\lambda_1 = -1$

By Theorem 4, the only self-dependent 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(a) 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 Related Work

There has been a great deal of research on applying loop transformations to optimise loop-oriented programs to expose parallelisim or otherwise improve data locality. These programs typically spend a considerable amount of time operating on arrays in loop nests. Unimodular loop transformations [1, 6, 14] can be used to transform a loop nest such that the transformed program consists of a sequential loop nest followed by a sequence of DOALL loops. In its full generality, the unimodular approach works only if all dependences in the loop nest are constant (i.e., $D = U = V = I$ in the notations of this paper) [13, 16]. In the case of affine dependences, there are two parallelisation strategies. One approximates the affine dependences with constant dependences and then applies the unimodular approach [16]. The other introduces constant dependences to uniformise a loop nest so that the uniformised program can be further parallelised [9, 11, 12]. In both cases, the artificial dependences introduced may constrain the amount of parallelism inherent in the original program.

The problem of transforming affine dependences into constant dependences has been the subject of study for many years in the systolic array community [8, 9, 15]. However, the focus there is to find a time schedule and a processor allocation so that the final transformed program represents a systolic array. This transformed program consists of essentially a sequential loop followed by a sequence of DOALL loops [4, 7].

The problem of parallelising the class of programs considered in this paper has been studied by several researchers [3, 5, 11, 12, 17]. Tzen and Ni [12] propose a dependence uniformisation technique and an index sychronisation scheme to reduce the synchronisation overhead. However, the artificial dependences they introduce can constrain the amount of parallelism inherent in the original program. For example, our parallelised program shown in Figure 6 for Example 5 runs in three steps while theirs requires index synchronisations to be executed inside the outer DOACROSS loop. Chen and Yew [3] later improve the dependence uniformisation technique so that more parallelism can be extracted. However, the artificial dependences they introduce still reduce the extractable parallelism in the original program. Shang, Hodzic and Chen [11] first uniformise the program and then find an optimal time scheduling for the uniformised program. In comparison with the unimodular approach, they can use more than one wavefront to parallelise the uniformised program. Zaafrani and Ito [17] divide the iteration space into two parallel regions and one sequential region according to the dependences of the program. The amount of parallelism exposed is limited by the sequential region. All these methods except [11] characterise the dependences of the program by their convex hull and attempt to extract as much parallelism as possible from the program. Finally, Ju and Chaudhary improve all these methods by identifying more parallel regions based on a more precise charaterisation of the data dependences. Unlike these previous methods, our method is centered around the concepts of eigenvectors and eigenvalues. For the class of programs considered in the paper, we can discover more parallelism than these previous methods in such a way that progressively more wavefronts can be executed in parallel.

5 Conclusion

In this paper, we studied 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 concepts of eigenvalues and 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. In general, our method discovers far more parallelism than any existing hyperplane-based unimodular approach.

The methodology outlined in this paper generalises directly to n-dimensional nested loops. One future work is to extend our method to handle the case with several pairs of references. Another future work is to take into account architecture-dependent details such as communication and synchronisation costs when parallelising a program.

6 Acknowledgements

Thanks to the referees for their constructive comments. This work is supported by an Australian Research Council Grant A10007149.

References

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundation*. Kluwer Academic Publishers, 1993.

[2] U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.

[3] D.-K. Chen and P.-C. Yew. On effective execution of non-uniform DOACROSS loops. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):463–476, 1996.

[4] P. Feautrier. Automatic parallelization in the polytope model. In G. R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer Verlag, 1996.

[5] J. Ju and V. Chaudhary. Unique sets oriented parallelization of loops with non-uniform dependences. *The Computer Journal*, 40(6):322–338, 1997.

[6] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, Feb. 1974.

[7] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer Verlag, 1993.

[8] G. M. Megson and L. Rapanotti. Uniformization techniques for reducible integral recurrence equations. In *Algorithms & Parallel VLSI Architectures III*, pages 283–296. Elsevier Science Publishers B. V., 1995.

[9] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, Oct. 1989.

[10] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177ff., 1988.

[11] W. Shang, E. Hodzic, and Z. Chen. On uniformization of affine dependence algorithms. *IEEE Trans. on Computers*, 45(7):827–839, Jul. 1996.

[12] 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.

[13] 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.

[14] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[15] Y. Wong and J. M. Delosme. Transformation of broadcasts to propagations in systolic algorithms. *J. Parallel and Distributed Computing*, 14(2):121–145, Feb. 1992.

[16] Y. Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. In *7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, Aug 1994.

[17] A. Zaafrani and M. Ito. Parallel region execution of loops with irregular dependences. In *Int. Conference on Parallel Processing*, pages 11–19, 1994.