Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Automatic parallelization based on multi-dimensional scheduling

Alain Darte
Frédéric Vivien

September 1994

Research Report N^{O} 94-24

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Automatic parallelization based on multi-dimensional scheduling

Alain Darte
Frédéric Vivien

September 1994

Abstract

In the scope of uniform recurrence equations, we study an algorithm first proposed by Karp, Miller and Winograd for detecting cycles of null weight in cyclic graphs. We show how this algorithm can be used for generating multi-dimensional schedules that express all the potential parallelism contained in a computable system of uniform recurrence equations. We then apply this technique to imperative programs based on loop nests and whose dependences are described in the framework $(\mathbb{Z}, +, -, *)$, the most popular way of describing dependences. We are able to show that our technique is an optimal parallelization technique in the sense that no more parallelism can be detected than provided by our new algorithm.

Keywords: automatic parallelization, multi-dimensionnal scheduling, loop nest, systems of recurrence equations, dependence analysis

Résumé

Dans le cadre des systèmes d'équations récurrentes uniformes, nous étudions un algorithme initialement proposé par Karp, Miller et Winograd de détection de cycles de poids nul dans les graphes cycliques. Nous montrons comment un tel algorithme peut être utilisé pour générer des ordonnancements multi-dimensionnels qui expriment tout le parallélisme potentiel d'un système d'équations récurrentes uniformes calculable. Nous appliquons alors cette technique aux programmes impératifs composés de nids de boucles et dont les dépendances sont décrites dans la grammaire $(\mathbb{Z}, +, -, *)$, la description des dépendances la plus répandue. Nous montrons que notre technique de parallélisation est optimale en ce sens qu'il ne peut être détecté plus de parallélisme que n'en trouve notre algorithme.

Mots-clés: parallélisation automatique, ordonnancement multi-dimensionnel, nid de boucles, systèmes d'équations récurrentes, analyse de dependances

Automatic parallelization based on multi-dimensional scheduling

Alain Darte
(on leave from CNRS, LIP, ENS Lyon)
Advanced Computer Research Institute
Lyon, France
e-mail: darte@acri.fr

Frédéric Vivien
LIP, CNRS URA 1398
Ecole Normale Supérieure de Lyon
Lyon, France
e-mail: Frederic.Vivien@lip.ens-lyon.fr

September 1994

Contents

1 Introduction

1.1 Parallelism and dependence analysis

Parallel languages - i.e. languages for parallel machines - can be separated into two general classes: languages with explicit parallelism and languages with implicit parallelism.

In the first case, the user has to propose by himself a parallel version of its application, to specify which portion of code is in fact parallel code, to choose between a macro or micro-tasking approach, to write down the program of each processor, to take care of communications and synchronizations between different processors. This is the case for message-passing languages extensions of sequential languages, such as C+MPI.

In the second case, all the work is supposed to be done by the compiler: parallelism detection, choice of the granularity, data placement, communications messages, and so on. This is the case for purely sequential languages such as Fortran77 or even Fortran90.

Of course, between these two extremities, all intermediate languages exist. For example, a language such as HPF allows explicit parallelism through **FORALL** directives, but also implicit parallelism through data parallelism.

However, whatever the language, the problem of detecting parallel code takes place, either for the user or for the compiler and it is one of the first problem that has to be solved. It may be useful for macro-tasking, micro-tasking, pipelining and vectorizing, among others. It is important before all other optimizations to know where the parallelism is and whether it is important enough to deserve parallel execution.

Parallelism detection directly depends upon the performance of dependence analysis techniques. The more precise the dependence analysis, the more parallelism can be detected. Powerful techniques (Banerjee [Ban88], Pugh [Pug92], Feautrier [Fea91] by precision order), have been developed. However, the more precise the dependence analysis, the more expensive and sophisticated the algorithm, going from the complexity of solving diophantine equations, to the complexity of integer linear programming.

Sometimes, even with the most sophisticated algorithm, it is not always possible to derive at compile-time all dependences with precision. In this case, the alternative approach is to be pessimistic and to consider that a dependence exists when the analyser is not able to decide. A subclass of sequential imperative programs has been proven to be of interest for compilation optimizations, the class of programs constituted by nested loops, possibly parametrized, with regular access to arrays and simple instructions (no function calls for example for which an interprocedural analysis would be needed). For this subclass, quite an efficient dependence analysis is feasible, and thus, one can hope to be able to detect parallel code easily.

In general, parallelism detection is simply a checking: the dependence analyzer is used to check if a loop transformation is correct or not (i.e. if dependence relations are not inverted), if a loop is parallel or not (i.e. if there is no dependence between different iterations of the loop). Some heuristics are used to choose, among all possible transformations, one transformation which could lead to parallel loops. This is the principles of parallelizers based on loop interchange, loop skewing, and so on [LW92, KP93].

Another approach is to view all loops as a whole and to try to apply a global transformation on the nested loops. Kennedy and Allen [KA87] proposed an algorithm based on a study of the strongly connected components of the dependence graph and on the depth of the dependences. However, loop restructurations, such as loop skewing or even loop interchange, are not taken into account in their algorithm.

However, a global approach, that takes also into account loop restructurations, is possible. This approach uses the notion of scheduling functions, introduced long time ago, for systems of uniform recurrence equations and systolic arrays design methodologies [KMW67, Qui87, Rao85]. This is the basis of Feautrier's work [Fea91] and of Darte and Robert's work [DR92]. This approach is more a parallelism detection than a parallelism checking.

This paper is devoted to the construction of such scheduling functions: in section 1.2, we present the material needed for the next sections, definition of system of uniform recurrence equations, nested loops terminology, graph terminology, definitions of multi-dimensional schedules and of separating hyperplanes.

In section 2, we present an algorithm designed by Karp, Miller and Winograd [KMW67] for detecting null weight cycles in cyclic graphs and we study more deeply its properties. We show how to derive nearly optimal multi-dimensional schedules, that means schedules whose latency are of the same order than the length of the longest dependence path.

In section 3, we show how this study can be applied to the automatic parallelization of nested loops, even when the dependence analysis is not exact but given in the framework $(\mathbb{Z}, +, -, *)$ [Ban88, Pug92]. We first consider the case of perfectly nested loops and we extend the method to more general loop nests.

Finally, in section 4, we summarize our results, we show the differences and relationships between our study and previous works, and we give concluding remarks.

1.2 Definitions and notations

1.2.1 Regular nested loops

In this paper, we restrict ourselves to the main program structures for which dependence analysis has been proven to be feasible, programs constituted by nested loops (also called *loop nest*) satisfying:

- all loop counters are integer variables,
- all loop bounds are affine functions of loop counters of surrounding loops, and possibly of program parameters,
- all loop increments are equal to 1.

In general, for the dependence analysis to be precise enough, it is required that these nested loops do not contain other control structures, such that **if** statements or jumps. We call such program structures, **regular nested loops**. Loops can be **perfectly** nested or not.

Each statement S_i is contained in some nested loops, say l_i loops, and will be executed as many times as there are possible combinations of the l_i loop counters of the surrounding loops. Each of these executions can be described by a l_i-dimensional vector I_i, whose components are the values of the different loop counters, starting from the outermost loop. Such a vector I_i is called **iteration vector** of statement S_i.

The execution order of the instances of S_i is the sequential order, i.e. the strict lexicographic order $\gg_l$ on the iteration vectors. $\gg_l$ is defined for two n-dimensional vectors x and y by induction:

- $n = 1$: $(x_1) \gg_l (y_1)$ if and only if $x_1 \geq y_1 + 1$
- $n > 1$: $(x_1, x_2, \ldots, x_n) \gg_l (y_1, y_2, \ldots, y_n)$ if and only if

$$\begin{cases} x_1 \geq y_1 + 1 \text{ or} \\ x_1 = y_1 \text{ and } (x_2, \ldots, x_n) \gg_l (y_2, \ldots, y_n) \end{cases}$$

The execution order between two instances of two distinct statements is a little bit more complicated. Consider two statements S_i and S_j surrounded by $l_{i,j}$ common loops, statement S_i being written before S_j in the program code. Let $\tilde{I}_i$ and $\tilde{I}_j$ be the iteration vectors of S_i and S_j truncated at the $l_{i,j}$ component. Then instance I_i of S_i is executed before instance I_j of statement S_j if and only if $\tilde{I}_i \gg_l \tilde{I}_j$ (different iteration of the $l_{i,j}$ surrounding loops) or $\tilde{I}_i = \tilde{I}_j$ (same iteration but S_i is considered first).

Dependence analysis is based on this concept. Dependences between statements are found by detecting dependences between instances of statements. The difference between two truncated iteration vectors of dependent instances is called the **distance vector**. This difference is carried out so that the distance vector be lexicographically positive. Generating all distance vectors is not reasonable and sometimes not feasible, thus, dependence analysis methods aim to give a condensed representation of the set of distance vectors. When the distance vector does not depend on the iteration vectors, the dependence is said uniform, and the value of the distance vectors is called **dependence vector**. Otherwise, one can represent the set of distance vectors by an approximation in the framework $(\mathbb{Z}, +, -, *)$ [Ban88, Pug92] or, when it is possible, in an exact but condensed representation as the DFG of Feautrier [Fea91]. Whatever the representation of distance vectors, dependence relations between statements are expressed by a graph, one vertex per statement, one edge per dependence relation, each edge being valued by the representation of the corresponding set of distance vectors. Such a graph is called **reduced dependence graph** by opposition to the **expanded dependence graph** which describes dependences between instances of statements.

Graph notations: In the following, G will be the reduced dependence graph, V the set of its vertices, E the set of its edges. Each vertex v corresponds to an iteration domain - the set of values the iteration vector takes - that we will denote by $\mathcal{D}_v$. $w(e)$ will represent the weight of an edge e. $t(e)$ the vertex from which edge e is directed and $h(e)$ the vertex to which edge e is directed.

1.2.2 System of uniform recurrence equations

Definition 1 *A system of* ***uniform recurrence equations*** *is a set of equations of the form:*

$$V_i(z) = f_i(V_{i_1}(z - d_{i_1,i}), \ldots, V_{i_{m_i}}(z - d_{i_{m_i},i}))$$

where V_i is a variable defined for all integer points included in a n-dimensional polyhedron $\mathcal{D}_i$. $d_{i,j}$ are vectors in Z^n and f_i is a strict function with m_i arguments.

A system of uniform recurrence equation may seem identical to perfectly nested loops with uniform dependences. However, in this framework, the implicit execution order **is not** given by the sequential order but by the owner computes rule: the left-hand side of each equation is computed only if all arguments of the right-hand side have been computed. Thus, a system of uniform recurrence equations is not always computable - for example, if an instance of a variable depends on itself - and the dependence vectors $d_{i,j}$ are not necessarily lexicographically positive. However, one can define as for regular nested loops, the notion of iteration vectors and reduced dependence graph.

1.3 Multi-dimensional scheduling

The goal of multi-dimensional schedules is to describe execution orders as loops execution. They are an extension of linear schedules (wave-front or hyperplane method [Lam74, DKR91]), which correspond in term of parallelization to the possibility of rewritting nested loops with a sequential outermost loop and all inner loops parallel. With multi-dimensional schedules, it is possible to express nested loops with more than one sequential loop.

Definition 2 *A mono-dimensional schedule is a function that gives an execution order to each instance of statement or variable as an integer - i.e. a mapping from $V \times \mathbb{Z}^n$ to $\mathbb{Z}$ - and that preserves dependence relations.*

A function T, with values in $\mathbb{Z}$, is a schedule if, and only if, for all iteration vectors p and q, for all statements v and w, $T(v,p) \geq T(w,q) + 1$ if (v,p) depends on (w,q) (we note: $(w,q) \rightarrow (v,p)$).

The notion of schedule is extended to functions with values in $\mathbb{Q}$ that satisfy the constraint:

$$T(v,p) \geq T(w,q) + 1 \qquad (1)$$

each time $(w,q) \rightarrow (w,p)$.

In the following, we will consider only rational functions, easier to build and to use and we will come back to integer schedules by applying the floor function to any rational schedule.

Among all mono-dimensional schedules, some interesting classes of schedules are linear schedules, affine schedules and affine schedules with constant linear part.

Definition 3 *An **affine schedule** is a mapping T of the form:*

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z} \\ T:\ (v,\ p) & \rightarrow & X_v p + \rho_v \end{array}$$

where the X_v's are n-dimensional vectors and the ρ_v's are constants.

If all X_v are the same, T is an **affine schedule with constant linear part**, and if all ρ_v are null, then T is a **linear schedule**. Note that for affine schedules with constant linear part, inequation 1 reduces to:

$$\forall e \in G,\ X w(e) + \rho_{h(e)} - \rho_{t(e)} \geq 1$$

Such a vector X is called a **strictly separating hyperplane**.

When one simply has $X w(e) + \rho_{h(e)} - \rho_{t(e)} \geq 0$, X is said to be a **weakly separating hyperplane** for edge e.

We are now ready to define multi-dimensional schedules introduced by Karp, Miller and Winograd [KMW67], Rao [Rao85] and Feautrier [Fea92a].

Definition 4 *A multi-dimensional schedule (of dimension d) is a function that gives an execution order to each instance of statement or variable as a d-dimensional vector with integer components - i.e. a mapping from $V \times \mathbb{Z}^n$ to $\mathbb{Z}^d$ - and that preserves dependence relations.*

You can imagine the successive components as corresponding to days, hours, minutes, seconds, and so on, ...The sequential execution order is, for example, a multi-dimensional schedule.

Such schedules are extended to rational schedules as previously. The compatibility with dependence relations is not anymore an inequality on values but can be expressed on d-dimensional vectors with the multi-dimensional ordering $\gg_l$.

The latency of a multi-dimensionnal scheduling is the number of elements of the set of the floor value of the multi-dimensionnal time-vectors.

We will be interesting in the following in multi-dimensional schedules whose components are affine functions.

Definition 5 *An **affine multi-dimensional schedule** (of dimension d) is a mapping T of the form:*

$$\begin{array}{rcl} V \times \mathbb{Z}^n & \longrightarrow & \mathbb{Z}^d \\ T:\ (v,\ p) & \rightarrow & (X_v^{(1)} p + \rho_v^{(1)}, \ldots, X_v^{(d)} p + \rho_v^{(d)}) \end{array}$$

where the $X_v^{(i)}$'s are n-dimensional vectors and the $\rho_v^{(i)}$'s are constants.

When at a given level i, vectors $X_v^{(i)}$ are identical, the inequation 1 reduces to:

$$\begin{array}{rl} 1 \leq i \leq k_e - 1 & X^{(i)} w(e) + \rho_{h(e)} - \rho_{t(e)} = 0 \\ \text{for some } k_e \leq d & X^{(k_e)} w(e) + \rho_{h(e)} - \rho_{t(e)} \geq 1 \end{array}$$

Thus, the existence of such multi-dimensional schedules is strongly related to the notion of weakly and strictly separating hyperplanes. This is the object of this paper.

2 Computability and scheduling of uniform recurrence equations

2.1 Linear scheduling is not sufficient

Parallelization techniques based on linear scheduling permit to transform loop nests into a very particular structure of loop nests, where all loops are parallel except the outermost loop. We say in this case that a parallelism of degree $(n-1)$ (where n is the number of nested loops) has been detected. This is always the case for example for perfect uniform loop nests.

However, when parallelizing arbitrary nested loops, techniques based on linear schedules are not sufficient. It may indeed happen that the nested loops to be parallelized contain some degree of parallelism but not enough to be expressed by a linear schedule. This is also the case when scheduling systems of uniform equations.

In the example of figure 1, there exists no linear schedules, but the system of uniform equations can be parallelized, as it will be shown in section 2.6.3. It contains indeed parallelism of degree 1.

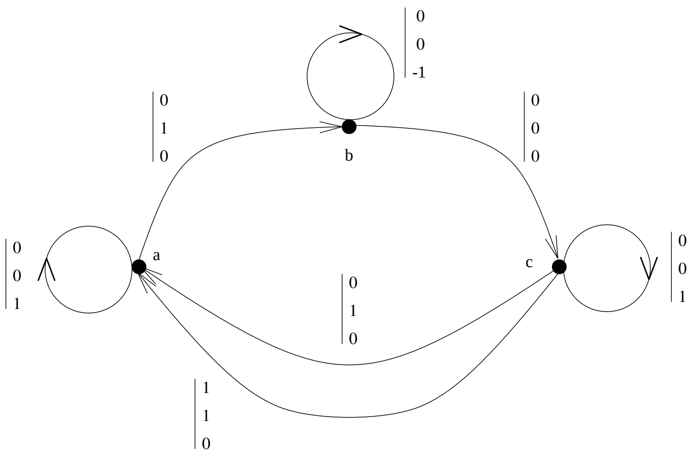

Figure 1: Example with no linear schedule

System of uniform equations:

$$\begin{cases} a(i,\ j,\ k) = c(i-1,\ j-1,\ k) + c(i,\ j-1,\ k) + a(i,\ j,\ k-1) \\ b(i,\ j,\ k) = a(i,\ j-1,\ k) + b(i,\ j,\ k+1) \\ c(i,\ j,\ k) = b(i,\ j,\ k) + c(i,\ j,\ k-1) \end{cases}$$

We will see that the non existence of such a linear schedule is linked to the existence of a multi-cycle of null weight in the dependence graph and to the problem of computability of the system of uniform equations.

The next sections are thus devoted to the problem of determining if a system of uniform equations is computable or not. This problem is also related to the problem of building multi-dimensional schedules and we will show how this can be useful for parallelizing loop nests that are more general than the simplest case of perfect uniform loop nests.

2.2 Detection of null weight cycles in uniform recurrence equations

Consider a system of uniform recurrence equations, defined by its iteration domain $\mathcal{D}$ and its reduced dependence graph G. When $\mathcal{D}$ is bounded, the system is computable if and only if the expanded dependence graph has no cycle. If $\mathcal{D}$ is sufficiently large, G describes exactly, but in a condensed form, all dependence relations: therefore, there exists a cycle in the expanded dependence graph if and only if G has a cycle of null weight.

In this section, we explain how the problem of detecting null weight cycles in a reduced dependence graph can be solved. We present a modified version of the algorithm first proposed by Karp, Miller and Winograd [KMW67], then studied among others by Rao [Rao85] and Kosaraju and Sullivan [KS88]. This algorithm, based on a recursive decomposition of the graph G, is the starting point of all the results presented in this paper.

2.2.1 Definitions and notations

Matrix notations We denote by C the *connection matrix* of the reduced dependence graph G defined as follows: each row of C corresponds to a vertex of G and each column of C to an edge of G. If the j-th edge of G is oriented from vertex i to vertex k, then we let $C_{i,j} = -1$ and $C_{k,j} = +1$, otherwise for all $l \notin \{i, k\}$, $C_{l,j} = 0$. A special case is for self-dependence edges for which we let $C_{l,j} = 0$, for all l.

We denote by D the *dependence matrix* whose columns are the dependence vectors. Of course, edges of G are considered in the same order for D and for C.

Finally, we denote by B, the block matrix whose first rows are the rows of C and whose last rows are the rows of D.

Example: For the example given by figure 1 with 3 vertices and 7 edges, we have:

$$C = \begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & -1 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} C \\ D \end{bmatrix}$$

Subgraph of null weight multi-cycles The previous matrix notations allow us to formulate the notion of multi-cycle (i.e. union of cycles) of null weight. If there exists a vector q (with non negative integer components) such that $Cq = 0$, then G has a multi-cycle that uses q_i times the i-th edge of G. Moreover, if $Dq = 0$ (thus $Bq = 0$), then the multi-cycle is a multi-cycle of null weight. Thus, detecting a null weight multi-cycle can be easily done by checking if the following system has a rational solution:

$$\left\{ \; q \geq 0, \quad q \neq 0, \quad Bq = 0 \; \right\} \tag{2}$$

(If there exists a rational solution, then by scaling its components, there exists an integral solution).

However, detecting a null weight cycle is much more difficult. A null weight cycle is a null weight multi-cycle but the contrary is false. However, when all edges of a null weight multi-cycle form a strongly connected graph, a null weight cycle can be found. This is the underlying idea of the decomposition algorithm of Karp, Miller and Winograd.

Let G' be the subgraph of G generated by the null weight multi-cycles of G. G' can be obtained in two steps:

- Delete all edges that do not belong to a multi-cycle of null weight, i.e. edges e_i for which there is no solution q of problem 2 with $q_i > 0$.
- Keep only vertices that belong to at least one non deleted edge.

We will show in section 2.4 that the subgraph G' can be built more efficiently by solving only one linear program.

2.2.2 Decomposition algorithm

Determining if a system of uniform recurrence equations is computable can be done by applying the following algorithm to its dependence graph G.

Algorithm

i. Decompose G into strongly connected components G_1, G_2, $\ldots$, G_s, and call step (ii) on each G_i.

ii. Build G' the subgraph of G generated by all edges that belong to a null weight multi-cycle of G.

- If G' is strongly connected then G is not computable.
- If G' is an empty graph, G is computable.
- Otherwise call step (i) on G'.

We need some lemmas before being able to prove the correctness of the algorithm.

Lemma 1 (Karp, Miller and Winograd) *A graph G contains a cycle of null weight if and only if its subgraph G' does.*

Proof: G' is the subgraph of G generated by the set of edges that belong to a multi-cycle of null weight. A cycle is a multi-cycle, thus G' contains all null-weight cycles of G. If G contains a null weight cycle, so does G'. Reciprocally, any cycle of G' is a cycle of G. $\square$

Corollary 1 *If the subgraph G' of a graph G is empty, G has no null weight cycle.*

Corollary 2 *G has a null weight cycle if and only if one of the connected components G'_i of G' does.*

Lemma 2 (Karp, Miller and Winograd) *If the subgraph G' of a graph G is strongly connected, G has a null weight cycle.*

Proof: If G' is strongly connected, there exists in G' a cycle $\mathcal{C}$ passing through all its vertices. Let the successive edges of $\mathcal{C}$ be e_1, e_2, $\ldots$, e_m. Each edge e_i of $\mathcal{C}$ is contained by definition of G' in a null weight multi-cycle $\mathcal{C}_i$ of G, hence of G'. This multi-cycle can be decomposed into three parts $\mathcal{C}_i = e_i + \pi_i + \mathcal{C}'_i$, where $\mathcal{C}'_i$ is a multi-cycle, and $e_i + \pi_i$ is a cycle of G'. π_i is a path from the vertex to which e_i is directed to the vertex from which e_i is directed. We can now construct a cycle in G' has follows: the cycle traverses the edges e_1, e_2,$\ldots$, e_m, then returns along the paths π_m, $\ldots$, π_2, π_1. Since each vertex of G' is visited in this process, it is possible to insert the remaining multi-cycles $\mathcal{C}'_i$ at some convenient points. The weight of the resulting cycle is null because it is the sum of the weights of the $\mathcal{C}_i$. The construction of such a cycle is illustrated by figure 2. $\square$

Corollary 3 *The decomposition algorithm is correct.*

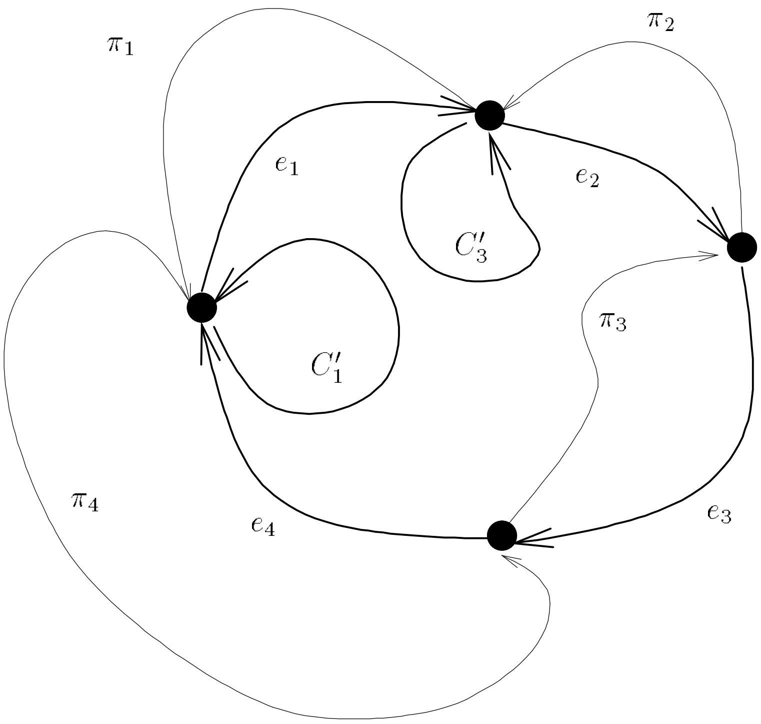

Figure 2: Construction of a null weight cycle in a strongly connected G'

Proof: Note first that the decomposition algorithm always ends, when applied to a finite graph. When step (ii) is applied to a graph F, either the algorithm ends because F' is strongly connected or null, or the step (ii) is called on strictly **smaller** subgraphs of F (the strongly connected components).

If the decomposition ends because one of the G'_i is strongly connected, then this subgraph G'_i has a null weight cycle as shown by lemma 2. This cycle is also a cycle of G, thus G has a null weight cycle and the system of uniform recurrence equations is not computable. On the other hand, if all recursive calls end because the corresponding subgraphs G'_i are empty, G has no null weight cycle (corollaries 1 and 2) and the corresponding system of uniform recurrence equations is computable. □

Definition 6 *We denote by d the depth of the decomposition algorithm, i.e. the longest sequence of recursive calls.*

If the first call ends directly (that means G' is empty or strongly connected), no further recursive calls are made and $d = 1$.

At depth i of the algorithm, strongly connected components are built. For each of them, the subgraph G' is built at step (ii) and it is decided whether a recursive call is necessary. In the following, we denote by $H_{i,j}$, the j-th subgraph of G that has been generated at depth i. For example, $H_{1,j}$ is the j-th strongly connected component of G. Each $H_{1,j}$ has a G'. If it is neither empty nor strongly connected, it generates one of the $H_{2,j}$'s.

At depth $i > 1$, each G' satisfies by construction that no vertex is "lonely" in the sense that no edge passes through it. So as to keep this property even for depth 1, we will first delete from G all "lonely" vertices (they are indeed unuseful as far as cycles and scheduling are concerned). If all vertices of G are "lonely", we will say to be coherent that the depth of the algorithm is $d = 0$.

2.3 Depth of the decomposition algorithm and longest dependence path

Once we know that a system of uniform recurrence equations is computable or not, it is interesting to give an idea of the length of the longest path in the expanded dependence graph. This length gives an upper bound on the sequentiality of the system of recurrence equations and thus gives also a lower bound on the parallelism it contains.

In [DKR91], it is shown that for a single uniform recurrence equation, this length is equivalent to the latency of the optimal linear schedule, on full dimensional domains whose size tends to infinity. Remember that the latency of a linear schedule is nothing but the number of iterations of the sequential outermost loop. To say it briefly, on domains of size parametrized by N, the latency of the optimal linear schedule is in λN for some constant λ and so does the length of the longest dependence path. Both are in N and the multiplicative constants are the same.

Here, when linear schedules do not always exist, the length of the longest path is not necessarily linear in N anymore, it can be in kN^p for some constants k and p. We will not try to be precise on the multiplicative constant k, we will just focus on p the power of N. Theorem 1 shows that this power is actually equal to the depth of the decomposition algorithm 2.2.2. This has been shown the first time by Sailesh Rao [Rao85], but we consider that his proof is not precised enough to be convincing. We present here a detailed proof of this result.

Theorem 1 (Rao, Vivien) *If d is the depth of the decomposition algorithm 2.2.2 applied to a dependence graph G, there exists a path in the expanded dependence graph of length at least kN^d for some positive constant k.*

In the following proof, we consider that the iteration domain is a n-dimensional cube defined by $\{p \mid 0 \leq p_i \leq N,\ 1 \leq i \leq n\}$. For more general iteration domains, the proof can be immediately extended to domains that contain a n-dimensional cube of size N and that are contained in a n-dimensional cube of size λN for some constant λ.

Proof: The proof uses paths in the reduced dependence graph G and paths in the expanded dependence graph included in the iteration domain $\mathcal{D}$. To avoid making a confusion between both, vertices in G will be denoted by letter v and vertices in $\mathcal{D}$ by letter p.

For a path l of G, $l = \sum_{i=1}^{r} e_i$ where $h(e_i) = t(e_{i+1})$, we define two quantities $m(l)$ and $M(l)$ as follows:

$$m(l) = \max\{t \in \mathbb{Z}^- \mid t.1_n \leq \sum_{i=1}^{s} w(e_i), 1 \leq s \leq r\}$$

$$M(l) = \min\{t \in \mathbb{Z}^+ \mid \sum_{i=1}^{s} w(e_i) \leq t.1_n, 1 \leq s \leq r\}$$

that represent the lower and upper bounds of a n-dimensional cube in which a path in $\mathcal{D}$ corresponding to l in G can be easily drawn.

For all $i < d$, we associate to the graph $H_{i,j}$ a multi-cycle $q_{i,j}$ that defines its corresponding $H'_{i,j}$, subgraph of null weight multi-cycles. $q_{i,j}$ defines for each strongly connected component $H_{i+1,j'}$ of $H'_{i,j}$ a cycle $C_{i+1,j'}$. Furthermore, $\sum_{j'} w(C_{i+1,j'}) = Dq_{i,j} = 0$ by construction.

For each strongly connected component $H_{1,j}$ of G, we define $C_{1,j}$ a cycle passing through all its vertices.

Now, we define the starting points $v_{i,j}$ of the cycles $C_{i,j}$ as follows:

- start from an arbitrary vertex $v_{1,j}$ in the j-th strongly connected component of G.
- follow the edges of $C_{1,j}$ and let the first vertex you reach that belongs to $C_{2,k}$ be $v_{2,k}$.
- define recursively all $v_{i+1,l}$ by traversing cycle $C_{i,j}$ starting from $v_{i,j}$.

Finally, let $m_{i,j} = m(C_{i,j})$ and $M_{i,j} = M(C_{i,j})$, where edges in $C_{i,j}$ are considered in the order defined from $v_{i,j}$.

We are now ready to build a path in G and its corresponding dependence path in $\mathcal{D}$ of length $\Omega(N^d)$. Let k be a positive integer. We will fix its value later on.

Path $C_j(k)$ in G

Define in each strongly connected component of G a cycle $C_j(k)$ as follows: traverse k times the cycle $C_{1,j}$ starting from $v_{1,1}$ and each time the vertex $v_{2,j'}$ is reached in this traversal for the first time, traverse, before going further on $C_{1,j}$, k times the cycle $C_{2,j'}$. Recursively, traverse in the same way, all cycles $C_{i,j}$. In such a path, $C_{i,j}$ will be traversed k^i times.

Let $L(P)$ be the length of a path P and let $\mathcal{L} = \max_j L(C_{d,j})$, then at least one of the $C_j(k)$'s satisfies $L(C_j(k)) \geq k^d \times \mathcal{L}$ (note that we could be a lot more precise but this is not needed for what we want to show).

We consider such a cycle $C_{j_0}(k)$ and we build an actual dependence path, that is to say, a chain of points p, **all in** $\mathcal{D}$, that corresponds to the cycle $C_{j_0}(k)$. This point is the weakness of Rao's proof: it is not explained precisely how to control the path and avoid it to go out of the domain boundaries.

Path $P(k)$ in $\mathcal{D}$

We start from a point $p(k)$ in $\mathcal{D}$ and we convert the path in G into a path in $\mathcal{D}$ by adding successively the corresponding dependence vectors.

Consider a point p of path $P(k)$ and let $\theta(i,j)$ be the number of times $C_{i,j}$ has been entirely traversed before reaching p and $l_{i,j}$ be the remaining part, a sub-path of $C_{i,j}$.

$$p = p(k) + w\left(\sum_{i,j} \theta(i,j) C_{i,j} + \sum_{i,j} l_{i,j}\right)$$

We have to show that such a point p belongs to $\mathcal{D}$ if k and $p(k)$ are well chosen.

For $i > 1$, consider $H_{i-1,j}$ and the strongly connected components $H_{i,j'}$ of $H'_{i-1,j}$: cycle $C_{i-1,j}$ has been entirely traversed $\theta(i-1,j)$ times. Thus, each cycle $C_{i,j'}$ has been traversed at least $k \times \theta(i-1,j)$ times: $\theta(i,j') \geq k \times \theta(i-1,j)$. It may have been traversed more than this, if cycle $C_{i-1,j}$ is currently being traversed (i.e. if $l_{i-1,j}$ is not empty). In this case, during the last traversal of $C_{i-1,j}$, $C_{i,j'}$ has been traversed again a certain number of times, between 0 and k, depending if p has been reached before (case 3), during (case 4) or after (case 2) such a traversal. Thus, $\theta(i,j') \leq k \times \theta(i-1,j) + k$. More precisely, we have equality in this last inequation only if $C_{i,j'}$ has been entirely traversed, i.e. when $l_{i,j'}$ is empty. See figure 3, for a drawing of all possible cases.

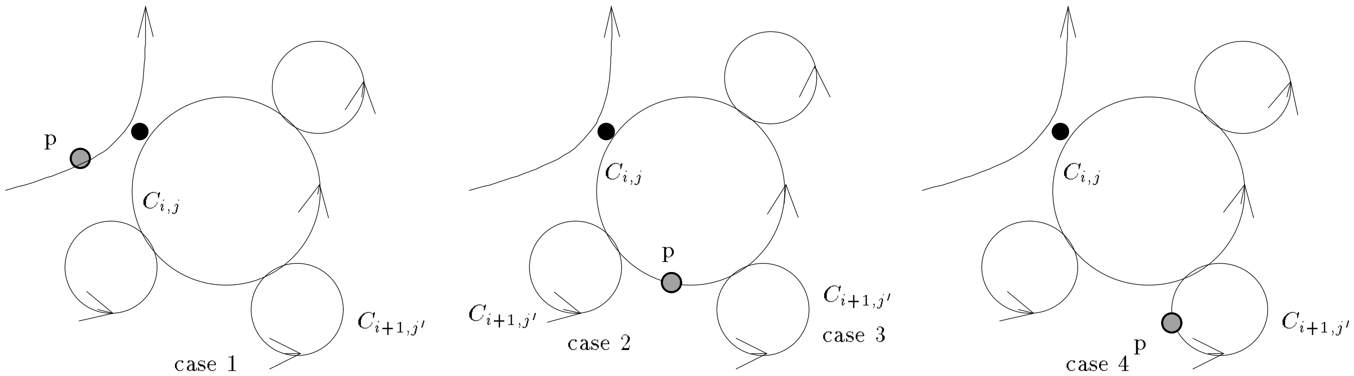

Figure 3: Possible positions for point p

We let $\theta'(i,j') = \theta(i,j') - k \times \theta(i-1,j)$ for $i > 1$ and $\theta'(1,j) = \theta(1,j)$. From the previous study, we get:

$$\theta'(i,j') \leq k \text{ if } l_{i,j'} = \emptyset \text{ and } \theta'(i,j') \leq k-1 \text{ otherwise.}$$

Recall that for a given $H_{i-1,j}$, we have by construction $\sum_{j'} w(C_{i,j'}) = 0$, thus:

$$\begin{aligned} w\left(\textstyle\sum_{j'} \theta'(i,j') C_{i,j'}\right) &= w\left(\textstyle\sum_{j'} (\theta(i,j') - k \times \theta(i-1,j)) C_{i,j'}\right) \\ &= w\left(\textstyle\sum_{j'} \theta(i,j') C_{i,j'}\right) - k \times \theta(i-1,j) \times 0 = w\left(\textstyle\sum_{j'} \theta(i,j') C_{i,j'}\right) \end{aligned}$$

Now, when $(i-1)$ varies from 1 to $d-1$, we add these equations, depth after depth, and we get:

$$w\left(\sum_{i \geq 2,j} \theta(i,j) C_{i,j} + \sum_{i \geq 2,j} l_{i,j}\right) = w\left(\sum_{i \geq 2,j} \theta'(i,j) C_{i,j} + \sum_{i \geq 2,j} l_{i,j}\right)$$

and this can be extended to $i = 1$ because of the choice of $\theta'(1,j)$.

Finally, because:

- $m_{i,j}.1_n \leq w(C_{i,j}) \leq M_{i,j}.1_n$.
- $m_{i,j}.1_n \leq w(l_{i,j}) \leq M_{i,j}.1_n$.
- $\theta'(i,j') \leq k$ if $l_{i,j'} = \emptyset$ and $\theta'(i,j') \leq k-1$ otherwise.
- $w\left(\sum_{i,j}\theta(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right) = w\left(\sum_{i,j}\theta'(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right)$

we obtain:

$$(k \times \sum_{i,j} m_{i,j})1_n \ \leq w\left(\sum_{i,j}\theta(i,j)C_{i,j} + \sum_{i,j} l_{i,j}\right) \ \leq (k \times \sum_{i,j} M_{i,j})1_n$$

which implies:

$$p(k) + k(\sum_{i,j} m_{i,j})1_n \leq p \leq p(k) + k(\sum_{i,j} M_{i,j})1_n$$

To conclude, we take $K = \sum_{i,j}(M_{i,j} - m_{i,j})$, $k = \lfloor \frac{N}{K} \rfloor$ and $p(k) = -k \times \sum_{i,j} m_{i,j}.1_n$. We have produced a dependence path in the n-dimensional cube of size N whose length is at least $k^d \times \mathcal{L}$, i.e. of length $\Omega(N^d)$. □

2.4 Construction and properties of G'

We come back in this section to the construction of the subgraph G' and we show that it is closely related to the construction of separating hyperplanes, thus of multi-dimensional schedules.

Step (ii) of the decomposition algorithm, which consists in the construction of G', can be done in solving only one linear program. We will show indeed that the edges of G' are exactly the edges e_i for which $v_i = 0$ in any optimal solution of linear program 3.

$$\min\left\{ \ \sum_i v_i \quad | \quad q \geq 0, \quad v \geq 0, \quad q+v \geq 1, \quad Bq = 0 \ \right\} \tag{3}$$

Note first that linear program 3 has a finite solution. $q = 0$ with $v = 1$ for all i is indeed a solution.

Lemma 3 *For any optimal solution (q,v) of program 3:*

- $q_i \neq 0 \Leftrightarrow v_i = 0$.
- $q_i = 0 \Leftrightarrow v_i = 1$.

Proof: Consider an optimal solution (q,v) of program 3. Note that the only constraint on v_i is $q_i + v_i \geq 1$. Thus $q_i = 0 \Rightarrow v_i \geq 1 \Rightarrow v_i = 1$ because $\sum_i v_i$ is minimal. Furthermore, $v_i = 0 \Rightarrow q_i \neq 0$. It remains to show that $q_i \neq 0 \Rightarrow v_i = 0$.

Let $q_0 = \min\{q_i \mid q_i \neq 0\}$. We can build another solution (q',v') with $q' = \frac{1}{q_0}q$, $v'_i = 0$ if $q_i \neq 0$ and $v'_i = v_i$ otherwise. $\sum_i v'_i \leq \sum_i v_i$ and we have thus found a better solution, except if v has not changed, i.e. if $v_i = 0$ each time $q_i \neq 0$. □

Lemma 4 *For any optimal solution (q,v) of program 3:*

- $v_i = 0 \Leftrightarrow e_i \in G'$

Proof: Any solution (q, v) of program 3 corresponds to a null weight multi-cycle ($Bq = 0$) whose participating edges are edges e_i for which $q_i \neq 0$. Thus, $v_i = 0 \Rightarrow q_i \neq 0 \Rightarrow e_i \in G'$.

Reciprocally, let e_i be an edge of G'. By definition of G', there exists a null weight multi-cycle (i.e. a vector $\tilde{q}$ with $B\tilde{q} = 0$) such that $\tilde{q}_i \geq 1$. Now, consider an optimal solution (q, v) of program 3. We can form another solution (q', v') with $q' = q + \tilde{q}$, $v'_j = v_j$ if $j \neq i$ and $v'_i = 0$. This solution is a better solution except if we already have $v_i = 0$. □

Therefore, solving program 3 gives directly G': the edges of G' are the edges e_i such that $v_i = 0$.

Now, to better understand what is behind this linear program, let us consider its dual. Program 3 can be written in a canonical form as:

$$\min \left\{ \textstyle\sum_i v_i \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = 1 + w, \quad Bq = 0 \right\} \tag{4}$$

Its dual can be written as:

$$\max \left\{ \textstyle\sum_i z_i \quad | \quad z \geq 0, \quad 0 \leq z_i \leq 1, \quad X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq z_i \right\} \tag{5}$$

Inequality $z_i \geq 0$ corresponds to variable w_i, inequality $z_i \leq 1$ to variable v_i while inequality $X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq z_i$ corresponds to variable q_i. The dual solution has an interesting property as shown by the following lemma.

Lemma 5 *For any optimal solution (z, X, ρ) of the dual program 5:*

- $e_i \in G' \Leftrightarrow X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} = 0$
- $e_i \notin G' \Leftrightarrow X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq 1$

Proof: We use the complementary slackness theorem [Sch86], which shows the link between strict inequalities in the dual and null variable in the primal.

Variable v**:** there is an optimal solution of 5 with $z_i < 1$ if and only if there is no optimal solution of 3 with $v_i > 0$, i.e. any optimal solution of 3 satisfies $v_i = 0$, i.e. $e_i \in G'$.

Variable w**:** $q_i + v_i >= 1$ is always an equality (i.e. $w_i = 0$) when $q_i = 0$. When $q_i \neq 0$, there exists an optimal solution with $q_i + v_i > 1$ (simply multiply q by some sufficiently large positive number). Now, there is an optimal solution of 5 with $z_i > 0$ if and only if there is no optimal solution of 3 with $w_i > 0$, i.e. any optimal solution of 3 satisfies $q_i = 0$, i.e. $e_i \notin G'$.

Variable q**:** there is an optimal solution of 5 with strict inequality in:

$$X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq z_i$$

if and only if there is no optimal solution of 3 with $q_i > 0$, i.e. $e_i \notin G'$.

This proves the result. □

Lemma 5 shows that considering the dual provides separating hyperplanes, which are strictly separating hyperplanes for the edges not in G' and weakly separating hyperplanes for edges in G'. Furthermore, for each subgraph $H_{i.j}$ in the decomposition, these hyperplanes are those that are the "most often strict": the number of edges, for which such an hyperplane is strict, is maximal.

Lemma 5 can be generalized (this generalization will be used in section 3) to the case where a cost $\lambda_i > 0$ (a "delay" along edge e_i) is attributed to each edge e_i. This does not not change the decomposition algorithm:

Lemma 6 *Let $G = (V, E)$ be a dependence graph. For any positive values $\lambda_1, \ldots, \lambda_{|E|}$, there exist constants $\rho_1, \rho_2, \ldots, \rho_{|V|}$ and a vector X of dimension n such that:*

- $e_i \in G' \Leftrightarrow X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} = 0$
- $e_i \notin G' \Leftrightarrow X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq \lambda_i$

Proof: The linear program:

$$\min \left\{ \textstyle\sum_i \lambda_i v_i \quad | \quad q \geq 0, \quad v \geq 0, \quad w \geq 0, \quad q + v = 1 + w, \quad Bq = 0 \right\} \tag{6}$$

has the same optimal solutions than the linear program 4, if all λ are positive. Its dual program is:

$$\max \left\{ \textstyle\sum_i z_i \quad | \quad z \geq 0, \quad 0 \leq z_i \leq \lambda_i, \quad X w(e_i) + \rho_{h(e_i)} - \rho_{t(e_i)} \geq z_i \right\} \tag{7}$$

The proof is then similar to lemma 5. □

During the decomposition algorithm, one can associate to each vertex v of G, a sequence of vectors $X_v^{(1)}, \ldots, X_v^{(d_v)}$, obtained by considering the dual problem 5. d_v is the depth of the decomposition algorithm where vertex v is removed: $v \in H_{d_v,j}$ but $\notin H'_{d_v,j}$. This sequence of vectors has an important property.

Theorem 2 *The $d_v - 1$ first separating hyperplanes $X_v^{(1)}, \ldots, X_v^{(d_v-1)}$ associated to a vertex v of G are linearly independent. Furthermore, when G is computable, all separating hyperplanes $X_v^{(1)}, \ldots, X_v^{(d_v)}$ associated to v are linearly independent.*

Proof: To simplify the notations, let G_1 be the strongly connected component of G that contains v and let G_i be the subgraph of G generated at depth i of the algorithm, that contains v. G_i is a strongly connected component of G'_{i-1}. Let the constants associated to $X_v^{(i)}$ in the dual program be $\rho_v^{(i)}$.

Suppose that for some index $k \leq d_v$, $X_v^{(1)}, \ldots, X_v^{(k-1)}$ are linearly independent and $X_v^{(1)}, \ldots, X_v^{(k-1)}, X_v^{(k)}$ are linearly dependent. That means $X_v^{(k)} = 0$ if $k = 1$ or that there exist rational numbers $\{\lambda_i\}_{1 \leq i \leq k-1}$ such that $X_v^{(k)} = \sum_{i=1}^{k-1} \lambda_i . X_v^{(i)}$ if $k > 1$.

Let $\mathcal{C}$ be a cycle of G_k. Let us show that $X_v^{(k)} w(\mathcal{C}) = 0$. If $k = 1$, this is obvious because $X_v^{(k)} = 0$. If $k > 1$, we have by construction of the vectors $X_v^{(i)}$, and because of lemma 5:

$$\forall e \in G_k : \; X_v^{(i)} w(e) - \rho_{h(e)}^{(i)} + \rho_{t(e)}^{(i)} = 0 \tag{8}$$

Thus, summing equations 8 for a given i along the edges of $\mathcal{C}$, shows that for all i, $1 \leq i \leq k-1$, $X_v^{(i)} w(\mathcal{C}) = 0$. Since $X_v^{(k)}$ is a linear combination of the vectors $X_v^{(i)}$, $X_v^{(k)} w(\mathcal{C}) = 0$.

However, by construction of $X_v^{(k)}$,

$$\forall e \in G_k : \; X_v^{(k)} w(e) - \rho_{h(e)}^{(i)} + \rho_{t(e)}^{(i)} \geq z_e \tag{9}$$

where $z_e = 1$ if $e \notin G'_k$ and $z_e = 0$ if $e \in G'_k$. Inequation 9 shows that $X_v^{(k)} w(\mathcal{C})$ is actually greater than the number of edges in $\mathcal{C}$ that do not belong to G'_k. Thus, every edge of $\mathcal{C}$ belongs to G'_k.

This is true for every cycle of G_k, G_k is strongly connected by construction, thus $G_k = G'_k$ and G'_k is also strongly connected. The decomposition algorithm has thus stopped here at level k, with the conclusion that G is not computable.

This proves that $X_v^{(1)}, \ldots, X_v^{(d_v)}$ are linearly independent when G is computable and that $X_v^{(1)}, \ldots, X_v^{(d_v-1)}$ are linearly independent in general. □

2.5 Time complexity of the algorithm

We are now ready to give an upper bound on the time complexity of the decomposition algorithm.

Theorem 3 *On a dependence graph G, with $|E|$ edges, $|V|$ vertices and with n-dimensional weights, the decomposition algorithm has a time complexity of $O((n+1)Z)$ where Z is the time complexity of the resolution of a linear program with $(2 \times |E|)$ variables and $(3 \times |E| + n + |V|)$ equations or inequations.*

Proof: Let Z be the time complexity of the resolution of program 3 for the construction of G'. Variables are q_e and v_e, one for for each edge, constraints are $q_e \geq 0$, $v_e \geq 0$ and $q_e + v_e \geq 1$, and $Bq = 0$ which is in fact a system of $n + |V|$ equations.

For all v, the vectors $X_v^{(1)}, \ldots, X_v^{(d_v-1)}$ are linearly independent has shown by lemma 2. Thus, $d_v - 1 \leq n$ (and even $d_v \leq n$ when G is computable). Thus, $d \leq n+1$: the depth of the calling tree is at most $n+1$.

Furthermore, the arguments of the different linear programming resolutions of programs 3 at depth i are disjoint subgraphs of G: the total complexity of all these resolutions at depth i is thus of the same order Z than a similar resolution on the whole graph G (for more details see [KS88]). Finally, once the current G' is computed, it remains to build its strongly connected components. This time complexity $(O(|E| + |V|))$ is negligible compared to Z. $\square$

Remarks:

- The time complexity proposed here is better than the one proposed by Kosaraju and Sullivan [KS88]. This difference comes from the fact that lemma 2 gives a better upper bound on the depth of the decomposition and from the fact that program 3 permits to build a subgraph G' in only one linear program and not one per edge of G.
- In practice, linear programs 3 and 5 can be simplified by replacing $Cq = 0$ by $q = \mu_1 q_1 + \ldots + \mu_m q_m$ where $q_1, \ldots, q_m$ form a basis of cycles. This reduces the number of inequalities in program 3 and the number of variables in program 5 (constants ρ disappear in this new formulation). The constants ρ can then be computed by an algorithmic approach, less expensive than a linear programming resolution, simply by computing the longest paths in a graph similar to G but where edge e_i has a weight equal to $z_i - Xw(e_i)$.

2.6 Multi-dimensional schedules for computable uniform dependence graphs

We are now ready to merge all previous results and propose nearly optimal multi-dimensional schedules for computable system of uniform recurrence equations.

2.6.1 Construction of multi-dimensional schedules

Consider a computable dependence graph G and assume first that G is strongly connected. We apply to G the decomposition algorithm, focusing this time on the dual program 5.

We build a multi-dimensional schedule for G, by building for each vertex v of G, the sequence of vectors $X_v^{(1)}, \ldots, X_v^{(d_v)}$ and the sequence of constants $\rho_v^1, \ldots, \rho_v^{d_v}$, obtained by considering the dual program 5 during the decomposition algorithm. Note that such separating hyperplanes X are not unique: once we know which edge corresponds to an equality like 8 and which one corresponds to an inequality like 9 with $z_e = 1$, one can choose another objective functions than $\sum_i z_i$. One can for example try to minimize the latency corresponding to the vector X, i.e. $\max_{p \in \mathcal{D},\ q \in \mathcal{D}} X(p-q)$. See [DKR91] for this kind of optimization.

For each vertex v, we complete the sequences of vectors $X_v^{(i)}$ and constants $\rho_v^{(i)}$ with zeros so as to obtain sequences of length d.

Lemma 7 *The multi-dimensional function T:*

$$\begin{array}{ccc} V \times \mathcal{D} & \longrightarrow & \mathbb{Z}^d \\ (v,\ p) & \rightarrow & (X_v^{(1)}p + \rho_v^{(1)},\ \ldots,\ X_v^{(d_v)}p + \rho_v^{(d_v)},\ 0,\ \ldots,\ 0) \end{array}$$

defines a multi-dimensional schedule.

Proof: We just have to show that for all edges e, for all $p \in \mathcal{D}$:

$$T(h(e), p) \gg_l T(t(e), p - w(e))$$

G is computable, thus the decomposition algorithm ended because all leaves of the calling trees ended with an empty G'. Thus, at some level of the decomposition, edge e has been removed. Let k be the level where edge e has been removed, i.e. $e \in G_k$ but $e \notin G'_k$. By construction, until level k, $h(e)$ and $t(e)$ belong to the same subgraph of G, thus their sequences of vectors $X^{(i)}$'s are the same until level k:

$$X_{h(e)}^{(1)} = X_{t(e)}^{(1)} = X^{(1)},\ \ldots,\ X_{h(e)}^{(k)} = X_{t(e)}^{(k)} = X^{(k)}.$$

Furthermore, until level $k-1$, vectors $X^{(i)}$'s are weakly separating hyperplanes for edge e and at level k, since e has been removed, $X^{(k)}$ is a strictly separating hyperplane for edge e.

Thus, the i-component $(i < k)$ of $T(h(e), p) - T(t(e), p)$ is equal to:

$$X^{(i)}p + \rho_{h(e)}^{(i)} - X^{(i)}p + X^{(i)}w(e) - \rho_{t(e)}^{(i)} = X^{(i)}w(e) + \rho_{h(e)}^{(i)} - \rho_{t(e)}^{(i)} = 0$$

and the k-th component satisfies:

$$X^{(k)}p + \rho_{h(e)}^{(k)} - X^{(k)}p + X^{(k)}w(e) - \rho_{t(e)}^{(k)} = X^{(k)}w(e) + \rho_{h(e)}^{(k)} - \rho_{t(e)}^{(k)} \geq 1$$

Finally, whatever are the rests of the sequences $X_{h(e)}^{(i)}$, $X_{t(e)}^{(i)}$, $\rho_{h(e)}^{(i)}$ and $\rho_{t(e)}^{(i)}$ after level k, one has $T(h(e), p) \gg_l T(t(e), p)$ for all $p \in \mathcal{D}$. □

Remark: the multi-dimensional schedules so obtained are not arbitrary schedules: they are indeed at each level affine schedules whose linear part is the same on the current graph G_k, we could call them, **locally** affine with constant linear part. This is an important property in practice, because it does not generate too complicated results, that would be difficult to use.

It remains to consider now arbitrary dependence graphs, i.e. not necessarily strongly connected. Let G_i be the i-th strongly connected component of G and let H be the graph obtained by merging all nodes of a strongly connected components into a single node. H is an acyclic graph, we can compute for each of its vertices v_i (i.e. for each $H_{1,i}$) the length of the longest path in H ending at v_i. Now adding at the beginning of each sequence of vectors X the null vector 0 and at the beginning of each sequence of constants ρ_v^i, the constant α_k if $v \in G_k$ gives a multi-dimensional schedule for G. Remark that the first dimension of this schedule will not correspond to a real loop, it is just a way of sequentializing the different components $H_{1,i}$'s when needed.

2.6.2 Latency of the multi-dimensional schedules

Lemma 8 *If d is the depth of the decomposition algorithm, then the multi-dimensional schedule associated with the decomposition has a latency in $O(N^d)$*

Proof: We assume the same properties on $\mathcal{D}$ than in theorem 1. $\mathcal{D}$ is contained in a n-dimensional cube of size λN for some positive constant λ. The latency corresponding to one non null vector X is thus linear in N and the latency corresponding to a family of k **linearly independent** vectors X is equivalent to μN^k, for some positive constant μ.

Let v be a vertex of G and let $X_v^{(1)}, \ldots, X_v^{(d_v)}$ be the sequence of vectors associated to it. Since G is computable, they are linearly independent(lemma 2), thus the latency corresponding to vertex v is in N^{d_v}. Finally, the total latency is equivalent to the maximum of the latency over all vertices, i.e. N^d. $\square$

Theorem 1 and lemma 8 shows that the multi-dimensional schedules obtained by this method are nearly optimal in the following sense:

Corollary 4 *The multi-dimensional schedule built in section 2.6.1 is nearly optimal: if d is the depth of the decomposition algorithm, the latency of the schedule is $O(N^d)$ and the length of the longest dependence path is $\Omega(N^d)$.*

This means that the system of uniform recurrence equations corresponding to G contains a parallelism of degree $(n-d)$ and that we are able to find it.

Question: is it possible to be more precise on the multiplicative factor of N^d, i.e. is it possible to build a schedule with latency equivalent to kN^d each time the length of the longest path is equivalent to kN^d?

2.6.3 Example

We come back to example 1. There is a multi-cycle of null weight, involving the three self-dependences. G' has three strongly connected components, but none of them contains a null weight multi-cycle and the algorithm stops at depth 2. Thus, there exists a multi-dimensional schedule whose latency is $O(N^2)$ and a dependence path of length $\Omega(N^2)$.

3 Application to the parallelization of regular nested loops

The scheduling algorithm presented in section 2.6 can be immediately applied to uniform loop nests, whose dependence graph is easy to build and similar to dependence graphs of uniform recurrence equations. It this case, the depth of the algorithm is always equal to 1, because there always exists a strictly separating hyperplane for all dependence edges.

Finding the dependences of more general nested loops is much more complicated. As said in the introduction, different dependence analysis methods have been proposed, from cheap but non precise methods to expensive but exact methods. Obviously, the more precise the analysis, the more parallelism can be detected.

When the dependence analysis is exact and expresses dependences as affine functions of the iteration vectors [Fea91], a similar decomposition algorithm can be derived by replacing inequalities such as 8 and 9, by more complicated inequalities that take into account the affine form of the dependences. This is a well known technique [Fea92b, DR92], that uses the affine form of Farkas lemma.

However, exact dependence analysis is expensive and we do believe that for most practical cases, it is not worth using it as far as parallelism detection is concerned. We claim that even with less precise dependences, it is possible to detect all parallelism. We first extend the decomposition algorithm to perfectly nested loops, for which we prove that the decomposition algorithm remains optimal, then to regular, but non necessarily perfectly, nested loops.

The principle of our method is to transform the original dependence graph into an equivalent uniform dependence graph, equivalent in the sense that the two important dual notions - null weight multi-cycles and separating hyperplanes - are preserved. This method can be applied to any dependence analysis that gives an approximation of the dependence polyhedron (sometimes called dependence cone) by precising its vertices and edges. We chosed to show how parallelizing nested loops when dependences are expressed in the most popular dependences representation, the set $(\mathbb{Z}, +, -, *)$ [Ban88, Pug92].

3.1 Generation of a uniform dependence graph

3.1.1 Dependence graph in $(\mathbb{Z}, +, -, *)$

As said in the introduction, a dependence vector between two statements is actually a condensed representation of all distance vectors between the different instances of these two statements. In the grammar $(\mathbb{Z}, +, -, *)$, each component of a dependence vector - an integer plus possibly one of the symbols $+$, $-$, $*$ - is an approximation of the set of values that the corresponding components of the distance vectors can take. The symbol $+$ represents any non negative value, the symbol $-$ any non positive value, and the symbol $*$ any value. For example, a component equals to $3-$ codes a dependence distance less or equal to 3 and a dependence vector equal to $(1, -)$ represents distance vectors $(1, -i)$ for any non negative value i.

Such a representation corresponds to a decomposition of the dependence vector in the canonical basis of $\mathbb{Z}^n$: $(\vec{f_1}, \ldots, \vec{f_n})$. Consider indeed an edge e of the dependence graph and $w(e)$ the corresponding dependence vector. Let r, s, t be vectors in $\{0,1\}^n$ defined as follows: $r_i = 1$ (respectively $s_i = 1$) (respectively $t_i = 1$) if the i-th component of $w(e)$ contains the symbol $+$ (respectively $-$) (respectively $*$). Let u be the vector whose components are the integer part of $w(e)$. For example, if $w(e) = (3+, 4-, +)$ then $u = (3, 4, 0)$, $r = (1, 0, 1)$, $s = (0, 1, 0)$ and $t = (0, 0, 0)$. In the following, we write $w(e) = u \oplus r \ominus s \otimes t$. With these notations, such a dependence vector means that statement

$h(e)$ depends on $t(e)$ by all dependences of the form:

$$\sum_{0 \le i \le n} \{u_i + \gamma_i * r_i - \lambda_i * s_i + (\mu_i - \nu_i) * t_i\} \vec{f_i} \tag{10}$$

where γ_i, λ_i, μ_i and ν_i are any elements of $\mathbb{N}^n$. Some of these dependences may not really exist, but they are nevertheless given by the dependence representation which is as said before an approximation. Therefore, any parallelism detection based on such a dependence analysis as to assume that they exist.

This is the base of our transformation.

3.1.2 Generation of an equivalent uniform dependence graph

In this section, we construct an equivalent uniform dependence graph $H = (W, F)$ from the non-uniform dependence graph $G = (V, E)$ with dependences expressed in $(\mathbb{Z}, +, -, *)$.

We start from $W = V$, $F = \emptyset$ and for each edge e of E, we add to H new vertices (we call them **intermediate vertices**) and new edges depending on the form of $w(e) = u \oplus r \ominus s \otimes t$. Let $a = t(e)$ and $b = h(e)$. e is oriented from vertex a to vertex b, where a and b are not necessarily distinct.

i. if $(r, s, t) = (0, 0, 0)$ (e is a uniform dependence vector).

- e is added to F with $w(e) = u$

ii. if $(r, s, t) \neq (0, 0, 0)$ but $t = 0$ (no symbol $*$).

- we add to V a new node a_1
- if $r_i \neq 0$, we add to E a new self-dependence edge e_i^1 on a_1, with weight $\vec{f_i}$
- if $s_i \neq 0$, we add to E a new self-dependence edge e_i^2 on a_1, with weight $-\vec{f_i}$
- we add to E an edge e^3, oriented from a to a_1, with weight u
- we add to E an edge e^4, oriented from a_1 to b, with null weight

iii. if $t \neq 0$ (at least one symbol $*$).

- we add to V two new nodes a_1 and a_2
- if $r_i \neq 0$ or $t_i \neq 0$, we add to E a new self-dependence edge e_i^1 on a_1, with weight $\vec{f_i}$
- if $s_i \neq 0$ or $t_i \neq 0$, we add to E a new self-dependence edge e_i^2 on a_2, with weight $-\vec{f_i}$
- we add to E an edge e^3, oriented from a to a_1, with weight u
- we add to E an edge e^4, oriented from a_1 to a_2, with null weight
- we add to E an edge e^5, oriented from a_2 to b, with null weight

Each simple path of H from a vertex a of G to a vertex b of G simulates with uniform dependences an edge of G. A simple path from a vertex a to a vertex b uses only once edges e^3, e^4 (and e^5 if $w(e)$ has the symbol $*$ in its components) but may use self-dependences on intermediate vertices an arbitrary number of times. This corresponds exactly to the possibility of having dependences such that those of formula 10.

Example: As an illustration, figures 4 and 5 show how an edge of weight $(2+, 3-, 7+)$ and an edge of weight $(2+, 3-, *)$ are transformed.

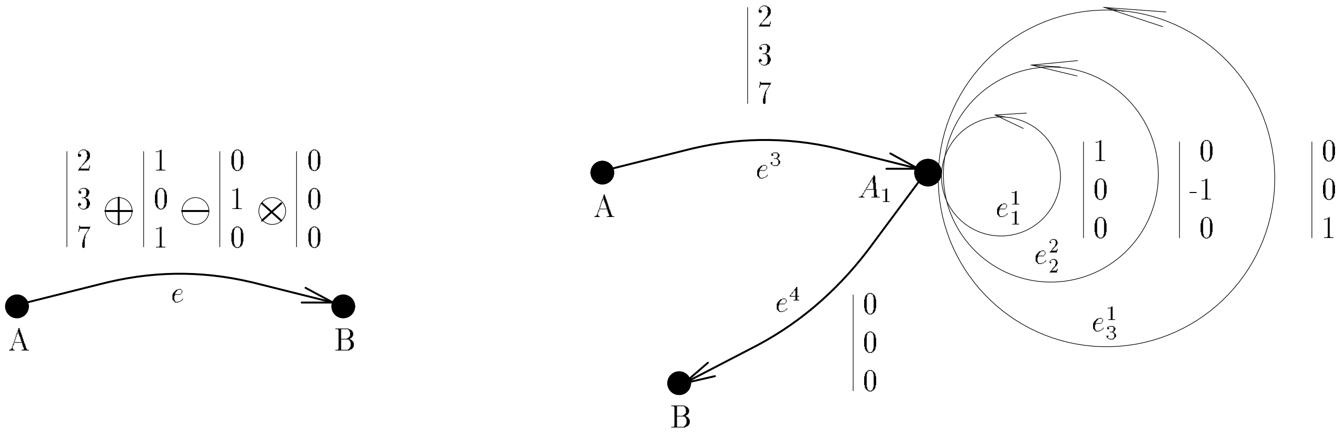

Figure 4: Example for case (ii)

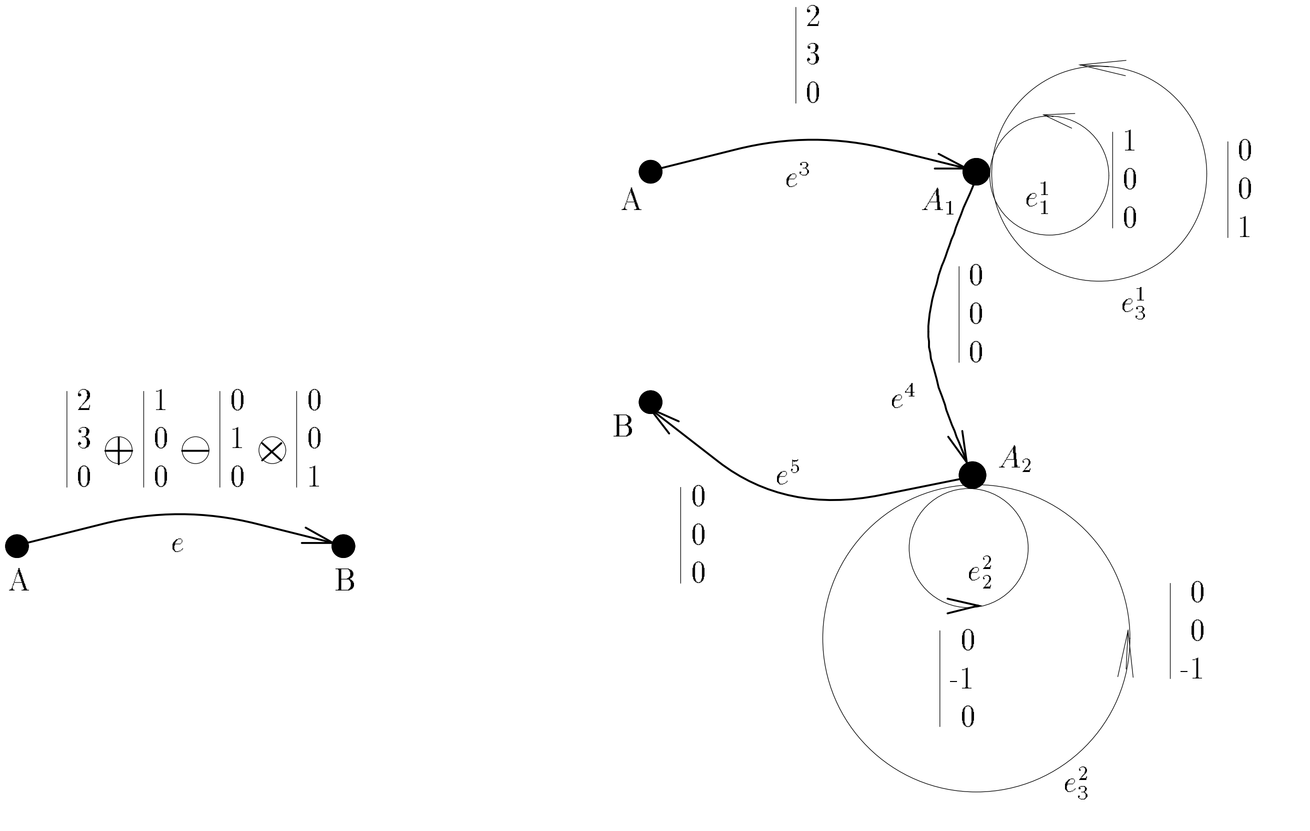

Figure 5: Example for case (iii)

3.1.3 Modified decomposition and scheduling algorithms

Applying directly the decomposition algorithm 2.2.2 to the graph H leads to valid multi-dimensional schedules for G. However, since several edges of H correspond to only one edge of G and since we are only interested in scheduling vertices of G and not intermediate vertices, one has to slightly modify the decomposition algorithm so as to correspond exactly to G, in terms of longest dependence paths (primal linear programs) and dependence constraints (dual linear programs).

Modified decomposition algorithm The modification we make comes from the following fact: a subgraph $H_{i,j}$ generated at level i of the decomposition can not contain a null weight cycle if all its vertices are intermediate vertices. Therefore, the recursion can be stopped. The depth of the algorithm is then smaller than (or equal to) the depth of algorithm 2.2.2 directly applied to H. This new definition of the depth is the right one for obtaining a result similar to theorem 2.3.

i. Decompose H into strongly connected components H_1, H_2, ..., H_s, and call step ii on each H_i.

ii. Build H' the subgraph of H generated by all edges that belong to a null weight multi-cycle of H.

   - If H' is strongly connected then G is not computable.
   - If no vertex of H' belongs to G then G is computable.
   - If H' is an empty graph then G is computable.
   - Otherwise call step i on H'.

Remark: we kept in the algorithm the possibility that G be non computable, because such an algorithm could be also applied to affine recurrence equations. Here, because we started from nested loops, this can never happen.

Theorem 4 *Let G be a non-uniform graph with dependences in $(\mathbb{Z}, +, -, *)$, and let H be the graph associated to G by algorithm 3.1.2. If d is the depth of the decomposition algorithm defined in 3.1.3 on the graph H, then there is in G a dependence path of length $\Omega(N^d)$.*

Proof: The proof is similar to the proof of theorem 2.3. We associate to each level i of the decomposition the same cycles $C_{i,j}$. The dependence path in H is obtained the same way by traversing k^i times the cycles at level i. Now, we have to convert this path into a path in G. Since vertices a_1 and a_2 can be reached only if edges e^3, e^4 and e^5 are used, one can replace each succession of edges in H involving a_1 and a_2 by one dependence edge in G. This changes the length of the path: a sub-path between a and b of length $\Omega(N)$ can be replace by a sub-path of length 1! However, this does not change the number of times an edge belonging to $C_{i,j}$ and involving a vertex of G (i.e. not an intermediate vertex) is used. It remains $\Omega(k^i)$. Thus, the result still holds, but with the new definition of the depth, i.e. with the modified decomposition algorithm. $\square$

Modified scheduling algorithm Before giving the scheduling algorithm, let us take a look at the constraints that must be satisfied for dependence vectors expressed in $(\mathbb{Z}, +, -, *)$.

Consider the case of an edge e in G, from a to b, that generates two intermediate vertices in H (this is the general case). Edge e is simulated by three "regular" edges e^3, e^4 and e^5, and at most n self-dependences e_i^1 and e_i^2. This corresponds to all dependences vectors of the form:

$$d_{\mu^1,\mu^2} = \sum_{i=1}^{n} \{\mu_i^1 w(e_i^1) + \mu_i^2 w(e_i^2)\} + w(e^3) + w(e^4) + w(e^5)$$

where the μ_i^1's and μ_i^2's are arbitrary non negative values.

A vector X is, for such an edge, a weakly (respectively strictly) separating hyperplane if for all μ^1, μ^2, one has $X d_{\mu^1,\mu^2} + \rho_b - \rho_a \geq 0$ (resp. $X d_{\mu^1,\mu^2} + \rho_b - \rho_a \geq 1$) for some constants ρ_a and ρ_b. This is actually equivalent to:

$$\begin{cases} X w(e_i^1) \geq 0, \ X w(e_i^2) \geq 0, \text{ for all } i \\ X\left(w(e^3) + w(e^4) + w(e^5)\right) + \rho_b - \rho_a \geq 0 \\ \text{respectively } X\left(w(e^3) + w(e^4) + w(e^5)\right) + \rho_b - \rho_a \geq 1 \end{cases} \tag{11}$$

Remark: this is the reason why it this possible in general to go from a representation of the dependence polyhedron expressed by linear inequalities - here a representation with the μ_i^1 and μ_i^2 - to a representation of this polyhedron by vertices and edges - here a representation with the edges e_i^1, e_i^2, e^3, e^4 and e^5. In general, this technique holds for any dependence analysis as soon as the representation with vertices and edges of the dependence polyhedron is accessible.

Inequations 11 show that one has just to find weakly separating hyperplanes that are at some level strictly separating hyperplanes for the edges of H that are not self-dependences on intermediate vertices. Furthermore, one just needs that $X\left(w(e^3) + w(e^4) + w(e^5)\right) + \rho_b - \rho_a \geq 1$ and not necessarily $X w(e^3) + \rho_{a_1} - \rho_a \geq 1$, $X w(e^4) + \rho_{a_2} - \rho_{a_1} \geq 1$ and $X w(e^5) + \rho_b - \rho_{a_2} \geq 1$.

Thus, the right linear program to consider is linear program 7 with the sum of the costs λ_i corresponding to e^3, e^4 and e^5 equal to 1, for example 1/3, 1/3 and 1/3 (or 1/2 and 1/2 when only one intermediate vertex is generated). Lemma 6 assume that all λ be positive, but here because e^4 and e^5 belong to a null weight multi-cycle each time e_3 does, one can choose $\lambda = 1$ for e^3 and $\lambda = 0$ for edges e^4 and e^5, without changing the structure of the decomposition algorithm.

Remark: as said in section 2.5, what really matters is constraints on cycles and not constraints on edges. Inequation $X\left(w(e^3) + w(e^4) + w(e^5)\right) + \rho_b - \rho_a \geq 1$ contains all important informations. The choice of the delays do not really matter as long as their sum is equal to 1.

The modified scheduling algorithm is thus the same as the scheduling algorithm of section 2.6, except that different costs are attributed to the edges of H. All costs are set to 1 except for edges of type e^4 and e^5 whose cost is set to 0.

Finally, the result of theorem 4 and the above construction of a d-multi-dimensional schedule lead to following corollary:

Corollary 5 *if d is the depth of the modified decomposition algorithm, one can build a schedule whose latency is $O(N^d)$ whereas the length of the longest dependence path is $\Omega(N^d)$.*

This proves that our parallelization technique is optimal in the sense that no more parallelism can be detected. However, this optimality depends on the quality of the dependence analysis: it may be possible to detect more parallelism if a more precise analysis removes some false dependences.

3.1.4 Example

Here is an example of perfectly nested loops with affine dependences:

```
for i = 0 to n do
    for j = 0 to n do
        for k = 0 to n do
            a(i, j, k) = c(i, j − 1, k) + c(i − 1, j − 1, k) + a(i, j, k − 1)
            c(i, j, k) = a(i, j − 1, k + i) + c(i, j, k − 1)
        endfor
    endfor
endfor
```

The non-uniform dependence graph is shown on figure 6. The uniform dependence graph obtained from the non-uniform one is identical to the dependence graph of figure 1 (where a_1 was renamed b).

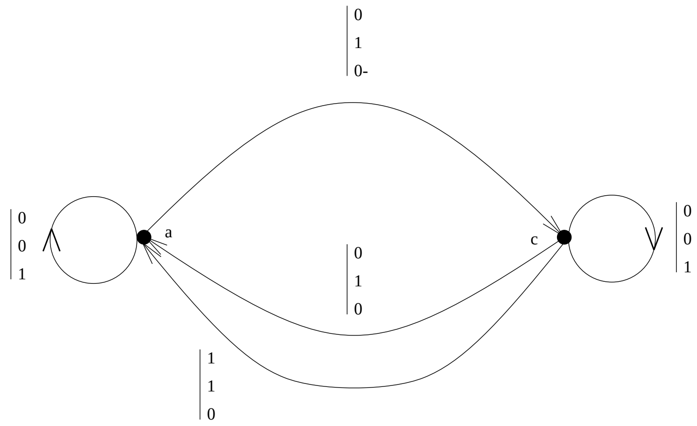

Figure 6: The non-uniform dependence graph

The depth of the decomposition algorithm is 2. There is a multi-cycle of null weight, involving the three self dependences. H' has three strongly connected components, but none of them contains a null weight cycle and the algorithm stops at depth 2. Each cycle of the graph gives a constraint for $X^{(1)}$, depending whether the corresponding edges are in H' or not:

$$X^{(1)}\begin{pmatrix}0\\0\\1\end{pmatrix}=0 \quad X^{(1)}\begin{pmatrix}0\\0\\1\end{pmatrix}=0 \quad X^{(1)}\begin{pmatrix}0\\0\\-1\end{pmatrix}=0$$

$$X^{(1)}\begin{pmatrix}0\\2\\0\end{pmatrix}\geq 2 \qquad X^{(1)}\begin{pmatrix}1\\2\\0\end{pmatrix}\geq 2$$

A possible solution is $X_a^{(1)} = X_{a_1}^{(1)} = X_c^{(1)} = (0,1,0)$. Constants are $\rho_a^1 = \rho_c^1 = \rho_{a_1}^1 = 0$. At level two, each vertex has its own scheduling vector, constraints are given by the self dependences. We only keep vertices that are not intermediate vertices: $X_a^{(2)} = X_c^{(2)} = (0,0,1)$. This corresponds to the following "loops" execution:

```
for j = 0 to n do              /* (vector (0,1,0)) */
    for k = 0 to n do              /* (vector (0,0,1)) */
        forall i = 0 to n do
```

$a(i,\ j,\ k) = c(i,\ j-1,\ k) + c(i-1,\ j-1,\ k) + a(i,\ j,\ k-1)$
$c(i,\ j,\ k) = a(i,\ j-1,\ k+i) + c(i,\ j,\ k-1)$
endforall
endfor
endfor

Thus, our technique shows that a simple interchange of loops permits to parallelize the nested loops. However, when applying Kennedy and Allen's algorithm [KA87], one finds no parallelism at all:

- At level 0, the algorithm finds no parallelism because of the edge between c and a of weight $(1, 1, 0)$. This edge deleted, the graph is still strongly connected.
- At level 1, the algorithm finds no parallelism because of the edge between c and a of weight $(0, 1, 0)$. This edge deleted, the graph contains three strongly connected components and no parallelism can be found at level three.

3.2 Application to non perfect loop nests

We now consider the case of non perfect loop nests. We show how transforming the dependence graph of a non perfect loop nest with dependences in $(\mathbb{Z}, +, -, *)$ into an equivalent dependence graph corresponding to a perfect loop nest. This new dependence graph is then scheduled as in section 3.1.3.

3.2.1 Equivalent perfect loop nest

Consider a non perfect loop nest N_1 and its dependence graph G.

Let $i_1, \ldots, i_l$ be the loop counters of N_1 and let $S_1, \ldots, S_s$ be the statements of N_1. Loop counters and statements are considered in the textual order.

Let $U(i)$ and $L(i)$ be the upper and lower bounds that define loop counter i. For all statement S of N_1, let $I(S)$ be the set of loop counters corresponding to the loops that surround S. Let $B(S)$ (resp. $A(S)$) be the set of loop counters declared textually before (resp. after) S and that do not correspond to loops surrounding S.

Let $P(i_k) = 1$ if $i_k \in B(S)$ for some statement S and let $P(i_k) = 0$ otherwise. Let $N(i_k) = 1$ if $i_k \in A(S)$ for some statement S and let $N(i_k) = 0$ otherwise.

Consider the perfect loop nest N_2 defined as follows:

for $i_1 = L(i_1) - N(i_1)$ to $U(i_1) + P(i_1)$ do
...
for $i_l = L(i_l) - N(i_l)$ to $U(i_l) + P(i_l)$ do
if $\{\forall i_k \in B(S_1),\ i_k = U(i_k) + P(i_k)\}$
and $\{\forall i_k \in A(S_1),\ i_k = L(i_k) - N(i_k)\}$ do S_1
...
if $\{\forall i_k \in B(S_s),\ i_k = U(i_k) + P(i_k)\}$ and
$\{\forall i_k \in A(S_s),\ i_k = L(i_k) - N(i_k)\}$ do S_s
endfor
...
endfor

One can check that:

Lemma 9 *The perfect loop nest N_2 is equivalent to the non perfect loop nest N_1.*

Remark: in practice, there is no need to build these nested loops. They just help understanding the transformation of the dependence graph.

The dependence graph F associated to N_2 has the same structure as graph G, except that in F, all dependence vectors have the same dimension (in G, the dimension of a dependence vector between S_1 and S_2 is equal to the number of common loops surrounding S_1 and S_2).

Lemma 10 *A dependence vector w in G from statement S_i to statement S_j corresponds to the dependence vector w' in F defined as follows:*

- *if $i_k \in I(S_i) \cap I(S_j)$, dimension k corresponds to a loop surrounding both statements: $w'_k = w_k$.*
- *if $i_k \notin I(S_i) \cup I(S_j)$, dimension k correspond to a loop involving neither S_i nor S_j: $w'k = 0$.*
- *Otherwise, if $i < j$, S_i is declared before S_j: $w'k = (1+)$ and if $i > j$, S_i is declared after S_j: $w'k = (-1-)$.*

Proof: One has just to test all possible distance vectors in loop nest N_2. □

Now we schedule F with algorithm 3.1.3. Each statement S in N_2 is scheduled by a family of l-dimensional vectors. Considering only the components of these vectors that correspond to loop counters in $I(S)$ gives a multi-dimensional schedule for statement S in N_1. This will come clear in the examples.

Remark: another way to treat non perfect loop nests is to work first on the loops that are common to all statements and then recursively to schedule the inner loops. When working on extern common loops, one has to consider for each dependence vector the sub-vector that corresponds to the studied loops. This comes from the fact that symbols $+$ and $-$ in dependence vectors between statements that do not share the same surrounding loops induce null components in the separating hyperplane X. It can be shown that this approach is fundamentally equivalent to the one previously exposed.

3.2.2 Examples

Here are two examples that illustrate our scheduling algorithm. The first one shows the power of our technique on a quite complicated example, the second one shows that this complicated algorithm is still able to treat very simple cases!

First example Here is an example of non-perfectly nested loops with uniform dependences:

```
for i = 1 to n do
    for j = 1 to n do
        a(i,j) = a(i,j-1) + b(i,j-1,1)                     /* (instruction S1) */
        for k = 1 to n do
            b(i,j,k) = b(i+1,j-1,k-1) + a(i,j-1)  /* (instruction S2) */
        endfor
    endfor
endfor
```

In this example there are four dependencies: S_1 depends on S_2, S_2 on S_1, and S_1 on itself with the same dependence vector: $(0,\ 1)$; S_2 depends on itself with the dependence vector $(1,\ -1,\ -1)$. The four dependence vectors belong to the same strongly connected component of the dependence graph.

In the new perfect loop nest equivalent we build, these dependence vectors become respectively: $(0,\ 1,\ 1-)$, $(0,\ 1,\ 1+)$, $(0,\ 1,\ 0)$ and $(1,\ -1,\ -1)$. The uniform dependence graph associated has a null weight multi-cycle but this multi-cycle does not involve vertices of the original dependence graph. Thus, the depth of the modified decomposition algorithm is only one and a strictly separating hyperplane can be found: the vector $X = (2,\ 1,\ 0)$ with null constants is a suitable solution.

The third component of the strictly separating hyperplane is necessarily equal to zero because of the simultaneous presence of dependence vectors with a third component containing a "+", and of dependence vectors with a third component containing a "-". The strong connectivity of the dependence graph and the dependences between instructions inside and outside the loop on k insure this simultaneous presence. An equivalent level by level approach is thus to consider only the two first components of all dependence vectors: $(0,\ 1)$, $(0,\ 1)$, $(0,\ 1)$, and $(1,\ -1)$. This set of vectors admits for strictly separating hyperplane the vector $(2,\ 1)$ with null constants.

Finally for parallelizing the loop nest, we use the linear scheduling (the constants are null) of vector $(2,\ 1)$ for instruction S_1, and the linear scheduling of vector $(2,\ 1,\ 0)$ for instruction S_2. We obtain one external sequential loop that contains one (resp. two) parallel loops for S_1 (resp. S_2).

Remark: to visualize the result of such a transformation, use for example TINY: skew the second loop (loop j) by factor 2 and interchange the two first loops (i and j)). Then, the two most internal loops are parallel!

Second example As second example of parallelization of non-perfectly nested loops, let us consider a program of "LU" matrix decomposition. On this simple example, our algorithm finds obviously the intuitive parallelization.

```
for k = 1 to n do
    for i = k + 1 to n do
        a(i, k) = a(i, k)/a(k, k)                /* (instruction A) */
        for j = k + 1 to n do
            a(i, j) = a(i, j) - a(k, j) * a(i, k)          /* (instruction B) */
        endfor
    endfor
endfor
```

The perfect loop nest associated to these non perfectly nested loops is of depth of 3 $(k,\ i,\ j)$ and contains no null weight multi-cycle. A possible solution is $X_A^{(1)} = X_B^{(1)} = (2,0,0)$, $\rho_A^1 = 0$ and $\rho_B^1 = 1$ and corresponds to the following nest:

```
for k = 1 to n do
    forall i = k + 1 to n do
            a(i, k) = a(i, k)/a(k, k)
    endforall
    forall i = k + 1 to n do
            forall j = k + 1 to n do
                    a(i, j) = a(i, j) - a(k, j) * a(i, k)
            endforall
```

```
    endforall
endfor
```

Once again, one could work level by level, by considering only the first component on all vectors.

4 Conclusion

4.1 Basis of our results

Our work is mainly based on three papers:

i. the seminal paper by Karp, Miller and Winograd [KMW67] that presented for the first time an algorithm for determining the computability of a system of uniform recurrence equations. Most ideas were already expressed in this paper, but not completely developed.

ii. Rao's thesis [Rao85] where we found the main idea for generating a dependence path whose length is of the same order than the latency of an optimal schedule (theorem 2.3). We extended Rao's proof that was incomplete.

iii. Feautrier's work [Fea92a] on multi-dimensional scheduling. Feautrier presented as a greedy heuristic his method based on a linear program similar to program 5 which maximize the number of satisfied dependence constraints. We showed that this method is optimal.

We showed how Karp, Miller and Winograd's algorithm can be applied to the parallelization of uniform loop nests, of perfect loop nests and of non perfect loop nests. We justified our technique by optimality results, by showing the link between cycles in the dependence graph and linear dependence constraints (i.e. separating hyperplanes).

4.2 Advantages of our method

We believe that our method has four main qualities:

- It can be applied to any dependence analysis as soon as edges and vertices of the dependence polyhedron are available. We showed in this paper how it can be applied to dependence graphs with dependence vectors in $(\mathbb{Z}, +, -, *)$. As an application, we plan to implement our parallelization technique in the loop restructuring research tool TINY, in which a powerful dependence analysis has been developed [Pug92].

- It maintains regularity and simplicity by trying to schedule, each time it is possible, all statements with the same scheduling vector (more precisely all statements in the current graph G' of the decomposition algorithm). A scheduling vector corresponds to a loop representation in terms of code generation: simpler scheduling functions lead to simpler code generation. Our method gives less complicated results, although near-optimal, than Feautrier's method which generates piece-wise affine multi-dimensional schedules. Our optimality results prove that there is no need to look a priori for arbitrary affine schedules. We will discuss in future work the problem of code generation. We hope that the reader is convinced that it is feasible.

- All parallelization techniques based on loops transformations (except tiling) correspond to a multi-dimensional schedule. Our method permits to build an optimal multi-dimensional schedule. Thus, among all parallelization methods based on loops transformations (heuristic choice of schedules such that Pugh's method [KP93] or level-by-level method such that Kennedy and Allen's method [KA87]), our method is the most powerful: no more parallelism can be detected by loops transformations.

- Thinking in terms of cycles (the dual approach of schedules) permits to reveal the limiting factor for parallelizing. Finding the null weight multi-cycles shows immediately why some loops can not be parallelized by a loop transformation. We believe that this remark can be useful for understanding how dependence analysis could be improved, without going to exact dependence analysis. We show in the next section two examples that illustrate this point.

4.3 Limitation due to the dependence analysis

As we said all along the paper, our method is optimal **with respect to** the dependence analysis. If the dependence analysis let us believe that a dependence exists, our method can fail to detect some parallelism if the false dependence generates some sequentiality: this problem is clear when considering the longest path in the uniform dependence graph associated to the nested loops. We give here two examples, the first one for perfect loop nests, the second one for non perfect loop nests. Both reveal a different type of weakness of a dependence analysis expressed in $(\mathbb{Z}, +, -, *)$.

When expressing a dependence in $(\mathbb{Z}, +, -, *)$, components of dependence vectors are independent. A dependence vector $(+, -)$ means that all dependence distances $(+\lambda, -\mu)$ (with λ and μ positive integer) are possible. There is no way to describe dependent components. The next example illustrates this problem:

```
for i = 0 to n do
    for j = 0 to n do
        a(i, j) = a(j, i) + a(i, j - 1)
    endfor
endfor
```

The three dependences for this example expressed in $(\mathbb{Z}, +, -, *)$ are $(+, -)$, $(+, -)$ and $(0, 1)$. Therefore, there is a null weight multi-cycle in the dependence graph and the nested loops can not be parallelized. Actually, they can not be parallelized because the dependence analysis makes us believe that a dependence vector such as $(1, -n)$ for example exists. An improvement on the dependence analysis would be to express the dependence between $a(j, i)$ and $a(i, j)$ as $+(1, -1)$: all dependence vectors are indeed of the form $(i - j, j - i)$. $+(1, -1)$ means all vectors $\lambda(1, -1)$ with $\lambda \geq 0$, which is a more precise representation of the dependence cone. In this case, the decomposition algorithm would find that the loops can be parallelized with vector $X = (2, 1)$.

When dealing with non perfect loop nests, a dependence analysis in $(\mathbb{Z}, +, -, *)$ describes only the components of the dependence distances that correspond to common loops surrounding the two concerned statements. No information is given about other loops indices. For example, no difference is made between these two loop nests:

```
for i = 0 to n do
    a(i, 0) = a(i - 1, 1)
    for j = 1 to n do
        a(i, j) = a(i, j - 1)
    endfor
endfor
```

```
for i = 0 to n do
    a(i, 0) = a(i - 1, n)
    for j = 1 to n do
        a(i, j) = a(i, j - 1)
    endfor
enfor
```

However, the first one could be parallelized whereas the second one is sequential.

Future work will be to study efficient code generation (not only loop rewriting but also data distribution) and possible improvements on the dependence analysis.

Dedication We would like to dedicate this work to Hervé Leverge who died a few months ago. Hervé was a bright researcher in the field of systolic methodology design. He liked to smoke on bridge parapets. One day, for an unknown reason, he slipped and failed.

References

[Ban88] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2:133–149, 1988.

[DKR91] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[DR92] Alain Darte and Yves Robert. Scheduling uniform loop nests. Technical Report 92-10, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, February 1992. published in IEEE Trans. Parallel Distributed Systems, August 1994.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992. Available as Technical Report 92-78, Laboratoire MASI, Université Pierre et Marie Curie, Paris, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992. Available as Technical Report 92-28, Laboratoire MASI, Université Pierre et Marie Curie, Paris, May 1992.

[KA87] Ken Kennedy and J.R. Allen. Automatic translations of fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, jul 1967.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, April 1993.

[KS88] S. Rao Kosaraju and Gregory F. Sullivan. Detecting cycles in dynamic graphs in polynomial time (preliminary version). In ACM Press, editor, *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 398–406, May 1988.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, feb 1974.

[LW92] M.S. Lam and M. E. Wolf. Automatic blocking by a compiler. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 537–542, 1992.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, aug 1992.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In Françoise Fogelman Soulie, Yves Robert, and Maurice Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987.

[Rao85] Sailesh K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, oct 1985.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.