Geometric Scheduling of 2-D Uniform Dependence Loops *

Ioannis Drositis, Theodore Andronikos, Aggelos Kokorogiannis,
George Papakonstantinou and Nectarios Koziris

Computing Systems Laboratory,
Computer Science Division,
Department of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou Campus, 15773 Zographou, Athens, Greece

Email: {jdros}@cslab.ntua.gr

Abstract

One of the primary tasks in the area of uniform dependence loops, is predicting the execution propagation, as well as finding an optimal time schedule. In this work, the problem of scheduling using wavefront prediction is presented. The geometric concepts of time instance subspaces and execution pattern are introduced. A quite simple and low complexity scheduling algorithm is presented. The index space is split into geometric subspaces and any point can be located in them. Each point is then scheduled according to the subspace where it belongs.

***Keywords**: Execution pattern, uniform dependence loop, scheduling.*

1 Introduction

One of the primary tasks when dealing with a nested loop, is scheduling the computations, while preserving the data dependences imposed by the initial (lexicographic) loop ordering. Predicting the execution wavefront is quite promising, as it can lead up to new scheduling prospectives.

The consept of using a linear transformation for time scheduling was introduced by Lamport [9] with the idea of *hyperplane*. As the hyperplane method was prooved to be nearly optimal from Darte [5], many researchers focused on it and extend it quite a lot [11, 6].

On the other hand, general scheduling approaches were never used in loop parallelization. General multiprocessor scheduling with precedence constrains was proved to be NP-complete, even when the number of available processors is unbounded [10]. However, in our case, things are much more simple, as the index space is usually limited. Other researchers model the index space *Directed Acyclic Graph* (DAG) of Unit Execution Time (UET) and the communication delays between computations (which are to be executed on different processors) are ignored (e.g. [8]). Restricting to constant or small communication delays (SCT), decreases the complexity, resulting in polynomial time solutions as in [4].

Our paper presents a new geometric methodology of predicting the execution propagation of a uniform nested loop index space. Due to the existence of dependence vectors, only a certain set of points can be executed at every moment. The geometric border of this set of points forms a polygonal shape called *pattern* with special attributes and characteristics. We shall describe how these patterns are formed, the rules that they follow and whether they can be computed non-exhaustively. Whether we can predict their shape and at what complexity cost.

Under a different perspective, the index space can be partitioned into a sequence $T_1, T_2, \ldots$ of (disjoint) subsets, such that T_n contains precisely those points whose earliest computation time is n. By definition, the (pre-computed) boundary points form T_0. We prove that the aforementioned wavefront-like polygonal pattern is precisely the geometric representation of the above subsets (see Figure 1).

Our goal is to partition the index space into patterns and then schedule points according to the pattern they belong to. For example, if $\mathbf{j} = (j_x, j_y)$ is included in pattern k, it can be computed at $t_j = k$ (Figure 1). Basic terminology and definitions are presented in Section 2, while principal study follows in Sections 3 and 4.

*This work was partially supported by the General Secretariat of Research and Development.

2 Terminology and Definitions

In this paper, we focus on 2-dimensional uniform dependence loops, which have the following form:

```
for (i1=0; i1 <= u1; i1++)
     ...
     for (in=0; in <= un; in++) {
          S1(I);
          S2(I);
          ...
     } ...  }
```

where $S_1, \ldots, S_k$ are assignment statements and $J = \{(i_1, \ldots, i_n) \mid 0 \leq i_r \leq u_r, 1 \leq r \leq n\} \subset \mathbf{N}^2$ is the index space. We intend to generalize our results on higher dimensions in a subsequent paper.

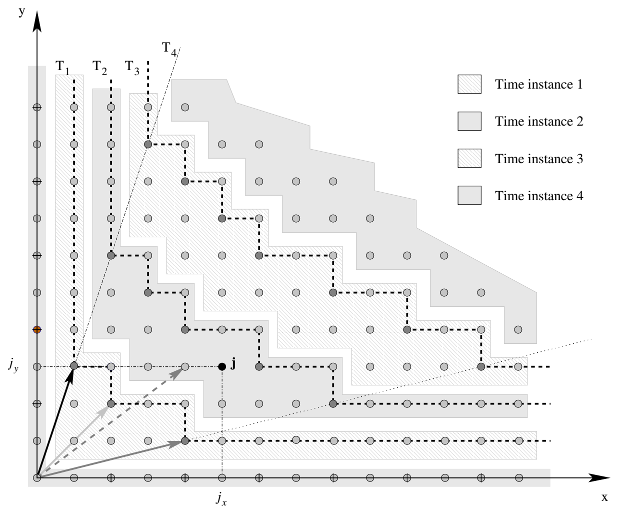

Figure 1: *Consider an algorithm with 3 dependence vectors* $\mathbf{d}_1 = (1,3)$, $\mathbf{d}_2 = (2,2)$, $\mathbf{d}_3 = (4,1)$ *and* $\mathbf{d}_4 = (4,3)$. *Each shaded polygon depicts the subset of J that can be executed at a certain moment. Patterns for moments 1-4 are shown. The dashed lines mark the pattern borders, while on the axes lie the boundary points.*

Our scheduling policy executes every index point at its *earliest computation time* (ECT). This policy guarantees the optimal execution time of the entire loop. Consider any doubly nested loop along with its index space and its uniform dependence vectors. We proceed now to define the most important concepts of our work:

- The index space is partitioned into disjoint **time subsets** denoted T_k, $k \geq 0$, such that T_k is the subset of J, with earliest computation time k. T_0 denotes the boundary points (all points that lie on the axes).

- The geometric shape of T_1 is called **initial pattern** (see Figure 1) and denoted as Pat_1.

- We call Pat_k the **pattern corresponding to moment** k. The intuition behind the pattern notion is that Pat_k is the geometric representation of T_k. We also define Pat_0 as $\{(0,0)\}$. Figure 1 depicts the patterns of moments $1, 2, 3$ and 4.

- **Pattern points** are the ones that define the polygon shape of the pattern. The remaining points just complete the polygon area. **Pattern vectors** are the dependence vectors whose end-points are the pattern points of the initial pattern.

- **Dependence difference vectors** are the difference vectors of the m pattern vectors. They are at most $m-1$ and are defined as: $\mathbf{a}_1 = \mathbf{d}_2 - \mathbf{d}_1, \mathbf{a}_2 = \mathbf{d}_3 - \mathbf{d}_1, \ldots, \mathbf{a}_{m-1} = \mathbf{d}_m - \mathbf{d}_1$.

By the term *pattern boundary* we usually refer to pattern points only, which are painted in darker shade in all figures. *Time instance* is also referred to as *step* or *moment*.

3 Analytical Pattern Computation

The geometric procedure of the pattern formation is briefly present bellow. Certain computational geometry algorithms are used, which are described in [7]. The crucial feature is that the general pattern (that of an arbitrary moment) depends *only* on the initial pattern form.

The initial pattern contains all points that can be computed initially ($t = 1$). Each dependence vector imposes a certain restriction, so only a subspace of J can be executed, due to each vector. By intersecting these subspaces of all dependence vectors initial pattern is formed.

A similar (subspace-intersection) procedure can be followed for the general k-moment pattern, by considering all subspaces that contribute to its pattern formation. This procedure implies the (obvious) method of finding the pattern exhaustively, by spanning the pattern vectors from the beginning of the axes. In other words, the pattern vectors are the ones that produce the *innermost concave wavefront* of points that are ready for execution. Step k pattern is formed by following the pattern vectors, starting from ($k-1$)-step pattern.

Although its complexity is clearly *exponential*, we use the results of this procedure as a point of reference because of its simplicity and accuracy. The dissadvantage of this method is that point-location, and consequently scheduling, are not so straightforwardly and efficiently derived (in the general case).

In [7] the formal description of the general k-step pattern is analysed. For a general algorithm with m pattern vectors, the k-step pattern-boundary points are given from the following expression: $\mathbf{j} = k \cdot \mathbf{d}_{xmin} + k_1 \cdot \mathbf{a}_1 + k_2 \cdot \mathbf{a}_2 + \ldots +$

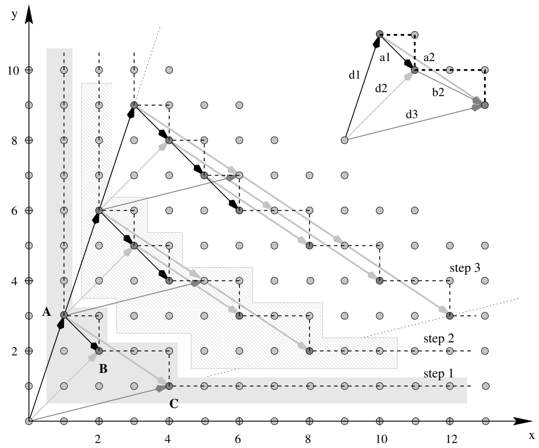

Figure 2: *Formation of the 3 first pattern boundary points by the difference vectors* $\mathbf{a}_i$*, i = 1, 2, ..., m-1 and* $\mathbf{d}_{xmin} \equiv \mathbf{d}_1$*. Every point* $\mathbf{j} \in Pat_1$ *is given from* $\mathbf{j} = \mathbf{d}_{xmin} + k_1 \cdot \mathbf{a}_1 + k_2 \cdot \mathbf{a}_2$*, where* $k_1, k_2 \in N$ *and* $k_1 + k_2 \leq 1$.

$k_{m-2} \cdot \mathbf{a}_{m-2} + k_{m-1} \cdot \mathbf{a}_{m-1}$, where $\mathbf{d}_{xmin}$ is the pattern vector with the minimum x-coordinate, $k_1, k_2, \ldots, k_{m-1} \in N$ and $k_1 + k_2 + \ldots + k_{m-1} \leq k$ (see Figure 2). Consequently, the formal description of the problem can be stated as:
Scheduling problem: *given any index point* $\mathbf{j}_i = (x_i, y_i)$*, find its earliest computation time k, according to the following constrains:*

- $k = k_1 + k_2 + \ldots + k_{m-1}$
- $k \cdot d_{xmin_x} + k_1 \cdot a_{1_x} + k_2 \cdot a_{2_x}, \ldots, k_{m-2} \cdot a_{m-2_x} + k_{m-1} \cdot a_{m-1_x} \leq x_i$
- $k \cdot d_{xmin_y} + k_1 \cdot a_{1_y} + k_2 \cdot a_{2_y}, \ldots, k_{m-2} \cdot a_{m-2_y} + k_{m-1} \cdot a_{m-1_y} \geq y_i \wedge k = minimum.$

From the description of the constrains, one can see that it can be solved analytically using *integer linear programming* techniques.

4 Polynomial Pattern Computation

In this section we present an efficient algorithm that calculates the formation of any k-step pattern. Although the exact border of the pattern is not always calculated precisely, the subset of points that belong to each pattern can be executed at the same moment. This in effect provides an efficient solution to our problem.

It is proven in [7] that this scheduling technique is asymptotically optimal, in the sense that the execution time assigned to each point is *at most one* more than its earliest computation time. In Figure 3, the exhaustive methodology that described previously, as well as the polynomial one are shown for a given doubly nested loop. Our enables an efficient scheduling.

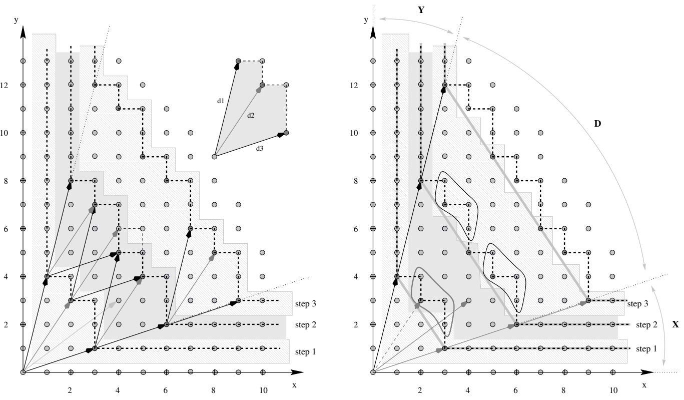

Figure 3: *Pattern formation by the exhaustive and the non-exhaustive method for an example algorithm with 4 dependence vectors.* ***a****) By the exhaustive method (left) the accurate pattern is computed, but a point-locating and scheduling algorithm is not easily achieved.* ***b****) Our method (right), although it computes a trapezoid (less accurate) k-step pattern, leads to an efficient scheduling. The points mentioned are the ones omitted for the next time instance.*

As it is shown in Figure 3, the index space is split into several cone subspaces (which are angles in 2-D). Although we are not following a recursive procedure, we can predict the *outline* of the k-step pattern. Certain geometric subpatterns, called *regions*, are formed. These regions have simpler representation than the accurate patterns, which simplifies point-location and thus scheduling.

4.1 Some More Definitions

Our method uses the dependence vector set. Given the m dependence vectors $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m$, we give the following definitions:

Cone: The convex subspace that is formed by two vectors $\mathbf{a}$ and $\mathbf{b}$ (in 2-D) is called a *cone* and is denoted $Con(\mathbf{a}, \mathbf{b})$. Formally, $Con(\mathbf{a}, \mathbf{b}) = \{\mathbf{j} \in N^2 \mid \mathbf{j} = \lambda_1 \cdot \mathbf{a} + \lambda_2 \cdot \mathbf{b}$, where $\lambda_1, \lambda_2 \geq 0\}$.

Y-Cone: The cone formed by the y-axis and the pattern vector with the least x-coordinate $\mathbf{d}_{xmin}$ (Figure 3).

X-Cone: The cone formed by the pattern vector with the least y-coordinate $\mathbf{d}_{ymin}$ and the x-axis.

D-Cone: The cone formed by vectors $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$: $Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$.

Cone Vectors: Among pattern vectors, some of them have the following properties:

- all other pattern vectors can be written as a linear combination of *exactly one* pair of them with positive coefficients

- the sum of these positive coefficients is always greater than 1.

In other words, the *cone vectors* form a sort of "base set" for the pattern vectors, and they are unique for a certain set of dependence vectors (see Figure 4). Note that $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ are always included in CV.

Cone Region t: Region t of a cone $Con(\mathbf{a}, \mathbf{b})$ is denoted as: $Con_i(\mathbf{a}, \mathbf{b}) = \{\mathbf{j} \in Con(\mathbf{a}, \mathbf{b}) \mid \mathbf{j} = \lambda_1 \cdot \mathbf{a} + \lambda_2 \cdot \mathbf{b}$, where $i < \lambda_1 + \lambda_2 \leq i + 1\}$ (Figure 4). Region Reg_t of moment t, is formed by combining the consequtive regions of all cones.

4.2 Patterns and Cones

The method presented in the following sections, is based on the *cone geometric theory* and it computes the k-step pattern outline. The index-space is divided into several *cones*. Every point located into such a cone, can be written as a linear combination of the vectors that form the cone, with *positive* coefficients. The cone vectors also determine the execution order among the points included in the cone, as they determine the optimal execution *hyperplane* for the cone, as proven in [1]. In other words, certain cone-subspaces, called *regions* are formed, as in Figure 4.

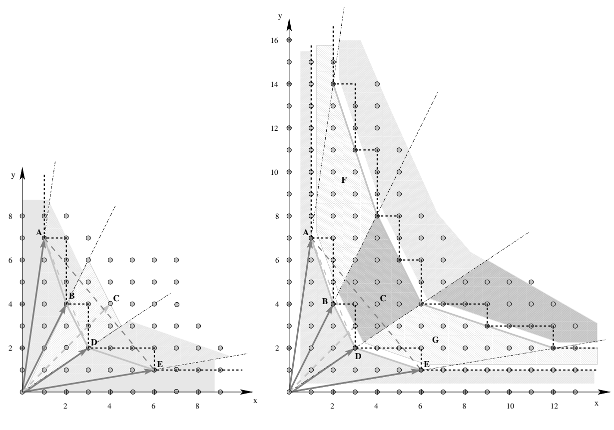

Figure 4: *The index space of an example algorithm with 5 dependence vectors* $\mathbf{d}_1 = (1, 7)$*,* $\mathbf{d}_2 = (2, 4)$*,* $\mathbf{d}_3 = (3, 2)$*,* $\mathbf{d}_4 = (4, 4)$ *and* $\mathbf{d}_5 = (6, 1)$*.* $CV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5 \rangle$*. The 5 cones that formed are shaded respectively (left). The first 3* regions *of all cones are also shown.*

Consider the cone formed by vectors $\mathbf{d}_2$ and $\mathbf{d}_3$ in Figure 4 and an index point $\mathbf{j}(j_x, j_y)$. If $\mathbf{j} = \lambda_1 \cdot \mathbf{d}_2 + \lambda_2 \cdot \mathbf{d}_3$, then we distinguish the following cases: **i)** if $\lambda_1 > 0$ and $\lambda_2 > 0$, $\mathbf{j}$ lies inside $Con(\mathbf{d}_2, \mathbf{d}_3)$ e.g. point $\mathbf{C} \in Con_1(\mathbf{d}_2, \mathbf{d}_3)$ and **ii)** if $\lambda_1 < 0$ or $\lambda_2 < 0$, $\mathbf{j}$ lies outside $Con(\mathbf{d}_2, \mathbf{d}_3)$ e.g. points $\mathbf{F}$ and $\mathbf{G}$.

If the cone that a given point is located into is known, we can retrieve the execution moment for it. The region that the point belongs to, denotes the time instance that the point can be executed. The region number for a given point can be computed as follows (see Figure 4):

$\triangleright$ ***if point*** $\mathbf{j} \in$ **Y*-cone*:** [region number] = $\lceil \frac{j_x}{d_{xmin_x}} \rceil$.

$\triangleright$ ***if point*** $\mathbf{j} \in$ **X*-cone*:** [region number] = $\lceil \frac{j_y}{d_{ymin_y}} \rceil$.

$\triangleright$ ***if point*** $\mathbf{j} \in Con(\mathbf{a}, \mathbf{b})$**:** [region number] = $\lceil \lambda_1 + \lambda_2 \rceil$, where $\mathbf{j} = \lambda_1 \cdot \mathbf{a} + \lambda_2 \cdot \mathbf{b}$, as above.

For more details on cones see [7] and [1].

4.3 Pattern-Outline Computation and Scheduling

The whole procedure is described by the following steps: **a)** *pattern vectors* are computed; **b)** *cone vectors* are computed; *–the index space is partitioned into cones and regions*; **c)** The actual *schedule* can be performed, by locating the given points among a pair of *cone* and *region*.

Each of the above steps is presented as a separate algorithm. Given a nested loop with a dependence vector set $\mathbf{D} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m\}$, the pattern vectors can be computed as follows.

Algorithm I *(Pattern-vector computation)*

INPUT: The m dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_m$ of the algorithm.

OUTPUT: Pattern vector list $PV_D = \langle \mathbf{d}_{p1}, \mathbf{d}_{p2}, \ldots, \mathbf{d}_{pl} \rangle$, where $l \leq m$.

1. Sort dependence vectors lexicographically, according to x and y and put the ordered set in a list, say DL. The pattern-vector list is currently empty: $PV_D = \emptyset$.
2. The first item DL is moved into PV_D. Now: $PV_D = \langle \mathbf{d}_{p1} \rangle$, $|PV_D| = 1$.
3. Starting from the 2nd item of DL, check its y-coordinate. If it is less than the y-coordinate of the vector last inserted in PV_D, insert vector in list also; else discard vector.
4. Repeat previous step for all remaining vectors in DL.
5. Pattern-vectors are in $PV_D = \langle \mathbf{d}_{p1}, \mathbf{d}_{p2}, \ldots, \mathbf{d}_{pl} \rangle$. Note that $\mathbf{d}_{p1} \equiv \mathbf{d}_{xmin}$ and $\mathbf{d}_{pl} \equiv \mathbf{d}_{ymin}$.

The complexity of the algorithm is $O(m \cdot log(m)) + O(m) = O(m \cdot log(m))$.

Computation of the *cone vectors* which follows, is the major part of the sceduling procedure. We present a computational geometry algorithm that computes them. It works with the output of of the previous algorithm PV_D and computes the *cone vector* list CV.

Algorithm II *(Cone-vector computation)*

DivideConeVector(PV_D, $\mathbf{d}_{xmin}$, $\mathbf{d}_{ymin}$)

INPUT: Pattern-vector list $PV_D = \langle \mathbf{d}_{p1}, \ldots, \mathbf{d}_{pl} \rangle$.

OUTPUT: Cone vector list $CV = \langle \mathbf{d}_{c1}, \mathbf{d}_{c2}, \ldots, \mathbf{d}_{ck} \rangle$, $k \leq l \leq m$.

1. Let $CV' = PV_D - \{\mathbf{d}_{xmin}, \mathbf{d}_{ymin}\}$.

2. Discard all vectors from CV' that their endpoints lie above the line that connects the endpoints of $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$.

3. If $CV' = \emptyset$ then return $\langle \mathbf{d}_{xmin}, \mathbf{d}_{ymin}\rangle$.
If $CV' = \langle \mathbf{d}_p \rangle$ then return $\langle \mathbf{d}_{xmin}, \mathbf{d}_p, \mathbf{d}_{ymin}\rangle$.

4. Let $s = (d_{ymin_y} - d_{xmin_y})/(d_{ymin_x} - d_{xmin_x})$; let $\mathbf{p}_i$ be the endpoint of a vector $\mathbf{d}_i \in PV_D$ that minimizes the expression $p_{i_y} - s \cdot p_{i_x}$ (this point is called *furthest point*).

5. Let $CV_l' = \{\mathbf{d} \in CV' : d_x < p_{i_x} \wedge \mathbf{d} \in \langle \mathbf{d}_{xmin}, \mathbf{d}_i \rangle\}$ and $CV_r' = \{\mathbf{d} \in CV' : d_x > p_{i_x} \wedge \mathbf{d} \in \langle \mathbf{d}_i, \mathbf{d}_{ymin}\rangle\}$.

6. Return the concatenation of $DivideConeVector(\langle \mathbf{d}_{xmin}, CV_l' \rangle, \mathbf{d}_{xmin}, \mathbf{d}_i)$ and $DivideConeVector(\langle CV_r', \mathbf{d}_{ymin}\rangle, \mathbf{d}_i, \mathbf{d}_{ymin})$.

The above algorithm follows the *divide-and-conquer* technique. The complexity achieved is $O(m \cdot log(h))$, where h is the number of cone vectors (usually quite smaller than m). For more details, one can refer to [7].

4.3.1 Pattern-Outline Computation

Starting from the index-space origin, we apply all cone-vectors. By connecting their endpoints, initial pattern outline is figured. By applying them twice and connecting their endpoints, the pattern of step $k = 1$ is formed and so on. The procedure is shown in Figure 4 and is formally described below.

Algorithm III *(k-step Pattern-Outline Computation)*
Input: Pattern-vector list PV_D and a step (integer) number $k \geq 0$.
Output: A set of points that constitute k-step pattern outline $Pat_{out}(k)$.

1. Compute algorithm cone-vectors, by performing *Algorithm II* on PV_D.

2. Apply all cone vectors k times to index-space origin –usually point $(0, 0)$. The outline of Pat_k, $Pat_{out}(k)$ consists of all these endpoints.

Note that $Pat_k \neq Pat_{out}(k)$; in other words, there are certain points in Pat_k that there are *not* included in $Pat_{out}(k)$. This accentuates the only dissadvantage of this method: some points are “placed” on $(k + 1)$-th region, instead of the k-th (Figure 3).

4.3.2 Scheduling

Any index space point $\mathbf{j}$ can be located into the formed regions, and thus correctly scheduled. The following algorithm has the details.

Algorithm IV *(Scheduling in a Multi-Cone Space)*
INPUT: Dependence vector set D and an index point $\mathbf{j}$.
OUTPUT: Time instance t_j that point $\mathbf{j}$ can be executed.

1. *Algorithm I* is applied on D and pattern vector list PV_D, as well as $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ are computed.

2. The l cone vectors are computed by applying *Algorithm II* onto PV_D. After this step: $CV = \langle \mathbf{d}_{c_1}, \mathbf{d}_{c_2}, \ldots, \mathbf{d}_{c_l}\rangle$, where $\mathbf{d}_{c_1} \equiv \mathbf{d}_{xmin}$ and $\mathbf{d}_{c_l} \equiv \mathbf{d}_{ymin}$.

3. For every cone-vector pair $(\mathbf{d}_{c_i}, \mathbf{d}_{c_{i+1}})$ of *subsequent* vectors $\mathbf{d}_{c_i}, \mathbf{d}_{c_{i+1}} \in CV$, solve the equation: $\mathbf{j} = \lambda_{i1} \cdot \mathbf{d}_{c_i} + \lambda_{i2} \cdot \mathbf{d}_{c_{i+1}}$ and compute λ_{i1} and λ_{i2}:
   - If for $(\mathbf{d}_{xmin}, \mathbf{d}_{c_1})$, $\lambda_{11} \geq 0$ and $\lambda_{12} < 0$: $\mathbf{j} \in Y$.
   - Else, if for $(\mathbf{d}_{c_{l-1}}, \mathbf{d}_{ymin})$, $\lambda_{l1} < 0$ and $\lambda_{l2} \geq 0$: $\mathbf{j} \in X$.
   - Else, if $\lambda_{i1} \geq 0$ and $\lambda_{i2} \geq 0$: $\mathbf{j} \in Con(\mathbf{d}_{c_i}, \mathbf{d}_{c_{i+1}})$.

4. Perform schedule for point $\mathbf{j}$:

   ▷ if $\mathbf{j} \in Y \Rightarrow t_j = \lceil \frac{j_x}{d_{xmin\,x}} \rceil = \lceil \lambda_{12} \rceil$

   ▷ if $\mathbf{j} \in X \Rightarrow t_j = \lceil \frac{j_y}{d_{ymin\,y}} \rceil = \lceil \lambda_{11} \rceil$

   ▷ if $\mathbf{j} \in Con(\mathbf{d}_{c_i}, \mathbf{d}_{c_{i+1}}) \Rightarrow t_j = \lceil \lambda_{i1} + \lambda_{i2} \rceil$.

One can note that the above scheduling algorithm performs a constant mistake: there are certain points that although they are ready for computation at time instance t_k, they are computed at the next time instance $t_k' = t_k + 1$. This is examplified in Figures 3 and 5.

It is proven in [2] that this side-effect doesn't affect optimality, as it results in a *constant* delay. The following example demonstrates *Algorithm IV* as well as the “mistake” just mentioned.

Example
Suppose an algorithm with dependence vectors $\mathbf{d}_1 = (1, 7)$, $\mathbf{d}_2 = (2, 4)$, $\mathbf{d}_3 = (3, 2)$, $\mathbf{d}_4 = (4, 4)$ and $\mathbf{d}_5 = (6, 1)$ (Figure 5). By applying first two steps of *Algorithm IV*, the cone-vector list is computed: $CV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5\rangle$. The index space can now be partitioned into 5 cones: Y, $Con(\mathbf{d}_1, \mathbf{d}_2)$, $Con(\mathbf{d}_2, \mathbf{d}_3)$, $Con(\mathbf{d}_3, \mathbf{d}_5)$ and X. Scheduling point $C(5, 5)$ poses the following equations: for $Con(\mathbf{d}_1, \mathbf{d}_2) \Rightarrow (\lambda_1, \lambda_2) = (-1, 3)$, for $Con(\mathbf{d}_2, \mathbf{d}_3) \Rightarrow (\lambda_1, \lambda_2) = (5/4, 5/8)$, for $Con(\mathbf{d}_3, \mathbf{d}_5) \Rightarrow (\lambda_1, \lambda_2) = (25/9, -5/9)$.

From the above, it is denoted that point C resides in cone $Con(\mathbf{d}_2, \mathbf{d}_3)$, and it is: $\lceil \lambda_1 + \lambda_2 \rceil = \lceil 15/8 \rceil = 2$; in other words, point C can be executed at time instance $t_C = 2$. Following the same procedure for points F, G

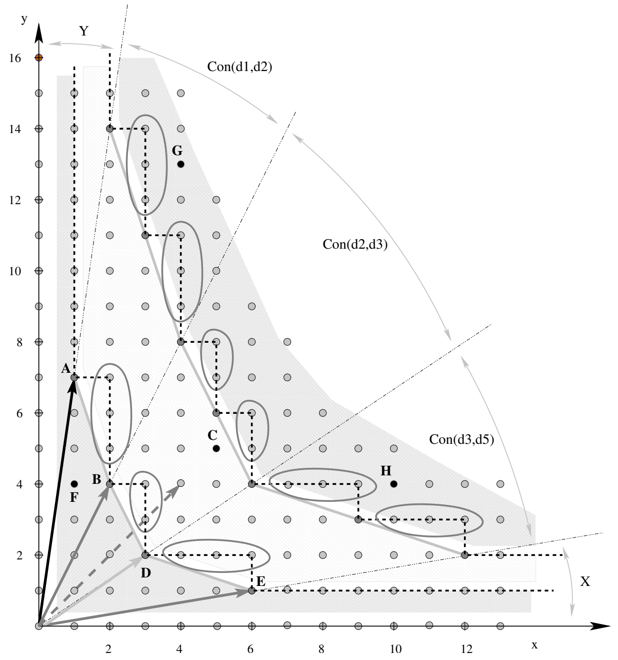

Figure 5: *The index space of an algorithm with 5 dependence vectors* $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$, $\mathbf{d}_4$ *and* $\mathbf{d}_5$*;* $CV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5 \rangle$*. The 5 cones that formed are shaded respectively and the first 3* regions *of all cones are shown. Delayed index points are indicated.*

and H, we can verify their computation moments: $F \in Con(\mathbf{d}_1, \mathbf{d}_2) \Rightarrow t_F = 1$, $G \in Con(\mathbf{d}_1, \mathbf{d}_2) \Rightarrow t_G = 3$ and $H \in Con(\mathbf{d}_3, \mathbf{d}_5) \Rightarrow t_H = 3$ Note at Figure 5 the index points that their execution is delayed by one time instance.

4.4 Computational Complexity of the Polynomial Computation

PATTERN-OUTLINE COMPUTATION: The complexity of *Algorithm III* will be studied. Computation of PV_D needs $O(m \cdot log(m))$ (*Algorithm I* must be performed). *Algorithm II* can be performed in $O(m \cdot log(h))$ time and $O(m)$ space, as proved in [3]; h is the number of the cone vectors (usually quite less than m). For the second step of *Algorithm III*, all cone-vectors should be applied $k + 1$ times to the origin of J, which needs $O((k+1) \cdot h)$ time. Supposing that $Cx(pout_k)$ is the complexity of k-step pattern-outline computation, it is: $Cx(pout_k) = O(m \cdot log(m)) + O(k \cdot h)$.

SCHEDULING: The complexity of *Algorithm IV* will be studied. CV_D computation needs $O(m \cdot log(h))$ time and $O(m)$ space, as before. Solving the $\lceil h/2 \rceil$ 2-D equations of step 3, needs $O(\lceil h/2 \rceil)$ time. So, scheduling a point in 2-D needs: $O(m \cdot log(m)) + O(k \cdot h)$ time.

These results prove the *polynomial* complexity of this method. Although they referred to 2-D problems, they mark out the efficiency of this technique, as far as scheduling is concerned.

5 Conclusion

In this paper we presented a nearly optimal scheduling method for uniform dependence nested loop algorithms. We described an exponential procedure for computing the precise k-step pattern boundary, which is the best we can get so far. Our major part was an assymptotically optimal scheduling method that uses geometry techniques.

We believe that our methodology can be generalized to handle an arbitrary n-dimensional uniform dependence loop and we are currently working towards this goal.

References

[1] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. An efficient algorithm for the optimal linear schedule of uniform dependence algorithms. In *Proceedings of the First ECPD International Conference on Advanced Robotics and Intelligent Automation*, pages 140–145, Athens, Greece, 1995.

[2] T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou, and P. Tsanakas. Lower bounds for parallel time and number of processors for the class of uniform dependence algorithms. Technical Report DSCLAB-TR-95-03, Digital Systems and Computers Laboratory - N.T.U.A., 1995.

[3] T. M. Chan. *Output-Sensitive Construction of Convex Hulls*. PhD thesis, The University of British Columbia, October 1995.

[4] P. Chretienne. Task scheduling with interprocessor communication delays. *European Journal of Operational Research*, 57:348–354, 1992.

[5] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1.2:73–81, 1991.

[6] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):814–822, 1994.

[7] I. Drositis, T. Andronikos, and G. Papakonstantinou. 2-d pattern prediction and scheduling. Technical Report CSLAB-TR-00-01, Computing Systems Laboratory - N.T.U.A., 2000.

[8] N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal time and efficient space free scheduling for nested loops. *The Computer Journal*, 39(5):439–448, 1996.

[9] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 37(2):83–93, February 1974.

[10] C. Papadimitriou and M. Yannakakis. Toward an architecture-independent analysis of parallel algorithms. *SIAM Journal of computers, Extended Abstract in Proceedings STOC 1988*, 19:322–328, 1990.

[11] W. Shang, M. T. O'Keefe, and J. Fortes. On loop transformations for generalized cycle shrinking. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):193–204, 1994.