Geometric Scheduling of 2-D UET-UCT Uniform Dependence Loops*

Ioannis Drositis, Theodore Andronikos, George Manis,
George Papakonstantinou and Nectarios Koziris

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

Email: jdros@cslab.ntua.gr

Abstract

Finding an optimal time schedule 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 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*: UET-UCT loop scheduling.*

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. 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 (fine-grain).

Time scheduling of loop-based iterations with uniform dependencies, was introduced by Karp in [12] (for uniform recurrence equations) and elaborated by Lamport in [14], who partitioned the index space into hyperplanes. All iteration-indexed points belonging to the same hyperplane can be concurrently executed. Darte in [7], proved that this method is nearly optimal, while the problem of finding the hyperplane 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 [15], linear programming in subspaces [18], integer programming [7], or graph-theoretic approaches [4]. All these approaches consider unit execution time for each iteration and *zero* communication for each communication step (UET model).

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 Garey [11]), researchers have given attention to other methods such heuristics, approximation algorithms etc.

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 [17] and by Picouleau in [16]. Even the case of unlimited processors, when task duplication is allowed, is in general polynomially intractable [16]; when the index space is limited another optimal approach in presented in [13]. Restricting to special cases decreases dramatically the complexity, resulting in polynomial time solutions as in [6, 4].

In [5] 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

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

graphs and trees with UCT [19].

A nearly optimal scheduling approach for 2-D UET uniform nested loops is presented in [9], based on the geometric properties of the index space and the dependence vectors. The index space of such a loop, can be partitioned into a sequence $ECT_1, ECT_2, \ldots$ of (disjoint) subsets, such that ECT_t contains precisely those points whose earliest computation time is t. The geometric representation of these subsets form certain wavefront-like polygonal shapes called *patterns* (Figure 1). The aforementioned technique, partitions the index space into geometric regions that approximate patterns and have simpler representation, and schedules computations according to the region they belong to.

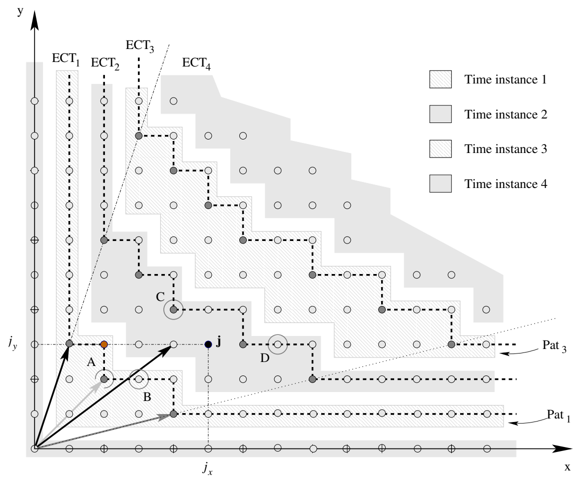

Figure 1. *Consider an algorithm with dependence vectors* $\mathbf{d}_1 = (1,3)$*,* $\mathbf{d}_2 = (2,2)$*,* $\mathbf{d}_3 = (4,1)$ *and* $\mathbf{d}_4 = (4,3)$*. We shown the four initial patterns.*

In this work, we present an extension of the above approach, considering communication delays that arise in more general architectures. We do this by introducing a transformation of UET-UCT problems to equivalent UET problems. This is the major contribution of this paper, as no other relative method has been proposed so far. All present UET-UCT scheduling techniques refer to grid or extended grid architectures [2] and to graph and tree algorithms [16, 17]. The UET-UCT polynomial scheduling algorithm is nearly optimal, as it achieves a makespan of constant delay. This algorithm can be generalized for higher dimensions.

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.

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);
        ...
        Sk(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 [14]; the dependence vectors are "uniform", i.e., constant all over the index space.

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_t, $t \geq 0$, such that ECT_t contains the set of points of J, whose earliest computation time is t. 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 ECT_t is called **pattern** of instance t and is denoted as Pat_t (see Figure 1). For consistency we define $Pat_0 = ECT_0$.

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 endpoints* satisfy the following condition: $J_{\mathbf{d}_1} \cup \ldots \cup J_{\mathbf{d}_m} = J_{\mathbf{p}_1} \cup \ldots \cup J_{\mathbf{p}_k}$. Their number is denoted as $k \leq m$.

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

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

Given the m dependence vectors $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m$, we give the following definitions:

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 $\lambda_1, \lambda_2 \geq 0\}$.

The cones formed by the x and y axes and the dependence vectors with the least x and y-coordinate, respectively, are called Y-cone and X-cone. These vectors are called $\mathbf{d}_{xmin}$ and $\mathbf{d}_{ymin}$ and the cone they defined is called D-cone= $Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$ –see Figure 4.

*We assume that the origin of the dependence vectors is the origin of the axes.

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

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

Cone Region: The i-th region of a cone $Con(\mathbf{a}, \mathbf{b})$ is the set $Con_i(\mathbf{a}, \mathbf{b}) = \{\mathbf{j} \in Con(\mathbf{a}, \mathbf{b}) \mid \mathbf{j} = \lambda_1 \cdot \mathbf{a} + \lambda_2 \cdot \mathbf{b}$, where $i < \lambda_1 + \lambda_2 \leq i+1\}$ –see Figure 3.

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

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

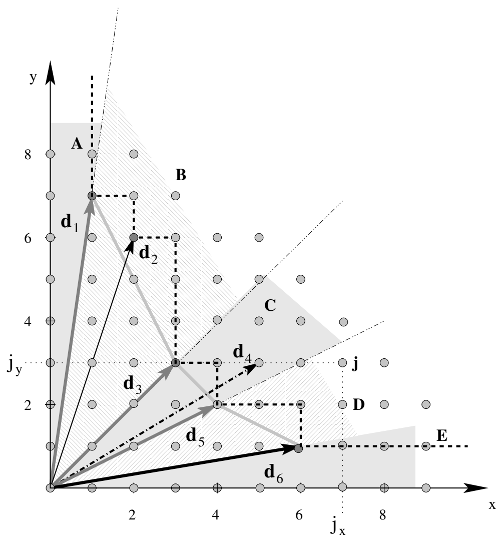

Figure 2. *Except* $\mathbf{d}_2$ *and* $\mathbf{d}_4$, *all other vectors belong to the cone vector sequence:* $CV = \langle \mathbf{d}_1, \mathbf{d}_3, \mathbf{d}_5, \mathbf{d}_6 \rangle$. *The 5 cones that are formed are colored respectively.*

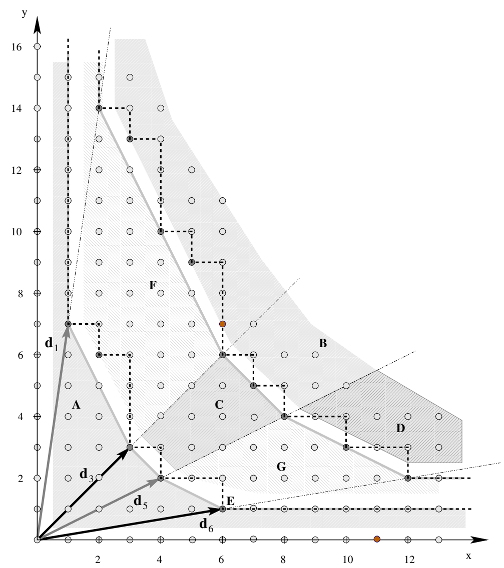

Figure 3. *For the algorithm shown in Figure 2, we show the first 3* regions *of all cones and cone regions* $Con_2(\mathbf{d}_3, \mathbf{d}_5)$ *and* $Con_3(\mathbf{d}_5, \mathbf{d}_6)$.

We also define, the problem that we are going to study. Given a nested loop algorithm, with index space J and dependence vector sequence DV:

Given any index point, compute the time instance that it should be executed; in other words, specify the region that includes this point.

3 Scheduling with Cones & Regions

Our method 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 4 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 $\lambda_2 > 0 \Rightarrow \mathbf{j} \in Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$
- if $\lambda_1 > 0$ and $\lambda_2 < 0 \Rightarrow \mathbf{j} \in Y$-cone
- if $\lambda_1 < 0$ and $\lambda_2 > 0 \Rightarrow \mathbf{j} \in X$-cone

For example, considering Figure 4, 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 $\lceil \frac{2}{1} \rceil = 2$.

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 II*).

After that, we simply calculate in which specific region $\mathbf{j}$ belongs, using the following expressions, as shown in Figure 4:

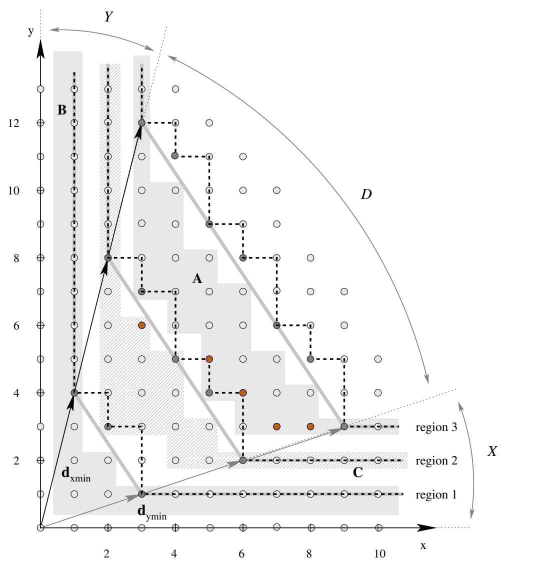

Figure 4. *Point locating in a 3-cone index space, with* $\mathbf{d}_{xmin} = (1, 4)$ *and* $\mathbf{d}_{ymin} = (3, 1)$.

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

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

▷ if $\mathbf{j} \in Con(\mathbf{d}_{xmin}, \mathbf{d}_{ymin})$ ⇒ [#region] = $\lceil \lambda_1 + \lambda_2 \rceil$, where $\mathbf{j} = \lambda_1 \cdot \mathbf{d}_{xmin} + \lambda_2 \cdot \mathbf{d}_{ymin}$ (e.g., $A \in \mathcal{R}_3$).

So, summing up, we partition the 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.

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.

3.1 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 use the **minimum cover** operator, denoted $\mathcal{MC}$. This operator 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:

$$J_{\mathbf{p}_1} \cup J_{\mathbf{p}_2} \cup \ldots \cup J_{\mathbf{p}_k} = J_{\mathbf{d}_1} \cup J_{\mathbf{d}_2} \cup \ldots \cup J_{\mathbf{d}_m}$$

Using this operator, we have that $PV = \mathbf{p}_1, \ldots, \mathbf{p}_k = \mathcal{MC}(\langle \mathbf{d}_1, \ldots, \mathbf{d}_m \rangle)$.

Algorithm I *(Cone-vector computation)*
`DivideConeVector(`PV, $\mathbf{d}_{xmin}$, $\mathbf{d}_{ymin}$`)`
INPUT: algorithm pattern-vectors $PV = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$
OUTPUT: the r cone vectors $CV = \langle \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_r \rangle$, $r \leq k \leq m$

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

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

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 3.

3.2 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 $Con(\mathbf{c}_k, \mathbf{c}_{k+1})$ 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 II *(Scheduling in a Multi-Cone Space)*
INPUT: $DV = \langle \mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m \rangle$ and an index point $\mathbf{j}$.
OUTPUT: the time instance t_j that $\mathbf{j}$ can be executed.

1. *Algorithm I* is applied on DV 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 I* 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 $\lambda_{12} < 0 \Rightarrow \mathbf{j} \in Y$-cone
- Else, if for $(\mathbf{c}_{r-1}, \mathbf{d}_{ymin})$, $\lambda_{r1} < 0$ and $\lambda_{r2} \geq 0 \Rightarrow \mathbf{j} \in X$-cone
- Else, if $\lambda_{i1} \geq 0$ and $\lambda_{i2} \geq 0 \Rightarrow \mathbf{j} \in Con(\mathbf{c}_i, \mathbf{c}_{i+1})$.

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

4 Pattern Formation in UCT Problems

Up to this point, we have assumed that any interprocessor 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 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 index points 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.

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 5. We call this dependence vector the *communication vector* and denote it 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).

4.1 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:

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

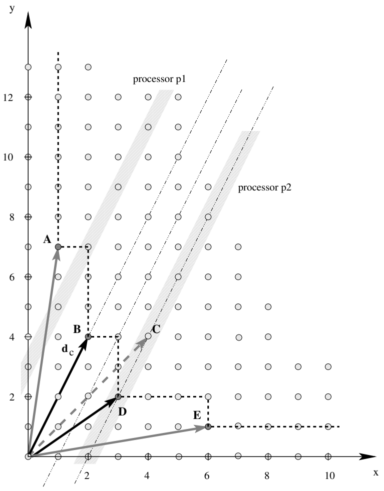

Figure 5. *In UET-UCT problems, a unit time overhead is imposed by communication along all dependence vectors, except $\mathbf{d}_c$. Index points that belong to the same execution chain formed by $\mathbf{d}_c$, are executed by the same processor. In this example $\mathbf{d}_c = (2, 4)$.*

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, we get pattern vectors by applying $\mathcal{MC}$ operator: $PV = \mathcal{MC}(DV) = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$. 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 $d_{jy} < p_{iy}$, $1 \leq i \leq k$ and $1 \leq j \leq m$.

$PV = \mathcal{MC}(DV) = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k \rangle$.

$DV_2 = \langle \mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m \rangle$.

$DV_{UET} = \langle \mathbf{d}_1, \ldots, \mathbf{d}_{c-1}, 2\mathbf{d}_c, \mathbf{d}_{c+1}, \ldots, \mathbf{d}_m \rangle$.

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

4.2 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):

$\mathbf{p} = i_1 \cdot \mathbf{p}_1 + i_2 \cdot \mathbf{p}_2 + \ldots + i_k \cdot \mathbf{p}_k$, nstances $(2, 4, \ldots)$.

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:

Figure 6. *An algorithm with 5 dependence vectors:* $\mathbf{d}_1 = (1,7)$*,* $\mathbf{d}_2 = (2,4)$*,* $\mathbf{d}_3 = (4,6)$*,* $\mathbf{d}_4 = (5,2)$ *and* $\mathbf{d}_5 = (7,1)$*, while* $\mathbf{d}_c = \mathbf{d}_2 = (2,4)$*. The dashed polygons show the points that are executed at the even time instances (2, 4, . . .). Even time instances are indicated by the straight lines.*

- 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 6.

The following lemmata and theorems prove the analytical expression of the general pattern outline. Their proofs can be found in [10].

Lemma 4.1 $\mathcal{MC}(\langle PV + \mathbf{d}_c\rangle \cup DV_2) = \mathcal{MC}(DV_{UET})$.

Lemma 4.2 $\mathcal{MC}(\langle DV_{UET} + \mathbf{d}_c\rangle \cup \langle PV + DV_2\rangle) = \mathcal{MC}(PV + DV_{UET})$.

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

- $pat_0 = \langle(0,0)\rangle$
- $pat_1 = \langle \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k\rangle = PV$
- $pat_t = \mathcal{MC}(\langle pat_{t-2} + DV_2\rangle \cup \langle pat_{t-1} + \mathbf{d}_c\rangle)$

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

Theorem 4.1 *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.

4.3 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 7. 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 that $\mathbf{j}$ and $\mathbf{j}'$ belong to t region of the equivalent UET problem, that defined above. Using the results of Theorem 4.1, we can express pat_{2k+1} in relation to pat_{2i} and pat_{2i+2}, only. It is:

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

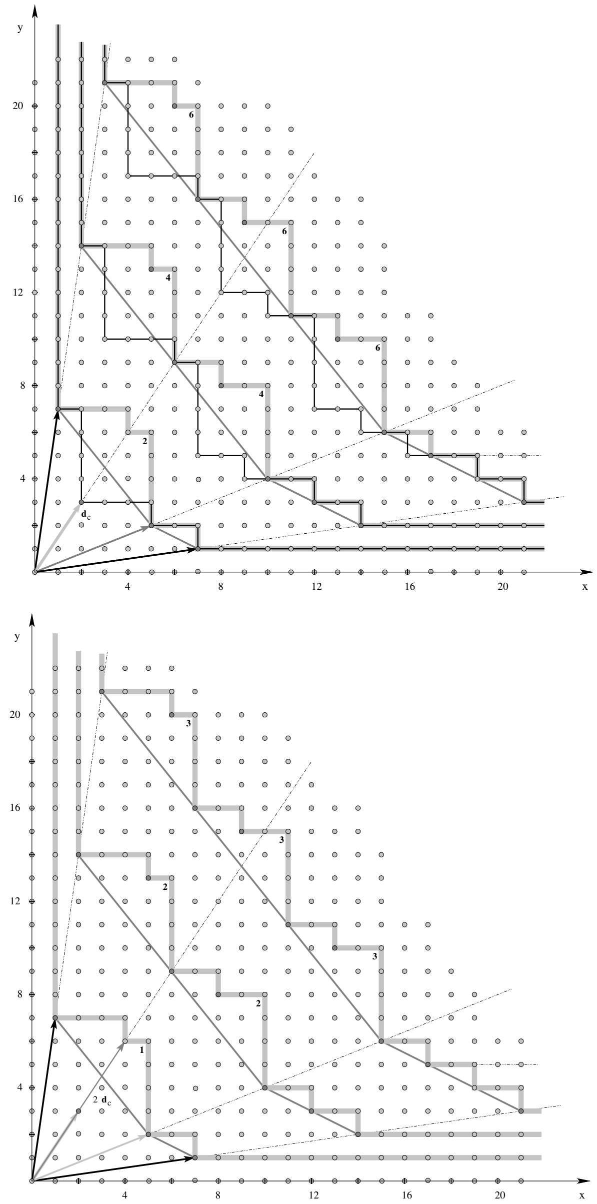

Figure 7. *The execution propagation of a UET-UCT problem with dependence vectors* $\mathbf{d}_1 = (1,7)$*,* $\mathbf{d}_2 = (2,3)$*,* $\mathbf{d}_3 = (5,2)$ *and* $\mathbf{d}_4 = (7,1)$*, with* $\mathbf{d}_c = \mathbf{d}_2$ *and a UET problem with dependence vectors* $\mathbf{d}_1$*,* $2\mathbf{d}_2$*,* $\mathbf{d}_3$ *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,* $i = 1, 2, \ldots$*.*

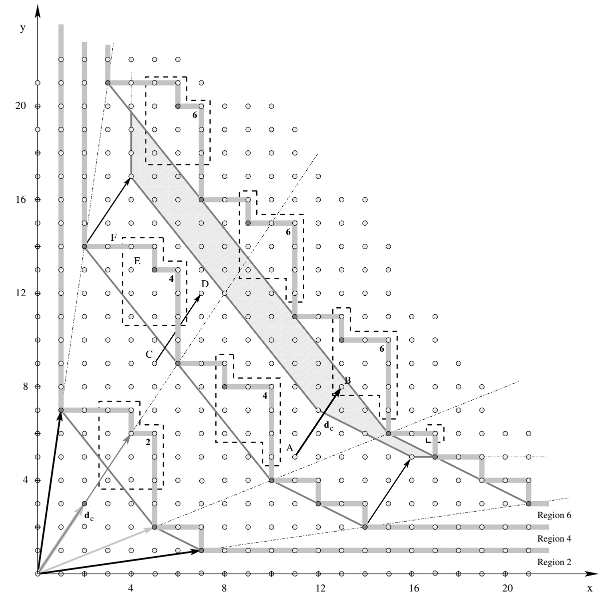

Figure 8. *Here we depict a doubly nested loop with 4 dependence vectors:* $\mathbf{d}_1 = (1,7)$*,* $\mathbf{d}_2 = (2,3)$*,* $\mathbf{d}_3 = (5,2)$ *and* $\mathbf{d}_5 = (7,1)$*. Zero communication cost is achieved along* $\mathbf{d}_2$ *(*$\mathbf{d}_c = \mathbf{d}_2$*).*

This means that if $\mathbf{j}$ belongs to the t region in the equivalent UET problem, then $ECT(\mathbf{j}) = 2t+1$, if and only if $\mathbf{j} - \mathbf{d}_c$ belongs to the t region too. If this is not the case, then $ECT(\mathbf{j}) = 2t$. This fact is illustrated in Figure 8.

The following algorithm is a revision of *Algorithm II* 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.

Algorithm III *(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 II* 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 II* 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 8. Regions are named by even numbers only, as they are the boundary of even time instance pattern outlines. Points $\mathbf{B}$ and $\mathbf{D}$ are located, using *Algorithm II*, 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 $\mathbf{C}$ will be executed at time instance $t_C = 5$ and point $\mathbf{B}$ 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.

4.4 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 $\mathbf{E}$ and $\mathbf{F}$ in Figure 8 correspondingly). Hence, our scheduling algorithm is asymptotically optimal, with a constant delay of at most 2.

4.5 Generalization to Higher Dimensions

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.

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. 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.

5 Conclusion

In this paper we presented a nearly optimal scheduling method for uniform dependence nested loop algorithms, considering the general UET-UCT model. We described an exponential procedure for computing the precise k-step pattern boundary, which is the best we can get so far. A nearly optimal geometric scheduling algorithm for the general UET-UCT model is proposed. Finally, we presented a new method for transforming a UET-UCT problem to a UET equivalent one. This last method is a major improvement, as it is the first method presented do far.

Our method has been implemented and tested in numerous examples and 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] 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.

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

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

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

[9] 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)*, Korea, 2001.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.