On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops

Alain Darte and Frédéric Vivien*

Laboratoire LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07
e-mail: [Alain.Darte,Frederic.Vivien]@lip.ens-lyon.fr

Abstract

We explore the link between dependence abstractions and maximal parallelism extraction in nested loops. Our goal is to find, for each dependence abstraction, the minimal transformations needed for maximal parallelism extraction. The result of this paper is that Allen and Kennedy's algorithm is optimal when dependences are approximated by dependence levels. This means that even the most sophisticated algorithm cannot detect more parallelism than found by Allen and Kennedy's algorithm, as long as dependence level is the only information available. In other words, loop distribution is sufficient for detecting maximal parallelism in dependence graphs with levels.

1 Introduction

Many automatic loop parallelization techniques have been introduced over the last 25 years, starting from the early work of Karp, Miller and Winograd [17] in 1967 who studied the structure of computations in repetitive codes called systems of uniform recurrence equations. This work defined the foundation of today's loop compilation techniques. It has been widely exploited and extended in the systolic array community (among others [22, 26, 27, 28, 7] are directly related to it), as well as in the compiler-parallelizer community: Lamport [20] proposed a parallel scheme - the hyperplane method - in 1974, then several loop transformations were introduced (loop distribution/fusion, loop skewing, loop reversal, loop interchange, ...) for vectorizing computations, maximizing parallelism, maximizing locality and/or minimizing synchronizations. These techniques have been used as basic tools for optimizing algorithms, the most two famous being certainly Allen and Kennedy's algorithm [1, 2], designed at Rice in the PFC system, and Wolf and Lam's algorithm [29], designed at Stanford in the SUIF compiler.

*Supported by the CNRS-INRIA project *ReMaP*.

At the same time, dependence analysis has been developed so as to provide sufficient information for checking the legality of these loop transformations, in the sense that they do not change the final result of the program. Different abstractions of dependences have been defined (dependence distance [23], dependence level [1, 2], dependence direction vector [30, 31], dependence polyhedron/cone [16], ...), and more and more accurate tests for dependence analysis have been designed (Banerjee's tests [3], I test [19, 24], Δ test [13], λ test [21, 14], PIP test [10], PIPS test [15], Omega test [25],...).

In general, dependence abstractions and dependence tests have been introduced with some particular loop transformations in mind. For example, the dependence level was designed for Allen and Kennedy's algorithm, whereas the PIP test is the main tool for Feautrier's method for array expansion [10] and parallelism extraction by affine schedulings [11, 12]. However, very few authors have studied, in a general manner, the links between both theories, dependence analysis and loop restructuring, and have tried to answer the following two dual questions:

- What is the minimal dependence abstraction needed for checking the legality of a given transformation?
- What is the simplest algorithm that exploits at best all information provided by a given dependence abstraction?

With the answer to the first question, we can adapt the dependence analysis to the parallelization algorithm, and avoid implementing an expensive dependence test if it is not needed. This question has been deeply studied in Yang's thesis [33], and summarized in Yang, Ancourt and Irigoin's paper [32].

Conversely, with the answer to the second question, we can adapt the parallelization algorithm to the dependence analysis, and avoid using an expensive parallelization algorithm if we know that a simpler one is able to find the same degree of parallelism, and for a smaller cost. This question has been addressed by Darte and Vivien in [6] for dependence abstractions based on a polyhedral approximation.

Completing this work, we propose in this paper, a more precise study of the link between *dependence abstractions* and *parallelism extraction* in the particular case of *dependence levels*. Our main result is that, in this context, Allen and Kennedy's parallelization algorithm is optimal for parallelism extraction, which means that even the most sophisticated algorithm cannot detect more parallel loops than Allen and Kennedy's algorithm does, as long as dependence level is the only information available. In other words, loop distribution is sufficient for detecting maximal parallelism in dependence graphs with levels. There is no need to use more complicated transformations such as loop interchange, loop skewing, or any other transformations that could be invented, because there is an intrinsic limitation in the dependence level abstraction itself that prevents detecting more parallelism.

The rest of the paper is organized as follows. In Section 2, we explain what we call maximal parallelism extraction for a given dependence abstraction and we recall the definition of dependence levels. Section 3 presents Allen and Kennedy's algorithm in its simplest form - which is sufficient for what we want to prove. The proof of our result is then subdivided into two parts. In

Section 4, we build a set of loops that are equivalent to the loops to be parallelized, in the sense that they have the same dependence graph. Then, we prove that these loops contain exactly the degree of parallelism found by Allen and Kennedy's algorithm (the proof is postponed in the appendix). Finally, Section 5 summarizes the paper.

2 Theoretical framework

For simplifying the notations, we first restrict to the case of perfectly nested loops. We explain at the end of this paper how our optimality result can be extended to non perfectly nested loops.

2.1 Notations

The notations used in the following sections are:

- $f(N) = O(N)$ if $\exists\ k > 0$ such that $f(N) \leq kN$ for all sufficiently large N.
- $f(N) = \Omega(N)$ if $\exists\ k > 0$ such that $f(N) \geq kN$ for all sufficiently large N.
- $f(N) = \Theta(N)$ if $f(N) = O(N)$ and $f(N) = \Omega(N)$.
- If X is a finite set, $|X|$ denotes the number of elements in X.
- $G = (V, E)$ denotes a directed graph with vertices V and edges E.
- $e = (x, y)$ denotes an edge from vertex x to vertex y.

2.2 Dependence graphs

The structure of perfectly nested loops can be captured by an ordered set of statements $S_1, \ldots, S_s$ (where S_i is textually before S_j if $i < j$) and an iteration domain $\mathcal{D} \subset \mathbb{Z}^n$ that describes all possible values of the loop counters; n is the number of nested loops. Given a statement S, to each n-dimensional vector $I \in \mathcal{D}$ corresponds a particular execution (called instance) of S, denoted by $S(I)$.

2.2.1 EDG, RDG and ADG

Dependences (or precedence constraints) between instances of statements define the **expanded dependence graph (EDG)** also called iteration level dependence graph. The vertices of the EDG are all possible instances $\{S_i(I) \mid 1 \leq i \leq s \text{ and } I \in \mathcal{D}\}$. There is an edge from $S_i(I)$ to $S_j(J)$, denoted by $S_i(I) \Longrightarrow S_j(J)$, if executing instance $S_j(J)$ before instance $S_i(I)$ may change the result of the program, i.e. if $S_i(I)$ and $S_j(J)$ satisfy Bernstein's conditions [4].

Definition 1 *For all* $1 \leq i, j \leq s$, *we define the* **distance set** $E_{i,j}$ *by:*

$$E_{i,j} = \{(J - I) \mid S_i(I) \Longrightarrow S_j(J)\} \quad (E_{i,j} \subset \mathbb{Z}^n)$$

In general, the EDG (and the distance sets) cannot be computed at compile-time, either because some information is missing (such as the values of size parameters or even worse, exact accesses to memory), or because generating the whole graph is too expensive. Instead, dependences are captured through a smaller, (in general) cyclic, directed graph, with s vertices (one per statement in the loop nest), called the **reduced dependence graph (RDG)** (or statement level dependence graph). The RDG can contain more than one edge to represent a single dependence. Each edge e has a label $w(e)$. This label has a different meaning depending upon the dependence abstraction that is used: it represents [1] a set $D_e \subset \mathbb{Z}^n$ such that:

$$\forall i, j,\ 1 \le i, j \le s,\ E_{i,j} \subset \left(\bigcup_{e=(S_i,S_j)} D_e \right) \tag{1}$$

In other words, the RDG describes, in a condensed manner, an iteration level dependence graph, called (maximal) **apparent dependence graph (ADG)**, that is a superset of the EDG. The ADG and the EDG have the same vertices, but the ADG has more edges, defined by:

$$S_i(I) \Longrightarrow S_j(J) \text{ (in the ADG) } \Leftrightarrow \exists\ e = (S_i, S_j) \text{ (in the RDG) such that } (J - I) \in D_e.$$

Equation 1 and Definition 1 ensure that the ADG is a super-approximation of the EDG.

2.2.2 Dependence level abstraction

In Sections 3 and 4, we will focus mainly on the case of RDGs labeled by one of the simplest dependence abstractions, namely the dependence level. The reader can find a similar study for other dependence abstractions in [6].

The **dependence level** associated to a dependence distance $J - I$ where $S_i(I) \Rightarrow S_j(J)$ is defined as ∞ if $J - I = 0$ or as the smallest integer l, $1 \le l \le n$ such that the l-th component of $J - I$ is non zero (and thus positive). A **reduced leveled dependence graph (RLDG)** is a reduced dependence graph whose edges are labeled by dependence levels.

Actually, with this definition, several values may be associated to a given edge of the reduced leveled dependence graph. To simplify the rest of the paper, we transform each edge labeled by k different levels into k edges with a single level. Therefore, in the following, a reduced leveled dependence graph is a multi-graph, for which each edge e has a label $l(e) \in [1 \ldots n] \cup \{\infty\}$. $l(e)$ is called the **level** of edge e. The **level** $l(G)$ of a RLDG G is the minimal level of an edge of G: $l(G) = \min\{l(e) \mid e \in G\}$.

2.2.3 Illustrating example

To better understand the links between the three concepts (EDG, RDG and ADG), let us consider a simple example, the SOR kernel:

[1] except for exact dependence analysis where it defines a subset of $\mathbb{Z}^n \times \mathbb{Z}^n$

Example 1

```
for i=1 to N
    for j=1 to N
        a(i, j) = a(i, j-1) + a(i-1, j)
    endfor
endfor
```

The EDG associated to Example 1 is given in Figure 1. The length of the longest path in the EDG is equal to $2 * N - 2$, i.e. $\Theta(N)$. As there are N^2 instances in the domain, we will say that the degree of intrinsic parallelism in this graph is 1.

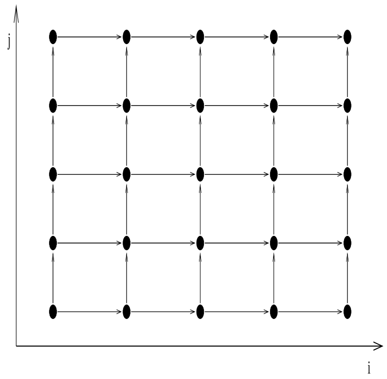

Figure 1: EDG for Example 1

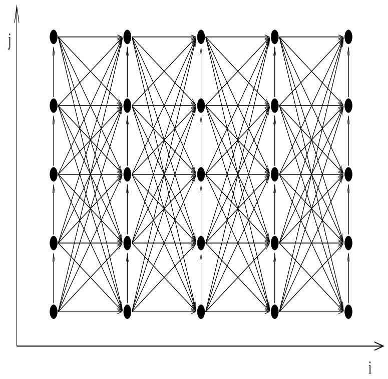

Figure 2: ADG for Example 1

The RDG has only one vertex. If it is labeled with dependence levels (i.e. if it is a RLDG), it has two edges with levels 1 and 2 (see Figure 3). Its corresponding ADG depicted in Figure 2 contains a path of length N^2. We will say that the degree of intrinsic parallelism in the RDG (to be precisely defined in Section 2.3) is 0 as there are N^2 instances in the domain.

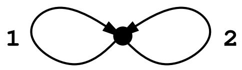

Figure 3: RLDG for Example 1

Actually, for this example, it is possible to build a set of loops - we call them the apparent loops (see Figure 4) - that have exactly the same RLDG as the original loops and that are purely sequential: there is indeed a path of length $\Omega(N^2)$ in the corresponding EDG (see Figure 5).

Since the original loops and the apparent loops cannot be distinguished by a parallelization algorithm using only the RLDG, no parallelism can be detected in this example, as long as the dependence level is the only information available. The goal of this paper is to generalize this fact to arbitrary RLDGs.

```
for i=1 to N
    for j=1 to N
        a(i, j) = 1 + a(i, j-1) + a(i-1, N)
    endfor
endfor
```

Figure 4: Apparent loops for Example 1

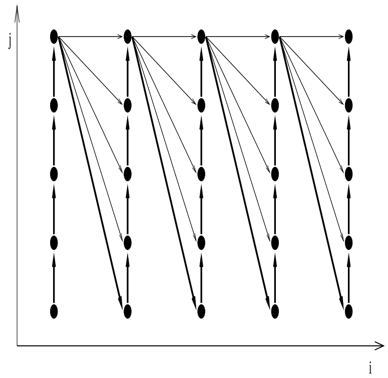

Figure 5: EDG for apparent loops

2.3 Characterizing maximal parallelism detection

In this paper, we do not consider parallelism such as parallelism exploited in DOACROSS loops, or parallelism exploited through software pipelining. We are interested only in understanding if large set of independent computations can be detected, and if they can be described by parallel loops. To make the link with a language like HPF, we are interested in detecting loops that, in HPF, can be preceded by the directive `!HPF independent`. We mark such loops by the keyword **forall** (we will write **forseq** for a loop that has been detected as a sequential loop). Remark that our **forall** loop does not change the semantic of the code, it should not be confused with a FORALL loop like in Fortran90, which corresponds in general to an array statement.

In this context, a naive definition for maximal parallelism detection is to say that we are looking for the maximal number of nested **forall** loops for each statement, or that we try to transform the code so that it contains as many nested **forall** loops as possible. Unfortunately, such a definition is not consistent, because of transformations such as loop coalescing that can change the number of nested loops. Indeed, two nested **forall** loops are equivalent to a single **forall** loop with more iterations, as the following example illustrates.

Example 2

The two following codes are equivalent, they reveal the same amount of parallelism, but the first one has more parallel loops.

```
forall i = 1 to 1000
    forall j = 1 to 100
        a(i,j) = 0
    endforall
endforall
```

```
forall I = 0 to 99999
    a((I div 100) + 1, (I mod 100) + 1) = 0
endforall
```

Therefore, one need to be more precise if we want to give a consistent definition that measures the amount of parallelism contained in a code. We consider that the only information available concerning the dependences in a set of loops L is the RDG associated to L. Any parallelism detection algorithm that transforms L into an equivalent code L_t has to preserve all dependences summarized in the RDG, i.e. all dependences described in the ADG: if $S_i(I) \Rightarrow S_j(J)$ in the ADG then $S_i(I)$ must be computed before $S_j(J)$ in the transformed code L_t.

Definition 2 *We define the* **latency** $\mathcal{T}(L_t)$ *of a transformed code* L_t *as the minimal number of clock cycles needed to execute* L_t *if:*

- *an unbounded number of processors is available.*
- *executing an instance of a statement requires one clock cycle.*
- *any other operation requires zero clock cycle.*

Of course the latency defined by Definition 2 is not equal to the real execution time. However, it allows us to define a notion of **degree of parallelism** that is completely independent on the target architecture, and that reveals intrinsic properties of the RDG of the initial sequential code. Actually, we need a more precise definition of latency, a latency for each statement of the code that we call the S-latency. The S-latency is defined as the latency except that only operations that are instances of statement S requires one clock cycle.

We can now define the degree of parallelism detected by an algorithm, with respect to a given dependence abstraction. However, to avoid the problem due to loop coalescing, and illustrated in Example 2, we need to consider parameterized codes. In the following definitions, we assume that all RDGs and ADGs are defined using the same dependence abstraction.

Definition 3 *Let A be a parallelism detection algorithm. Let L be a set of nested loops and let G be its RDG. Apply algorithm A to G and suppose, when transforming the loops, that the iteration domain $\mathcal{D}_S$ associated to statement S is contained in (resp. contains) a n_S-dimensional cube of size $O(N)$ (resp. $\Omega(N)$). Then, the* S**-degree of sequentiality extraction** *(resp.* **parallelism extraction***) for A in G is d_S (resp. $n_S - d_S$) where d_S is the smallest non negative integer such that the S-latency of the transformed code is $O(N^{d_S})$.*

Remark that Definition 3 has still a small weakness: it does not allow us to distinguish between codes that are purely sequential and codes that reveal independent sets of operations of size $\log(N)$. Nevertheless, it allows us to link the latency of a code with the length of the paths in the ADG, and the S-latency with the S-length of the paths in the ADG. The S-length of a path is the number of its vertices that are instances of statement S.

Indeed, since two operations linked by an edge in the ADG cannot be computed at the same clock cycle in the transformed code L_t, the latency of L_t, whatever the parallelization technique used, is larger than the length of the longest path in the ADG. More precisely, we have the following:

- If an algorithm is able to transform the initial loops into a transformed code whose S-latency is $O(N^{d_S})$, then the S-length of any dependence path is $O(N^{d_S})$.

- Equivalently, if the ADG contains a path of S-length that is not $O(N^{d_S})$, then whatever the parallelization technique you use, the latency of the transformed code cannot be $O(N^{d_S})$.

This leads to the following definitions:

Definition 4 *Let G be a RDG and suppose that the iteration domain $\mathcal{D}_S$ associated to statement S is contained in (resp. contains) a n_S-dimensional cube of size $O(N)$ (resp. $\Omega(N)$). Let d_S be the smallest non negative integer such that the S-length of any dependence path in the ADG is $O(N^{d_S})$. Then, we say that the S-***degree of sequentiality** *(resp.* **parallelism***) in G is d_S (resp. $n_S - d_S$) or that G contains d_S (resp. $n_S - d_S$) degrees of sequentiality (resp. parallelism).*

Remark that Definitions 3 and 4 ensure that the S-degree of parallelism extraction is always smaller than or equal to the S-degree of parallelism in a RDG.

Definition 5 *An algorithm A performs maximal parallelism extraction (or is said* **optimal for parallelism extraction***) if for each RDG G, for each statement S in G, the S-degree of parallelism extraction for A in G is equal to the S-degree of intrinsic parallelism in G.*

Defining the optimality for parallelism extraction starting from the S-latency, rather than simply the latency, allows us to discuss the quality of parallelism detection algorithms even for statements that do not belong to the most sequential part of the code.

Now, with these definitions, the optimality of a parallelism detection algorithm A can be proved as follows. Consider a set of nested loops L. Denote by G the RDG associated to L for the given dependence abstraction and by G_a its corresponding ADG. Let d_S be the S-degree of sequentiality extraction for A in G. Then, we have at least two ways for proving the optimality of A.

i. Build, for each statement S, a dependence path in G_a whose S-length is not $O(N^{d_S - 1})$.

ii. Build a set of loops L' whose RDG is also G and whose EDG contains, for each statement S, a dependence path whose S-length is not $O(N^{d_S - 1})$. The loops L' are called apparent loops (see Example 1 again).

Note that (ii) implies (i) since the EDG of L' is a subset of G_a (L and L' have the same RDG). Therefore, proving (ii) is, a priori, more powerful. In particular, it reveals the intrinsic limitations, for parallelism extraction, due to the dependence abstraction itself: even if the degrees of intrinsic parallelism in L and L' may be different at run-time (i.e. their EDGs may be different), they cannot be distinguished by the parallelization algorithm, since L and L' have the same RDG. In other words, a parallelism detection algorithm will parallelize L and L' in the same way. Therefore, since L' is parallelized optimally, the algorithm is considered optimal with respect to the dependence abstraction that is used.

The first technique is used in [9] for polyhedral approximations of dependences. In this paper, we will use the second technique. Both techniques are equivalent for uniform dependence graphs since in this case the EDG and the ADG are equal. Figure 6 recalls the links between the original loops L, the apparent loops L' and their EDGs, RDGs and ADGs.

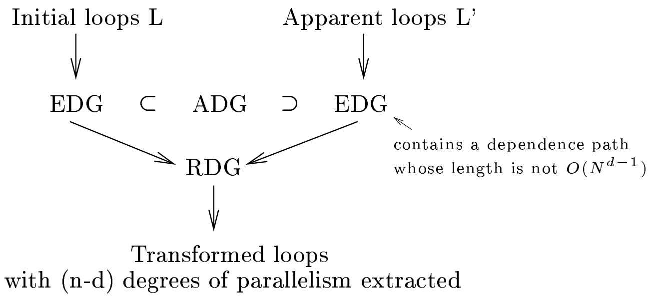

Figure 6: Links between L, L' and their EDGs, ADGs and RDGs

One could argue that the latency (and the S-latency) of a transformed code is not easy to compute. Indeed, in the general case, the latency can be computed only by executing the transformed code with a fixed value of N. However, for most known parallelizing algorithms, the S-degree of parallelism extraction (but not necessarily the S-latency) can be computed simply by examining the structure of the transformed code, as shown by lemma 1. In this case, we retrieve the intuitive definition of S-degree of parallelism equal to the number of nested **forall** loops that surround statement S.

Lemma 1 *Assume that each statement S of the initial code L appears only once in the transformed code L_t and is surrounded in both L and L_t by n_S loops. Furthermore, assume that the iteration domain $\mathcal{D}_t$ described by these n_S loops contains a n_S-cube $\underline{\mathcal{D}}$ of size $\Omega(N)$ and is contained in a n_S-cube $\overline{\mathcal{D}}$ of size $O(N)$. Then, the number of parallel loops that surround S is the S-degree of parallelism extraction.*

Proof Consider a given statement S of the initial code L. To simplify the arguments of the proof, denote by L_r the code obtained by removing from L_t everything that does not involve the instances of S: the latency of L_r is, by definition, the S-latency of L_t. Furthermore, L_r is a set of n_S perfectly nested loops that surround statement S. Let $\overline{L}$ (resp. $\underline{L}$) be the code obtained by changing the loop bounds of L_r so that they describe $\overline{\mathcal{D}}$ (resp. $\underline{\mathcal{D}}$) instead of $\mathcal{D}_t$.

Since $\underline{\mathcal{D}} \subset \mathcal{D}_t \subset \overline{\mathcal{D}}$, the latency of L_r is larger than the latency of $\underline{L}$ and smaller than the latency of $\overline{L}$. Furthermore, since $\overline{\mathcal{D}}$ and $\underline{\mathcal{D}}$ are n_S-cubes, the latency is easy to compute: the latency of $\underline{L}$ is $\Omega(N^d)$ and the latency of $\overline{L}$ is $O(N^d)$, where d is the number of sequential loops that surround S. Therefore, the latency of L_r is $\Theta(N^d)$ and the S-degree of parallelism extraction in L_r (and thus the S-degree of parallelism extraction in L) is $(n_S - d)$, i.e. the number of parallel loops that surround statement S. ■

3 Allen and Kennedy's algorithm

Allen and Kennedy's algorithm has first been designed for vectorizing loops. Then, it has been extended so as to maximize the number of parallel loops and to minimize the number of synchronizations in the transformed code. It has been shown (see details in [5, 34]) that for each statement of the initial code, as many surrounding loops as possible are detected as parallel loops. Therefore, one could think that what we want to prove in this paper has been already proved! However, looking precisely into the details of Allen and Kennedy's proof reveals that what has actually been proved is the following: consider a statement S of the initial code and L_i one of the surrounding loops. Then L_i will be marked as parallel if and only if there is no dependence at level i between two instances of S. This result proves that the algorithm is optimal among all parallelization algorithms that describe, in the transformed code, the instances of S with exactly the same loops as in the initial code. This does not prove a general optimality property as in Definition 5. In particular, this does not prove that it is not possible to detect more parallelism with more sophisticated techniques than loop distribution and loop fusion. This paper gives an answer to this question.

First, we recall Allen and Kennedy's algorithm, in a very simple form, since we are interested only in detecting parallel loops and not in the minimization of synchronization points. The initial call is ALLEN-KENNEDY(G, 1) where G is the RLDG of the code to parallelize.

ALLEN-KENNEDY(G, k)

i. remove from G all edges of level $< k$.

ii. compute the strongly connected components of G.

iii. **for every** strongly connected component C in topological order **do**

  (i) **if** C is reduced to a single statement S, with no edge, **then** generate **forall** loops in all remaining dimensions, i.e. from level k to level n_S, and generate code for S.

  (ii) **else**

    i. let $l = l_{\min}(C)$.

    ii. generate **forall** loops from level k to level $l-1$, and a **forseq** loop for level l.

    iii. call ALLEN-KENNEDY(C, $l+1$).

```
for i=1 to N
    for j=1 to N
        a(i, j) = i
        b(i, j) = b(i, j-1) + a(i, j) × c(i-1, j)
        c(i, j) = 2 × b(i, j) + a(i, j)
    endfor
endfor
```

Figure 7: Code for Example 3

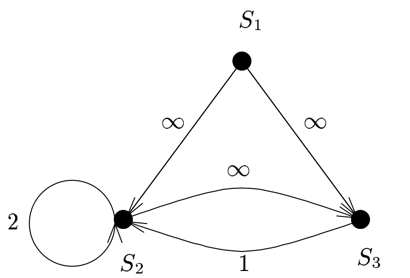

Figure 8: RLDG for Example 3

Example 3

We illustrate Allen and Kennedy's algorithm on the code given in Figure 7. There are three statements S_1, S_2 and S_3 in textual order. The first call is ALLEN-KENNEDY(G, 1) that detects two strongly connected components $\mathcal{V}_1 = \{S_1\}$ and $\mathcal{V}_2 = \{S_2, S_3\}$. The first component $\mathcal{V}_1$ has no edge. Therefore, the algorithm generates two parallel loops and the code for S_1. The second component has a level 1 edge, thus the algorithm generates a sequential loop and recursively calls ALLEN-KENNEDY($\mathcal{V}_2$, 2). See Figure 9.

Edges of level strictly less than 2 are then removed. Two strongly connected components appear $\mathcal{V}_{2,1} = \{S_2\}$ and $\mathcal{V}_{2,2} = \{S_3\}$ in this order. The level of $\mathcal{V}_{2,1}$ is 2, therefore the algorithm generates one sequential loop and recursively calls ALLEN-KENNEDY($\mathcal{V}_{2,1}$, 3), which generates the code for S_2. The second component $\mathcal{V}_{2,2} = \{S_3\}$ has no edge, thus the algorithm generates directly one parallel loop and the code for S_3. See Figure 10 for the final code.

```
forall i=1 to N
      forall j=1 to N
            a(i, j) = i
      endforall
endforall
forseq i=1 to N

      ALLEN-KENNEDY(V2, 2)

endforseq
```

Figure 9: Code after one call

```
forall i=1 to N
      forall j=1 to N
            a(i, j) = i
      endforall
endforall
forseq i=1 to N
      forseq j=1 to N
            b(i, j) = b(i, j-1) + a(i, j) × c(i-1, j)
      endforseq
      forall j=1 to N
            c(i, j) = 2 × b(i, j) + a(i, j)
      endforall
endforseq
```

Figure 10: Final parallelized code

4 Generation of apparent loops

We now show how to built the apparent loops L' defined in Section 2.3. We present a systematic procedure called *Loop_Nest_Generation*, that builds, from a reduced leveled dependence graph G, a perfect loop nest L' whose RLDG is exactly G. In the appendix, we will prove that the S-length of the longest path in the EDG of L' is of the same order as the S-latency of the code parallelized by ALLEN-KENNEDY, for each statement S of G, thereby proving the optimality of Allen and Kennedy's algorithm with respect to the dependence level abstraction.

Let $G = (E, V)$ be a RLDG. We assume that G has been built in a consistent way from some nested loops. Therefore, vertices can be numbered according to the topological order defined by the edges whose level is ∞ (loop independent dependences): $v_i \xrightarrow{\infty} v_j \Rightarrow i < j$. We denote by d the **dimension** of G: $d = \max\{l(e) \mid e \in E \text{ and } l(e) < \infty\}$.

The apparent loops L' corresponding to G consist of d perfectly nested loops, with $|V|$ statements, denoted by $T_1, \dots, T_{|V|}$. Each statement is of the form $a_i[I] = \text{rhs}(i)$ where a_i is a d-dimensional array and $\text{rhs}(i)$ is the right-hand side that defines array a_i. Most dependences in L' are uniform dependences, except dependences corresponding to edges that we call **critical** edges.

Critical edges are defined as follows. During the recursive calls to ALLEN-KENNEDY, we select some edges that we consider in a special way when generating the apparent loops. This selection is done at step iii(ii)i of the algorithm (see the different steps of the algorithm in Section 3): one of the edges with level equal to $l_{\min}(C)$ is marked as a critical edge. In the following, E_c is the set of critical edges of G and "@" denotes the operator of expression concatenation.

Loop_Nest_Generation(G)

Initialization:
For $i = 1$ **to** $|V|$ **do** $\text{rhs}(i) \leftarrow$ "1"

Computation of the statements of L':
For each $e = (v_i, v_j) \in E$ **do**
if $l(e) = \infty$ **then** $\text{rhs}(j) \leftarrow \text{rhs}(j)$ @ "$+a_i[I_1, \dots, I_d]$"
if $l(e) < \infty$ and $e \notin E_c$ **then** $\text{rhs}(j) \leftarrow \text{rhs}(j)$ @ "$+a_i[I_1, \dots, I_{l(e)-1}, I_{l(e)} - 1, I_{l(e)+1}, \dots, I_d]$"
if $e \in E_c$ **then** $\text{rhs}(j) \leftarrow \text{rhs}(j)$ @ "$+a_i[I_1, \dots, I_{l(e)-1}, I_{l(e)} - 1, \underbrace{N, \dots, N}_{d-l(e)}]$"

Code generation for L':
For $i = 1$ **to** d **do** generate ("**For** $I_i = 1$ **to** N **do**")
For $i = 1$ **to** $|V|$ **do** generate ("$a_i[I_1, \dots, I_d] :=$" @ $\text{rhs}(i)$)

Lemma 2 *The reduced leveled dependence graph of L' is G.*

Proof Denote by G' the reduced leveled dependence graph associated to L'. Note that, in L', there is only one *write* for each index vector I and each array a_i: this *write* occurs in statement T_i, at iteration I. Therefore, the dependences in L' that involves array a_i correspond to a dependence between this unique *write* and some *read* on this array. Each *read* on array a_i in the right-hand side of a statement corresponds, by construction of L', to one particular edge e in the graph G. Therefore, G and G' have the same vertices and the same edges. It remains to check that the level of all edges is the same in G and G', which is obvious. ■

Back to Example 1

The reduced leveled dependence graph of Example 1 is drawn in Figure 3. It has a single edge e of level 1. Thus, e is marked as critical. e generates a read $\text{a}(i-1, N)$ in L'. When e is deleted from the RLDG, the new graph contains a single edge e' of level 2. This edge is also selected as critical. e' generates a read $\text{a}(i, j-1)$. Finally, the apparent loops generated for Example 1 are the loops of Figure 11, as promised in Section 2.2.2.

Back to Example 3

The reduced leveled dependence graph of Example 3 is drawn in Figure 8. It has two strongly connected components, the first one with no edge. The component $\mathcal{V}_2 = \{S_2, S_3\}$ is of level 1, with a single edge of level 1, the edge e from S_3 to S_2. Thus, e is selected as critical. It generates a read c$(i-1, N)$ in the right hand side of statement T_2. When e is removed, $\mathcal{V}_2$ is broken into two strongly connected components, one with S_2 and one with S_3. The self-dependence e' for S_2 of level 2 is also selected as critical, as the only remaining edge. e' generates a read b$(i, j-1)$ in the right-hand side of T_2. Finally, the apparent loops for Example 3 are the loops of Figure 12. The reader can check that the EDG of the apparent loops contains a path of S_1-length $\Omega(1)$, a path of S_2-length $\Omega(N^2)$ and a path of S_3-length $\Omega(N^2)$, thus contains the same amount of parallelism than found by ALLEN-KENNEDY.

```
for i=1 to N
    for j=1 to N
        a(i, j) = 1 + a(i, j-1) + a(i-1, N)
    endfor
endfor
```

Figure 11: Apparent loops for Example 1

```
for i=1 to N
    for j=1 to N
        a(i, j) = 1
        b(i, j) = 1 + b(i, j-1) + a(i, j) + c(i-1, N)
        c(i, j) = 1 + b(i, j) + a(i, j)
    endfor
endfor
```

Figure 12: Apparent loops for Example 3

We denote by d_S the number of calls to ALLEN-KENNEDY (the initial call excluded) that concern a statement S, i.e. the number of calls ALLEN-KENNEDY(H, k) such that S is a vertex of H. Since one sequential loop is generated by such calls, d_S is also the number of sequential loops that surround S in the parallelized code, and as shown by Lemma 1, d_S is the S-degree of sequentiality extraction (see Definition 3) for ALLEN-KENNEDY. In the appendix, we prove the following result:

Theorem 1 *Let L be a set of loops whose RLDG is G. Use Algorithm Loop_Nest_Generation to generate the apparent loops L'. Then, for each strongly connected component G_i of G, there is a path in the EDG of the apparent loops L' which visits, $\Omega(N^{d_s})$ times, each statement S in G_i.*

The proof is long, technical, and painful. It can be omitted at first reading. The important corollary is the following:

Corollary 1 *Allen and Kennedy's algorithm is optimal for parallelism extraction in reduced leveled dependence graphs (optimal in the sense of Definition 5).*

Proof Let G be a RLDG defined from n perfectly nested loops L. $n - d_S$ is the S-degree of parallelism extraction in G. Furthermore, Algorithm *Loop_Nest_Generation* generates a set of d perfectly nested loops L', whose RLDG is exactly G (Lemma 2) and such that, for each strongly connected component G_i of G, there is a path in the EDG associated to L', which visits, $\Omega(N^{d_S})$ times, each statement S in G_i (Theorem 1). If $d = n$, the corollary is proved.

It may be possible however that $d < n$. In this case, in order to define n apparent loops L'' instead of d apparent loops L', simply add $(n - d)$ innermost loops in L' and complete all array references with $[I_{d+1}, \ldots, I_n]$. This does not change the RLDG since, in L'', there is no dependence in the innermost loops, except possibly loop independent dependences. Actually, the $(n - d)$ innermost loops are parallel loops and the path defined by Theorem 1 in the EDG of L' can be immediately translated into a path of same structure in the EDG of L'', simply by considering the $n - d$ last values of the iteration vectors as fixed (the EDG of L' is the projection of the EDG of L'' along the last $(n - d)$ dimensions). The result follows. ■

This proves that as long as the only information available is the RDG, it is not possible to detect more parallelism that found by Allen and Kennedy's algorithm. Is it possible to detect more parallelism if the structure of the code, i.e. the way loops are nested (but not the loop bounds), are given? The answer is no: it is possible to enforce L' to have the same nesting structure as L. The procedure is similar to Procedure *Loop_Nest_Generation*, but with the following modification:

- The left-hand side of statement S_i is $a_i(I)$ where I is the iteration vector corresponding to the loops that surround S_i in the initial code L. Thus, the dimension of the array associated to a statement is equal to the number of surrounding loops.

- The right-hand side of statement S_i is defined as in Procedure *Loop_Nest_Generation*, except that iteration vectors are completed by values equal to N if needed.

A theorem that generalizes Theorem 1 to the non perfectly nested case can be given, with a similar proof. We do not want to go into the details, the perfect case is painful enough. We just illustrate the non perfect case by the following example:

Example 4

Consider the non perfectly nested loops of Figure 13 (this is the program called "petersen.t" given with the software Petit, see [18], obtained after scalar expansion).

This code has exactly the same RLDG as Example 3. Furthermore, it is well known that with information on direction vectors for example, the S_1-degree, S_2-degree, and S_3-degree of parallelism are respectively 1, 1 and 0. The code can be parallelized with one parallel loop of size $\Theta(N)$ around S_1, one sequential loop of size $\Theta(N)$ around S_3 and two parallel loops of size $\Theta(N)$ around S_2, one sequential and one parallel. However, with Allen and Kennedy's algorithm, the S_1-degree, S_2-degree, S_3-degree of parallelism extraction are respectively 1, 0 and 0. This is because it is not possible, with only the RDG and the structure of the code, to distinguish between the code in Figure 13 and the apparent code in Figure 14, for which the S_1-degree, S_2-degree, S_3-degree of of parallelism are respectively 1, 0 and 0.

```
for i=2 to N
        s(i) = 0
        for l=1 to i-1
                s(i) = s(i) + a(l, i) × b(l)
        endfor
        b(i) = b(i) - s(i)
endfor
```

Figure 13: Code for Example 4

```
for i=1 to N
        a(i) = 1
        for j=1 to N
                b(i, j) = 1 + b(i, j-1) + a(i) + c(i-1)
        endfor
        c(i) = 1 + a(i) + b(i, N)
endfor
```

Figure 14: Apparent loops for Example 4

5 Conclusion

We have introduced a theoretical framework in which the optimality of algorithms that detect parallelism in nested loops can be discussed. We have formalized the notions of degree of parallelism extraction (with respect to a given dependence abstraction) and of degree of parallelism (contained in a reduced dependence graph). This study explains the impact of a given dependence abstraction on the maximal parallelism that can be detected: it determines whether the limitations of a parallelization algorithm are due to the algorithm itself or are due to the weaknesses of the dependence abstraction.

In this framework, we have studied more precisely the link between dependence abstractions and parallelism extraction in the particular case of *dependence level*. Our main result is the optimality of Allen and Kennedy's algorithm for parallelism extraction in reduced leveled dependence graphs. This means that even the most sophisticated algorithm cannot detect more parallelism, as long as dependence level is the only information available. In other words, loop distribution is sufficient for detecting maximal parallelism in dependence graphs with levels.

The proof is based on the following fact: given a set of loops L whose dependences are specified by level, we are able to systematically build a set of loops L' that cannot be distinguished from L (i.e. they have the same reduced dependence graph) and that contains exactly the degree of parallelism found by Allen and Kennedy's algorithm. We call these loops the apparent loops. We believe this construction of interest since it better explains why some loops appear sequential when considering the reduced dependence graph while they actually may contain some parallelism. This is mainly a theoretical result, that gives an upper bound on the parallelism that can be detected.

A Appendix: proof of the optimality theorem

In this section, we denote by L the initial loops and by G the reduced leveled dependence graph associated to L. We denote by L' the "apparent loops" generated by Procedure *Loop_Nest_Generation* applied to G. We denote by d_S (see Section 4) the number of sequential loops detected by Allen and Kennedy's algorithm, which surround statement S, when processing G. We show that for each statement S in L', there is a dependence path in the expanded dependence graph of L' that contains $\Omega(N^{d_S})$ instances of the statement S. More precisely, we build a dependence path that satisfies this property simultaneously for all statements of a strongly connected component of G. This path is built by induction on the depth (to be defined) of a strongly connected component of G. In Section A.3, we study the initialization of the induction, whose general case is studied in Section A.4. The proof by induction itself and the optimality theorem are presented in Section A.5.

To make the proof clearer, we give in Section A.2 a schematic description of the induction. Before that, we need to introduce some new definitions, which is done in Section A.1.

Due to space limitation, this appendix contains only the intermediate results needed to prove the whole theorem, and the proof of one of these results. The other proofs can be found in [8].

A.1 Some more definitions

We extend the notion of **depth** to graphs. Remember that we defined the depth d_S of a statement S in Section 4 as the number of calls to ALLEN-KENNEDY, the initial code excluded, that concern statement S when processing the graph G.

Let H be a subgraph of G that contains S: we define similarly $d_S(H)$ as the number of calls to ALLEN-KENNEDY that concern statement S when processing the graph H (instead of G). Note that $d_S = d_S(G)$. Finally, we define $d(H)$, the depth of H, as the maximal depth of the vertices (i.e. statements) of H:

Definition 6 *The* **depth** *of a reduced dependence graph H is:*

$$d(H) = \max\{d_v(H) \mid v \in V(H)\}$$

where $V(H)$ denotes the vertices of H.

The proof of the theorem is based on an induction on the depths of the strongly connected components of G that are built in ALLEN-KENNEDY.

Given a path P in a graph, $P = x_1 \xrightarrow{e_1} x_2 \xrightarrow{e_2} \dots \xrightarrow{e_{k-1}} x_k$, we define the tail of P as the first statement visited by P, denoted by $t(P) = x_1$, and we define the head of P as the last statement visited by P, denoted by $h(P) = x_k$. We define the weight of a path as follows:

Definition 7 *Let P be a path of a reduced leveled dependence graph G, for which $l(e)$ denotes the level of an edge e. We define the* **weight** *of P at level l, denoted $w_l(P)$, as the number of edges in P whose level is equal to l: $w_l(P) = |\{e \in P \mid l(e) = l\}|$.*

A.2 Induction proof overview

In the following, H denotes a subgraph of G which appears in the decomposition of G by ALLEN-KENNEDY. The induction is an induction on the depth of H.

We want to prove by induction that if S is a statement of H, there is in the EDG of L' a dependence path whose S-length is $\Omega(N^{d_S(H)-1})$. From this result, we will be able to build the desired path.

First of all, we prove in Theorem 2 that if $d(H) = 1$ there exists in the EDG of L' a path which visits all statements of H, whose tail and head correspond to the same statement, for two different iterations, and whose starting iteration vector can be fixed to any value in a certain sub-space of the iteration space. We write that this path is a "cycle", as it corresponds to a cycle in the RDG.

Then, we prove that whatever the depth of H, this property still holds: there exists in the EDG of L' a path which visits all statements of H, whose tail and head correspond to the same statement, and whose starting iteration vector can be fixed to any value in a certain sub-space of the iteration space. Furthermore, we can connect on this "cycle" the different "cycles" built for the subgraphs $H_1, \ldots, H_c$ of H which appear at the first step of the decomposition of H by ALLEN-KENNEDY. Each of these "cycles" can be connected a number of times linear in N, the domain size parameter. This leads to the desired property, which is proved in Theorem 3.

Remark that the subgraphs $H_1, \ldots, H_c$ have depths strictly smaller than the depth of H, this is why the induction is made on the depths of the graphs. Furthermore, all subgraphs H_i are strongly connected by construction. As a consequence, their level is an integer, i.e $l(H_i) \neq \infty$.

A.3 Initialization of the induction: $d(H) = 1$

The initial case of the induction is divided in three parts: in Section A.3.1 we state the hypotheses and notations, in Section A.3.2 we prove some intermediate results and in Section A.3.3 we build the desired path from the results that we previously established.

A.3.1 Data, hypotheses and notations

In this subsection, we recall or define the data, hypotheses and notations which will be valid throughout Section A.3. In particular, Property 1, Lemma 3 and Theorem 2 are proved under the hypotheses listed below.

- H is a subgraph of G which appears in the decomposition of G by ALLEN-KENNEDY. We suppose that H is of depth one, $d(H) = 1$.
- l is the level of H: $l = l(H)$.
- f is the critical edge of H.
- $C(H)$ is an arbitrarily chosen cycle of H which visits all statements in H and includes f.
- $C(H)$ is split up into $C(H) = f, P_1, f, P_2, f, \ldots, f, P_{k-1}, f, P_k$, where, for all i, $1 \leq i \leq k$, the path P_i does not contain the edge f (decomposition shown on Figure 15). Remark that the first statement of P_i (i.e. $t(P_i)$) is the head of edge f (i.e. $h(f)$), and the last statement of P_i (i.e. $h(P_i)$) is the tail of edge f (i.e. $t(f)$), as $C(H)$ is a cycle.
- ξ is an integer greater than the maximal length of all cycles $C(H)$.

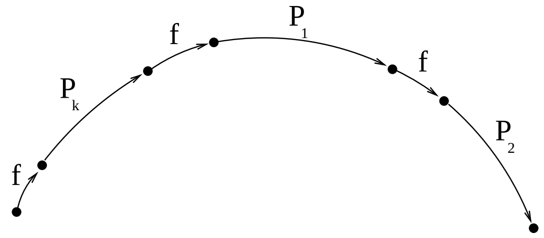

Figure 15: The decomposition of $C(H)$

- As all iteration vectors are of size d, if I is a vector of size $l'-1$, for $1 \leq l' \leq d$, and if j is an integer, then $(I, j, N, \dots, N)$ denotes the iteration vector $(I, j, \underbrace{N, \dots, N}_{d-l'})$.

A.3.2 A few bricks for the wall (part I)

We first prove in Property 1 that there is no critical edge in a path P_i. This property is used to build in Lemma 3 a path in the EDG of L' whose tail and head correspond in G to the same statement, and whose projection in the RLDG of L' is exactly the path $f + P_i$. The existence of such a path is conditioned by the value of the iteration vector associated to the starting statement. The result is used in Section A.3.3 in order to prove Theorem 2.

Property 1 *Let i be an integer in $[1 \dots k]$. Then, P_i contains no critical edge.*

Lemma 3 *Let i be an integer in $[1 \dots k]$, I a vector in $[1 \dots N]^{l-1}$, j an integer in $[1 \dots N - (1 + w_l(P_i))]$, and K a vector in $[\xi \dots N]^{d-l}$. Then, there exists, in the EDG of L', a dependence path from iteration $(I, j, N, \dots, N)$ of statement $t(f)$ to iteration $(I, j + 1 + w_l(P_i), K)$ of the same statement, path which corresponds in H to the use of f followed by all edges of P_i,*

A.3.3 Conclusion for the initialization case

In the next corollary (Corollary 2), we simply concatenate the paths built in Lemma 3 for the different paths P_i. This builds a path in the EDG of L' whose projection in the RLDG of L' is $C(H)$. For the sake of regularity, we turn $|V|$ times on this path as this gives us, for the subgraphs H of depth 1, exactly the result that is proved on subgraphs of greater depths (Theorem 3). We have then the path announced in the proof overview (Section A.2).

Corollary 2 *Let I be a vector in $[1 \dots N]^{l-1}$, j an integer in $[1 \dots N - |V| w_l(C(H))]$, and K a vector in $[\xi, \dots, N]^{d-l}$. Then, there exists in the expanded dependence graph of L' a dependence path Φ which visits all nodes of H, which starts at instance $(I, j, N, \dots, N)$ of statement $t(f)$, and which ends at instance $(I, j + |V| w_l(C(H)), K)$ of the same statement.*

The result for the initialization case of the induction established, we can study the general case.

A.4 General case of the induction

We first formulate formally the induction hypothesis in Section A.4.1. In Section A.4.2, we give some definitions. In Section A.4.3, we prove some intermediate results. Finally, in Section A.4.4, we build the desired path from the previously established results.

A.4.1 Induction hypothesis

We present in this section the induction hypothesis used in the induction proof in Section A.5. The induction hypothesis requires quite complicated conditions mainly for technical reasons. The induction hypothesis $\mathcal{IH}(k)$ is parameterized by the depth k of the subgraphs H. Schematically, $\mathcal{IH}(k)$ is true if, for all subgraphs H that appear during the processing of G, there exists a dependence path in the EDG of L' which visits each statement S of H $\Omega(N^{d_S(H)-1})$ times.

Induction hypothesis at depth k: $\mathcal{IH}(k)$

We suppose that N is greater than $(d+1)|V|\xi$. The **induction hypothesis $\mathcal{IH}(k)$** is true, if and only if, for all subgraphs H of G strongly connected with at least one edge (H is a subgraph of G that appears during the processing of G) whose depth is strictly less than k ($d(H) \leq k$), there exists a path Φ, in the EDG of L', from instance $(I, j, N, \ldots, N)$ of statement $t(C(H))$ to instance $(I, j + |V|w_{l(H)}(C(H)), K)$ of the same statement, $\forall\ I \in [1 \ldots N]^{l(H)-1}$, $\forall\ j \in [1 \ldots N - |V|w_{l(H)}(C(H))]$, $\forall\ K \in [N - (d+1-d(H))|V|\xi \ldots N]^{d-l(H)}$ and such that for each statement S in H the S-length of Φ is $\Omega(N^{d_S(H)-1})$.

Before going further, we give here some remarks on the importance of the different hypotheses, conditions and values on the components of the index vector of the starting and ending statements of the built paths:

- The $l(H)-1$ first components of the index vectors are constant along this dependence path. This simplify our work when we connect this cycle with a cycle built for the subgraph of smaller depth: we will just have to look at the $d - l(H) + 1$ last components.
- The $l(H)$-th component is increased by a constant factor, namely $|V|w_{l(H)}(C(H))$. In particular this factor is independent of N. Joined to the freedom we have for the value of the $d-l$ last components of the ending statement index vector (variable K), this will allow us to connect consecutively $\Omega(N)$ times the cycles built for smaller depths.

The subgraphs of depth 1 do satisfy this induction hypothesis:

Theorem 2 *$\mathcal{IH}(1)$ is true.*

A.4.2 Data, hypotheses and notations

We recall or define in this subsection the data, hypotheses and notations which will be valid all along Section A.4. In particular, Property 2, Lemmas 4 and 5, and Theorem 3 are proved under the hypotheses listed here.

- H is strongly connected with at least one edge as it appears during the processing of G.
- l is the level of H: $l = l(H)$.
- We suppose $\mathcal{IH}(k)$ true for $k < l$: the goal of this section is to show that $\mathcal{IH}(l)$ is true.
- $H_1, \ldots, H_c$ are the c subgraphs of H on which ALLEN-KENNEDY is recursively called. In particular, H_i satisfies $\mathcal{IH}(d(H_i))$.
- Φ_i, $1 \leq i \leq c$, denotes the dependence path defined for H_i by $\mathcal{IH}(d(H_i))$.
- f is the critical edge of H.
- $C(H)$ is an arbitrarily chosen cycle of H which visits all statements in H and includes f.
- ξ is an integer greater than the maximal length of all cycles $C(H)$.
- The cycle $C(H)$ can be decomposed into: $C(H) = f, P'_1, f, P'_2, f, \ldots, f, P'_{k-1}, f, P'_k$, where, for i, $1 \leq i \leq k$, the path P'_i does not contain the edge f. Remark that, since $C(H)$ is a cycle, the first statement in the path P'_i (i.e. $t(P'_i)$) is the head of the edge f (i.e. $h(f)$), and that the last statement in P'_i (i.e. $h(P'_i)$) is the tail of edge f (i.e. $t(f)$).
- C' is the concatenation of $|V|$ times the cycle $C(H)$. C' is naturally decomposed into: $C' = f, P_1, f, P_2, f, \ldots, f, P_{|V|k-1}, f, P_{|V|k}$, where $P_i = P'_{(i \bmod k)}$.
- As the number of strongly connected components (which is at least c) in H (subgraph of G) is smaller than the number of vertices ($|V|$) in G, we have $c \leq |V|$. As C' contains $|V|$ occurrences of each path P'_i, for $1 \leq i \leq k$, we can define $s_1, \ldots, s_c$ as occurrences of $t(C(H_1))$, $\ldots$, $t(C(H_c))$ in C' such that each path P_i (for $1 \leq i \leq |V|k$) contains at most one of the s_m ($1 \leq m \leq |V|$).
- If P is a path, and p and q are two vertices of P, then $p \xrightarrow{P} q$ denotes the sub-path of P which starts at vertex p and which ends at vertex q.
- If p is a vertex of a path P of H, we denote by $\delta(P, p, l')$ the first vertex of $p \xrightarrow{P} h(P)$ followed by a critical edge whose level is strictly less than l'.

A.4.3 A few bricks for the wall (part II)

We prove in Property 2 that a path P of H which does not contain the critical edge f of H cannot contain any critical edge of level less than or equal to l. Then we prove in Lemma 4 that for any such path P of length smaller than ξ, we can build in the EDG of L' a path whose projection on H is P. Moreover, we express exactly the instances of the statements visited by P.

Property 2 *Let P be a path of H which does not contain the edge f. The critical edges contained in P are of levels greater than or equal to $l + 1$.*

Lemma 4 *Let P be a path of H which does not contain the edge f and whose length is strictly less than ξ. Let I be a vector in $[1 \dots N]^{l-1}$, j an integer in $[1 \dots N]$, and K a vector in $[\xi \dots N]^{d-l}$. Then, there exists a dependence path $\mathcal{P}$ in the EDG of L' whose projection onto H is equal to P, and which ends at instance $(I, j + w_l(P), K)$ if $1 \leq j + w_l(P) \leq N$. Furthermore, the path $\mathcal{P}$ visits the statement that corresponds to a vertex p of P at iteration $\left(I, j + w_l(P) - w_l\left(p \xrightarrow{P} h(P)\right), K'\right)$ where, for l' in $[l{+}1 \dots d$, $K'_{l'-l} = N - w_{l'}\left(p \xrightarrow{P} \delta(P, p, l')\right)$ if $\delta(P, p, l')$ exists and $K'_{l'-l} = K_{l'-l} - w_{l'}\left(p \xrightarrow{P} h(P)\right)$ otherwise.*

In Lemma 5 we apply Lemma 4 to the paths P_i, so as to build a path corresponding to $f + P_i$ in the EDG of L'. There are two main differences between Lemmas 3 and 5. The first one is the existence in the paths P_i of critical edges, which requires Lemma 4. The second one corresponds to the cycles Φ_m: because of the desired property on the number of occurrences of each statement in the built path, if P_i contains the vertex s_m, we have to turn $\Omega(N)$ times around Φ_m while building the path corresponding to P_i. This construction is illustrated on Figure 16.

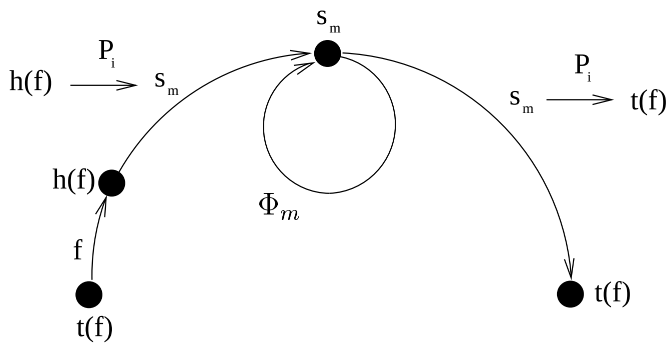

Figure 16: As P_i contains the vertex s_m, we turn round Φ_m

Lemma 5 *Let i be an integer in $[1 \dots |V|k]$, I a vector in $[1 \dots N]^{l-1}$, j an integer in $[1 \dots N - (1 + w_l(P_i))]$, and K a vector in $[N - (d{+}1 - d(H))|V|\xi \dots N]^{d-l}$. Then, there exists in the EDG of L' a dependence path $\mathcal{P}_i$ which goes along f and all edges of P_i, which starts at the instance $(I, j, N, \dots, N)$ of statement $t(f)$ and which ends at instance $(I, j + 1 + w_l(P_i), K)$ of the same statement. Furthermore, if s_m is a vertex of P_i, for m in $[1 \dots c]$, then this dependence path visits $\Omega(N^{d_S(H)-1})$ times each statement S of H_m.*

Proof: Consider a vector I, an integer j and a vector K according to the theorem hypotheses.

Two cases can occur, depending if one of the vertices of P_i is s_m for some m, $1 \leq m \leq c$. First of all, note that:

i. By definition, P_i does not contain the edge f.

ii. P_i is strictly shorter than $C(H)$ whose length is smaller than ξ. Thus the length of P_i is strictly less than ξ.

iii. By hypothesis, $N \geq (d+1)|V|\xi \geq (d+2-d(H))|V|\xi$ (since $d(H) \geq 1$). Therefore, the $(d-l)$-dimensional set $[N-(d+1-d(H))|V|\xi \ldots N]^{d-l}$ is a subset of $[|V|\xi \ldots N]^{d-l}$, and thus of $[\xi \ldots N]^{d-l}$.

These two properties permit to apply Lemma 4 to sub-paths of P_i in the two following cases:

- We suppose that no vertex s_m belongs to P_i.

  Because of the preceding remarks, we can apply Lemma 4 to P_i. This gives a path from instance $(I, j+1, K')$ of statement $t(P_i) = h(f)$ to instance $(I, j+1+w_l(P_i), K)$ of statement $h(P_i) = t(f)$, where K' is defined by Lemma 4. Then we concatenate, in front of this path, the edge corresponding to f from instance $(I, j, N, \ldots, N)$ of $t(f)$ to instance $(I, j+1, K')$ of statement $h(f)$. This leads to the desired result.

- We now suppose that s_m is a vertex of P_i for some m, $1 \leq m \leq c$. By construction, if $m' \neq m$, $s_{m'}$ is not a vertex of P_i (see Section A.4.2). We build the path backwards, starting from the ending vertex of P_i.

  The previous remarks allow us to apply Lemma 4 to the sub-path of P_i: $s_m \xrightarrow{P_i} h(P_i)$. This gives a path from instance $(I, j+1+w_l(P_i) - w_l\left(s_m \xrightarrow{P_i} h(P_i)\right), K')$ of statement s_m to instance $(I, j+1+w_l(P_i), K)$ of statement $h(P_i) = t(f)$, where K' is defined by Lemma 4. Remark that $w_l(P_i) - w_l\left(s_m \xrightarrow{P_i} h(P_i)\right) = w_l\left(t(P_i) \xrightarrow{P_i} s_m\right)$. We call this first path $\mathcal{P}_1$.

  We want, starting from this instance of s_m, to turn (backwards) $\Omega(N)$ times round the cycle Φ_m. To do so, we use the induction hypothesis $\mathcal{IH}(d(H_m))$ for H_m. We have first to check that the path described in $\mathcal{IH}(d(H_m))$ can be inserted in front of instance $(I, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right), K')$ of s_m. In other words, we have to check that the $d - l(H_m) + 1$ last components of $(I, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right), K')$ satisfy the hypothesis stated in $\mathcal{IH}(d(H_m))$.

  Since, $l(H_m) > l$, the components that have to be considered are the components of K'. We thus have to check that:

  - $K'_{l'-l} > N - (d+1-d(H_m))|V|\xi$ for $l' > l(H_m)$.
  - $1 + |V|w_{l(H_m)}(C(H_m)) \leq K'_{l(H_m)-l} \leq N$.

  For that, consider the $l'-l$-th component of K': once again, we have two cases to consider:

  - $\delta(P_i, s_m, l')$ exists. Then: $K'_{l'-l} = N - w_{l'}\left(s_m \xrightarrow{P_i} \delta(P_i, s_m, l')\right)$ and $s_m \xrightarrow{P_i} \delta(P_i, s_m, l')$ is a sub-path of P_i whose length is strictly less than ξ. Thus, $w_{l'}\left(s_m \xrightarrow{P_i} \delta(P_i, s_m, l')\right) < \xi$, and $K'_{l'-l} > N - \xi$. Now, since $d+1-d(H_m) \geq 1$, $N - \xi \geq N - (d+1-d(H_m))|V|\xi$. This proves $K'_{l'-l} > N - (d+1-d(H_m))|V|\xi$.
  - $\delta(P_i, s_m, l')$ does not exist. Then, $K'_{l'-l} = K_{l'-l} - w_{l'}\left(s_m \xrightarrow{P_i} h(P_i)\right)$. $s_m \xrightarrow{P_i} h(P_i)$ is a sub-path of P_i whose length is strictly less than ξ. Thus, $w_{l'}\left(s_m \xrightarrow{P_i} h(P_i)\right) < \xi$, and $K'_{l'-l} > K_{l'-l} - \xi > K_{l'-l} - |V|\xi$. Now, since $K_{l'-l} \geq N - (d+1-d(H))|V|\xi$ by hypothesis, we get $K'_{l'-l} > N - (d+2-d(H))|V|\xi = N - (d+1-d(H_m))|V|\xi$.

This proves the first inequality. The second is implied by the first one since $N \geq (d+1)|V|\xi$.

We can thus apply the induction hypothesis. We get a dependence path from instance $(I,\, j+1+ w_l\left(t(P_i) \xrightarrow{P_i} s_m\right),\, K'_1,\, \dots,\, K'_{l(H_m)-l-1},\, K'_{l(H_m)-l} - |V|w_{l(H_m)}(C(H_m)),\, N,\, \dots,\, N)$ of statement s_m to instance $(I, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right), K')$ of the same statement. We call this second path $\mathcal{P}_2$.

We now apply the induction hypothesis, in the particular case where all components of K (with the notations of $\mathcal{IH}(d(H_m))$) are equal to N and we get a dependence path from instance $(J, k'' - |V|w_{l(H_m)}(C(H_m)), N, \dots, N)$ of statement s_m, to instance $(J, k'', N, \dots, N)$ of the same statement, if J is a vector of $[1, \dots, N]^{l(H_m)-1}$, and if k'' is an integer such that $1 + |V|w_{l(H_m)}(C(H_m)) \leq k'' \leq N$. Furthermore, this path visits, $\Omega(N^{d_S(H_m)-1})$ times, each statement S in H_m.

We let $\lambda_m = \left\lfloor \frac{K'_{l(H_m)-l} - \xi}{|V|w_{l(H_m)}(C(H_m))} - 1 \right\rfloor$. λ_m is chosen so that we can use λ_m times the induction hypothesis in the form stated above. As $K'_{l(H_m)-l}$ has been proved to be $\Omega(N)$, and as all other quantities are constant that depend only on G and not on N, λ_m is $\Omega(N)$ too. Therefore, we get a dependence path, from instance $(I,\, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right),\, K'_1,\, \dots,\, K'_{l(H_m)-1-l},\, K'_{l(H_m)-l} - (\lambda_m+1)|V|w_{l(H_m)}(C(H_m)),\, N,\, \dots,\, N)$ to instance $(I,\, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right),\, K'_1,\, \dots,\, K'_{l(H_m)-1-l},\, K'_{l(H_m)-l} - |V|w_{l(H_m)}(C(H_m)),\, N,\, \dots,\, N)$ of the same statement s_m. We call this path $\mathcal{P}_3$. By construction, $\mathcal{P}_3$ visits each statement S in H_m $\lambda_m\Omega(N^{d_S(H_m)-1})$ times, i.e. $\Omega(N^{d_S(H_m)}) = \Omega(N^{d_S(H)-1})$ times.

Note that by choice of λ_m, $K'_{l(H_m)-l} - (\lambda_m+1)|V|w_{l(H_m)}(C(H_m)) \geq \xi$ and this also holds for all other components of K'. We can thus apply once again Lemma 4, and we get a path, that we call $\mathcal{P}_4$, from instance $(I, j+1, K'')$ of statement $h(f)$ to instance $(I,\, j+1+w_l\left(t(P_i) \xrightarrow{P_i} s_m\right),\, K'_1,\, \dots,\, K'_{l(H_m)-1-l},\, K'_{l(H_m)-l} - (\lambda_m+1)|V|w_{l(H_m)}(C(H_m)),\, N,\, \dots,\, N)$ of statement s_m, where K'' is defined by Lemma 4.

Then, we concatenate in front of this path the dependence corresponding to the edge f from instance $(I, j, N, \dots, N)$ of statement $t(f)$ to instance $(I, j+1, K'')$ of statement $h(f)$.

Finally, concatenating paths $f + \mathcal{P}_4$, $\mathcal{P}_3$, $\mathcal{P}_2$ and $\mathcal{P}_1$ leads to the desired path.

■

A.4.4 Conclusion for the general case

The previous lemma leads to the desired theorem (i.e. $\mathcal{IH}(l)$ where $l = l(H)$) simply by concatenating the different paths given by Lemma 5 for each P_i.

Theorem 3 *If $\mathcal{IH}(k)$ is true, for $k < l$, then $\mathcal{IH}(l)$ is true. In other words, we have:*

Let I be a vector in $[1 \dots N]^{l-1}$, j an integer in $[1 \dots N - |V|w_l(C(H))]$, and K a vector in $[N - (d+1-d(H))\xi \dots N]^{d-l}$. Then, there exists a dependence path Φ in the EDG of L', from instance $(I, j, N, \dots, N)$ of statement $t(C(H))$ to instance $(I, j+|V|w_l(C(H)), K)$ of the same statement. Furthermore, Φ visits each statement v in H $\Omega(N^{d_v(H)-1})$ times.

A.5 The induction and the optimality theorem

We have almost established the proof by induction that we write below in a formal form.

Theorem 4 *$\mathcal{IH}(k)$ is true, for all $k \geq 1$. In other words, we have the following:*

Let H be a subgraph of G which appears in the decomposition of G by ALLEN-KENNEDY. *Assume that N is larger than $(d+1)|V|\xi$. Then for all I in $[1 \ldots N]^{l(H)-1}$, all j in $[1 \ldots N - |V|w_{l(H)}(C(H))]$, and all K in $[N-(d+1-d(H))\xi \ldots N]^{d-l(H)}$, there exists a dependence path Φ in the EDG of L', from instance $(I, j, N, \ldots, N)$ of statement $t(C(H))$, to instance $(I, j+|V|w_{l(H)}(C(H)), K)$ of the same statement. Furthermore, Φ visits each statement S in H $\Omega(N^{d_S(H)-1})$ times.*

This theorem is not the final result, as we want in fact a path which visits each statement S in H $\Omega(N^{d_S(H)})$ times. However, Theorem 5 is a simple extension of this result: we establish it by turning $\Theta(N)$ times on the path given by Theorem 4.

Theorem 5 *Let H be a subgraph of G which appears in the decomposition of G by* ALLEN-KENNEDY. *Assume that N is larger than $(d+1)|V|\xi$. Then there exists a dependence path Φ in the expanded dependence graph of L', which visits each statement S in H $\Omega(N^{d_S(H)})$ times.*

References

[1] J.R. Allen and K. Kennedy. PFC: a program to convert programs to parallel form. Technical report, Dept. of Math. Sciences, Rice University, TX, March 1982.

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

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, 1988.

[4] A. J. Bernstein. Analysis of programs for parallel processing. In *IEEE Trans. on El. Computers, EC-15*, 1966.

[5] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, Houston, TX, 1987.

[6] Alain Darte and Frédéric Vivien. A classification of nested loops parallelization algorithms. In *INRIA-IEEE Symposium on Emerging Technologies and Factory Automation*, pages 217–224. IEEE Computer Society Press, 1995.

[7] Alain Darte and Frédéric Vivien. Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.

[8] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. Technical Report 96-05, LIP, ENS-Lyon, France, February 1996. Extended version.

[9] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism in polyhedral reduced dependence graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press. To appear.

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

[11] 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.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992.

[13] G. Goff, K. Kennedy, and C.W. Tseng. Practical dependence testing. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[14] D. Grunwald. Data dependence analysis: the λ test revisited. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[16] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[17] 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, July 1967.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New user interface for Petit and other interfaces: user guide.* University of Maryland, June 1995.

[19] X.Y. Kong, D. Klappholz, and K. Psarris. The I test: a new test for subscript data dependence. In Padua, editor, *Proceedings of 1990 International Conference of Parallel Processing*, August 1990.

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

[21] Z.Y. Li, P.-C. Yew, and C.Q. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, pages 215–224, Crete, Greece, June 1989.

[22] D.I. Moldovan. On the analysis and synthesis of vlsi systolic arrays. *IEEE Transactions on Computers*, 31:1121–1126, 1982.

[23] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971.

[24] K. Psarris, X.Y. Kong, and D. Klappholz. Extending the I test to direction vectors. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

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

[26] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *The 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, June 1984. IEEE Computer Society Press.

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

[28] Vwani P. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms*. PhD thesis, Stanford University, December 1988.

[29] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

[30] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[31] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge MA, 1989.

[32] Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23(4):359–388, August 1995.

[33] Yi-Qing Yang. *Tests des dépendances et transformations de programme*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, Fontainebleau, France, 1993.

[34] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.