On Parallelization of UET/UET-UCT Loops*

George Papakonstantinou, Theodore Andronikos
and Ioannis Drositis

Computing Systems Laboratory,
Computer Science Division,
Department of Electrical and Computer Engineering,
National Technical University of Athens,
Zographou Campus, 157-73, Athens, Greece

Email: {papakon,jdros}@cslab.ntua.gr

Abstract

Finding an optimal time schedule, as well as predicting the execution propagation is one of the primary tasks in the area of parallelizing uniform dependence loops. Due to the existence of dependence vectors, the index space of such a loop, is split into subspaces of points that can be executed at different time instances. The geometric representation of these sets form certain polygonal shapes called *patterns*, with special attributes and characteristics. In this paper, we give a closed formula of the pattern wavefronts, as well as a condition for pattern outline uniformity. Then we present a scheduling technique that is based on the geometric attributes of the index space and the dependence vector set. Our strategy can be applied to architectures that consider unit execution-zero communication delay (UET) or unit execution-unit communication (UET-UCT) model, as a new method for transforming UET-UCT problems to UET equivalent ones is presented.

Keywords: Execution pattern, wavefront, UET-UCT loop scheduling.

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

1 Introduction

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

Iterations are considered as points in a multidimensional index space, and the dependence vectors are considered as directed arcs between the corresponding indexed points. We are focusing on iteration level parallelism.

Time scheduling of loop-based iterations with uniform dependencies, was introduced by Karp in [18] (for uniform recurrence equations) and elaborated by Lamport in [21], who partitioned the index space into hyperplanes. All iteration-indexed points belonging to the same hyperplane can be concurrently executed. Darte in [9], proved that this method is nearly optimal, while the problem of finding the hyperlane that results in the minimum makespan was solved in [1]. Moldovan, Shang, Darte and others applied the hyperplane method to find a linear optimal execution schedule, using diophantine equations [22], linear programming in subspaces [28], integer programming [9], or graph-theoretic approaches [4]. All these approaches consider unit execution time for each iteration and zero communication for each communication step (UET model).

Linear schedules in loop parallelization were used in almost every attempt to map a nested loop into systolic architectures, like in [22, 30]. Linear transformations affect all indiced computations uniformly, thus producing time schedules which fit into systolic arrays without imposing artificial communication overhead or job management requirements, which would increase circuitry complexity. Nevertheless, due to linearity's uniformity, tackling all communications uniformly is not the optimal approach. Loop index spaces are sometimes irregularly bounded, and boundary effects and non-uniformities cannot be well organized by a uniform transformation. An optimal time schedule has to keep up with such irregularities and organize them properly, without destroying the algorithm's initial data dependencies.

To cope with a more general approach of the scheduling problem, considering also communication delays, many researchers have focused on task scheduling, which is one of the most important and difficult problems of parallel computing. The algorithm is modeled with a task graph, generally a directed acyclic graph (DAG), where the computation and communication times are depicted as node and edge weights, respectively. Since the general scheduling problem is well known to be NP-complete (see Ullman [32] and Garey [16]), researchers have given attention to other methods such heuristics, approximation algorithms etc. Papadimitriou and Yanakakis [23] proposed a clever heuristic which guaranteed worst performance twice the optimum makespan, for a task graph with

arbitrary communication and computation times.

When both computation and communication times are restricted to have unit time length, it is known that scheduling UET-UCT graphs with a bounded number of processors is NP-complete as proved by Rayward-Smith in [26] and by Picouleau in [24]. Even the case of unlimited processors, when task duplication is allowed, is in general polynomially intractable [24]; when the index space is limited another optimal approach in presented in [20]. Restricting to special cases decreases dramatically the complexity, resulting in polynomial time solutions as in [17, 8, 4].

In [7] Chretienne presented an algorithm linear in the cardinality of the vertices of the graph, for optimal makespan on SCT (small communication time, thus including UCT) in-trees and out-trees. Also, there exist polynomial optimal solutions for series-parallel digraphs, bipartite graphs and trees with UCT [33].

Although scheduling tree and graph-modeled algorithms in UET/UET-UCT continues, scheduling nested loops with UCT remains an open problem. In [13], a nearly optimal scheduling approach for 2-D UET uniform nested loops is presented, based on the geometric properties of the index space and the dependence vectors. The index space of a uniform dependence loop, can be partitioned into a sequence $ECT_0, ECT_1, \ldots$ of (disjoint) subsets, such that ECT_n contains precisely those points whose earliest computation time is n. By definition, the (pre-computed) boundary points form ECT_0. The geometric representation of these subsets form certain wavefront-like polygonal shapes called *patterns* (Figure 1).

Based on the geometric properties of the formed patterns, the aforementioned technique, partitions the index space into geometric regions and schedules computations according to the region they belong to.

In this work, we present an extension of the above approach, considering communication delays that may arise in a more general target architecture. We achieve this by introducing a method for transforming a UET-UCT problem to a UET equivalent one. This is the major contribution of this paper, as no other relative method has been proposed so far, to the best of our knowledge. All UET-UCT scheduling techniques refer to grid or extended grid architectures and to graph and tree algorithms [26, 24, 2].

The presented UET-UCT polynomial scheduling algorithm is proved to be nearly optimal, as it achieves a makespan of constant delay from the earliest possible. This algorithm can be generalized to apply to higher dimensional problems, as it is also shown. Our method has been implemented and tested in numerous examples, proving its superiority over existing methods.

Basic terminology and definitions are presented in Section 2, while principal study for the UET case follows in Sections 3 and 4. In Section 5 the UET-UCT case is studied. We conclude in Section 6.

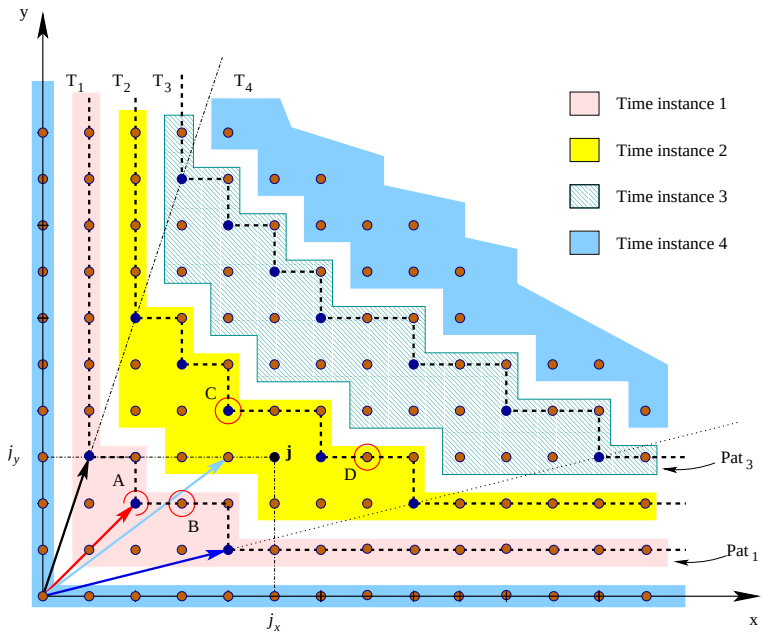

Figure 1: *Consider an algorithm with three dependence vectors* $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = (2, 2)$, $\mathbf{d}_3 = (4, 1)$ *and* $\mathbf{d}_4 = (4, 3)$. *Each shaded polygon depicts the subset of* J *that can be executed at a certain moment. This is what we call* pattern*; the four initial patterns are shown (for moments 1-4). The dashed lines mark the front of each pattern, called pattern outline. On the axes lie the pre-computed at moment* $t = 0$ *(boundary) points. One of the points included in* Pat_2 *is point* $\mathbf{j} = (j_x, j_y)$. *Note that* $\mathbf{d}_4$ *doesn't affect pattern border, as it's not a pattern vector.*

2 Terminology and Definitions

Uniform dependence loops, in the general n-dimensional case, have the following form:

```
for (i1=1; i1<=u1; i1++) {
    ...
    for (in=1; in<=un; in++) {
        S1(I);
        S2(I);
        ...
    }
    ...
}
```

where $S_1, \ldots, S_k$ are assignment statements and $J = \{(i_1, \ldots, i_n) \mid 1 \leq i_r \leq u_r, 1 \leq r \leq n\}$ is the index space, where $J \subset \mathbf{N}^n$. We follow the model that was introduced by Lamport in [21]; the dependence vectors are "uniform", i.e., constant all over the index space.

Due to the existence of dependence vectors, two arbitrary instances of an assignment statement S_r, i.e., $S_r(\mathbf{j})$ and $S_r(\mathbf{j}')$, cannot in general be executed independently. Consequently, the optimal parallel execution, which partitions the index space into sets of independent index points and executes all these sets in parallel (in different processors), becomes a challenging algorithmic task.

Our scheduling policy executes every index point at its *earliest computation time* (ECT), taking the existing dependence vectors into consideration. This policy guarantees the optimal execution time of the entire loop. We also assume that the execution of each computation requires a single unit of time (*Unit Execution Time* model). We now give a few definitions.

- The index space is partitioned into disjoint **time subsets** denoted ECT_i, $i \geq 0$, such that ECT_i contains the set of points of J, whose earliest computation time is i. By definition, ECT_0 denotes the boundary (pre-computed) points.

- Consider any doubly nested loop along with its index space and its uniform dependence vectors. The geometric shape of the subset of index points that can be computed at time $t = 1$ (ECT_1) is called **initial pattern** (see Figure 1) and is denoted as Pat_1.

- Using the algorithms described in [14], starting from the initial pattern, we can compute successive similar wavefront-like polygons which we denote by $Pat_i, i \geq 1$. We call Pat_k the **pattern corresponding to moment** i. The intuition behind the pattern notion is that Pat_i is the geometric representation of ECT_i. In Figure 1, we depict the patterns that correspond to moments $1, 2, 3$ and 4. For consistency we define $Pat_0 = ECT_0$.

- **Pattern outline** is the front boundary of each pattern, towards the execution direction, as shown in Figure 1, with the dotted line, for each pattern. It is denoted as pat_i, $i \geq 1$ and the points are given and computed in lexicographical order, unless noted. So, pat_k is a *sequence* of points, not a simple set.

- Let $\mathbf{d} = (d_x, d_y)$ be a 2-D point; then $J_{\mathbf{d}} = \{\mathbf{j} = (j_x, j_y) \in J \mid d_x \leq j_x \text{ and } d_y \leq j_y\}$.

- $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m$ are the dependence vectors of the algorithm. Their number is denoted m. In the rest of this paper, without loss of generality, we assume that the m dependence vectors $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m$ are lexicographically sorted, so that $\mathbf{d}_1$ is the one with the least x-coordinate. We use DV to denote the sequence of them: $DV = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m \rangle$.

- The sequence of **pattern vectors** $PV = \langle \mathbf{p}_1, \ldots, \mathbf{p}_k \rangle$, is the minimum subset of the dependence vectors, whose end-points[1] satisfy the following condition: $J_{\mathbf{d}_1} \cap \ldots \cap J_{\mathbf{d}_m} = J_{\mathbf{p}_1} \cap \ldots \cap J_{\mathbf{p}_k}$. Their number is usually denoted as $k \leq m$.

[1] We assume that the origin of the dependence vectors is the origin of the axes.

- **Pattern points** are those points that are necessary in order to define the polygon shape of each pattern outline. The rest of the points just complete the polygon area. Consider for example points A, B, C and D in Figure 1. Points A and C are pattern points; B and D are not. Note that the pattern points of the initial pattern, are the endpoints of the pattern vectors.

- **Difference dependence vectors** are the vectors that are the difference of the k pattern vectors. They are at most $m-1$ and we define two types of them: the *absolute* difference vectors: $\mathbf{a}_1 = \mathbf{p}_2 - \mathbf{p}_1, \mathbf{a}_2 = \mathbf{p}_3 - \mathbf{p}_1, \ldots, \mathbf{a}_{k-1} = \mathbf{p}_k - \mathbf{p}_1$, and the *relative* difference vectors: $\mathbf{b}_1 = \mathbf{p}_2 - \mathbf{p}_1, \mathbf{b}_2 = \mathbf{p}_3 - \mathbf{p}_2, \ldots, \mathbf{b}_{k-1} = \mathbf{p}_k - \mathbf{p}_{k-1}$ (see Figure 2).

- Given a sequence of points $A = \langle \mathbf{j}_1, \mathbf{j}_2, \ldots, \mathbf{j}_a \rangle$ and a vector $\mathbf{d}$, then $A + \mathbf{d} = \langle \mathbf{j}_1 + \mathbf{d}, \mathbf{j}_2 + \mathbf{d}, \ldots, \mathbf{j}_a + \mathbf{d} \rangle$.

- Given a sequence of points $A = \langle \mathbf{j}_1, \mathbf{j}_2, \ldots, \mathbf{j}_a \rangle$ and a sequence of vectors $B = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_b \rangle$, then $A + B = \langle A + \mathbf{d}_1, A + \mathbf{d}_2, \ldots, A + \mathbf{d}_b \rangle$.

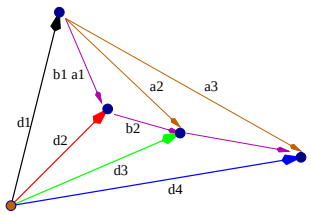

Figure 2: *Absolute and relative dependence difference vectors for an example algorithm with 4 dependence vectors.*

We also define, at this point, the two problems that we are going to study. Given a nested loop algorithm, with index space J and dependence vector set DV, it is:

- **Wavefront Problem**: *compute the arbitrary pattern outline pat_k that corresponds to time k.*

- **Scheduling Problem**: *given any index point, compute the time instance that it should be executed; in other words, specify the pattern that includes this point.*

3 Analytical Pattern Computation

In this section we study the problem of computing the pattern outline that corresponds to an arbitrary time instance k *–wavefront problem.* The crucial feature is that the arbitrary pattern outline pat_k depends only on the initial pattern pat_1.

The idea behind the formation of the pattern outline, is the intersection of certain subspaces that actually form it. We reveal the subspaces that contribute to the formation of each pattern outline and we present a geometric algorithm that computes it. Our study results to an analytical closed form that describes the general k-step pattern outline.

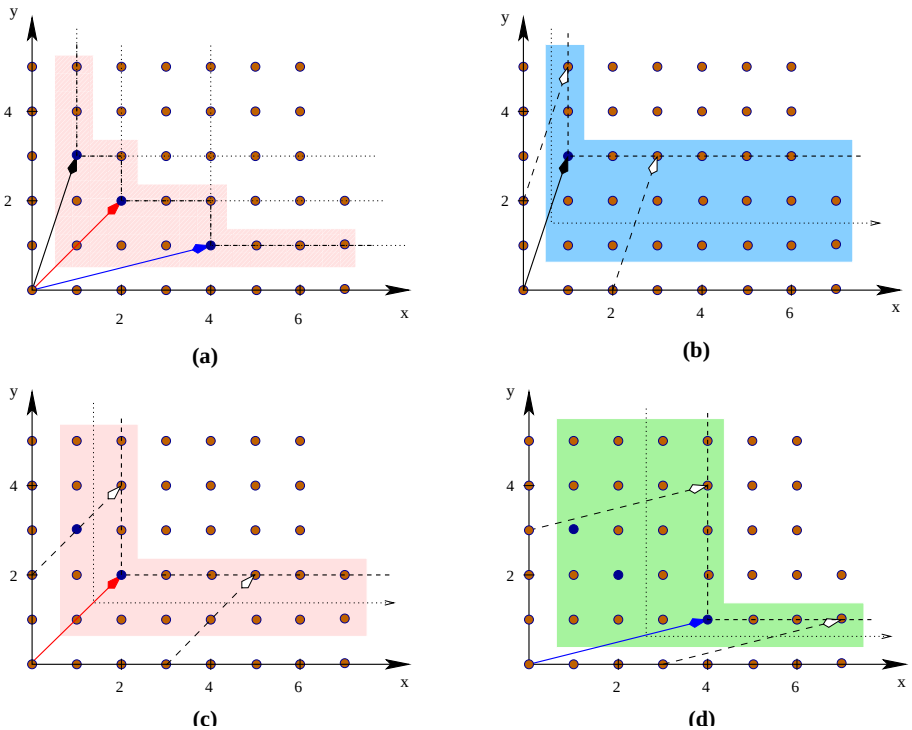

Figure 3: *Initial pattern for an example algorithm with 3 dependence vectors* ***(a)****. Subspaces imposed by each vector for* $t = 1$ ***(b-d)****. The intersection of the three subspaces form the initial pattern.*

3.1 Pattern Computation

The computation of the initial pattern shows the intuition behind our method. Given the precomputed boundary points, each dependence vector imposes a restriction to the points that can be computed at $t = 1$ (see Figure 3). By intersecting the subspaces for all dependence vectors, the initial pattern is formed, as shown in Figure 4.

We present a computational geometry *plane-sweeping*[2] algorithm that efficiently solves the general intersection problem in two dimensions. This algorithm takes as input m points $\mathbf{d}_1, \ldots, \mathbf{d}_m$ and returns the minimum subset $\mathbf{p}_1, \ldots, \mathbf{p}_k$ of those points that satisfy the following condition:

$$J_{\mathbf{p}_1} \cap J_{\mathbf{p}_2} \cap \ldots \cap J_{\mathbf{p}_k} = J_{\mathbf{d}_1} \cap J_{\mathbf{d}_2} \cap \ldots \cap J_{\mathbf{d}_m} \tag{1}$$

[2]For more on plane-sweeping algorithms the interested reader may refer to [11]

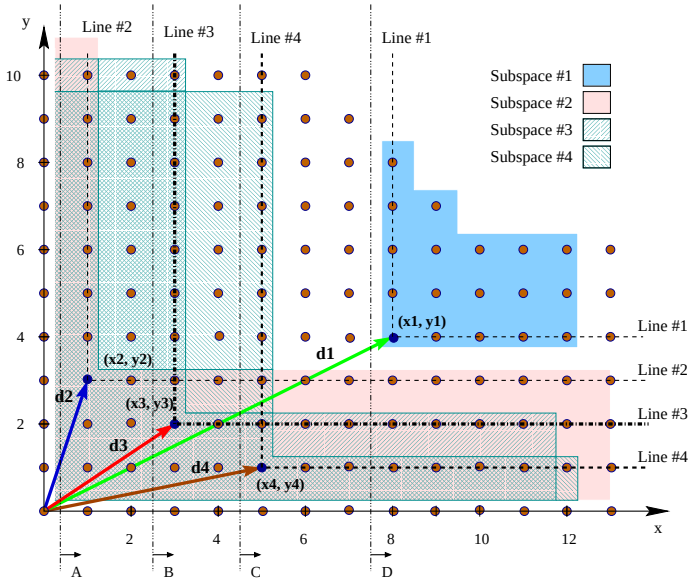

Figure 4: *Formation of the initial pattern through subspace intersection. Note that only pattern vectors* $\mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4$ *affect initial pattern boundary.*

We say that $\mathbf{p}_1, \ldots, \mathbf{p}_k$ are the **minimum cover** of $\mathbf{d}_1, \ldots, \mathbf{d}_m$ and we denote it by $\langle \mathbf{p}_1, \ldots, \mathbf{p}_k \rangle = \mathcal{MC}(\langle \mathbf{d}_1, \ldots, \mathbf{d}_m \rangle)$.

Algorithm I *(Minimum Cover)*

INPUT: m 2-D points $\langle \mathbf{j}_1, \ldots, \mathbf{j}_m \rangle$.

OUTPUT: The k 2-D points $\langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$, $k \leq m$, such that $\langle \mathbf{p}_1, \ldots, \mathbf{p}_k \rangle = \mathcal{MC}(\langle \mathbf{j}_1, \ldots, \mathbf{j}_m \rangle)$.

1. Initially $\mathcal{MC}(\langle \mathbf{j}_1, \ldots, \mathbf{j}_m \rangle) = \langle\ \rangle$ (the empty sequence).
2. From the origin of the x-axis, start sweeping with a vertical *scan-line* to the right.
3. While moving along the x-axis, add to minimum cover the first point that the scan-line comes across; this gives $\mathcal{MC}(\langle \mathbf{j}_1, \ldots, \mathbf{j}_m \rangle) = \langle \mathbf{p}_1 \rangle$.
4. Continue moving the scan-line to the right. For every point reached, its y-coordinate is checked; if it is less than the y-coordinate of the last added point, the current point is also added to the list.
5. Repeat Step 4, until all the points are crossed.
6. The resulting sequence is the minimum cover sequence: $J_{\mathbf{p}_1} \cap \ldots \cap J_{\mathbf{p}_k} = J_{\mathbf{j}_1} \cap \ldots \cap J_{\mathbf{j}_m}$.

To compute the initial pattern, we use the above algorithm with input the m dependence vectors:

$$pat_1 = \mathcal{MC}(\langle \mathbf{d}_1, \ldots, \mathbf{d}_m \rangle) = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle = PV \tag{2}$$

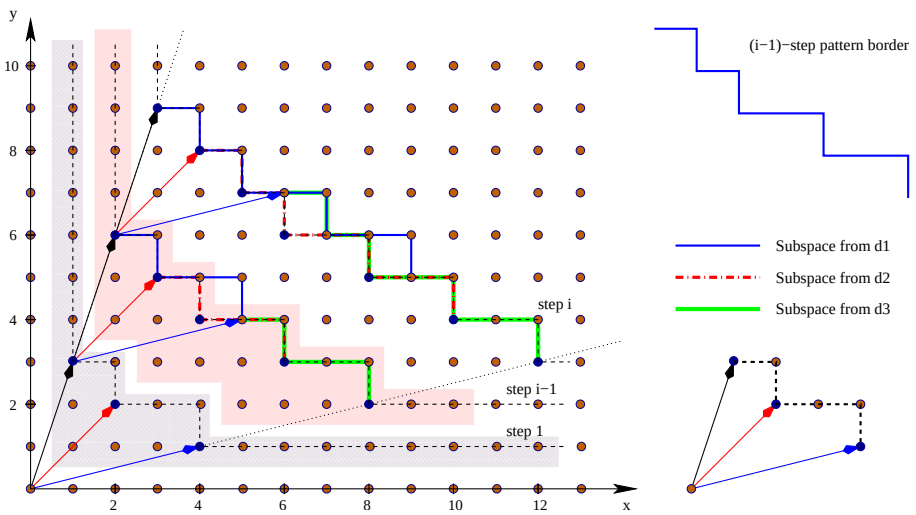

Figure 5: *Formation of i-step pattern in an example algorithm with 3 pattern vectors: the intersection of all transpositions of the $(i-1)$-step pattern according to all pattern vectors. The three images of the i-step border are drawn with different shades, as shown on the right of the figure. Initial pattern is also shown.*

A similar, subspace-intersection procedure, can be followed for the general i-step pattern outline by considering all subspaces that contribute to the formation of this pattern. These subspaces are the k transpositions applied to $(i-1)$-step pattern outline by each pattern vector, as in Figures 5 and 6.

The algorithm that computes the general i-step pattern, is based on the minimum cover algorithm that we presented before and is formally described as follows:

Algorithm II *(Formation of i-step pattern outline)*

INPUT: Dependence vector sequence $DV = \langle \mathbf{d}_1, \ldots, \mathbf{d}_m \rangle$ and a number $i \in \mathbf{N}$, $i > 0$ (i stands for moment).

OUTPUT: The point sequences of the outlines of all patterns until i.

1. $pat_1 = PV = \mathcal{MC}(DV)$ *–the k pattern vectors are calculated*

2. if $i = 1$, return pat_1

3. else, return $pat_i = \mathcal{MC}(pat_{i-1} + PV)$ *–the reader should note that $|pat_i| \leq m \cdot |pat_{i-1}|$*

Although it is the first algorithm that computes the exact i-step pattern outline, one can note that it works recursively, which means that for i-step pattern outline computation, all $1, 2, \ldots, (i-1)$-step outlines should be computed too.

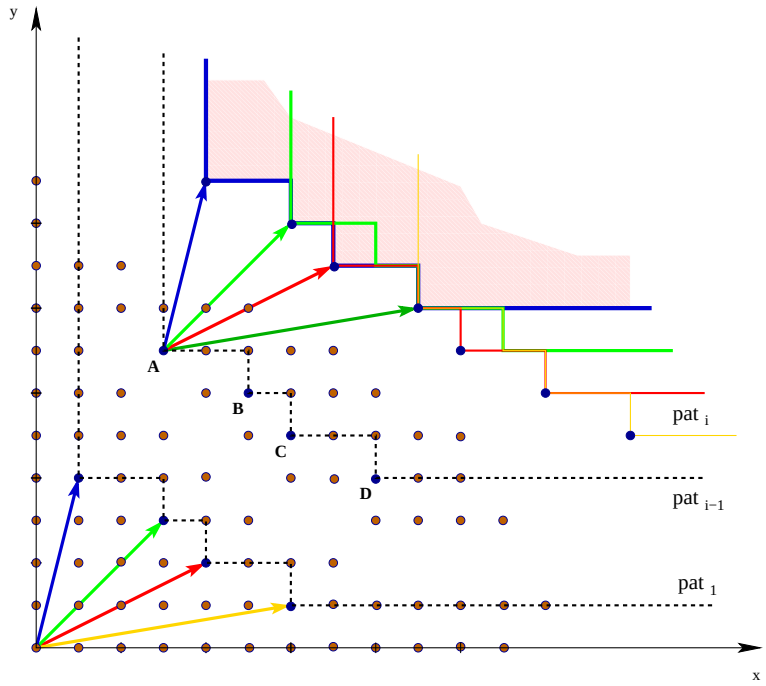

Figure 6: *Computing pat_i from the subspaces formed by the transpositions of pat_{i-1}, along each pattern vector. The intersection of the resulted subspaces, form pat_i. For simplicity, only the subspace along one vector is shaded.*

Also, several i-step points are computed from more than one points of the previous pattern, which indicates the procedure's inefficiency. However, it is the best we can get so far, for the general dependence vector case. Only in certain special cases, its outline can be computed quite efficiently, as shown in [12] and will be described further on.

3.2 Pattern Analytical Description

The above 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 i pattern is formed by following the pattern vectors, starting from $(i-1)$-step pattern.

It is obvious that Pat_1 consists of all points $\mathbf{j} = (j_x, j_y)$, such that $\mathbf{j} - \mathbf{d}_i \in ECT_0$ for any dependence vector $\mathbf{d}_i, 1 \leq i \leq m$. Hence, $Pat_1 = ECT_1$. For the same reasons it is clear that $Pat_i = ECT_i$.

Let $\mathbf{d}_{xmin}$ be the pattern vector with the minimum x-coordinate, as shown in Figure 7. Every dependence vector can be expressed as $\mathbf{d}_i = \mathbf{d}_{xmin} + \mathbf{a}_i$, and equivalently, any point $\mathbf{p}_j \in pat_1$ can be written as:

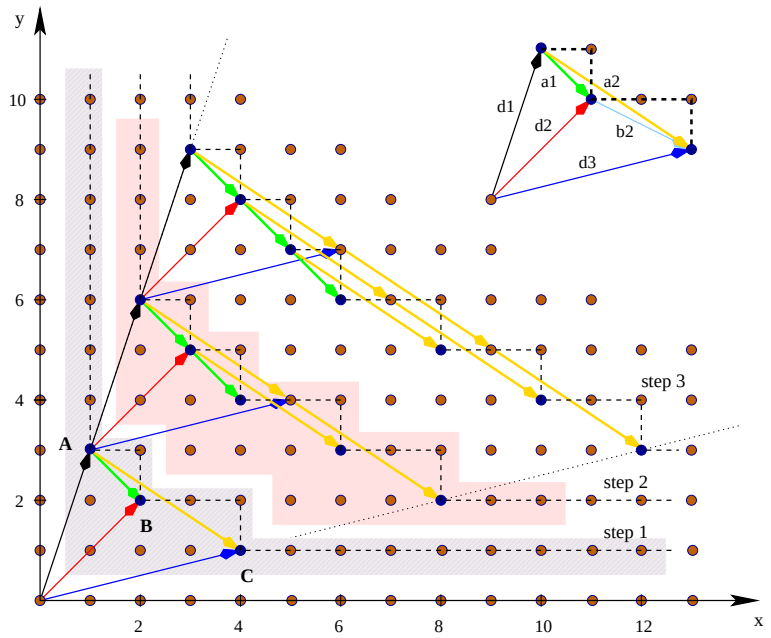

Figure 7: *Formation of the 3 first pattern boundary points by the difference vectors* $\mathbf{a}_i$, *i = 1, 2, . . . , m-1 and* $\mathbf{d}_{xmin} \equiv \mathbf{d}_1$. *Every point* $\mathbf{j} \in pat_1$ *is given from* $\mathbf{j} = \mathbf{d}_{xmin} + k_1 \cdot \mathbf{a}_1 + k_2 \cdot \mathbf{a}_2$, *where* $k_1, k_2 \in \mathbf{N}$ *and* $k_1 + k_2 \leq 1$. *For example, point* ***A*** *is given for* $k_1 = 0$ *and* $k_2 = 0$, *point* ***B*** *is given for* $k_1 = 1$ *and* $k_2 = 0$, *while point* ***C*** *is given for* $k_1 = 0$ *and* $k_2 = 1$.

$$\mathbf{p}_j = \mathbf{d}_{xmin} + k_1 \cdot \mathbf{a}_1 + \ldots + k_{m-1} \cdot \mathbf{a}_{m-1},$$

where:

$$k_i \in \mathbf{N} \text{ and } k_1 + k_2 + \ldots + k_{m-1} \leq 1,\ 1 \leq i \leq m,\ 1 \leq j \leq k.$$

According to the above, the initial pattern outline pat_1 includes the points $\mathbf{j} = (j_x, j_y)$ such that:

- $k \cdot d_{xmin_x} + k_1 \cdot a_{1_x} + \ldots + k_{m-1} \cdot a_{m-1_x} \leq j_x$
- $k \cdot d_{xmin_y} + k_1 \cdot a_{1_y} + \ldots + k_{m-1} \cdot a_{m-1_y} = j_y$
- $j_y = minimum$ *(for a certain* j_x *may be multiple* j_y*)*
- $k_1 + k_2 + \ldots + k_{m-1} \leq 1 = t$.

As any pattern outline can be derived from the previous one, by taking the transpositions mentioned before, the above expression can recursively be generalized as follows. The general pattern outline pat_t includes all points $\mathbf{j} = (j_x, j_y)$ such that:

- $k \cdot d_{xmin_x} + k_1 \cdot a_{1_x} + \ldots + k_{m-1} \cdot a_{m-1_x} \leq j_x$
- $k \cdot d_{xmin_y} + k_1 \cdot a_{1_y} + \ldots + k_{m-1} \cdot a_{m-1_y} = j_y = min$

and

- $k_1 + k_2 + \ldots + k_{m-1} \leq t$, with $k_i \in \mathbf{N}, 1 \leq i \leq m-1$.

According to the above, the formal description of the two problems can be generally stated as:

Wavefront Problem: *given the number of the time instance t, we would like to find all index-space points that form t-step pattern outline pat_t. The formal restrictions that give point $\mathbf{j} = (j_x, j_y) \in pat_t$ are:*

- $k \cdot d_{xmin_x} + k_1 \cdot a_{1_x} + \ldots + k_{m-1} \cdot a_{m-1_x} \leq j_x$
- $k \cdot d_{xmin_y} + k_1 \cdot a_{1_y} + \ldots + k_{m-1} \cdot a_{m-1_y} = j_y = min$
- $k_1 + \ldots + k_{m-1} \leq t$, with $k_i \in \mathbf{N}$, for $1 \leq i \leq m-1$

Scheduling Problem: *given any index-space point $\mathbf{j} = (j_x, j_y)$, we would like to find the time-instance t that it can be executed. The restrictions for t are:*

- $k \cdot d_{xmin_x} + k_1 \cdot a_{1_x} + \ldots + k_{m-1} \cdot a_{m-1_x} \leq j_x$
- $k \cdot d_{xmin_y} + k_1 \cdot a_{1_y} + \ldots + k_{m-1} \cdot a_{m-1_y} \geq j_y$
- $k_1 + k_2 + \ldots + k_{m-1} \leq t$, $k_i \in \mathbf{N}$, $1 \leq i \leq m-1$, and
- $t = minimum$

From the description of the two problems, one can note that they can be solved analytically using *integer linear programming* techniques. However, the above expressions are the first analytical ones that give any point of t-step pattern outline for a general uniform dependence loop algorithm.

3.3 Pattern Uniformity Condition

The interesting point is that the above expression, can be quite simplified in certain cases. In these cases the pattern outline is *perfectly* scalable and i-step outline is more than obvious, as in Figure 8.

We will describe a condition that makes the pattern precisely normal and uniform. If the given condition holds, the i-step pattern can be computed by –simply– considering the initial pattern i times. The proofs of lemmas and theorems that follow, are presented in the Appendix.

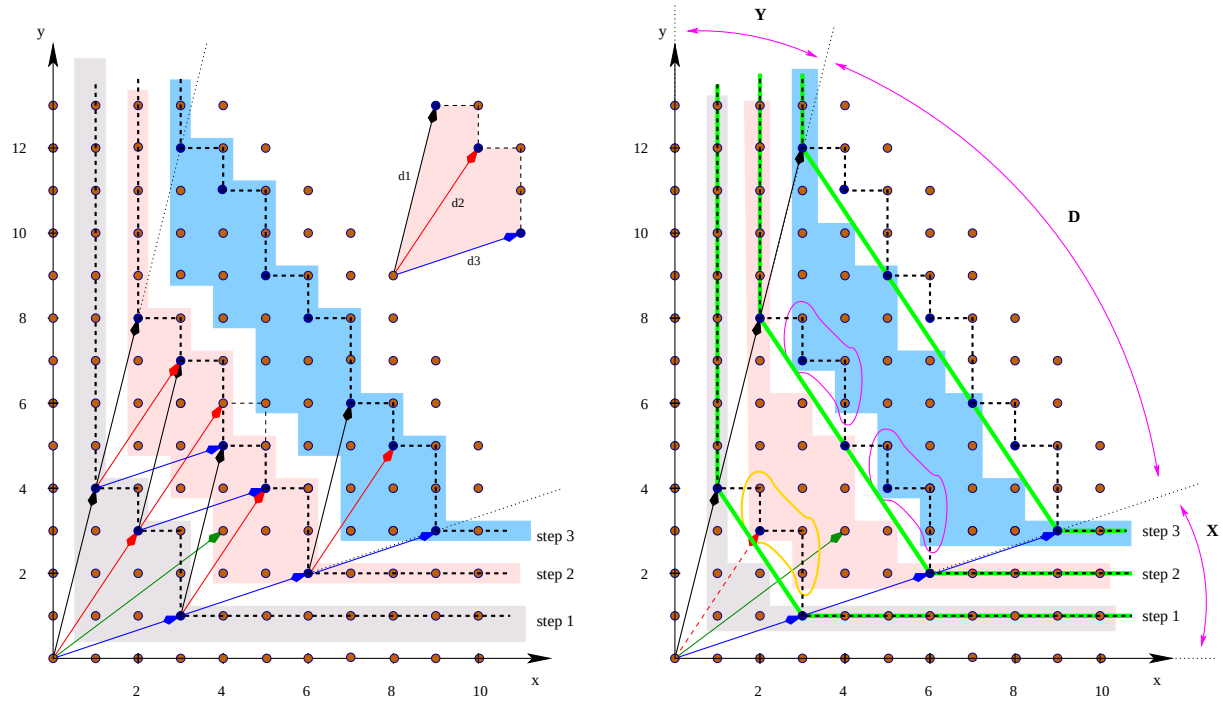

Figure 8: *An algorithm with 4 dependence vectors* $\mathbf{d}_1 = (1, 4)$, $\mathbf{d}_2 = (2, 3)$, $\mathbf{d}_3 = (3, 1)$ *and* $\mathbf{d}_4 = (4, 3)$, *which is not a pattern vector. The pattern outline is perfectly scalable: general i-step pattern outline is computed by considering the outline of the initial pattern i times.*

Lemma 3.1 *Consider a nested loop algorithm with two dependence difference vectors* $\mathbf{a}'_1$ *and* $\mathbf{a}'_2$ *and their slopes*[3] g_1 *and* g_2, *respectively. If* $g_1 \geq g_2$ *and* $a'_{1y} \geq a'_{2y}$ *holds, then the points that contribute to pattern outline formation are (as in Figure 21):*

- *the endpoints of* $i \cdot \mathbf{a}'_1$, *where* $0 \leq i \leq n$,
- *potentially, the endpoints of* $i \cdot \mathbf{a}'_2$, *that reach the triangles formed by* $\mathbf{a}'_1$.

Lemma 3.2 *Consider* m *dependence difference vectors* $\mathbf{a}_i$ *and their slopes* g_i, *such that* $g_1 \geq g_i$ *for* $2 \leq i \leq m-1$. *Without loss of generality* $a_{1y} \geq a_{iy}$, $2 \leq i \leq m-1$. *Then the previous Lemma holds for* $\mathbf{a}'_1 = \mathbf{a}_1$ *and* $\mathbf{a}'_2$ *any linear combination of the form* $k_2 \cdot \mathbf{a}_2 + k_3 \cdot \mathbf{a}_3 + \ldots + k_m \cdot \mathbf{a}_{m-1}$, *where* $0 \leq k_i \leq m-1$ *and* $k_2 + k_3 + \ldots + k_{m-1} = n$.

Theorem 3.1 *Consider a nested loop algorithm with* k *pattern vectors* $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k$. *Let* $\mathbf{a}_i = \mathbf{p}_i - \mathbf{p}_{i+1}$ *be the* $k-1$ *difference vectors, with slope values* g_i. *If the following hold, for* $2 \leq i \leq k-1$:

1. $g_1 \geq 2 \cdot g_i$,

[3] Using the coordinate system of Figures 21 and 22.

2. $a_{1y} \geq a_{iy}$ *and*

3. $a_{ix} \geq \frac{a_{1x}}{2}$

then the pattern outline is recurrent: it can be computed by replicating every pattern part by the first part (which is the pattern outline of the first time instance).

Many other conditions of uniformity can be proved, based on the same approach. Figures 5 and 7 give another such uniformity case: the initial pattern is split into certain subpatterns. By taking each subpattern i-times, the i-step pattern is precisely formed. It can be proved that the conditions for this case are:

- $g_1 > g_2 > \ldots > g_{m-1}$ and
- $a_{1y} < a_{2y} < \ldots < a_{(m-1)y}$

3.4 Pattern Computation Complexity

In this section we study the complexity of the above procedure for pattern computation (complexity of *Algorithms I* and *II*).

The input of *Algorithm I* is the sequence of m dependence vectors. These points are sorted in ascending order of their x-coordinate and then, checked for descending y-coordinate order, in other words, algorithm's complexity is:

$$\Theta(m \cdot \log m) + \Theta(m) = \Theta(m \cdot \log m).$$

On *Algorithm III*, complexity will be found recursively, as for calculating t pattern, $(t-1)$-pattern must have been calculated also. Supposing that $Cx(p_t)$ is the complexity of step-t pattern-outline computation, it is for step 1:

$$Cx(p_1) = O(m \cdot \log m),$$

as only *Algorithm I* is applied for this step. For the outline of the arbitrary step t it is $|pat_t| = m^t$, in the worst case. For computing next step outline pat_{t+1}, we get all points that can be reached by every point $\mathbf{p}_t \in pat_t$, along any pattern vector (at most m). All these $m^t \cdot m$ points need to be sorted. So:

$$Cx(p_{t+1}) = Cx(p_t) + m^t \cdot m + (m^{t+1} \cdot \log(m^{t+1})) \Rightarrow$$
$$Cx(p_{t+1}) = Cx(p_t) + m^{t+1} \cdot (1 + \log(m^{t+1})) \Rightarrow$$
$$Cx(p_{t+1}) = Cx(p_t) + O(m^{t+1} \cdot \log(m^{t+1})).$$

In other words:

$$Cx(p_2) = Cx(p_1) + m^2 \cdot \log(m^2) = O(m^2 \cdot \log(m^2)),$$
$$Cx(p_3) = Cx(p_2) + m^3 \cdot \log(m^3) = O(m^3 \cdot \log(m^3)),$$

$$\vdots$$

$$Cx(p_t) = O(m^t \cdot \log(m^t)).$$

So, the complexity of computing the t-step pattern outline pat_t, for a nested loop algorithm with m dependence vector is:

$$Cx(pat_t) = O(m^t \cdot \log(m^t))$$

This result proves the pattern-outline exponential computational cost that mentioned before.

4 Region Computation and Scheduling

In this section we present a nearly optimal technique for scheduling nested loop index points. The intuition behind this efficient algorithm, is the partitioning of the index space into a sequence of regions $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, ..., such that $\mathcal{R}_1 \subseteq ECT_1$, $\mathcal{R}_2 \subseteq ECT_2 \cup ECT_1$ and in general $\mathcal{R}_i \subseteq \bigcup_{t=1}^{i} ECT_t$.

It is proven in [14] 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. This means that we actually have that $\mathcal{R}_i \subseteq ECT_i \cup ECT_{i-1}$.

In Figure 9, we show the exhaustive methodology that was previously described, as well as our new method, for a given doubly nested loop. Notice that the exhaustive procedure calculates precisely the border of the pattern. Our procedure is not that accurate, since the pattern is always a trapezoid, but enables an efficient scheduling to follow. The index space is partitioned into several cones (which are angles in 2-D). Certain geometric formations, called *regions*, are formed. Regions have simpler representation than the accurate patterns, which simplify dramatically point-location and thus scheduling. Using the following algorithms, we can predict the i-step region, and therefore, we do not have to resort to recursive procedure.

4.1 Terminology and 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 dependence vector with the least x-coordinate $\mathbf{d}_{xmin}$ (see Figure 9).

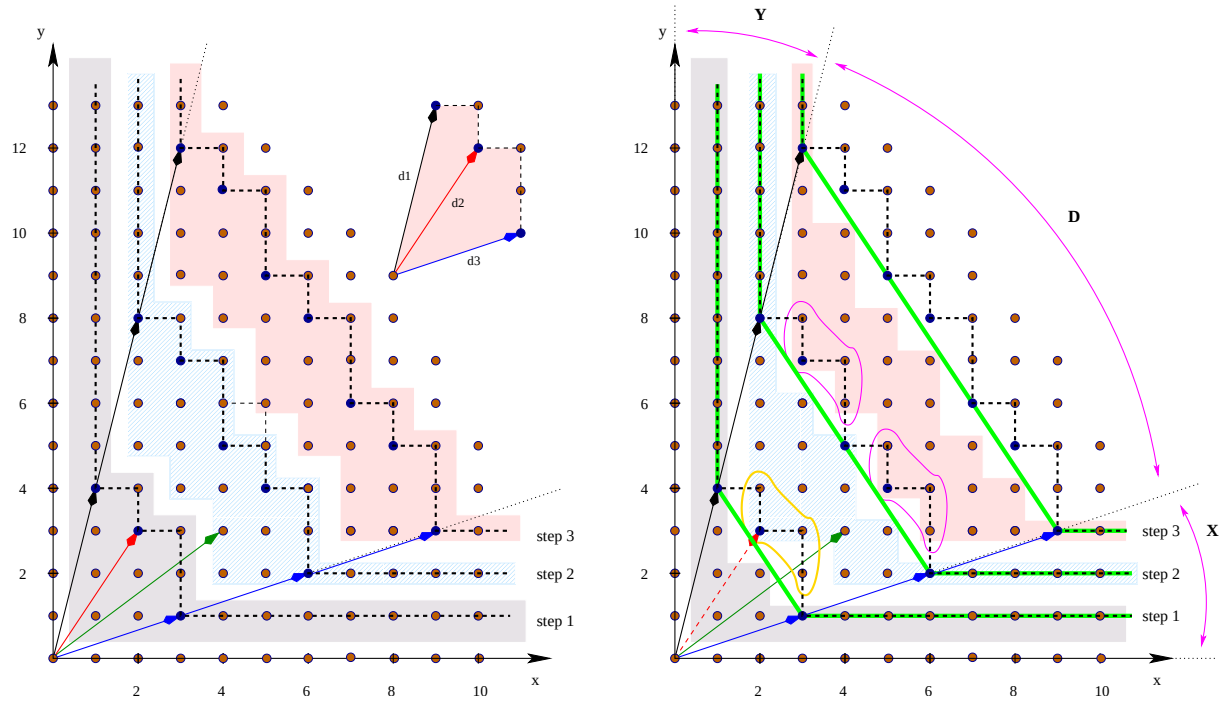

Figure 9: *Pattern formation by the exhaustive and the non-exhaustive method for an example algorithm with 4 dependence vectors.* ***a****) By the exhaustive method (left) the accurate pattern is computed, but a point-locating and scheduling algorithm is not easily achieved.* ***b****) Our method (right), 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.*

X-Cone: The cone formed by the dependence 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 $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_r$ are those dependence vectors that have the following property. They form a lexicographically ordered sequence $CV = \langle \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_r \rangle$, such that:

- $\mathbf{d} = \lambda_1 \cdot \mathbf{c}_i + \lambda_2 \cdot \mathbf{c}_{i+1}$, where $\lambda_1 + \lambda_2 > 1$, for every $\mathbf{d} \in DV$, $\mathbf{d} \neq \mathbf{c}_i, \mathbf{c}_{i+1}$ and for *exactly one* pair of $\langle \mathbf{c}_i, \mathbf{c}_{i+1} \rangle$.

The number of cone vectors is denoted by r. Note that $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ are always included in CV ($\mathbf{c}_1 = \mathbf{d}_{xmin}$ and $\mathbf{c}_r = \mathbf{d}_{ymin}$).

X_i-Region is the set: $X_i = \{\mathbf{j} = (j_x, j_y) \in X \mid (i-1) \cdot d_{xmin_y} < j_y < i \cdot d_{xmin_y}, \text{ where } i > 0\}$.

Y_i-Region is the set: $Y_i = \{\mathbf{j} = (j_x, j_y) \in Y \mid (i-1) \cdot d_{ymin_x} < j_x < i \cdot d_{ymin_x}, \text{ where } i > 0\}$.

Cone Region: The i-th region of a cone $Con(\mathbf{a}, \mathbf{b})$ is the set (see Figure 10):

$$Con_i(\mathbf{a}, \mathbf{b}) = \{\mathbf{j} \in Con(\mathbf{a}, \mathbf{b}) \mid \mathbf{j} = \lambda_1 \cdot \mathbf{a} + \lambda_2 \cdot \mathbf{b}, \text{ where } i < \lambda_1 + \lambda_2 \leq i + 1\}.$$

Region: Region $\mathcal{R}_i$ is denoted as the union of i-regions of all cones:

$$\mathcal{R}_i = \bigcup_{j=1}^{r-1} Con_i(\mathbf{c}_j, \mathbf{c}_{j+1}) \cup X_i \cup Y_i.$$

For instance, in Figure 10, we show an algorithm with 6 dependence vectors. In this case $CV = \langle \mathbf{d}_1, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_6 \rangle$. We also show regions $Con_2(\mathbf{d}_3, \mathbf{d}_5)$ and $Con_3(\mathbf{d}_5, \mathbf{d}_6)$, as well as $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$. Points A and E belong to $\mathcal{R}_1$, points F, C and G belong to $\mathcal{R}_2$ and B and D belong to $\mathcal{R}_3$.

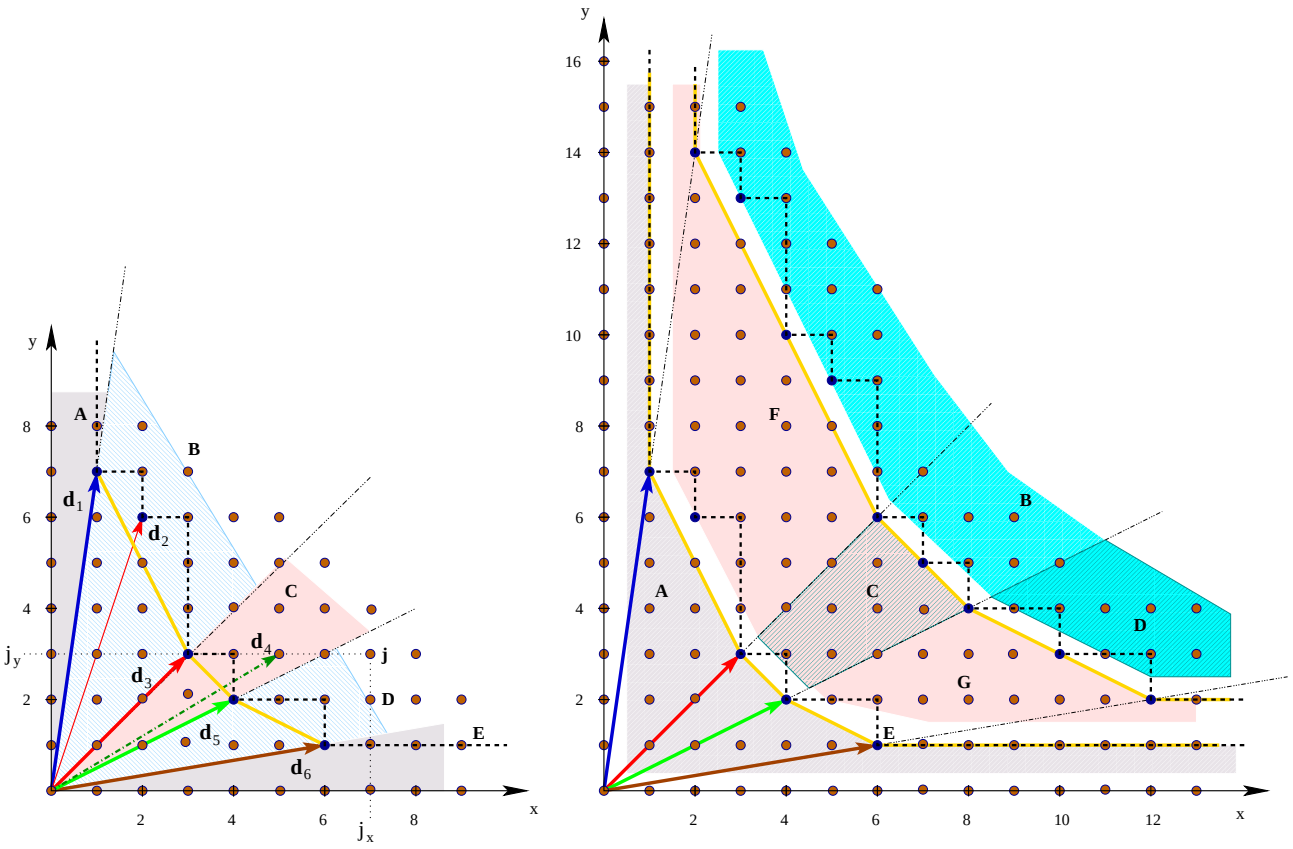

Figure 10: *The index space of an algorithm with 6 dependence vectors* $\mathbf{d}_1 = (1, 7)$, $\mathbf{d}_2 = (2, 6)$, $\mathbf{d}_3 = (3, 3)$, $\mathbf{d}_4 = (4, 2)$, $\mathbf{d}_5 = (5, 3)$ *and* $\mathbf{d}_6 = (6, 1)$. *Except* $\mathbf{d}_2$ *and* $\mathbf{d}_4$, *all other vectors belong to the cone vector sequence:* $CV = \langle \mathbf{d}_1, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_6 \rangle$. *The 5 cones that are formed are colored respectively. The first 3* regions *of all cones are also shown, as well as cone regions* $Con_2(\mathbf{d}_3, \mathbf{d}_5)$ *and* $Con_3(\mathbf{d}_5, \mathbf{d}_6)$ *(right).*

4.2 Cones and Regions

The method presented in the following sections, divides the index space 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. Intuitively, cone vectors determine the execution order among the points that belong in the cone, by determining the optimal *hyperplane* for the cone, as proved in [3].

Consider, for example, the cone formed by vectors $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ in Figure 11 and an index point $\mathbf{j} = (j_x, j_y)$. If $\mathbf{j} = \lambda_1 \cdot \mathbf{d}_{xmin} + \lambda_2 \cdot \mathbf{d}_{ymin}$, and $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ are lexicographically ordered, then we distinguish the following cases:

- **if** $\lambda_1 > 0$ **and** $\lambda_2 > 0$**,** then $\mathbf{j}$ lies inside $Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$ (e.g., point $\mathbf{A} \in Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$).
- **if** $\lambda_1 > 0$ **and** $\lambda_2 < 0$**,** then $\mathbf{j}$ lies inside Y-cone (e.g., point $\mathbf{B} \in Y$).
- **if** $\lambda_1 < 0$ **and** $\lambda_2 > 0$**,** then $\mathbf{j}$ lies inside X-cone (e.g., point $\mathbf{C} \in X$).

For example, considering Figure 11, if we solve equation $A = \lambda_1 \cdot \mathbf{d}_{xmin} + \lambda_2 \cdot \mathbf{d}_{ymin}$ for (λ_1, λ_2) we get $(\lambda_1, \lambda_2) = (\frac{17}{11}, \frac{9}{11}) \Rightarrow A \in Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$. Similarly, for point B we get $(\lambda_1, \lambda_2) = (\frac{35}{11}, \frac{-8}{11}) \Rightarrow B \in Y$-cone. Finally, for point C we get $(\lambda_1, \lambda_2) = (\frac{-5}{11}, \frac{42}{11}) \Rightarrow C \in X$-cone. Note also that $A \in \mathcal{R}_3$, as $\lceil \frac{17}{11} + \frac{9}{11} \rceil = 3$, while $B \in \mathcal{R}_1$, as $\lceil \frac{1}{1} \rceil = 1$ and $C \in \mathcal{R}_2$, as $\lceil \frac{2}{1} \rceil = 2$.

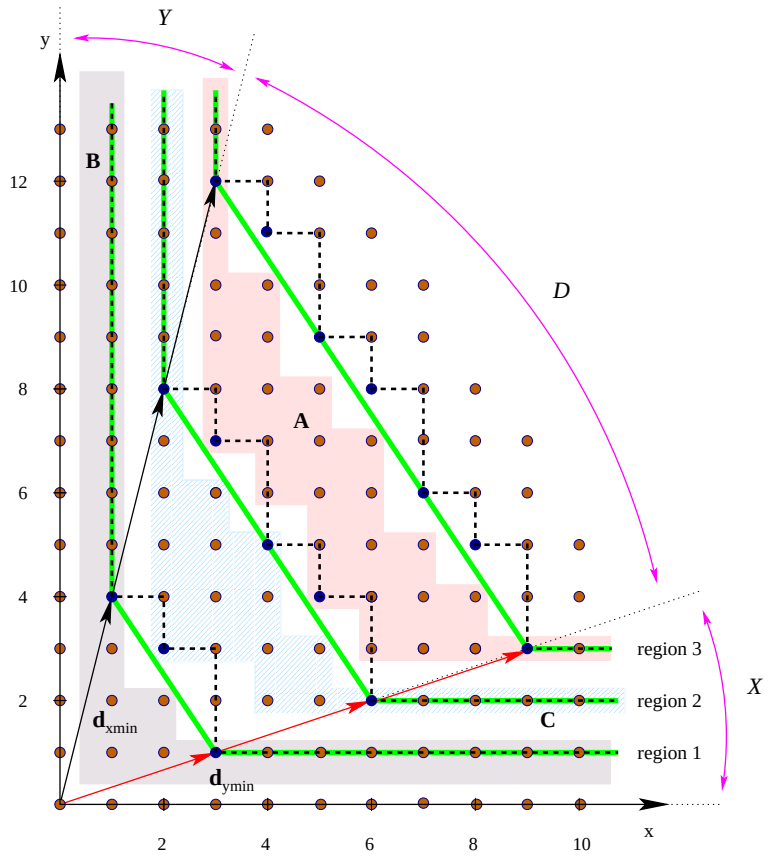

Figure 11: *Point locating in a 3-cone index space, with* $\mathbf{d}_{xmin} = (1, 4)$ *and* $\mathbf{d}_{ymin} = (3, 1)$.

The basic idea behind our algorithm is the following: if point $\mathbf{j}$ belongs in region $\mathcal{R}_i$, then $\mathbf{j}$ is scheduled for execution at moment i. So, in order to find out when $\mathbf{j}$ is to be executed, we must first calculate in which region $\mathbf{j}$ belongs. The first step is to determine in which cone $\mathbf{j}$ belongs; e.g., if it belongs to either X or Y, or $Con(\mathbf{c}_i, \mathbf{c}_{i+1})$ for a pair of cone vectors. This is done by solving the equations $\mathbf{j} = \lambda_1 \cdot \mathbf{c}_i + \lambda_2 \cdot \mathbf{c}_{i+1}$, $1 \leq i \leq r-1$, and checking the sign of λ_1 and λ_2. For details see *Algorithm IV*.

After that, we simply calculate in which specific region $\mathbf{j}$ belongs, using the following expressions, as shown if Figure 11:

▷ ***if point*** $\mathbf{j}$ ***lies in the*** Y***-cone:*** [region number] $= \lceil \frac{j_x}{d_{xmin_x}} \rceil$ (e.g., point $B \in \mathcal{R}_1$).

▷ ***if point*** $\mathbf{j}$ ***lies in the*** X***-cone:*** [region number] $= \lceil \frac{j_y}{d_{ymin_y}} \rceil$ (e.g., point $C \in \mathcal{R}_2$).

▷ ***if point*** $\mathbf{j}$ ***lies in*** $Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$***:*** [region number] $= \lceil \lambda_1 + \lambda_2 \rceil$, where $\mathbf{j} = \lambda_1 \cdot \mathbf{d}_{xmin} + \lambda_2 \cdot \mathbf{d}_{ymin}$, as above (e.g., point $A \in \mathcal{R}_3$).

So, summing up, we partition th index space into a sequence of cones and each cone is further partitioned into regions. Inside a cone, the region number determines the time instance that the points of the region are executed. The whole procedure is described by the following steps:

1. We compute the *cone vectors* out of the given dependence vectors; the index space can be partitioned into cones and regions –the i-step pattern can be also computed.

2. The actual *schedule* can be performed, by locating the given points among a pair of *cone* and *region*.

Given a nested loop with a dependence vector sequence $DV = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m \rangle$, each of the above steps is presented as a separate algorithm.

4.3 Cone-Vector and Region Computation

The computation of the cone vectors, is the major part of the scheduling procedure. We present a computational geometry algorithm that computes them. It works with the pattern-vector sequence $PV = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$ and computes the cone vector sequence CV. Note that $\mathbf{p}_1 = \mathbf{d}_{xmin}$ and $\mathbf{p}_k = \mathbf{d}_{ymin}$. In order to have PV as input we have used *Algorithm I* with input DV. During the algorithm, sequence CV' is also used, where $CV' = \langle \mathbf{p}_2, \ldots, \mathbf{p}_{k-1} \rangle = CV - \{\mathbf{d}_{xmin}, \mathbf{d}_{ymin}\}$.

Algorithm III *(Cone-vector computation)*

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

INPUT: The pattern-vector sequence $PV = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$ of the algorithm.

OUTPUT: A sequence of the r cone vectors $CV = \langle \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_r \rangle$, where $r \leq k \leq m$.

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

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

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

4. Else if $CV' = \langle \mathbf{c} \rangle$, then return $\langle \mathbf{d}_{xmin}, \mathbf{c}, \mathbf{d}_{ymin} \rangle$.

5. Else, let $s = (d_{ymin_y} - d_{xmin_y})/(d_{ymin_x} - d_{xmin_x})$; let $\mathbf{p}_i = (p_{i_x}, p_{i_y}) \in PV$ a vector that minimizes the expression $p_{i_y} - s \cdot p_{i_x}$ (this point is called *furthest point*).

6. Let $CV'_l = \langle \mathbf{c} \in CV' : c_x < p_{i_x} \wedge \mathbf{c} \in \langle \mathbf{d}_{xmin}, \mathbf{p}_i \rangle \rangle$ and $CV'_r = \langle \mathbf{c} \in CV' : c_x > p_{i_x} \wedge \mathbf{c} \in \langle \mathbf{p}_i, \mathbf{d}_{ymin} \rangle \rangle$.

7. Return the concatenation of `DivideConeVector(`$\langle \mathbf{d}_{xmin}, CV'_l \rangle, \mathbf{d}_{xmin}, \mathbf{p}_i$`)` and `DivideConeVector(`$\langle CV'_r, \mathbf{d}_{ymin} \rangle, \mathbf{p}_i, \mathbf{d}_{ymin}$`)`.

For better understanding, we show an example in Figure 12. This algorithm has six dependence vectors $\mathbf{d}_1$, $\mathbf{d}_2$, ..., $\mathbf{d}_6$. It is $PV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4, \mathbf{d}_5 \rangle$. In (a), a line segment is drawn between the endpoints of $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$: $\overline{AE}$. Vectors $\mathbf{d}_1$ and $\mathbf{d}_5$ are inserted in cone-vector sequence: $CV = \langle \mathbf{d}_1, \mathbf{d}_5 \rangle$. From the remaining endpoints, the furthest (towards axes origin) is chosen: point C. In (b), the vector that corresponds to this endpoint $\mathbf{d}_3$ is added: $CV = \langle \mathbf{d}_1, \mathbf{d}_3, \mathbf{d}_5 \rangle$.

Initial cone $Con(\mathbf{d}_1, \mathbf{d}_5)$ is split into $Con(\mathbf{d}_1, \mathbf{d}_3)$ and $Con(\mathbf{d}_3, \mathbf{d}_5)$. In (c) the algorithm is further applied onto $Con(\mathbf{d}_1, \mathbf{d}_3)$ and pattern vector $\mathbf{d}_2$ is also added to CV, while on $Con(\mathbf{d}_3, \mathbf{d}_5)$ the procedure has already come to an end. So, cone-vector list finally includes $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$ and $\mathbf{d}_5$. All cones formed are shown on (d).

The above algorithm follows the *divide-and-conquer* technique. The complexity achieved is $O(m \cdot \log r)$, where m is the number of dependence vectors and r the number of cone vectors (usually quite smaller than m).

Now that the cone vectors are computed, the i-step region can be also computed quite efficiently. Starting from the origin, we apply all cone-vectors. By connecting their endpoints, the initial region $\mathcal{R}_1$ is formed. By applying them twice and connecting their endpoints, the step 2 region $\mathcal{R}_2$ is formed and so on. The procedure is shown in Figure 10.

4.4 Scheduling

The index space of a given nested loop algorithm is partitioned into cones and regions, by calculating the cone-vectors of the algorithm. Given an index point $\mathbf{j}$, we first find the cone $Con(\mathbf{c}_k, \mathbf{c}_{k+1})$

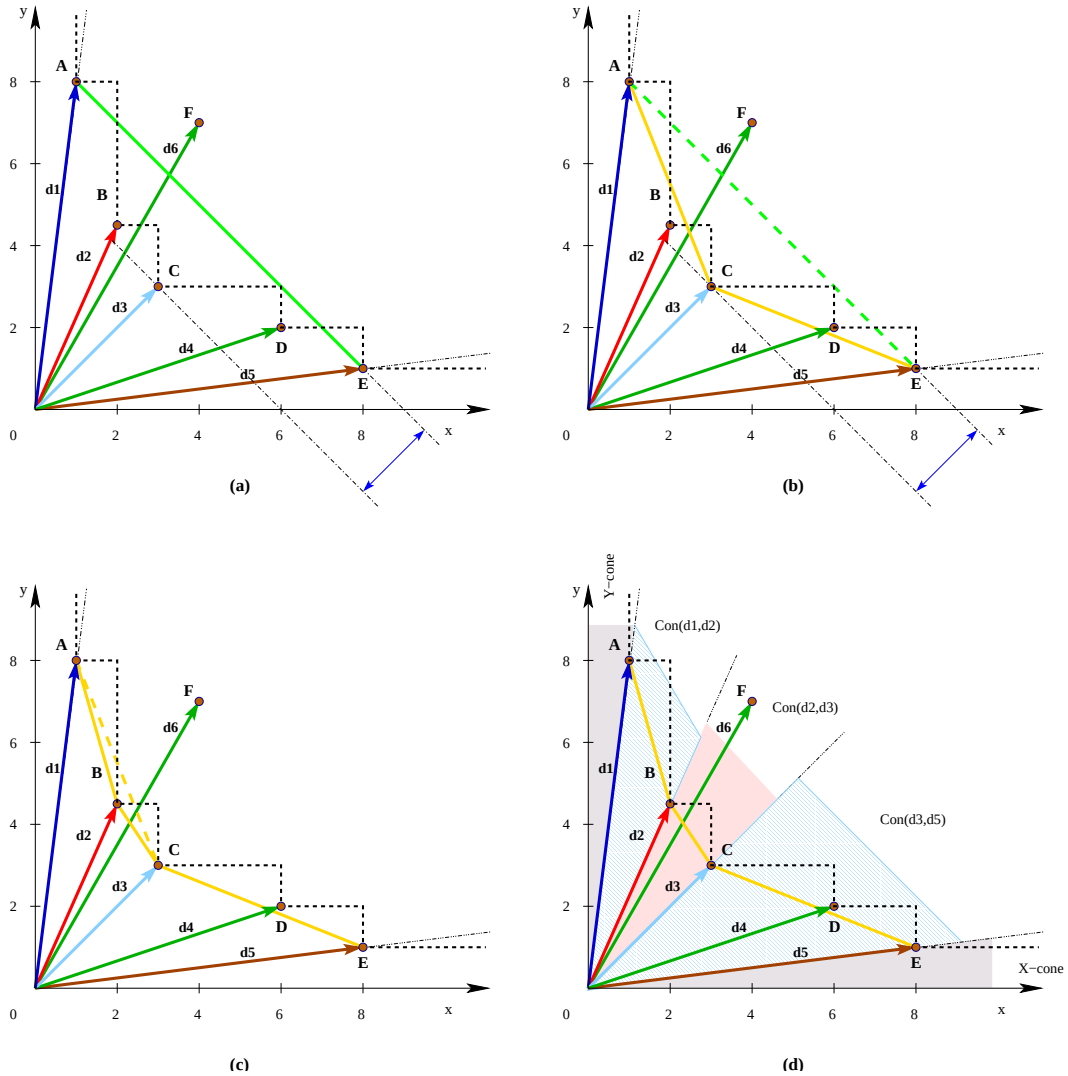

Figure 12: ***a)*** *Pattern vectors* $\mathbf{d}_{xmin}$ *and* $\mathbf{d}_{ymin}$ *are inserted in the cone-vector list and furthest point is computed.* ***b)*** *Vector* $\mathbf{d}_3$ *that corresponds to furthest point is inserted into cone-vector list CV.* ***c)*** *Algorithm continues on the two subcones formed by* $\mathbf{d}_3$*. On* $Con(\mathbf{d}_1, \mathbf{d}_3)$ *one more vector is added, while on* $Con(\mathbf{d}_3, \mathbf{d}_5)$ *the procedure has finished.* ***d)*** *The cone-vector set, as well as the five cones formed are shown.*

where $\mathbf{j}$ belongs, and then region $Con_i(\mathbf{c}_k, \mathbf{c}_{k+1})$, where $\mathbf{j}$ belongs. Finally, we schedule $\mathbf{j}$ to be executed at moment i. This is done via the following algorithm.

Algorithm IV *(Scheduling in a Multi-Cone Space)*

INPUT: The dependence vector set $D = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m\}$ of the algorithm and an index point $\mathbf{j}$.

OUTPUT: The time instance t_j that point $\mathbf{j}$ can be executed.

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

computed.

2. The r cone vectors are computed by applying *Algorithm III* onto PV. After this step: $CV = \langle \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_r \rangle$, where $\mathbf{c}_1 = \mathbf{d}_{xmin}$ and $\mathbf{c}_r = \mathbf{d}_{ymin}$.

3. Find out at which cone our point $\mathbf{j}$ is located: for every cone-vector pair $(\mathbf{c}_i, \mathbf{c}_{i+1})$ of *subsequent* vectors $\mathbf{c}_i, \mathbf{c}_{i+1} \in CV$, solve the equation: $\mathbf{j} = \lambda_{i1} \cdot \mathbf{c}_i + \lambda_{i2} \cdot \mathbf{c}_{i+1}$ and compute λ_{i1} and λ_{i2}.
   - If for $(\mathbf{d}_{xmin}, \mathbf{c}_2)$, $\lambda_{11} \geq 0$ and $\lambda_{12} < 0$: point $\mathbf{j}$ lies inside cone Y.
   - Else, if for $(\mathbf{c}_{r-1}, \mathbf{d}_{ymin})$, $\lambda_{r1} < 0$ and $\lambda_{r2} \geq 0$: point $\mathbf{j}$ lies inside cone X.
   - Else, if $\lambda_{i1} \geq 0$ and $\lambda_{i2} \geq 0$: point $\mathbf{j}$ lies inside cone $Con(\mathbf{c}_i, \mathbf{c}_{i+1})$.

4. Now that we know the cone that point $\mathbf{j}$ belongs to, we can schedule its execution according to the following:

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

   ▷ if the point lies in cone $X \Rightarrow t_j = \lceil \frac{j_y}{d_{ymin\,y}} \rceil = \lceil \lambda_{11} \rceil$

   ▷ if point lies in cone $Con(\mathbf{c}_i, \mathbf{c}_{i+1}) \Rightarrow t_j = \lceil \lambda_{i1} + \lambda_{i2} \rceil$.

The following example demonstrates the scheduling procedure of *Algorithm IV*.

Example I

Suppose an 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 $\mathbf{d}_5 = (6, 1)$, as in Figure 13. 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:

$$\text{locating point } C\text{:} \begin{cases} Con(\mathbf{d}_1, \mathbf{d}_2): & \lambda_1 \cdot \mathbf{d}_1 + \lambda_2 \cdot \mathbf{d}_2 = (5,5) \Rightarrow \lambda_1 = -1 \text{ and } \lambda_2 = 3 \\ Con(\mathbf{d}_2, \mathbf{d}_3): & \lambda_1 \cdot \mathbf{d}_2 + \lambda_2 \cdot \mathbf{d}_3 = (5,5) \Rightarrow \lambda_1 = 5/8 \text{ and } \lambda_2 = 5/4 \\ Con(\mathbf{d}_3, \mathbf{d}_5): & \lambda_1 \cdot \mathbf{d}_3 + \lambda_2 \cdot \mathbf{d}_5 = (5,5) \Rightarrow \lambda_1 = 25/9 \text{ and } \lambda_2 = -5/9 \end{cases}$$

From the above, it is denoted that point C resides in cone $Con(\mathbf{d}_2, \mathbf{d}_3)$. For this cone 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 $F = (1, 4)$, $G = (4, 13)$ and $H = (10, 4)$, we can verify their computation moments:

point F: $\lambda_1 \cdot \mathbf{d}_1 + \lambda_2 \cdot \mathbf{d}_2 = (1,4) \Rightarrow \lambda_1 = 2/5$ and $\lambda_2 = 3/10 \Rightarrow F \in Con(\mathbf{d}_1, \mathbf{d}_2)$ and $t_F = 1$

point G: $\lambda_1 \cdot \mathbf{d}_1 + \lambda_2 \cdot \mathbf{d}_2 = (4,13) \Rightarrow \lambda_1 = 1$ and $\lambda_2 = 3/2 \Rightarrow G \in Con(\mathbf{d}_1, \mathbf{d}_2)$ and $t_G = 3$

point H: $\lambda_1 \cdot \mathbf{d}_3 + \lambda_2 \cdot \mathbf{d}_5 = (10,4) \Rightarrow \lambda_1 = 14/9$ and $\lambda_2 = 8/9 \Rightarrow H \in Con(\mathbf{d}_3, \mathbf{d}_5)$ and $t_H = 3$

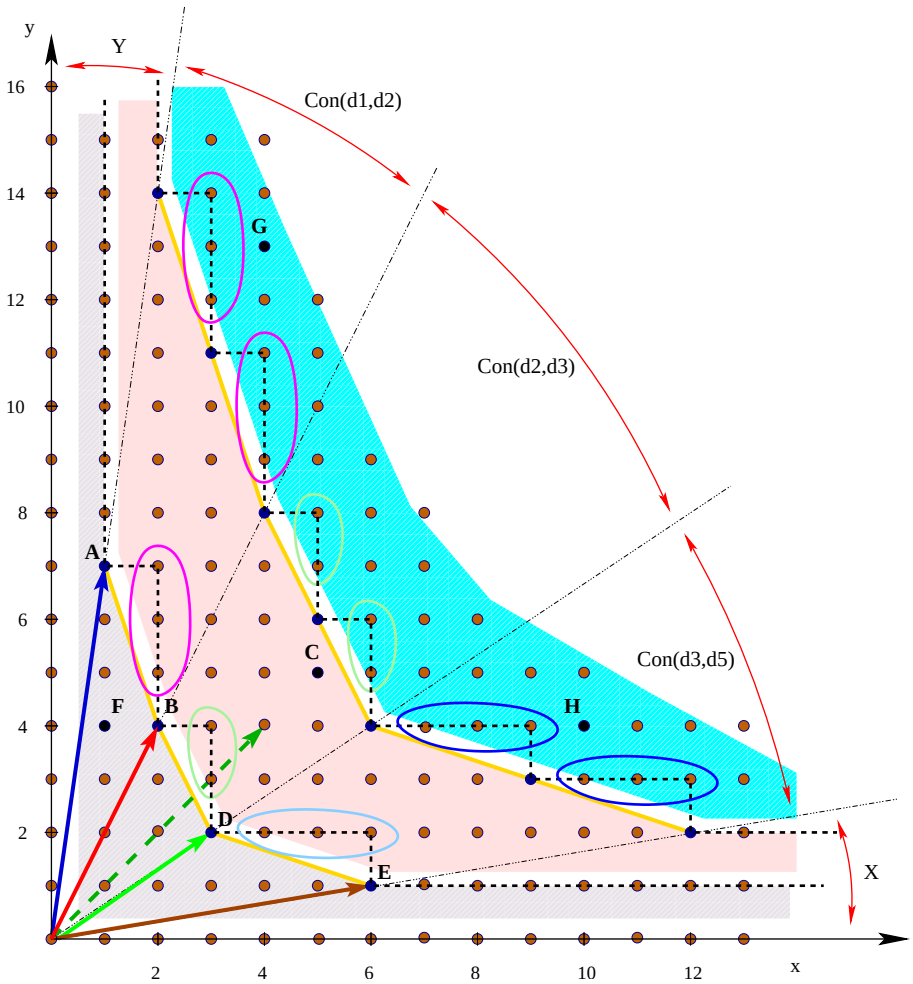

Figure 13: *The index space of an example algorithm with 5 dependence vectors* $\mathbf{d}_1 = (1, 7)$, $\mathbf{d}_2 = (2, 4)$, $\mathbf{d}_3 = (3, 2)$, $\mathbf{d}_4 = (4, 4)$ *and* $\mathbf{d}_5 = (6, 1)$. *Except* $\mathbf{d}_4$, *all other vectors belong to cone vector sequence:* $CV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5 \rangle$. *The 5 cones that formed are colored respectively and the first 3* regions *of all cones are shown.*

Note that execution begins at moment 1. Also note at Figure 13, that the execution of certain index points, is delayed by one time instance. This shows that our scheduling scheme is nearly optimal, as we will explain later on. ■

4.5 Relation to Convex Hull

We must emphasize the equivalence of the problem of finding the cone vectors with that of computing the convex hull of the pattern-vector endpoints and of three more points (d_{xmin_x}, y_{max}), (x_{max}, y_{max}) and (x_{max}, d_{ymin_y}), as shown in Figure 14, where x_{max},y_{max} the upper bounds for x and y-coordinate respectively. If we compute the convex hull and then subtract these three points,

the remaining ones are the endpoints of the cone vectors. For more on this relevant method see [6] and [19][4].

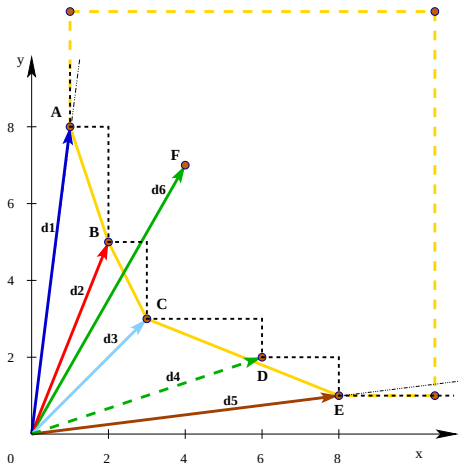

Figure 14: *The convex hull of the pattern-vector endpoints plus the points* (d_{xmin_x}, y_{max}), (x_{max}, y_{max}) *and* (x_{max}, d_{ymin_y}). *The points that form this hull are the cone-vector endpoints plus the three extra points.*

4.6 Issues of Validity and Optimality

It is clear that this scheduling algorithm is valid because it respects the dependencies among the index points. Any point $\mathbf{j}$, belonging to region $\mathcal{R}_i$ is executed at moment i. The fact that $\mathbf{j}$ belongs to region $\mathcal{R}_i$ means that $ECT(\mathbf{j}) \leq i$. Let's see that in more detail.

- Consider a point $\mathbf{j} \in \mathcal{R}_1$. Then, as shown in Figure 15, $\mathbf{j} - \mathbf{d} \notin J^2$ for every $\mathbf{d} \in PV$. Thus, $ECT(\mathbf{j}) = 1$ and $\mathcal{R}_1 \subseteq ECT_1$.

- Consider now a point $\mathbf{j} \in \mathcal{R}_i$. Then, as one can see in Figure 15, $\mathbf{j}-\mathbf{d} \in \bigcup_{l=1}^{i-1} \mathcal{R}_l$ or $\mathbf{j}-\mathbf{d} \notin J^2$ for every $\mathbf{d} \in PV$. If $\mathbf{j}' = \mathbf{j} - \mathbf{d} \in \bigcup_{l=1}^{i-1} \mathcal{R}_l$, then $ECT(\mathbf{j}') \leq i-1$ and $\mathbf{j}'$ was executed at the latest at moment $i-1$. Then we can execute $\mathbf{j}$ at moment i without violating any precedence constraints. Thus $ECT(\mathbf{j}) \leq i$ and $\mathcal{R}_i \subseteq \bigcup_{k=1}^{i} ECT_k$.

The above facts demonstrate the validity of our scheduling scheme. Let us turn now to the question of optimality; is our scheduling optimal and if not, what is the worst case scenario?

Our algorithm delays the execution of any point $\mathbf{j}$, at most one moment, in the worst case. That is, $\mathbf{j} \in \mathcal{R}_i \Rightarrow ECT(\mathbf{j}) = i-1$ or $ECT(\mathbf{j}) = i$ and $\mathcal{R}_i \subseteq ECT_{i-1} \cup ECT_i$. Hence, our scheduling policy is asymptotically optimal. To show that this is the case, consider Figure 16. Any point that belongs to ECT_i and is not included in region $\mathcal{R}_i$, is delayed by one time instance (near optimal

[4]The algorithm that we described above, is a revision of *QuickHull* algorithm [5] for 2-D convex hulls.

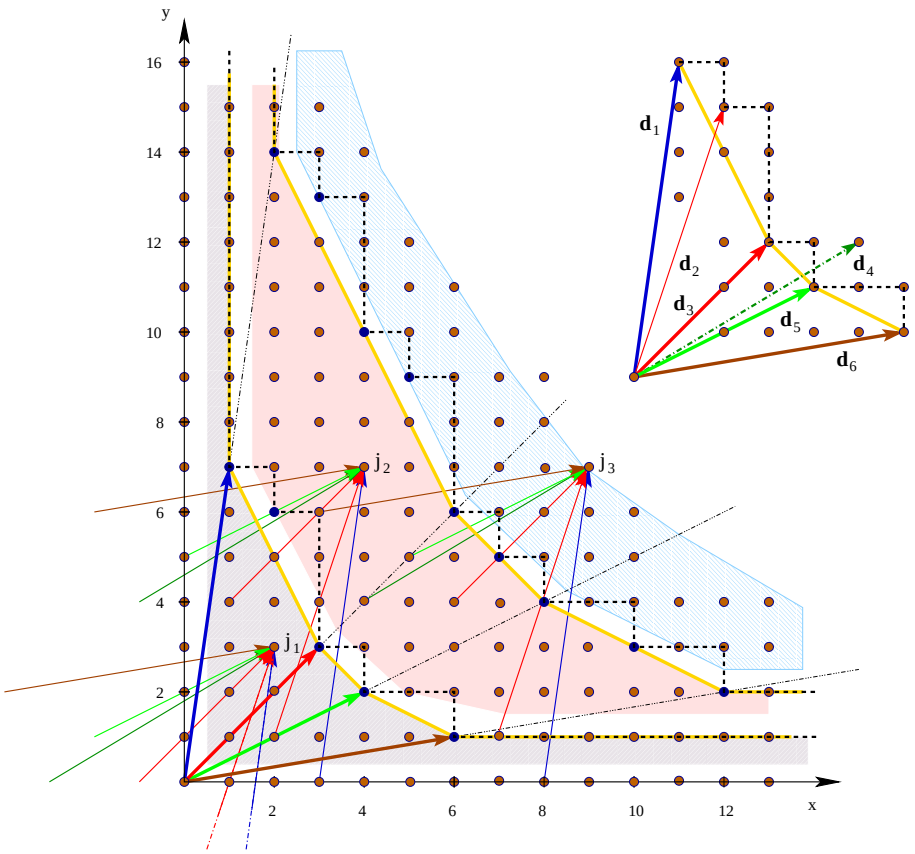

Figure 15: *The index space of Example 1. It is* $PV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_6 \rangle$ *and* $CV = \langle \mathbf{d}_1, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_6 \rangle$. *The index space is partitioned into regions, while the dotted lines denote the* ECT_i *boundary. We consider the cases that* $\mathbf{j} = \mathbf{j}_1 \in \mathcal{R}_1$, $\mathbf{j} = \mathbf{j}_2 \in \mathcal{R}_2$ *and* $\mathbf{j} = \mathbf{j}_3 \in \mathcal{R}_3$. *On the upper right corner all dependence, pattern and cone vectors are shown.*

execution); i.e., point $\mathbf{j}_{npt}$ in figure. On the contrary, all points of ECT_i that are included in $\mathcal{R}_i$, are executed at time instance i (optimal execution); i.e., point $\mathbf{j}_{opt}$ in the previous figure. All points that are delayed by one time instance are also noted, while all other points are scheduled for execution at their earliest computation time.

The above prove that our scheduling procedure is nearly optimal, with a constant delay of, at most, one time instance.

5 Pattern Formation in UCT Problems

Up to this point, we have assumed that any inter-processor communication action needs zero time and only computations take non-zero time (*unit execution time* problems - UET). Now we consider a more realistic model, where communication between different processors requires one

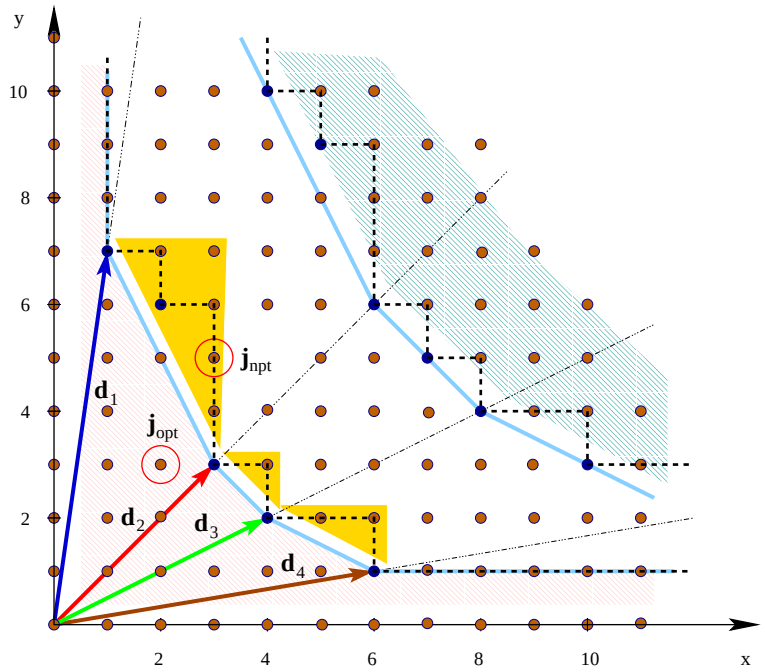

Figure 16: *Optimality issues while scheduling Example 1. The index points of* ECT_1 *that belong to the shaded triangles of* $\mathcal{R}_2$*, i.e.,* $\mathbf{j}_{npt}$*, are delayed by one time instance. They are scheduled at* $t = 2$*, although they can be executed at* $t = 1$*. All other points of* ECT_1 *are scheduled at their earliest computation time* $t = 1$*.*

unit of time, the so called UET-UCT model.

The main difference with the UET model is the communication overhead that is imposed by any interprocessor communication. This means that two successive points, according to dependence vector $\mathbf{d}$, can be computed at successive time instances, only if they are executed at the same processor; in any other case, an extra time instance is needed for communication.

The above denote that in UET-UCT problems there are certain sequences of points that are computed by the same processor. Each such sequence can be viewed as a chain of computations, that is created by a certain dependence vector, as shown in Figure 17. This dependence vector is called *communication vector* and it will be referred to as $\mathbf{d}_c$. A unit time overhead is imposed by communication along all dependence vectors, except $\mathbf{d}_c$.

In the following sections, the methods presented for the problem of pattern prediction, as well as for scheduling a nested loop, according to the UET model, will be revisited, taking into account the communication delay (UET-UCT model).

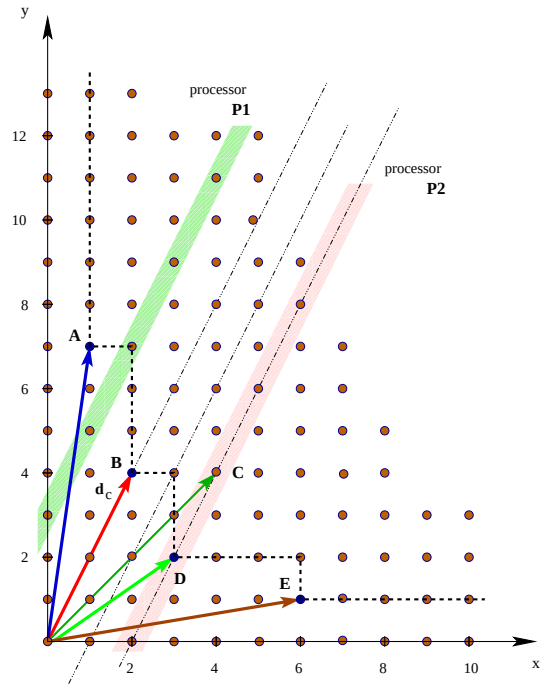

Figure 17: *In unit execution-unit communication time problems (UET-UCT), a unit time overhead is imposed by communication along all dependence vectors, except* $\mathbf{d}_c$*. Index points that belong to the same execution chain formed by* $\mathbf{d}_c$*, are executed by the same processor. In this example* $\mathbf{d}_c = (2, 4)$*.*

5.1 UET-UCT Model Definitions

The main notion in the UET-UCT model, is the communication overhead that is imposed by any interprocessor communication. This means that all dependence vectors, except $\mathbf{d}_c$, impose a unit time overhead. So, the time subset ECT_t, of points with ECT t, for the UET-UCT problems, are recursively defined as follows:

- $ECT_0 = \{boundary\ precomputed\ points\}$
- $ECT_1 = \{\mathbf{j} \in J \mid \mathbf{j} - \mathbf{d}_i \in ECT_0, \text{ for every } \mathbf{d}_i \in DV\}$
- $ECT_t = \{\mathbf{j} \in J \mid \mathbf{j} - \mathbf{d}_i \in ECT_{t-2}, \text{ for every } \mathbf{d}_i \neq \mathbf{d}_c \text{ and } \mathbf{j} - \mathbf{d}_c \in ECT_{t-1}\}$, where $t \geq 2$.

The notion behind the above definition is that: ECT_t is generated by ECT_{t-1} points, according to dependence vector $\mathbf{d}_c$, and by ECT_{t-2}, according to any other dependence vector. The extra time instance denotes the communication overhead (a unit of time).

Some more definitions follow:

- Given the sequence $DV = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m \rangle$ of dependence vectors, $\mathcal{MC}(DV) = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$, is the output of *Algorithm I* on DV. Consequently, for every vector $\mathbf{p}_i = (p_{ix}, p_{iy}) \in$

$\mathcal{MC}(DV)$ there isn't any vector $\mathbf{d}_j = (d_{jx}, d_{jy}) \in DV$ such that $d_{jx} \leq p_{ix}$ and $d_{jy} < p_{iy}$, for $1 \leq i \leq k$ and $1 \leq j \leq m$.

- $PV = \mathcal{MC}(DV) = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle = \langle \mathbf{p}_i \rangle$, $1 \leq i \leq k$
- $DV_2 = \langle \mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m \rangle$, $1 \leq j \leq m$ and $j \neq c$
- $DV_{UET} = \langle \mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, 2\mathbf{d}_c, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m \rangle$, $1 \leq r \leq m$

Note that PV includes all pattern vectors, DV_2 includes all pattern vectors except $\mathbf{d}_c$, while $DV_{UET} = DV_2 \cup 2\mathbf{d}_c$.

5.2 General Pattern Formation

We are searching for a formal expression of the outline of the t-step pattern. By the definition of ECT_1, any point $\mathbf{p} \in pat_1$ is given from the following expression (quite similar to UET case):

$$\mathbf{p} = i_1 \cdot \mathbf{p}_1 + i_2 \cdot \mathbf{p}_2 + \ldots + i_k \cdot \mathbf{p}_k\},$$

where $i_1 + i_2 + \ldots + i_k = 1$ and $\mathbf{p}_i, 1 \leq i \leq k$, the k pattern vectors of the algorithm.

The main difference from the UET case is depicted when the general t-step pattern is concerned. It depends on the *two* previous patterns, as at every processor P at time instance t, the points that are available are:

- all points calculated at moment $t-1$ by the same processor and
- all points calculated at moment $t-2$ by any other processor –*each of these points need an additional time instance to reach processor* P.

The above determine quite a different execution flow, as the one shown in Figure 18.

The following lemmata and theorems prove the analytical expression of the general pattern outline. Their proofs can be found in [14].

Lemma 5.1 $\mathcal{MC}(\langle PV + \mathbf{d}_c \rangle \cup DV_2) = \mathcal{MC}(DV_{UET})$.

Lemma 5.2 $\mathcal{MC}(\langle DV_{UET} + \mathbf{d}_c \rangle \cup \langle PV + DV_2 \rangle) = \mathcal{MC}(PV + DV_{UET})$.

The following expressions of the t-step pattern outline, for the UET-UCT case, can be derived obviously from the definition of ECT_t:

- $pat_0 = \langle (0,0) \rangle$
- $pat_1 = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle = PV$

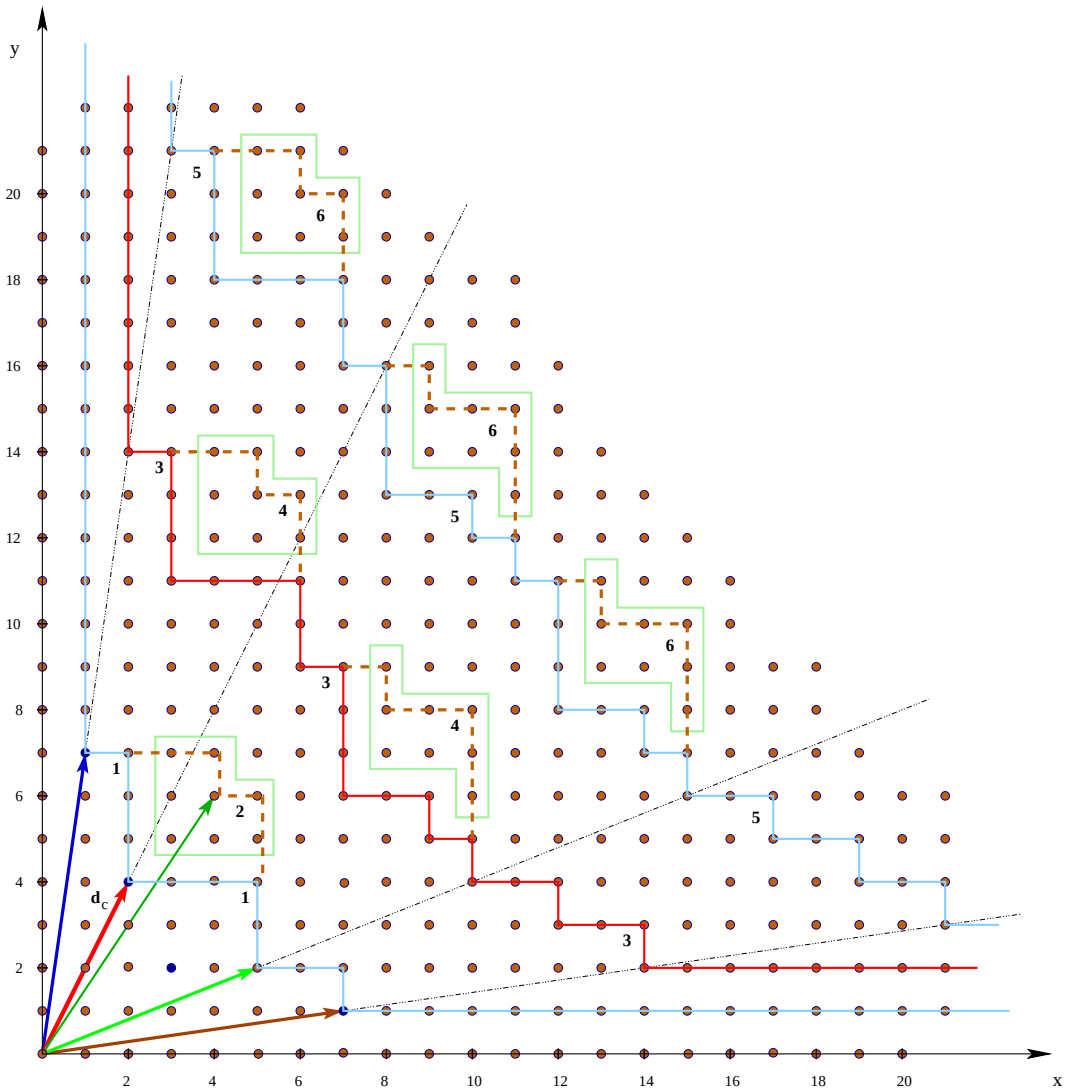

Figure 18: *In the this figure, an algorithm with 5 dependence vectors is shown* $\mathbf{d}_1 = (1, 7)$, $\mathbf{d}_2 = (2, 4)$, $\mathbf{d}_3 = (4, 6)$, $\mathbf{d}_4 = (5, 2)$ *and* $\mathbf{d}_5 = (7, 1)$. *The UET-UCT model is concerned with* $\mathbf{d}_c = \mathbf{d}_2 = (2, 4)$. *The initial pattern border is depicted by the light colored line, as pattern 5 boundary also. Dark line marks pattern 3 boundary, while the dashed polygons show the points that are executed at the even time instances (2, 4, . . .).*

- $pat_t = \mathcal{MC}(\langle pat_{t-2} + DV_2 \rangle \cup \langle pat_{t-1} + \mathbf{d}_c \rangle)$

Using the lemmata just presented, the above expressions will be transformed, according to the following theorem.

Theorem 5.1 *Given a doubly nested loop algorithm, with dependence vector sequence* $DV = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m \rangle$, *and pattern vector sequence* $PV = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$, *the outline of the i-step pattern can be written as:*

- $pat_0 = \langle (0,0) \rangle$
- $pat_{2t+1} = \mathcal{MC}(\{pat_{2t} + PV)$
- $pat_{2t+2} = \mathcal{MC}(\{pat_{2t} + DV_{UET})$.

The above theorem proves that the wavefront of the execution propagation, can be expressed formally, in the UET-UCT case. Furthermore, the pattern outline of the odd time instances, is identical to the outline of a UET algorithm with slightly changed dependence vector set: $DV_{UET} = \langle \mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, 2\mathbf{d}_c, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m \rangle$. Instead of the communication vector $\mathbf{d}_c$, it contains $2\mathbf{d}_c$.

For a given UET-UCT problem, the pure UET problem with the same PV, except that includes $2\mathbf{d}_c$ instead of $\mathbf{d}_c$, will be called equivalent UET algorithm.

5.3 Scheduling UET-UCT Loops

Due to the previous theorem, we get an equivalence of any UET-UCT problem, with a pure UET one, with slightly changed dependence vector set. This is clearly shown in Figure 19. A UET-UCT algorithm is concerned, and the wavefront of the execution is shown, while considering as $\mathbf{d}_c$, the noted dependence vector. Right besides, we show the execution wavefront of the UET algorithm that contains all dependence vectors except $\mathbf{d}_c$, plus $2\mathbf{d}_c$.

Note the equivalence of $2i$-th outline of the UET-UCT execution wavefront, with the i-th outline of the UET one, where $i = 1, 2, 3, \ldots$. This equivalence denotes that we can schedule any UET-UCT problem, according to the equivalent UET one. By considering the partitioning of the index space into regions, we schedule each point according to the region it belongs.

The only difference is that the calculated time instance t, of the equivalent UET problem, refers to $i = 2t$ and $i' = 2t + 1$ time instances of the original UET-UCT problem. So, the scheduling will be performed at three faces instead of two:

- specify the region t that any given point belongs, in the equivalent UET problem,
- specify the actual time instance $i = 2t$ or $i' = 2t + 1$ that the point should be scheduled at, by performing an additional test, and then
- schedule the given point at time i or i' according to the previous test.

What we need, is an additional test that distinguishes an index point $\mathbf{j}$ that belongs to the odd time instance ($i' = 2t + 1$) from an other $\mathbf{j}'$ that belongs to the even one ($i = 2t$), given the fact

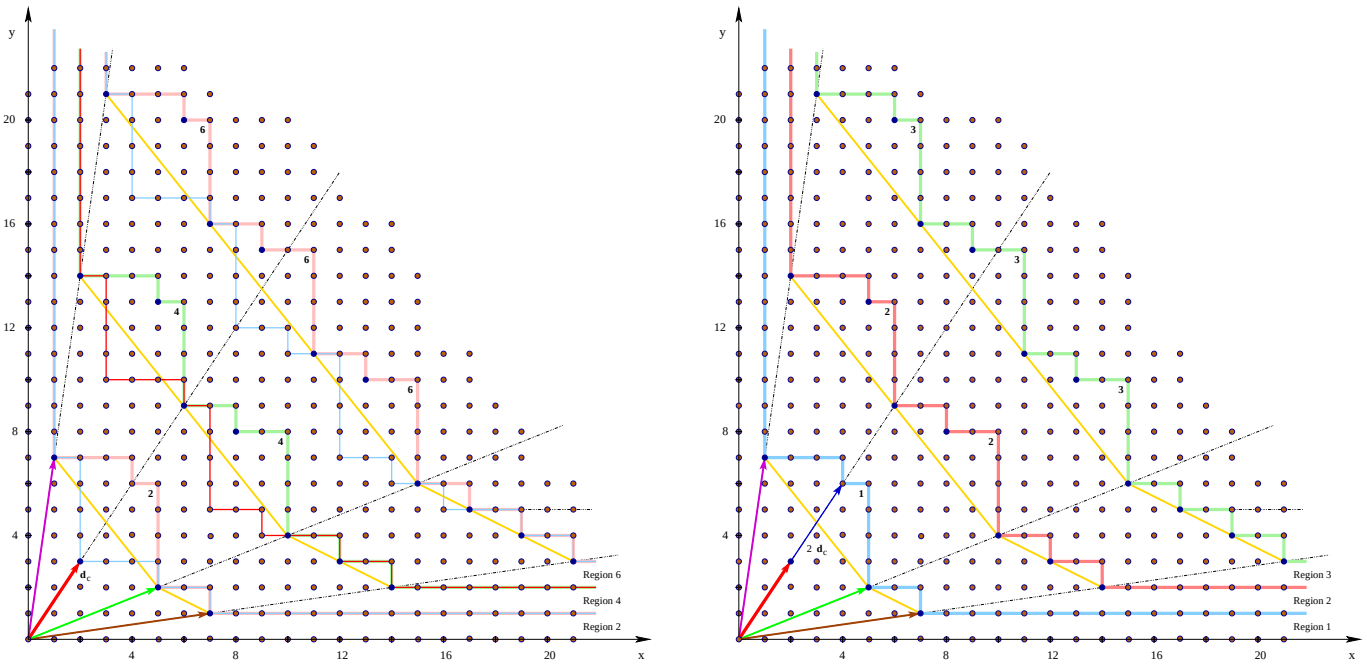

Figure 19: *The execution propagation of* ***a)*** *a UET-UCT problem with dependence vectors* $\mathbf{d}_1 = (1, 7)$, $\mathbf{d}_2 = (2, 3)$, $\mathbf{d}_3 = (5, 2)$ *and* $\mathbf{d}_4 = (7, 1)$, *with* $\mathbf{d}_c = \mathbf{d}_2$ *and* ***b)*** *a UET problem with dependence vectors* $\mathbf{d}_1$, $2\mathbf{d}_2$, $\mathbf{d}_3$ *and* $\mathbf{d}_4$. *Note the equivalence of* $2i$*-th outline of the UET-UCT problem with the* i*-th outline of the UET one, where* $i = 1, 2, 3, \ldots$.

that $\mathbf{j}$ and $\mathbf{j}'$ belong to t region of the equivalent UET problem, that defined above. Using the results of Theorem 5.1, we can express pat_{2k+1} in relation to pat_{2i} and pat_{2i+2}, only. It is:

$$\begin{aligned}
pat_{2i+1} &= \mathcal{MC}(pat_{2i} + PV) = \mathcal{MC}(\langle pat_{2i} + DV_2\rangle \cup \langle pat_{2i} + \mathbf{d}_c\rangle) \\
&= \mathcal{MC}(\langle pat_{2i} + DV_2\rangle \cup \langle pat_{2i} + \mathbf{d}_c\rangle \cup \langle pat_{2i} + 2\mathbf{d}_c\rangle) \\
&= \mathcal{MC}(\langle pat_{2i} + DV_2\rangle \cup \langle pat_{2i} + 2\mathbf{d}_c\rangle \cup \langle pat_{2k} + \mathbf{d}_c\rangle) \\
&= \mathcal{MC}(\langle pat_{2i} + DV_{UET}\rangle \cup \langle pat_{2i} + \mathbf{d}_c\rangle) \\
&= \mathcal{MC}(\mathcal{MC}(\langle pat_{2i} + DV_{UET}\rangle) \cup \langle pat_{2i} + \mathbf{d}_c\rangle) \\
&= \mathcal{MC}(pat_{2i+2} \cup \langle pat_{2i} + \mathbf{d}_c\rangle)
\end{aligned}$$

From the above expression, it is denoted that if $\mathbf{j}$ belongs to t region in the equivalent UET problem, it belongs to $i' = 2t + 1$ time instance, if and only if $\mathbf{j} - \mathbf{d}_c$ belongs to t region also. If this is not the case, it belongs to $i = 2t$ time instance. This fact is clearly explained in Figure 20.

The following algorithm is a revision of *Algorithm IV* that works in UET-UCT problems. It implements the procedure that was just described. The index space is partitioned into cones and regions and point location is performed. After that, the additional test is performed, in order to specify the exact execution instance for each point.

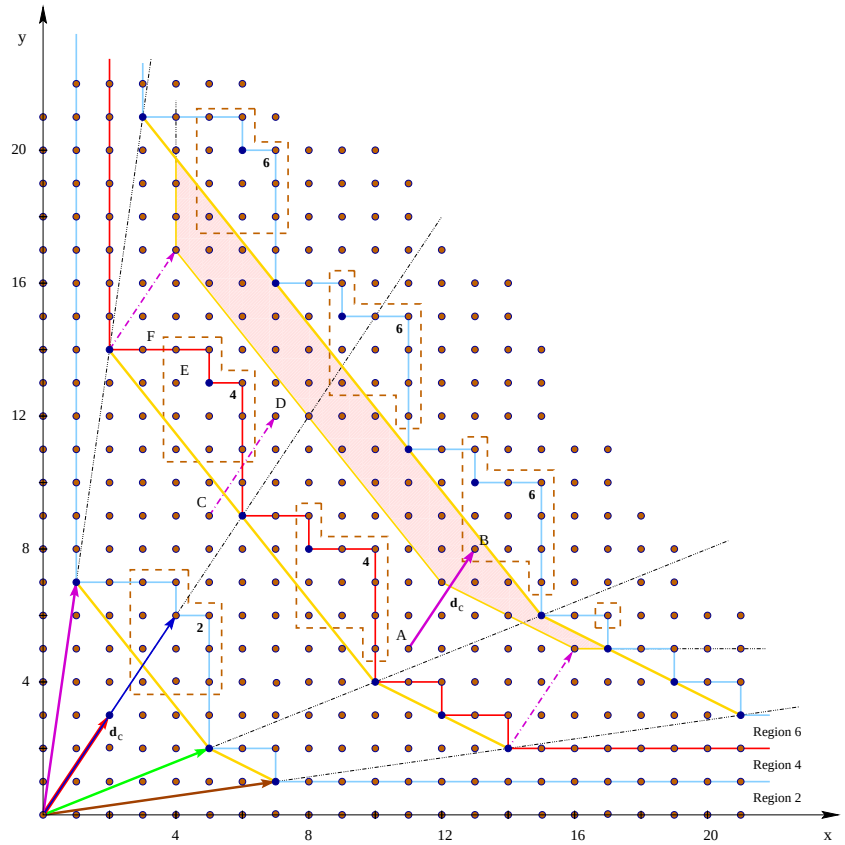

Figure 20: *Scheduling according to* Algorithm V. *The index space of a doubly nested loop with 4 dependence vectors* $\mathbf{d}_1 = (1,7)$, $\mathbf{d}_2 = (2,3)$, $\mathbf{d}_3 = (5,2)$ *and* $\mathbf{d}_5 = (7,1)$ *is shown. Zero communication cost is achieved along* $\mathbf{d}_2$ *(*$\mathbf{d}_c \equiv \mathbf{d}_2$*). The open polygonal lines indicate the outline of the points that can be executed at time less or equal to* $2k$*, where* $k = 1, 2, 3, \ldots$ *The dotted closed polygons denote the exact points that can be executed at time instance* $2k$*, while the shaded one shows the points that will be executed at time instance 6.*

Algorithm V *(Scheduling a UET-UCT Loop)*

INPUT: The dependence vector set $DV = \{\mathbf{d}_1, \ldots, \mathbf{d}_m\}$ of the algorithm, communication vector $\mathbf{d}_c$ and an index point $\mathbf{j}$.

OUTPUT: The time instance t'_j that point $\mathbf{j}$ can be executed.

1. Communication vector $\mathbf{d}_c$ is extracted from DV and $2\mathbf{d}_c$ is applied instead; a new dependence set is formed: $DV_{UET} = \{\mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, 2\mathbf{d}_c, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m\}$ –equivalent UET algorithm.

2. *Algorithm IV* is applied on DV_{UET} and point $\mathbf{j}$; time instance t_j is computed (t_j is the execution time in the equivalent UET problem). The candidate execution instances for point $\mathbf{j}$ are $2t_j - 1$ and $2t_j$.

3. *Algorithm IV* is applied now on DV_{UET} and point $\mathbf{j} - \mathbf{d}_c$; time instance t_{j-d_c} is computed.

▷ if $t_{j-d_c} = t_j \Rightarrow$ point j is executed (at the initial UET-UCT problem) at $t'_j = 2t_j$.

▷ if $t_{j-d_c} \neq t_j \Rightarrow$ point j is executed at $t'_j = 2t_j - 1$.

The above algorithm is explained in Figure 20. Regions are named by even numbers only, as they are the boundary of even time instance pattern outlines. Points **B** and **D** are located, using *Algorithm IV*, into region 6. In other words, the candidate time instances for their execution are 5 and 6.

However, point $\mathbf{A} = \mathbf{B} - \mathbf{d}_c$ resides into previous region, while point $\mathbf{D} = \mathbf{C} - \mathbf{d}_c$ resides in current region. So, point **C** will be executed at time instance $t_C = 5$ and point **B** at time instance $t_B = 6$.

The intuition behind that "movement" of pat_{2k} along $\mathbf{d}_c$ is that, although $\mathbf{d}_c$ imposes no communication cost, it does not disappear as a computational dependence, which may distinguish two consequent time instances.

5.4 Validity and Optimality

As far as algorithm's execution optimality is concerned, the points with *even* execution time are delayed at most by 1 time instance, while the points with *odd* execution time are delayed at most by 2 (points **E** and **F** in Figure 20 correspondingly). Hence, our scheduling algorithm is asymptotically optimal, with a constant delay of at most 2.

5.5 Generalization to Higher Dimensions

We are currently working on comparing exhaustively our method with the hyperplane ones [22, 30, 31, 15]. Preliminary results indicate that the proposed method offers a dramatic increase in performance. This increase is of the order of 10%-35%, with respect to the best previously known methods. Consider for example the following nested loop:

```
for (i=1; i<=26; i++)
    for (j=1; j<=26; j++) {
        a(i,j) = 2*a(i-1,j-3) + 4*b(i-2,j-2);
        b(i,j) = a(i-4,j-1) + 3*b(i-5,j-4);
    }
```

If we use the algorithm described in [30], the execution of the above loop requires 26 moments, whereas our method only takes 12 moments. Of course, Sheu et all give emphasis to mapping the index space onto an array of processors.

Although our method has been described for two dimensional problems, it can be generalized for n dimensions. For the transformation of UET-UCT problems to UET ones, a similar method can be described for the n-dimensional cases, considering the hypersurfaces, formed by the end-points of the dependence vectors, as pattern boundaries, leading to the same conclusion.

As far as the scheduling algorithm is concerned, we present the generalization idea for the three dimensions.

We need to generalize the cone-vector computation algorithm. In the presented algorithm for two dimensions, the pattern vectors that are initially concerned as cone-vectors are $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$. Since any vector $\mathbf{d} = (d_x, d_y)$, with $d_x \geq d_{xmin_x}$ or $d_y \geq d_{ymin_y}$, is not included in PV, we conclude that $\mathbf{d}_{xmin}$ has the maximum y-coordinate and $\mathbf{d}_{ymin}$ has the maximum x-coordinate. Then, cone-vector computation continued incrementally, by considering the distances of all pattern vector end-points from the line segment that connects the end-points of $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$, as explained in Figure 12.

For three dimensional algorithms, the dependence vectors that are initially concerned as cone-vectors, are the ones that have the maximum x, y and z-coordinate, respectively, in PV. These vector end-points form a *plane* (a line in 2-D, a hyperplane in n dimensions). Cone-vector computation can continue incrementally, as before, by considering the distances of all pattern vector end-points from the plane mentioned. The procedure will continue incrementally as above.

6 Conclusion

In this paper we presented an asymptotically optimal scheduling method for uniform dependence nested loop algorithms, considering the UET-UCT model. We described an exponential procedure for computing the precise t-step pattern wavefront, which is the best we can get so far. A nearly optimal scheduling algorithm, based on the geometric properties of the index space, was proposed, for any UET-UCT nested loop.

Finally, we presented a new method for transforming a UET-UCT problem to a pure UET equivalent one. This last method is a major improvement, as all implemented methods for the UET algorithms, can be now applied to UET-UCT ones. To the best of our knowledge, it is the first method presented so far. Our method has been implemented and tested in numerous examples, proving its superiority over existing methods. It can be generalized to handle an arbitrary n-dimensional uniform dependence loop.

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, G. Papakonstantinou, and P. Tsanakas. Optimal scheduling for UET/UET-UCT generalized n-dimensional grid task graphs. *Journal of Parallel and Distributed Computing*, 57(2):140–165, May 1999.

[3] 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.

[4] T. Andronikos, N. Koziris, Z. Tsiatsoulis, G. Papakonstantinou, and P. Tsanakas. Lower time and processor bounds for efficient mapping of uniform dependence algorithms into systolic arrays. *Journal of Parallel Algorithms and Applications*, 10(3-4):177–194, 1997.

[5] C. B. Barder, D. P. Dobkin, and H. Hundanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, December 1996.

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

[7] P. Chretienne. A polynomial algorithm to optimally schedule tasks over a virtual distributed system under tree-like precedence constraints. *European Journal of Operational Research*, 43:225–230, 1989.

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

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

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

[11] M. de Berg, M. van Kreveld, M. Overmans, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[12] I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric pattern prediction and scheduling of uniform dependence loops. In *Proceedings of the 5th Hellenic European Conference on Computer Mathematics and its Applications - HERCMA 2001*, Athens, 2001.

[13] I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric scheduling of 2-D uniform dependence loops. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS 2001*, pages 259–264, Korea, 2001.

[14] 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.

[15] I. Drositis, G. Goumas, N. Koziris, P. Tsanakas, and Papakonstantinou. Evaluation of loop grouping methods based on orthogonal projection spaces. In *Proceedings of the 2000 International Conference on Parallel Processing ICPP-2000*, pages 469–476, Toronto, Canada, 2000. IEEE Press.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, 1979.

[17] H. Jung, L. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor sceduling of dags with communication delays. In *Proceedings of the 1st ACM SPAA 1989*, pages 254–264, 1989.

[18] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform reccurence equations. *Journal of the Association of Computing Machinery*, 14(3):563–590, July 1967.

[19] D. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986.

[20] 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.

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

[22] D. Moldovan and J. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–11, 1986.

[23] 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.

[24] P. Picouleau. *Etude de Problèmes d' Optimization dans les Systèmes Distribués*. PhD thesis, Université Pierre et Marie Curie, 1992.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[26] V. J. Rayward-Smith. Uet scheduling with unit interprocessor communication delays and unlimited number of processors. *Discrete Applied Mathematics*, 18:55–71, 1987.

[27] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[28] W. Shang and J. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6):723–742, 1991.

[29] 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.

[30] J. Sheu and T. Tai. Partitioning and mapping nested loops on multiprocessors systems. *IEEE Transactions on Parallel and Distributed Systems*, 24:430–439, 1991.

[31] P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain Grouping: A method for partitioning loops onto mesh-connected processor arrays. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):941–955, September 2000.

[32] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[33] T. Varvarigou, V. Roychowdhury, and T. Kailath. Scheduling In and Out forests in the presence of communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 8(10), 1996.

Appendix

Lemma 4.1 *Consider a nested loop algorithm with two dependence difference vectors* $\mathbf{a}'_1$ *and* $\mathbf{a}'_2$ *and their slopes*[5] g_1 *and* g_2*, respectively. If* $g_1 \geq g_2$ *and* $a'_{1y} \geq a'_{2y}$ *holds, then the points that contribute to pattern outline formation are (as in Figure 21):*

- *the endpoints of* $i \cdot \mathbf{a}'_1$*, where* $0 \leq i \leq t$*,*
- *potentially, the endpoints of* $i \cdot \mathbf{a}'_2$*, that reach the triangles formed by* $\mathbf{a}'_1$*.*

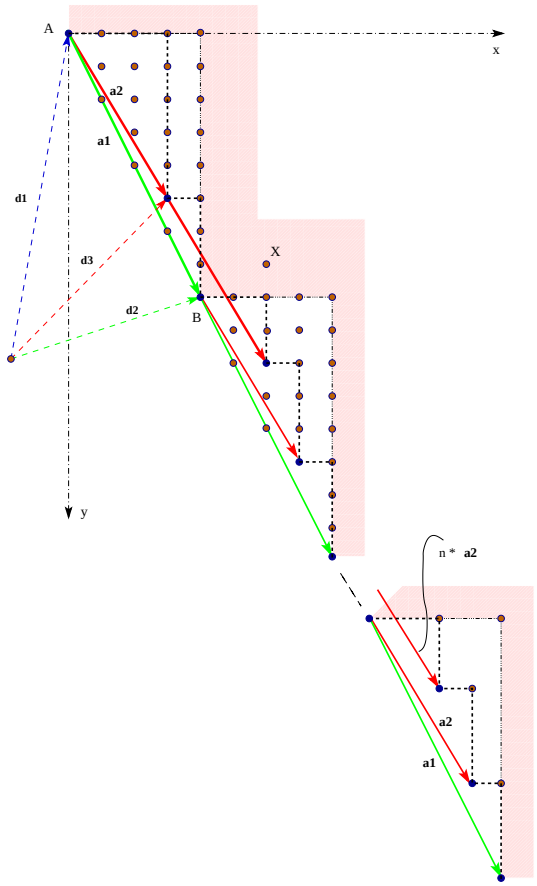

Figure 21: *Pattern outline depends on the slopes of the difference vectors. It is comprised by distinct parts, whose outline is the triangle formed by* $\mathbf{d}_{xmin}$ *and* $\mathbf{d}_{ymin}$*, as shown. The pattern points are determined by the endpoints of a linear combination of the algorithm dependence difference vectors.*

Proof[6]

Obviously, as difference vector $\mathbf{a}'_1$ is the one with the greatest slope, the end points of $k \cdot \mathbf{a}'_1$ will all

[5] Using the coordinate system of Figures 21 and 22.

[6] Based on the formulas of §3.3.

be pattern outline points. All extra (potential) points will be end points of vectors given by the expression: $k_1 \cdot \mathbf{a}'_1 + k_2 \cdot \mathbf{a}'_2$, where $0 \leq k_1, k_2 \leq i$ and $k_1 + k_2 = t$ (for the t time instance). Also, if a point X belongs to the shaded region shown, it doesn't affect the pattern, as there is a point with less x and y-coordinate which eliminates its influence (point B). ⊣

Lemma 4.2 *Consider m dependence difference vectors $\mathbf{a}_i$ and their slopes g_i, such that $g_1 \geq g_i$ for $2 \leq i \leq m-1$. Without loss of generality $a_{1y} \geq a_{iy}$, $2 \leq i \leq m-1$. Then the previous Lemma holds for $\mathbf{a}'_1 = \mathbf{a}_1$ and $\mathbf{a}'_2$ any linear combination of the form $k_2 \cdot \mathbf{a}_2 + k_3 \cdot \mathbf{a}_3 + \ldots + k_m \cdot \mathbf{a}_{m-1}$, where $0 \leq k_i \leq m-1$ and $k_2 + k_3 + \ldots + k_{m-1} = t$.*

Proof

Obvious from previous Lemma and $\mathbf{a}_i$ definitions. ⊣

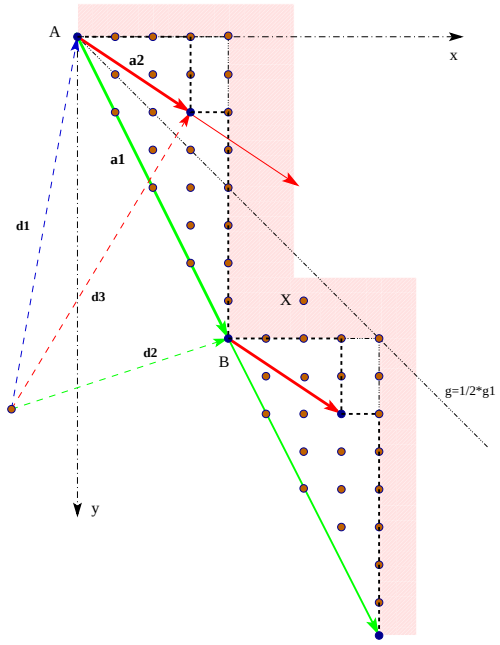

Figure 22: *Theorem explanation; consider the difference vector $\mathbf{a}_{gmax} \equiv \mathbf{a}_1$, as the one with the largest slope value g_{max}. If the slope values of all other difference vectors are less than the half of g_{max}, the outline of each pattern can be produced by a repeated "subpattern".*

Theorem 4.1 *Consider a nested loop algorithm with k pattern vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k$. Let $\mathbf{a}_i = \mathbf{p}_i - \mathbf{p}_{i+1}$ be the $k-1$ difference vectors, with slope values g_i. If the following hold, for $2 \leq i \leq k-1$:*

1. *$g_1 \geq 2 \cdot g_i$,*
2. *$a_{1y} \geq a_{iy}$ and*

3. $a_{ix} \geq \frac{a_{1x}}{2}$

then the pattern outline is recurrent: it can be computed by replicating every pattern part by the first part (which is the pattern outline of the first time instance).

Proof

It is obvious that every linear combination of the difference vectors $\mathbf{a}_i$, is located at the shaded area, because for the resultant slope g_{sum} it is: $2 \cdot g_{sum} \leq g_1$ and $a_{sum_x} \geq a_{1_x}$, because of (3). So, for the pattern outline formation only vectors $\mathbf{a}_i$ contribute. Every vector $k_i \cdot \mathbf{a}_i$, due to (1), (2) and (3), creates only one point at every triangle, at the same (relative) position. According to the above and from the two previously presented lemmata, the pattern outline is recurrent and it is formed by the pattern of the first time instance. $\dashv$