UNIVERSITÄT PASSAU

Practical Methods for Scheduling and Allocation in the Polytope Model

Diplomarbeit

Aufgabensteller: Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung

Betreuer: Martin Griebl

Bearbeiter: Wolfgang Meisl

27. September 1996

Abstract

This thesis deals with the automatic parallelization step of compilers for multi-processor computers. The focus is on practical methods for this task, which means methods that are general and fast. Thus, a wide variety of loop programs is accepted as input, including imperfectly nested loops with affine variable accesses and dependences. The scheduler assigns time steps to the operations (which are the instances of statements for the surrounding loops). The allocator places the operations onto a (virtual) processor grid. Both, the scheduler and the allocator produce multi-dimensional affine functions as output and both are based upon a special graph: the scheduler upon the polyhedral reduced dependence graph and the allocator upon the access graph.

Acknowledgements

I would like to thank the following people:
first of all, my tutor Martin Griebl for his substantial support. He always had time for my problems, gave many valuable suggestions and read this thesis more than once.
Prof. Christian Lengauer for proofreading this thesis, although he was very busy at that time. He gave detailed suggestions and improved the English of this thesis.
The current and former members of the LooPo team for many valuable suggestions and insights: Andreas Dischinger, Nils Ellmenreich, Peter Faber, Max Geigl, Robert Günz, Harald Keimer, Radko Kubias, Martina Schumergruber, Sabine Wetzel, Christian Wieninger and Alexander Wüst.
Finally, I want to thank Monika and Matthias Müller, who greatly improved my English (but who still doubt whether the language used in this thesis is really English).

Without all their help this thesis would not be what it is.

Contents

1 Introduction

The development of computers with multiple processors brought about a big problem: how do we obtain software that runs on several processors in parallel. First, people tried to find ad-hoc parallel algorithms for their problems. This method is much like programming sequential computers in assembler. The resulting programs are fast but programming is lengthy, the readability of the programs is poor and the risk of making mistakes is very high. Besides, the source program is specific to one special architecture. All these disadvantages apply to handmade parallel code, too. Parallelizing compilers are *the* important step towards portability and reliability.

It is known that the major part of a program's execution time is consumed inside loops or recursion. The *polytope model* has proved to be a very useful mathematical framework for automatic parallelization of loop programs. Each iteration is identified by a point in a multi-dimensional lattice. With the restriction that loop bounds are affine functions of indices of surrounding loops or of structure parameters, the index space of each statement forms a polytope.

Parallelization in this model means applying a transformation (i.e., an affine mapping) to the source polytope. The result is a target polytope with a possibly different shape embedded in a slightly different space. The number of dimensions is still the same, but we distinguish between time and space dimensions. The source polytope is spawned completely by time dimensions. After the transformation, some of them might have become space dimensions. They correspond to **forall** loops indicating detected parallelism [Len93].

The transformation must preserve dependence relations contained in the source program. Under these constraints, one tries to optimize some criteria like the latency of the target program or the communication volume.

So far, no good algorithm has been found for computing such a transformation in a single step. So two smaller subproblems have been studied: one for the time dimensions and one for the space dimensions. The projection on the time dimensions is called a *schedule*, that on the space dimensions an *allocation*. Schedule and allocation are computed separately and get merged subsequently [Wet95].

Recently, very precise methods for scheduling and allocation have been developed [Fea92a, Fea92b]. But they require a very accurate and, therefore, expensive dependence analysis. Moreover, the methods themselves are very slow, as large optimization problems have to be solved.

A compromise between optimality and processing time has to be found for a real-world compiler. One should not need a parallel supercomputer just to run a parallelizing compiler.

The methods presented in this thesis are not optimal in the sense of what can be achieved theoretically, but they produce good results relative to their less accurate input information. And — most importantly — they do this quite fast. To get an impression of the practical usability, both, the scheduler and the allocator have been implemented as part of the polyhedral loop parallelizer LooPo [Loo96, GL96].

This thesis is organized as follows: Chapter 2 introduces some notational conventions and conveys the general theoretical background used in the following chapters. Chapter 3 describes ways of representing dependences which are the basis for the scheduling method introduced in Chapter 4: the Darte-Vivien scheduler. Chapter 5 deals with an allocation according to a paper of Dion and Robert. The last chapter concludes with some remarks.

2 Notations and Prerequisites

2.1 Notations

It is assumed that the reader is familiar with the basic notions of linear algebra like vector spaces, systems of equations, matrices, etc.

As rational numbers are sufficient for the theory presented in this paper, the vector space used in most places is $\mathbb{Q}^n$.

Some type-setting and notational conventions are used in formulas throughout this thesis:

- $\mathbb{N}$ is used as symbol for the natural numbers including zero.
- Simple scalars in $\mathbb{Q}$, $\mathbb{Z}$, or $\mathbb{N}$ are denoted by lower-case letters: $x, y, z, \ldots$
- Vectors in $\mathbb{Q}^n$ or $\mathbb{Z}^n$ are denoted by boldface lower-case letters: $\mathbf{a}, \mathbf{b}, \mathbf{c}, \ldots$
- Matrices in $\mathbb{Q}^{m \times n}$ are denoted by capital letters: $A, B, C, \ldots$.
- $\mathbf{0}, \mathbf{1}$ are used as abbreviations for the vectors $(0, \ldots, 0)$ resp. $(1, \ldots, 1)$.
- A quantification of a variable x is denoted according to [DS90]:

  $$(Qx \ : \ Rx \ : \ Px)$$

  Q can be either $\forall$ or $\exists$, Rx represents the range of x and Px is the predicate over x.
- A function f that maps elements $a \in A$ to $b \in B$ is denoted by:

  $$f \ : \ A \longrightarrow B \ : \ a \longrightarrow b$$

- $|X|$ denotes the cardinality of some set X.

2.2 Vector Spaces, Matrices and Linear Functions

2.2.1 Definition

For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Q}^n$ the **scalar product** is defined as follows:

$$\mathbf{ab} \;=\; \sum_{i=0}^{n} a_i b_i$$

$\mathbf{a}, \mathbf{b}$ are said to be **orthogonal** iff $\mathbf{ab} = 0$.

The order used for $\mathbb{Q}^n$ is the **lexicographic order**. For $\mathbf{a}, \mathbf{b} \in \mathbb{Q}^n$:

$$\mathbf{a} \leq \mathbf{b} \;\Leftrightarrow\; (\exists\, l \;:\; 1 \leq l \leq n \;:\; (\forall\, j \;:\; j < l \;:\; a_i = b_i) \wedge (a_l \leq b_l))$$

The **Manhattan norm** is used as the norm for a vector $\mathbf{a} \in \mathbb{Q}^n$:

$$\|\mathbf{a}\| \;=\; \sum_{i=1}^{n} |a_i|$$

2.2.2 Definition

A matrix $A^{(l)}$ is called a **left inverse** of a matrix A if the following equation holds (I denotes the identity matrix):

$$A^{(l)}A = I$$

A matrix $A^{(r)}$ is called a **right inverse** of a matrix A if:

$$AA^{(r)} = I$$

A matrix A^{-1} is called the **inverse** of A if both equations hold:

$$AA^{-1} = I = A^{-1}A$$

2.2.3 Theorem

The following holds for inverses of an (m, n)-matrix A.

- A has an inverse iff A is square and of full rank.
- A has a right inverse iff A is flat $(m \leq n)$ and of maximum rank.
- A has a left inverse iff A is narrow $(m \geq n)$ and of maximum rank.

PROOF [Usm87, Chapter 3]

Remark

If A is invertible, its inverse is unique. This is not true for left or right inverses. There may be many matrices which satisfy the corresponding equation. [Usm87] refers to a comprehensive description of inverses.

2.2.4 Definition

The **kernel** $\ker(A)$ of a matrix $A \in \mathbb{Q}^{m\times n}$ is the following linear subspace of $\mathbb{Q}^n$:

$$\ker(A) \;=\; \{\, \mathbf{k} \;:\; \mathbf{k} \in \mathbb{Q}^n \;:\; A\mathbf{k} = \mathbf{0} \,\}$$

2.2.5 Definition

Whenever we refer to an **affine mapping** in this thesis, we have the following special definition in mind. We allow parameterized functions, but in a restricted way: parameters cannot appear as coefficients to variables or other parameters. For variable vector $\mathbf{i} \in \mathbb{Q}^n$, parameter vector $\mathbf{p} \in \mathbb{N}^m$ and coefficients $\mathbf{a} \in \mathbb{Q}^n$, $\mathbf{b} \in \mathbb{Q}^m$ and $c \in \mathbb{Q}$ we get for a one-dimensional function f:

$$f \;:\; \mathbb{Q}^n \longrightarrow \mathbb{Q} \;:\; (\mathbf{i};\ \mathbf{p}) \longrightarrow \mathbf{ai} + \mathbf{bp} + c$$

With scalars rather than vectors the function definition looks as follows:

$$f(i_1,\ldots,i_n;\ p_1,\ldots,p_m) = \sum_{k=1}^{n} a_k i_k \;+\; \sum_{l=1}^{m} b_l p_l \;+\; c$$

Multi-dimensional affine functions are defined accordingly in each dimension.

Example

valid:	$f(i_1, i_2;\ p)$	$=$	$3i_1 + i_2 - p + 1$
not valid:	$f(i;\ p)$	$=$	pi
	$f(i_1, i_2;\ p_1, p_2)$	$=$	$i_1 + p_1 p_2$

A **piecewise affine mapping** consists of a set of affine mappings guarded by range specifications for variables or parameters. To ensure a sound function definition, all ranges have to be disjunct.

A very handy representation is a binary tree with **if** nodes, **then** or **else** edges and leaves containing the elementary mappings.

Note that not every **then** edge may have a pairing **else** edge, since ranges might not be contiguous.

Example

$$
\begin{array}{llllll}
f(i;\, n) = & \mathbf{if}\ \ i \le n & \mathbf{then} & \mathbf{if}\ \ n \le m & \mathbf{then} & m+1 \\
 & & & & \mathbf{else} & n+1 \\
 & & \mathbf{else} & i+1 & &
\end{array}
$$

2.3 Graphs

One fundamental theory used in this thesis is graph theory. We review briefly some notions we are going to use. There are many good text books about graph theory. The German reader may refer to [Jun94].

2.3.1 Definition

A **graph** G is a pair (V, E) of a **vertex set** V and an **edge set** E.

The edges of a **directed** graph are pairs (v, w) of the relation $V \times V$. The first vertex of the pair is called the **source**, the second the **target**. The edges of an **undirected** graph are two-element sets $\{v, w\} \subset V$. E forms a multiset, i.e., multiple edges between two vertices are permitted.

A **subgraph** (or **component**) (V', E') is a graph where $V' \subset V$ and $E' \subset (E \cap (V' \times V'))$.

A **directed path** from $v \in V$ to $w \in V$ is a sequence of edges starting at v and ending at w. The target of the preceding edge must be the source of the following edge.

A **cycle** is a path from v to v. A **self loop** (or **elementary cycle**) is a cycle consisting of a single edge.

A **multicycle** is a union of cycles (not necessarily connected) (Fig. 2.1).

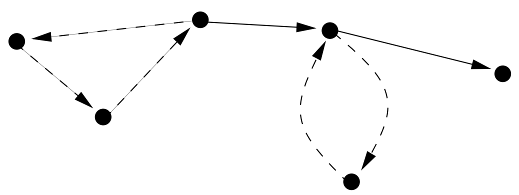


Fig. 2.1: Multicycle in a graph (dashed lines)

2.3.2 Definition

A directed graph G is called **strongly connected** iff there is a directed path from each vertex v to each other vertex w $(v \neq w)$.

An undirected graph is **connected** iff an undirected path exists from each vertex v to each other vertex w $(v \neq w)$.

A directed graph is **connected** iff the underlying undirected graph is connected (Fig. 2.2).

Fig. 2.2: Strongly connected graph   Connected graph

Remark

The strongly connected components of a directed graph $G = (V, E)$ can be computed in time $O(|V| + |E|)$ by the algorithm of Tarjan [ZC90].

This algorithm can also be used to obtain the connected components of a directed graph: construct a graph G' by adding an edge (w, v) for each edge (v, w) if it does not exist already (i.e., G' is the symmetric closure of G). The connected components of G are the strongly connected components of G'.

2.3.3 Definition

Edge weights are attached to every edge:

$$(\forall\, e \;:\; e \in E \;:\; (\exists\, \mathbf{w}_e \;:\; \mathbf{w}_e \in \mathbb{Q}^{m \times n}))$$

The **weight** (or **length**) $\mathbf{w}_\pi$ of a path π is the sum of its edge weights:

$$\mathbf{w}_\pi \;=\; \sum_{i=1}^{|\pi|} \mathbf{w}_{e_i}$$

A **shortest path** between two vertices is one with minimum length of all existing paths. A **longest path** is in analogy a path with maximum length.

Remark

Though efficient algorithms like the one of Dijkstra or Floyd are well known for the shortest path problem, this is not the case for the longest path problem. On the contrary: it is known to be NP-complete [Jun94]. Polynomial-time algorithms exist only for acyclic graphs. We will show in Section 4.2.13 that the problem of finding a schedule is strongly related to the problem of finding the longest path in a graph. For a special case, we will present an efficient algorithm.

2.3.4 Definition

A cycle is a path with infinite length. Nevertheless, it is possible to assign a (not unique) weight to it.

The **base weight** $\mathbf{w}_\mathcal{C}$ of a cycle $\mathcal{C}$ is the weight of a single traversal of the cycle. But all multiples $n\mathbf{w}_\mathcal{C}$ ($n \in \mathbb{N} \setminus \{0\}$) are weights of the cycle, too: n is the number of turns.

A multicycle need not to be a path. So the **weight of a multicycle** is defined as any nonempty sum of weights of any of its cycles.

Example

Multicycles of null weight are very important in Chapter 4. An example is depicted in Figure 2.3. It traverses twice the left cycle (of base weight (-1)) and once the right cycle (of base weight 2).

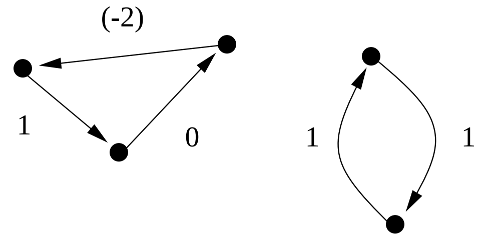


Fig. 2.3: Example for a null-weight multicycle

2.4 Polyhedra

The Polyhedron is another concept central to this thesis. Polyhedra are used to model subsets of $\mathbb{Q}^n$ or $\mathbb{Z}^n$, which can be treated quite efficiently. Two different representation are introduced.

2.4.1 Definition

A **hyperplane** is an affine subspace of our vector space $\mathbb{Q}^n$ with dimensionality $n-1$. It can be represented in two ways:

1. as the linear hull of $n-1$ linearly independent vectors $\mathbf{v}_i \in \mathbb{Q}^n$ shifted by a vector $\mathbf{b} \in \mathbb{Q}^n$:

$$H \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{x} = \sum_{i=1}^{n-1} a_i \mathbf{v}_i \;+\; \mathbf{b} \,\}$$

2. as the orthogonal space of a **normal vector** $\mathbf{a} \in \mathbb{Q}^n$ ($\mathbf{a} \neq \mathbf{0}$) and a constant b:

$$H \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{a}\mathbf{x} = b \,\}$$

Any hyperplane divides the vector space into two **halfspaces**. Mathematical representations are very similar. Just put an inequality in place of the equation:

$$H \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{a}\mathbf{x} \leq b \,\}$$

Remark

The orientation of the normal vector is important to halfspaces. The vector $\mathbf{a}$ and its reverse $(-\mathbf{a})$ mark the two halfspaces H^+ and H^- divided by a hyperplane.

Using the same vector $\mathbf{a}$ for both hyperplanes, we obtain:

$$\begin{aligned} H^+ &= \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{a}\mathbf{x} \geq b \,\} \\ H^- &= \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{a}\mathbf{x} \leq b \,\} \end{aligned}$$

So one can check easily which halfspace a point $\mathbf{x} \in \mathbb{Q}^n$ belongs to: just look at the sign of the scalar product $\mathbf{a}\mathbf{x}$.

2.4.2 Definition

Any intersection of finitely many halfspaces is called a **polyhedron**. According to the previous definition, it can be described as the solution space of a system of inequalities:

$$P(A, \mathbf{b}) \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; A\mathbf{x} \leq \mathbf{b} \,\}$$

A denotes an (m, n)-matrix and $\mathbf{b} \in \mathbb{Q}^m$.

- A **polytope** is a totally bounded polyhedron.
- A **polyhedral cone** is a polyhedron specified by a homogeneous system of inequalities:

$$C(A) \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; A\mathbf{x} \leq \mathbf{0} \,\}$$

2.4.3 Theorem (Decomposition Theorem)

A set $P \subset \mathbb{Q}^n$ is a polyhedron if and only if

$$P \;=\; \{\, \mathbf{x} + \mathbf{y} \;:\; \mathbf{x} \in Q,\; \mathbf{y} \in C \,\}$$

for some polytope $Q \subset \mathbb{Q}^n$ and some polyhedral cone $C \subset \mathbb{Q}^n$.

Proof [Sch86, p. 88]

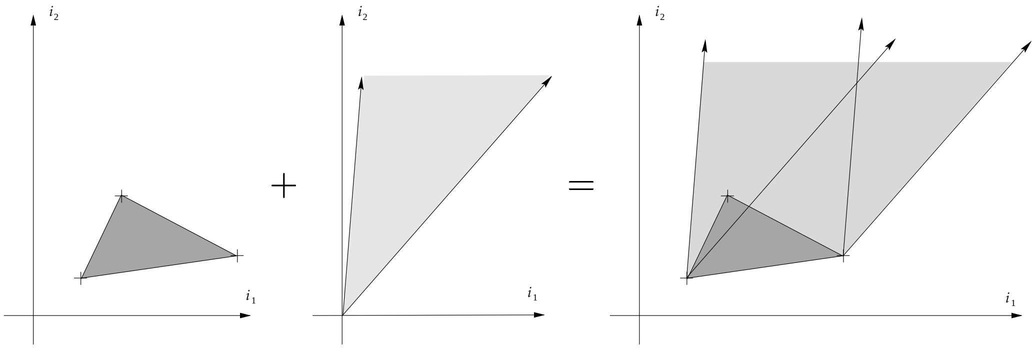


Fig. 2.4: Minkowski sum of a polytope and a cone

Remark

The previous theorem points the way to an alternative, constructive representation of a polyhedron. The next two theorems present constructive representations for polytopes and cones. Then an arbitrary polyhedron can be constructed according to Theorem 2.4.3 (called the *Minkowski sum*, Fig. 2.4) from the constructive representations of its polytope and cone.

2.4.4 Theorem (Main Theorem for Polytopes)

$P \subset \mathbb{Q}^n$ is the convex hull of a finite set of points $\{\mathbf{p}_1, \ldots, \mathbf{p}_m\} \subset \mathbb{Q}^n$:

$$P \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; a_i \in \mathbb{Q}^+,\; \sum_{i=1}^{m} a_i = 1 \;:\; \mathbf{x} = \sum_{i=1}^{m} a_i \mathbf{p}_i \,\}$$

Then P is a polytope.

Proof [Zie95]

2.4.5 Theorem (Main Theorem for Polyhedral Cones)

C is a nonnegative combination of a finite set of vectors $\{\mathbf{c}_1, \ldots, \mathbf{c}_m\} \subset \mathbb{Q}^n$:

$$C \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; a_i \geq 0 \;:\; \mathbf{x} = \sum_{i=1}^{m} a_i \mathbf{c}_i \,\}$$

Then C is a polyhedral cone.

Proof [Zie95]

2.4.6 Corollary

Each polyhedron $P = \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; A\mathbf{x} \leq \mathbf{b} \,\}$ can be defined equivalently as

$$P \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{x} = \sum_{i=1}^{\nu} a_i \mathbf{p}_i \;+\; \sum_{i=1}^{\varrho} b_i \mathbf{r}_i \;+\; \sum_{i=1}^{\lambda} c_i \mathbf{l}_i \,\}$$

for	a set of **points**	$\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_\nu\}$	$\subset \mathbb{Q}^n$,	$a_i \in \mathbb{Q}^+$,	$\sum_{i=1}^{\nu} a_i = 1$,
	a set of **rays**	$\mathcal{R} = \{\mathbf{r}_1, \ldots, \mathbf{r}_\varrho\}$	$\subset \mathbb{Q}^n$,	$b_i \in \mathbb{Q}^+$,	
	and a set of **lines**	$\mathcal{L} = \{\mathbf{l}_1, \ldots, \mathbf{l}_\lambda\}$	$\subset \mathbb{Q}^n$,	$c_i \in \mathbb{Q}$.	

We call these sets the **dual representation** of the polyhedron[1].

Proof

According to Theorem 2.4.3, P is the Minkowski sum of some polytope Q and some polyhedral cone C:

$$P \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; \mathbf{x} = \mathbf{x}_Q + \mathbf{x}_C,\;\; \mathbf{x}_Q \in Q,\; \mathbf{x}_C \in C \,\}$$

[1] We avoid the frequently used notion *parametric representation*, because we already use the term *parametric* for something completely different in Section 2.2.5.

By applying the Theorems 2.4.4 and 2.4.5 we get:

$$P \;=\; \{\, \mathbf{x} \in \mathbb{Q}^n \;:\; a_i,\; b_i \in \mathbb{Q}^+ \;,\; \sum_{i=1}^{\nu} a_i = 1 \;:\; \mathbf{x} = \sum_{i=1}^{\nu} a_i \mathbf{p}_i \;+\; \sum_{i=1}^{\gamma} b_i \mathbf{c}_i \,\}$$

Next we extract two sets $\mathcal{R}$ and $\mathcal{L}$ from $\mathcal{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_\gamma\}$:

$$\begin{aligned} \mathcal{R} &= \{\, \mathbf{c} \in \mathbb{Q}^n \;:\; \mathbf{c} \in \mathcal{C} \;\wedge\; (-\mathbf{c}) \notin \mathcal{C} \,\} \\ \mathcal{L} &= \{\, \mathbf{c} \in \mathbb{Q}^n \;:\; \mathbf{c} \in \mathcal{C} \;\wedge\; (-\mathbf{c}) \in \mathcal{C} \;\wedge\; \mathbf{c} \geq \mathbf{0} \,\} \end{aligned}$$

Putting things together, we can replace the sum over the elements of $\mathcal{C}$ in the equation above by two sums over elements of $\mathcal{R}$ and $\mathcal{L}$. For $\mathcal{L}$ we allow coefficients in $\mathbb{Q}$ rather than in $\mathbb{Q}^+$, as the negative combinations are correct in this case.

Then we have obtained exactly the equation stated in the theorem. □

2.4.7 Definition

A point (or ray) in $\mathcal{P}$ ($\mathcal{R}$) is called **extremal** if it cannot be generated by any combination of other points (rays).

Remark

- To avoid confusion with graph terminology, we do not use the common term **vertex** for extremal point.
- Minimal sets of points or rays contain only extremal elements. Minimal sets of lines are not unique and, therefore, no extremal lines exist.

2.4.8 Remark

- $\mathcal{L}$ is called the **lineality space** of P.
- If ray space $\mathcal{R}$ and lineality space $\mathcal{L}$ are empty, then P is a polytope.
- It is not required that the sets $\mathcal{P}$, $\mathcal{R}$ and $\mathcal{L}$ are minimal.

2.4.9 Remark

The conversion of a system of inequalities specifying a polyhedron into point, ray and line sets describing the same polyhedron is a nontrivial problem. The time complexity is necessarily exponential because of a result in polyhedral theory stating that the number of extremal points can be exponential on the number of inequalities.

Chernikova presented a conversion algorithm which is implemented, e.g., in [Wil93].

2.4.10 Theorem

f is an affine mapping, P is a polyhedron. Then the image $f(P)$ of P is again a polyhedron.

PROOF [Zie95]

2.4.11 Notation

A **wrapping polyhedron** $P(X)$ for some set $X \in \mathbb{Q}^n$ is any polyhedron which is a superset of X: $X \subset P(X)$.

2.5 Loop Programs

The syntax and semantics of real-world programming languages are much too complex for our purposes. It is essential to pick out a subset so that all allowed programs can be handled. The polytope model [Len93] has been designed for **loop programs**. Now we specify what input programs are exactly allowed.

- Program data can be divided into three classes:
  1. **Structure parameters** have a fixed, predefined value for the duration of a complete run of the program. They behave like constants and are often used to describe the size of a problem, e.g., sorting n elements. Their values are nonnegative integers.
  2. **Loop indices** are integer variables that are the counters of **for** loops (see below).
  3. **Variables** are arrays of some dimensionality and cell type.

     Access to array variables (read or write) must be addressable by affine functions of loop indices and structure parameters as defined in Definition 2.2.5, by **access functions**. For n indices i_j and m parameters p_k we get for an access to array x:

     $$x\left[\sum_{j=1}^{n} a_j i_j \;+\; \sum_{k=1}^{m} b_k p_k \;+\; a_0, \ldots\right]$$

- The only control structure is the **for** loop, in a Pascal-like semantics:

  for $i := l(i)$ **to** $u(i)$ **step** s **do**
      (∗ loop body ∗)
  od

  It consists of the following components:

  1. loop index (or loop counter) i to be incremented by stride s at the end of each loop turn,
  2. stride s, which must be an integer greater than or equal to 1,
  3. lower bound $l(i)$ and upper bound $u(i)$ for i, which are integer-valued expressions calculated *before* entering the loop. The following expressions are allowed:
     (a) affine functions of surrounding loop indices and structure parameters (again Definition 2.2.5 applies),
     (b) max() values of expressions in $l(i)$ resp. min() values in $u(i)$,
     (c) floor() or ceil() denoting lower or upper Gaussian brackets of expressions.
  4. The body of the loop is a syntactical block.

- Syntactical blocks are **for** loops or statements. A statement reads or writes a fixed sets of variables. In the following examples, we will simply use assignments (i.e., computations with just one write operation), but we do not depend on this restriction.

  $$V_{\text{write}} \;:=\; f(V_{\text{read}_1}, \ldots, V_{\text{read}_n})$$

  The set of all statements in the program is denoted by Ω_L.

- A loop program consists of a sequence of syntactical blocks.

A valid loop program is, for example:

for $i := 1$ **to** n **do**
    $a(i) := a(i-1)$ (∗ Statement 1 ∗)
    for $j := 1$ **to** $max(i+n, 5)$ **do**
        $b(i,j) := b(i,j-1) + a(i)$ (∗ Statement 2 ∗)
    od
    $c(i) := a(i)$ (∗ Statement 3 ∗)
od

Remark

The scheduler and allocator do not operate directly on the source program. They get their information from the preceding modules, mainly from the dependence analysis. Therefore, any program accepted by these modules is valid, as long as their output conforms with our restrictions. This allows much more complicated programs to be handled. Constructs, which may be hidden by preceding modules, may include negative stride, **if-then-else** or even **while** loops.

2.5.1 Definition

A loop index **carries** a statement s, if s is in the body of its loop. All the indices carrying a statement form its **iteration vector** $\mathbf{i} \in \mathbb{Z}^n$.

A concrete value of the iteration vector represents one atomic operation (in our model). It is also called an **instance** of a statement. The set of all possible values of $\mathbf{i}$ is called the **index space** (or **iteration domain**) I_s of s. For the iteration vector $\mathbf{i} = (i_1, \ldots, i_k)$, it is specified by the following system of inequalities:

$$\begin{array}{ccccc} l(i_1) & \leq & i_1 & \leq & u(i_1) \\ \vdots & & & & \vdots \\ l(i_k) & \leq & i_k & \leq & u(i_k) \end{array}$$

Remark

Due to the definition of $u(i_j)$ and $l(i_j)$, it is always possible to rearrange an inequality $i_j \geq l(i_j)$ or $i_j \leq u(i_j)$ to the system:

$$\begin{array}{ccccc} a_{1,1}i_1 + \cdots + a_{1,k}i_k & + & b_{1,1}p_1 + \cdots + b_{1,m}p_m & \leq & c_1 \\ \vdots & & & \vdots & \\ a_{n,1}i_1 + \cdots + a_{n,k}i_k & + & b_{n,1}p_1 + \cdots + b_{n,m}p_m & \leq & c_n \end{array}$$

Thus, the index space can be described as an intersection of finitely many halfspaces and, therefore, it forms a polyhedron. Since upper and lower bounds are finite, we know that the iteration domain is a finite set. So it is a polytope.

2.5.2 Remark (Polytopes versus Polyhedra)

If more complex expressions are allowed for loop bounds, the precise representation of the index space as a polytope is lost. The use of array variables with arbitrary contents as loop bounds results in rather unpredictable index spaces. This way, even **while** loops can be simulated [GL94]. What we can do in this case is to approximate the index space by a larger set. If we do not know exactly the extent in some direction we assume unboundedness. This generalization of the polytope model is known as the **polyhedron model**.

In the following chapters, it will be pointed out where polytopes are essential. This is not the case, e.g., for the core algorithm of the scheduler, but — unfortunately — for preparing its input and for postprocessing its output.

2.5.3 Definition

Perfectly nested loops form a subset of all possible loop programs with some nice properties. Every statement has the same surrounding loops. Thus, the iteration vectors are all of the same length and the index spaces are the same for all statements, which are all in the single **body** of the loop nest.

for $i_1 := l(i_1)$ **to** $u(i_1)$ **do**
    $\ddots$
    for $i_n := l(i_n)$ **to** $u(i_n)$ **do**
        (∗ Statement s_1 ∗)
        $\vdots$
        (∗ Statement s_n ∗)
    od
    $\vdots$
od

All other loop programs are called **imperfectly nested**.

3 Dependences

Variable accesses enforce an order on the flow of execution. The condition of Bernstein says that two subsequent accesses to the same variable cause a dependence if at least one is a write access [Ban93]. Thus, three basic types exist:

- read after write *(true dependence)*
- write after read *(anti dependence)*
- write after write *(output dependence)*

As we are not interested in the individual type, we treat them all the same.

3.1 Representing Dependences

A dependence δ between two statements s_i and s_j of a loop program consists of dependent pairs of instances of s_i and s_j.

$$s_i(\mathbf{i})\;\delta\;s_j(\mathbf{j})$$

denotes that instance $\mathbf{j}$ of s_j depends on instance $\mathbf{i}$ of s_i because of dependence δ. We refer to $s_i(\mathbf{i})$ as the **source** and to $s_j(\mathbf{j})$ as the **target** of a dependence pair.

δ is a preorder relation on instances of the two statements:

$$\delta \;=\; \left\{\; (\mathbf{i},\mathbf{j}) \;:\; \mathbf{i} \in I_{s_i},\; j \in I_{s_j} \;:\; s_i(\mathbf{i})\;\delta\;s_j(\mathbf{j}) \;\right\}$$

The set of all index vectors appearing as the first (second) component of a dependent pair is called the **source space** S_δ (the **target space** T_δ) of δ:

$$\begin{aligned} S_\delta &= \left\{\; \mathbf{i} \;:\; \mathbf{i} \in I_{s_i},\; j \in I_{s_j} \;:\; s_i(\mathbf{i})\;\delta\;s_j(\mathbf{j}) \;\right\} \\ T_\delta &= \left\{\; \mathbf{j} \;:\; \mathbf{i} \in I_{s_i},\; j \in I_{s_j} \;:\; s_i(\mathbf{i})\;\delta\;s_j(\mathbf{j}) \;\right\} \end{aligned}$$

The graph depicting this relation is called the **expanded dependence graph (EDG)**.

The size of the EDG depends in many cases on structure parameters and it is in general too large to be useful as input for schedulers. Condensed description for the dependence relation has to be found. There are several choices, but all are either restricted in some way or sacrifice on precision. The three representations introduced below are ordered by decreasing precision. The (anti) dependence from Statement s_2 to Statement s_1 in Figure 3.1 is used to demonstrate the differences of the representations.

for $i_1 := 1$ **to** n **do**
  for $i_2 := i_1$ **to** n **do**
    $a[i_1, i_2] := c[i_1, i_2]$ (∗ Statement s_1 ∗)
    $b[i_1, i_2] := a[2i_1, 2i_2] + d$ (∗ Statement s_2 ∗)
  od
od

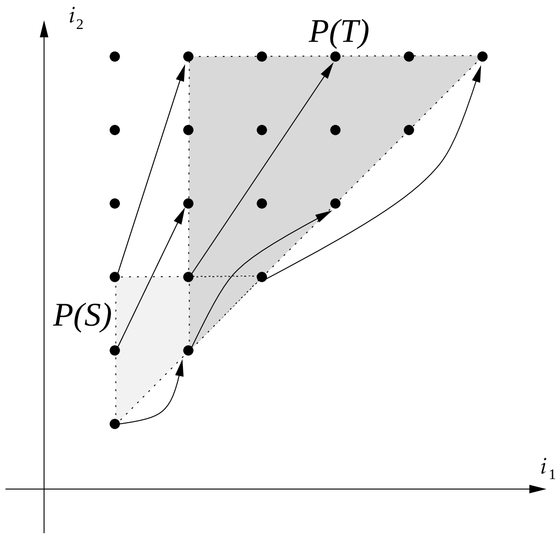


Fig. 3.1: Example for a dependence from s_2 to s_1 and its expanded dependence graph ($n = 6$)

3.1.1 Definition

As the source $s_i(\mathbf{i})$ of a dependence $s_i(\mathbf{i})\ \delta\ s_j(\mathbf{j})$ is unique, the dependence relation can be interpreted as the function

$$h\ :\ T_\delta \longrightarrow S_\delta\ :\ \mathbf{j} \longrightarrow \mathbf{i}$$

assigning to each target instance $\mathbf{j}$ a source $\mathbf{i}$. S_δ and T_δ are again the source and the target space of the dependence.

Due to our restrictions on array accesses and loop bounds, this is a (piecewise) affine mapping, it is called an ***h*-transformation** [Fea92a].

Remark

Though h-transformations are defined originally as piecewise affine functions, it is sufficient to regard them as affine functions. The separate cases in the definition of h can be viewed as separate functions h_k with smaller source and target spaces.

If the set T_δ is computed exactly, h-transformations are as accurate as the EDG. Again, the restrictions on input programs ensure that this set can be covered by a polytope.

Normally, the point sets T_δ and S_δ are only subsets of all integer lattice points inside the polytopes $P(T_\delta)$ and $P(S_\delta)$. There are "holes" in the lattice. The reason for this is that we do not know any additional property of the function h (e.g., unimodularity). Fortunately this is not much of a problem, as we will see in Section 3.2.1.

Note that h is generally not invertible, since one source may have many targets. Thus, reverse directions for dependence edges and h-transformations are essential.

For Figure 3.1, we get the following h-transformation (S_δ and T_δ are marked in Figure 3.1):

$$h \;:\; T_\delta \longrightarrow S_\delta \;:\; (i_1, i_2) \longrightarrow (\tfrac{1}{2}i_1, \tfrac{1}{2}i_2)$$

The dependence analysis that produces h-transformations is precise but expensive. Thus, simpler descriptions are needed.

3.1.2 Definition

Indices of the l loops, which are common to both statements involved in a dependence, appear as the first l entries of their iteration vectors. This is due to the nesting structure of loops. So an l-dimensional **distance vector** can be defined for $\mathbf{i} \in I_{s_i}$ and $\mathbf{j} \in I_{s_j}$:

$$\text{dist} \;:\; I_{s_i} \times I_{s_j} \longrightarrow \mathbb{Z}^l \;:\; (\mathbf{i}, \mathbf{j}) \longrightarrow (j_1 - i_1, \ldots, j_l - i_l)$$

As dependences can be caused only by operations executed before the current operation, distance vectors are always lexicographically nonnegative.

The set

$$E_\delta \;=\; \{\; \mathrm{dist}(\mathbf{i},\mathbf{j}) \;:\; \mathbf{i} \in I_{s_i},\; \mathbf{j} \in I_{s_j} \;:\; s_i(\mathbf{i})\; \delta\; s_j(\mathbf{j}) \;\}$$

is called **set of distance vectors** (or **distance set**).

dist is an affine function for the same reason as the h-transformation is. Since $P(S_\delta)$ and $P(T_\delta)$ are polytopes, again, due to Theorem 2.4.10, E_δ can be covered by a polytope.

If we omit all facets depending on structure parameters we get the **dependence polyhedron** $P(E_\delta)$.

The dependence polyhedron for our running example (Fig. 3.1) is:

$$P(E_\delta) \;=\; \left\{ (i_1, i_2) \in \mathbb{Q}^n \;\middle|\; \begin{array}{rcrcl} i_1 & & & \geq & 1 \\ -i_1 & + & i_2 & \geq & 0 \end{array} \right\}$$

It is shown in Figure 3.2.

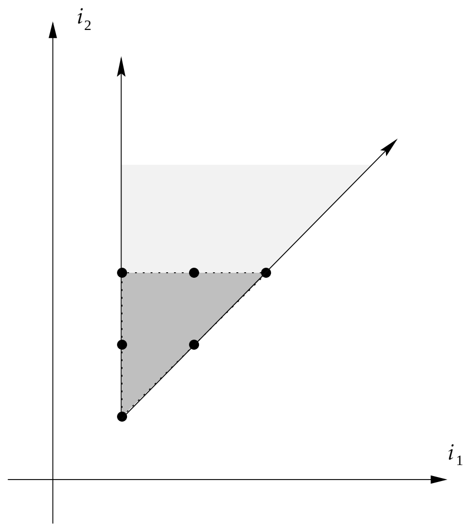


Fig. 3.2: Dependence polyhedron of Example 3.1

Remark

The EDG cannot be reconstructed from the dependence polyhedron, since only distances and no concrete pairs $(\mathbf{i}, \mathbf{j})$ are known. Nevertheless, a superset of the EDG, the **apparent dependence graph (ADG)** can be constructed. As Figure 3.3 shows, it contains a lot more dependence edges than the original EDG (the same set for each vertex). This may result in a loss of detectable parallelism as example 6 in [DV95a] demonstrates. But, on the other hand, the regularity of the ADG is the key to efficient scheduling algorithms.

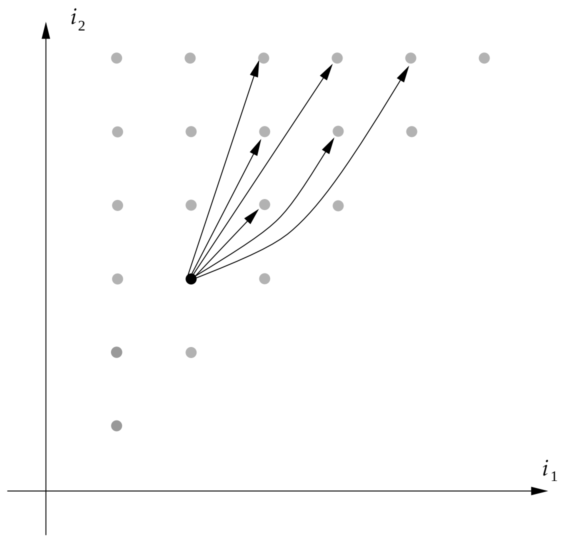


Fig. 3.3: Apparent dependence graph of Example 3.1 (detail)

Remark

An interesting special case for two statements sharing the same index space is $|E_\delta| = 1$ (i.e., all distance vectors are equal). Then $\mathbf{j} - \mathbf{i}$ is called a **uniform distance vector** and the dependence δ **uniform**.

Uniform dependences are easy to handle, since they are the same for all instances of the dependent statements. We can construct the **reduced dependence graph (RDG)** with one vertex per statement and one weighted edge per distance vector. This description is still exact.

We want to make use of an RDG in the general case, too. Replacing the single distance vectors by the sets $P(E_\delta)$ as edge labels results in a graph called the **polyhedral reduced dependence graph (PRDG)**.

We refer to the edges of the PRDG as **polyhedral edges** to emphasize their special weights.

The PRDG for Example 3.1 is shown in Figure 3.4.

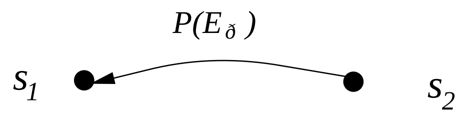


Fig. 3.4: Polyhedral reduced dependence graph of Example 3.1

3.1.3 Definition

A very compact but less accurate specification of the distance set is the **direction vector**. Each entry consists of an integer value and a qualifier. Formally, we define a direction vector $\mathbf{d}$ as follows:

$$\mathbf{d} \in \Big(\mathbb{Z} \times \{=, +, -, *\}\Big)^l$$

The direction vector is like a grammar for possible distance vectors in the set. The qualifiers have the following meaning:

$x=$	only value x
$x+$	all values higher than or equal to x
$x-$	all values lower than or equal to x
$*$	all values

The value "1" and the qualifier "=" can be omitted. So $(+, 2)$, for example, is a valid abbreviation for $(1+, 2=)$. Then, uniform distance vectors look the same as their corresponding direction vectors.

Since negative distance vectors are not possible, direction vectors like $(0-, \dots)$ or $(*, \dots)$ are not allowed.

Remark

Direction vectors can be viewed as special dependence polyhedra. In fact, each direction vector defines a polyhedron whose lines "$*$" and rays "$+, -$" follow the canonical base vectors of the index space.

Direction vectors are very popular since efficient dependence analysis is available for them [Pug92].

For Example 3.1, we get the direction vector $(1+, 1+)$. The corresponding polyhedron is depicted in Figure 3.5.

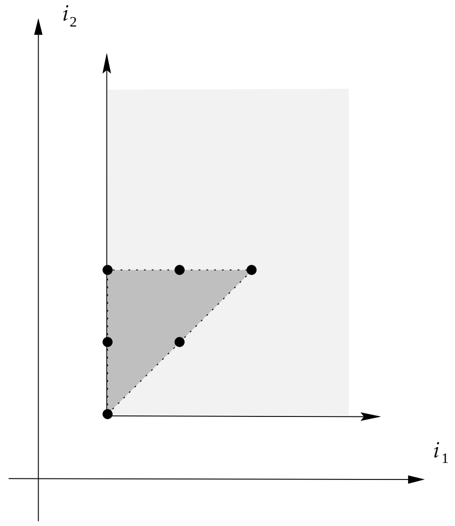


Fig. 3.5: Polyhedron described by the direction vector of Example 3.1

3.2 Converting Representations

In the LooPo project, the output of dependence analysis are h-transformations [Kei96]. Since the scheduler, as introduced in Chapter 4, expects dependence polyhedra, a conversion is necessary. Two methods are presented here: one for a conversion to direction vectors and one for a conversion to dependence polyhedra.

3.2.1 Direction Vectors

Only perfectly nested loops are considered for now. We will come back to the general case in Section 4.4.

The h-transformation comes as parametric affine function (Def. 2.2.5) plus two wrapping polyhedra for its source and target space:

$$h \;:\; P(T_\delta) \longrightarrow P(S_\delta) \;:\; \mathbf{j} \longrightarrow h(\mathbf{j})$$

Of course, if $P(T_\delta)$ is known precisely, the source space $P(S_\delta)$ is redundant, since it is the image of $P(T_\delta)$ under function h. But dependence analysis might not be that precise. For example, the index spaces of the source and target statements could have been used as an approximation of the source and target spaces.

We may formulate the set of dependent instances as:

$$\delta \;=\; \{\, (\mathbf{i},\mathbf{j}) \;:\; \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta) \;:\; \mathbf{i} = h(\mathbf{j}) \,\}$$

This way, we check, whether the results of the h-transformation are integral and inside the source space. As we only allow positive integer strides for our loops, rational results of the h-transformation mark holes in the target space. Holes in the source space are simply marked by not being a result of h for any lattice point of $P(T_\delta)$.

The distance set can be reformulated accordingly:

$$E_\delta \;=\; \{\ \mathrm{dist}(\mathbf{i},\mathbf{j})\ :\ \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta)\ :\ \mathbf{i} = h(\mathbf{j})\ \}$$

Each entry d_k of the direction vector $\mathbf{d}$ can be constructed separately.

First, the extent of the distance set in dimension k has to be determined. The Gaussian brackets below ensure that the limits are derived from a correct (i.e., integral) $(h(\mathbf{j}), \mathbf{j})$ pair. The integral minimum must be greater or equal to the total minimum. In analogy, the integral maximum is equal to or less than the total maximum.

$$\begin{aligned} l_k &= \lceil \min\{\ i_k - j_k\ :\ \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta)\ :\ \mathbf{i} = h(\mathbf{j})\ \}\rceil \\ u_k &= \lfloor \max\{\ i_k - j_k\ :\ \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta)\ :\ \mathbf{i} = h(\mathbf{j})\ \}\rfloor \end{aligned}$$

Both limits exist, because the distance set is contained in a (parameterized) polytope. But they may depend on structure parameters and are, therefore, not useful for us. We need to find parameter-less limits.

The easiest way to get rid of the parameters is to declare them as ordinary variables before computing the optimum. Then, the result of the optimization will not depend on parameters anymore. We can do so since we use parameters just like variables in our affine mappings (Def. 2.2.5). We need only to add inequalities for the parameters to the source and target polytopes. But since there is generally no upper bound for parameters, $P(S_\delta)$ and $P(T_\delta)$ turn into polyhedra. So “min” and “max” in the equations above turn into “inf” and “sup”.

$$\begin{aligned} l_k &= \lceil \inf\{\ i_k - j_k\ :\ \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta)\ :\ \mathbf{i} = h(\mathbf{j},\mathbf{p})\ \}\rceil \\ u_k &= \lfloor \sup\{\ i_k - j_k\ :\ \mathbf{i} \in P(S_\delta),\ \mathbf{j} \in P(T_\delta)\ :\ \mathbf{i} = h(\mathbf{j},\mathbf{p})\ \}\rfloor \end{aligned}$$

Then, the k-th entry of the direction vector is derived in the following way:

$$d_k \;=\; \begin{cases} l_k & l_k > -\infty\ \wedge\ l_k = u_k \\ l_k+ & l_k > -\infty\ \wedge\ l_k \neq u_k \\ u_k- & l_k = -\infty\ \wedge\ u_k < \infty \\ * & \text{otherwise} \end{cases}$$

Example

Again, the h-transformation for Example 3.1 is considered. We assume precise source and target spaces:

$$P(S_\delta) = \begin{bmatrix} i_1 & \geq & 1 \\ i_1 & \leq & \frac{1}{2}n \\ i_2 & \geq & i_1 \\ i_2 & \leq & \frac{1}{2}n \\ n & \geq & 0 \end{bmatrix} \qquad P(T_\delta) = \begin{bmatrix} i_1 & \geq & 2 \\ i_1 & \leq & n \\ i_2 & \geq & i_1 \\ i_2 & \leq & n \\ n & \geq & 0 \end{bmatrix}$$

$$h \;:\; P(T_\delta) \longrightarrow P(S_\delta) \;:\; (i_1, i_2) \longrightarrow (\tfrac{1}{2}i_1, \tfrac{1}{2}i_2)$$

The main task is to find the bounds according to the systems of inequalities of $P(S_\delta)$ and $P(T_\delta)$. In the author's implementation, PIP, the *Parametric Integer Programming* tool [FT90], is used for this purpose. Since no parameters are involved, also other linear optimizers are suitable. Our example is luckily so simple that we can find the bounds just by looking at the systems of inequalities.

In the following let $(i_1, i_2) \in P(T_\delta)$ and $(x_1, x_2) \in P(S_\delta)$.

First component:

$$\begin{array}{rccccc} \text{lower bound:} & l_1 & = & \lceil \inf \{\, x_1 \;:\; x_1 = i_1 - \frac{1}{2}i_1 \,\} \rceil & = & 1 \\ \text{upper bound:} & u_1 & = & \lfloor \sup \{\, x_1 \;:\; x_1 = i_1 - \frac{1}{2}i_1 \,\} \rfloor & = & \infty \end{array}$$

Second component:

$$\begin{array}{rccccc} \text{lower bound:} & l_2 & = & \lceil \inf \{\, x_2 \;:\; x_2 = i_2 - \frac{1}{2}i_2 \,\} \rceil & = & 1 \\ \text{upper bound:} & u_2 & = & \lfloor \sup \{\, x_2 \;:\; x_2 = i_2 - \frac{1}{2}i_2 \,\} \rfloor & = & \infty \end{array}$$

So we get $d_1 = 1+$ and $d_2 = 1+$ and the direction vector $(1+, 1+)$.

3.2.2 Dependence Polyhedra

Again, we only deal with perfectly nested loops. For other loop programs the method can be applied to the loops common to both statements. In this case, only a projection of the h-transformation to the common loop indices is used. Of course, the other indices have to be handled, too. We will come back to the general case in Section 4.4.3.

To eliminate structure parameters, we use the same trick as in the last section: we declare them as variables. Then, the h-transformation for a dependence surrounded by n common loops and parameterized by m parameters has the following dimensionality:

$$h \ : \ \mathbb{Q}^{n+m+1} \longrightarrow \mathbb{Q}^n \ : \ \mathbf{j} \longrightarrow H\mathbf{j}$$

where the $(n, n+m+1)$-matrix H is the homogeneous representation of h.

The distance set E_δ and hence the dependence polyhedron $P(E_\delta)$ is defined as the image of function dist for all dependent pairs of instances. Now we formulate a distance function $d_h(\mathbf{j})$ which is based on the given h-transformation:

$$d_h \ : \ T_\delta \longrightarrow E_\delta \ : \ \mathbf{j} \longrightarrow \mathrm{dist}(h(\mathbf{j}), \mathbf{j})$$

Then, we obtain a wrapping polyhedron of the distance set in the following way:

$$P(E_\delta) \ = \ d_h(P(T_\delta))$$

By applying the definitions of dist and h, we obtain:

$$d_h \ : \ P(T_\delta) \longrightarrow \mathbb{Q}^n \ : \ \mathbf{j} \longrightarrow (I - H)\mathbf{j} \qquad (*)$$

$I \in \mathbb{Q}^{n \times n+m+1}$ denotes the truncated identity matrix of $\mathbb{Q}^{n+m+1}$ (i.e., the projection $\mathbb{Q}^{m+n+1} \longrightarrow \mathbb{Q}^n$).

As d_h is an affine mapping and $P(T_\delta)$ is a polyhedron, the image $d_h(P(T_\delta))$ is a polyhedron, too (Theorem 2.4.10). It is the dependence polyhedron we are looking for.

To calculate the image of the polyhedron $P(T_\delta)$, it is necessary to convert it to its dual representation first. Then, d_h can be applied to each vector of these sets. At last, redundant (i.e., not extremal) elements can be deleted.

In the implementation, these three steps are performed by the `polylib`-library developed by Wilde and LeVerge [Wil93]. The result is the minimal dual representation of the dependence polyhedron.

Example

For Example 3.1 we get the following point, ray and line sets for $P(T_\delta)$ with parameter n as the last dimension:

$$\begin{array}{llcl} \text{points} & \mathcal{P} & = & \{(2,2,1)\} \\ \text{rays} & \mathcal{R} & = & \{(0,0,1),\ (0,1,1),\ (1,1,1)\} \\ \text{lines} & \mathcal{L} & = & \emptyset \end{array}$$

The polyhedron defined by these sets is shown in Figure 3.6.

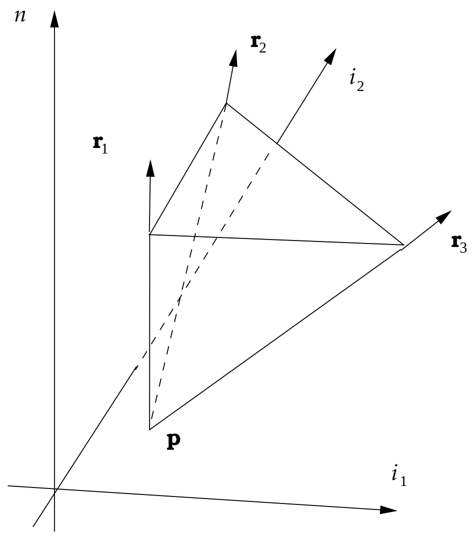


Fig. 3.6: Target space $P(T_\delta)$ for Example 3.1 including parameter n

Next, we apply distance function d_h of $(*)$ to each vector:

$$\begin{array}{lcl} \mathcal{P} & = & \{(1,1)\} \\ \mathcal{R} & = & \{(0,0),\ (0,\frac{1}{2}),\ (\frac{1}{2},\frac{1}{2})\} \end{array}$$

The first element of $\mathcal{R}$ is redundant and can be deleted. If needed, the other elements may be scaled to be integral. This is the expected result as shown in Figure 3.2.

Remark

The point set of $P(E_\delta)$ may contain nonintegral elements. As $E_\delta \subset \mathbb{Z}^n$, these points certainly do not belong to the distance set E_δ. For our purposes, the point set of the dependence polyhedron $P(E_\delta)$ is required to be integral, too. So we need a method for eliminating nonintegral points.

The integer hull $P_I(E_\delta)$ of E_δ (i.e., the convex hull of its elements) is the best dependence polyhedron that can be found. But finding it is a complicated task which need algorithms with execution time exponential in n. For more details see the *cutting planes* method in [Sch86, Chapter 23].

The only fast way is to find an integral **bounding box**, i.e., a polyhedron with facets that are all parallel to the canonical base vectors. We mentioned already that direction vectors represent special dependence polyhedra. They have exactly the required shape and can, therefore, be used as substitutes for $P(E_\delta)$.

In Figure 3.7 the different types of wrapping polyhedra are illustrated for an example point distribution.

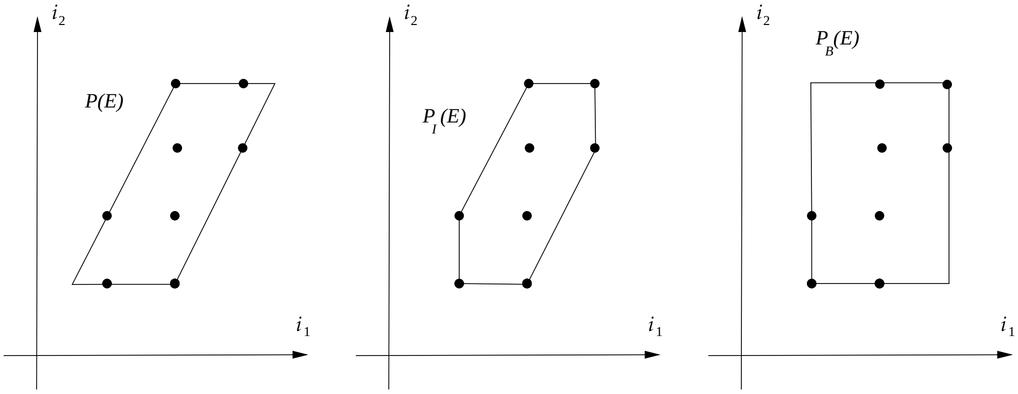


Fig. 3.7: Dependence polyhedron $P(E_\delta)$ with nonintegral extremal points, integer hull of E_δ and integral bounding box around E_δ

4 The Darte-Vivien Scheduler

The Darte-Vivien scheduler is basically a scheduler for systems of uniform recurrence equations (SUREs), which have been introduced by Karp, Miller and Winograd in 1967 [KMW67]. Their trail-blazing paper has inspired much research in the last few decades. In 1974, Lamport proposed a scheduler for a subclass of SUREs which is equivalent to perfectly nested loops with only uniform dependences [Lam74]. He discovered that, in this case, the result is always one sequential outermost loop and $n-1$ parallel inner loops.[1] However, these restrictions are quite severe, so that this scheduler is only useful in a limited way.

Darte and Vivien showed that the original theory is permissive enough to allow scheduling of a much wider variety of programs, including loop programs as defined in Section 2.5. Their contribution is, on the one hand, a method for finding schedules for every computable SURE and, on the other hand, methods for transforming dependences for loop programs into equivalent SUREs.

After some basic definitions concerning schedules in Section 4.1, we introduce SUREs, their differences to loop programs and methods for detecting computability and finding scheduling functions for them in Section 4.2. Then in Section 4.3, we present a transformation for dependence polyhedra of a perfect loop nest into a SURE. The subject of Section 4.4 is the extension of the methods to imperfectly nested loops. After some remarks on the actual implementation in Section 4.5, we conclude with a comparison with other scheduling methods in Section 4.6.

[1] The Lamport scheduler is also implemented as part of the LooPo project.

4.1 Basic Definitions

4.1.1 Definition

A **schedule** is a function which assigns integral time steps to all instances $s(\mathbf{i})$ of the statements $s \in \Omega_L$ of a program in such a way that dependent instances are executed in the correct order (i.e., source before target):

$$\Theta \;:\; \Omega_L \times I_{s \in \Omega_L} \longrightarrow \mathbb{N}$$

with

$$(\forall\, \mathbf{i}, \mathbf{j} \;:\; \mathbf{i} \in I_{s_1},\; \mathbf{j} \in I_{s_2},\; s_1(\mathbf{i})\, \delta\, s_2(\mathbf{j}) \;:\; \Theta(s_1, \mathbf{i}) \le \Theta(s_2, \mathbf{j}) + 1)$$

The latter property is called the **schedule property**.

The **latency** of a schedule is the number of time steps used, i.e., the highest minus the lowest time step.

4.1.2 Definition

A (one-dimensional) **affine schedule** is a set of affine functions X_s (one for each statement), which satisfy the schedule property:

$$\Theta \;:\; \Omega_L \times I_{s \in \Omega_L} \longrightarrow \mathbb{Q}^+ \;:\; (s, \mathbf{i}) \longrightarrow X_s \mathbf{i} + c_s$$

A **locally shifted affine schedule** is an affine schedule which has the same function X for every statement. Only the constants may differ:

$$\Theta \;:\; \Omega_L \times I_{s \in \Omega_L} \longrightarrow \mathbb{Q}^+ \;:\; (s, \mathbf{i}) \longrightarrow X \mathbf{i} + c_s$$

Note that such an X only can exist if the nesting depth of all statements is equal. If we omit the constants c_s, we obtain a **linear schedule**, which is exactly the same for all statements:

$$\Theta \;:\; I \longrightarrow \mathbb{Q} \;:\; \mathbf{i} \longrightarrow X \mathbf{i}$$

Remark

We allow rational time for affine schedules, which does not conform with the original definition of a schedule. But it does not mean that the granularity of time is refined. Since Θ must satisfy the schedule property, the time difference between two dependent operations is at least 1. We use the interpretation $\lfloor \Theta \rfloor$ of Θ in order to come back to integral results:

$$\lfloor \Theta \rfloor \;:\; \Omega_L \times I_{s \in \Omega_L} \longrightarrow \mathbb{N} \;:\; (s, \mathbf{i}) \longrightarrow \lfloor X_s \mathbf{i} + c_s \rfloor$$

Note that $\lfloor \Theta \rfloor$ satisfies the schedule property, if Θ does. As the Gaussian brackets would complicate the scheduling process, we omit them and expect that subsequent modules use the correct interpretation (which is $\lfloor \Theta \rfloor$).

4.1.3 Definition

Since X is a $(1, \dim(I_s))$-matrix, it can be interpreted as the normal vector of a hyperplane:

$$H \;=\; \{\, \mathbf{i} \in I_s \;:\; X\mathbf{i} = \mathbf{0} \,\}$$

If we restrict ourselves to perfectly nested loops (i.e., $(\forall\, s : s \in \Omega_L : I_s = I)$), we get the notion of a **strictly separating hyperplane**:

$$X \text{ linear schedule } \Leftrightarrow (\forall\, \mathbf{i}, \mathbf{j} \;:\; \mathbf{i} \in I,\; \mathbf{j} \in I,\; s_1(\mathbf{i})\; \delta\; s_2(\mathbf{j}) \;:\; X(\mathbf{j} - \mathbf{i}) \geq 1)$$

Thus, it can be viewed as a synonym for linear schedule.

We call X' (the normal vector of a) **weakly separating hyperplane**, if the scalar product is at least always nonnegative:

$$(\forall\, \mathbf{i}, \mathbf{j} \;:\; \mathbf{i} \in I,\; \mathbf{j} \in I,\; s_1(\mathbf{i})\; \delta\; s_2(\mathbf{j}) \;:\; X'(\mathbf{j} - \mathbf{i}) \geq 0)$$

Note that X' does not satisfy the schedule property anymore.

Remark

Geometrically, a linear schedule "slices" the index space with equidistant parallel hyperplanes (Fig. 4.1). The schedule property implies that there is a smallest possible distance between two hyperplanes. All statement instances in this slice can be computed in parallel.

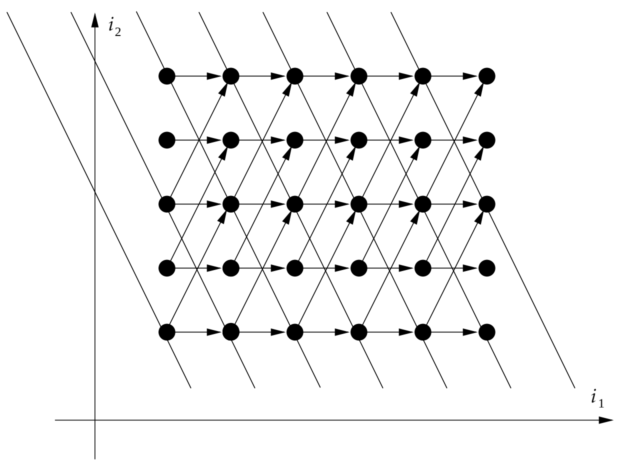


Fig. 4.1: Slicing the index space with parallel hyperplanes

4.1.4 Definition

Although for all loop programs there exists at least one schedule (the original sequential execution order), this schedule might not be affine since the degree of parallelism might not be high enough. To be able to schedule all loop programs, we make the extension to **multi-dimensional affine schedules**, defined as follows:

$$\Theta \ : \ \Omega_L \times I_{s\in\Omega_L} \longrightarrow \mathbb{Q}^n \ : \ (s, \mathbf{i}) \longrightarrow (X_s^1\mathbf{i} + c_s^1, \ldots, X_s^n\mathbf{i} + c_s^n)$$

with n as the dimensionality of the schedule. The lexicographic order of $\mathbb{Q}^n$ is used for checking the schedule property of Θ.

Remark

Multi-dimensional time seems to be confusing at first sight. To get an idea you may, for example, think of a clock. Hours establish the first, minutes the second and seconds the third dimension.

The great advantage of multi-dimensional affine schedules is their natural link to loop nests. An n-dimensional schedule can be interpreted as a perfect nest of n (sequential) loops. So for these programs one, (purely sequential) schedule can be immediately given: the identity $I \longrightarrow I \ : \ \mathbf{i} \longrightarrow \mathbf{i}$.

Geometrically, a multi-dimensional affine schedule can be viewed as a sequence of weakly separating hyperplanes followed by one strictly separating hyperplane, followed by a sequence of arbitrary hyperplanes. These are the sufficient and necessary conditions for multi-dimensional affine functions to have the schedule property (cf. with the definition of the lexicographic order (Def. 2.2.1)).

This already gives an idea of the strategy the Darte-Vivien scheduler uses: recursively divide the reduced dependence graph with weakly separating hyperplanes until a strictly separating hyperplane can be found for all edges of its subgraphs.

4.2 Systems of Uniform Recurrence Equations

Now, we turn our attention away from loop programs for a while. Let us examine systems of uniform recurrence equations, for they are useful for coding dependence relations and possible execution orders. The gap to loop programs is closed in Section 4.3.

First, we introduce some basics notions used by Karp et al.

4.2.1 Definition

A **system of uniform recurrence equations (SURE)** is a finite set of equations of the form:

$$V_k(\mathbf{i}) = \begin{cases} f(V_{k_1}(\mathbf{i} - \mathbf{d}_{k_1,k}), \ldots, V_{k_{m_k}}(\mathbf{i} - \mathbf{d}_{k_{m_k},k})) & \mathbf{i} \in I \\ z \in Z & \mathbf{i} \in \mathbb{Z}^n \setminus I \end{cases}$$

where

- k is the number of the current equation,
- $I \subset \mathbb{Z}^n$ is a totally bounded set, called the **index space**,
- $\mathbf{i} \in I$ is the **iteration vector**,
- V_k is initialized with a value z of some set Z outside the index space ($\mathbf{i} \notin I$) and computed by the given equation inside the index space,
- f_k is an arbitrary function with m_k arguments:

$$f_k \;:\; Z^{m_k} \longrightarrow Z$$

- $\mathbf{d}_{l,k} \in \mathbb{Z}^n$ $(1 \leq l \leq m_k)$ are called **distance vectors**.

4.2.2 Example

The following system will serve as an example throughout this section:

$$\begin{aligned} a(i,j,k) &= b(i-1,j-1,k) + a(i-1,j-1,k-1) \\ b(i,j,k) &= v(i,j,k) + b(i,j,k-3) \\ v(i,j,k) &= a(i,j-1,k) + v(i,j,k+1) \end{aligned}$$

with index space $I = \{0, \ldots, n\}^3$.

4.2.3 Definition (Dependence Graph)

Explicitly, SUREs describe only the flow of data, not the flow of execution. All values $V_{k_1}(\mathbf{i} - \mathbf{d}_{k_1,k}), \ldots, V_{k_{m_k}}(\mathbf{i} - \mathbf{d}_{k_{m_k},k})$ have to be computed before $V_k(\mathbf{i})$ can be computed. For that reason, $V_k(\mathbf{i})$ **depends** on these values. This implicitly restricts the execution order of the system.

In analogy to loop programs with uniform dependences, we can define an **expanded dependence graph (EDG)**, whose vertices are the elements of the index space, or the **reduced dependence graph (RDG)**, whose vertices are the equations (cf. Sec. 3.1). The RDG is constructed in the following way:

- Each vertex v_k corresponds to one variable V_k.
- Each dependence $V_l(\mathbf{i} - \mathbf{d}_{l,k})\ \delta\, V_k(\mathbf{i})$ becomes an edge from V_l to V_k, weighted by the vector $\mathbf{d}_{l,k}$.

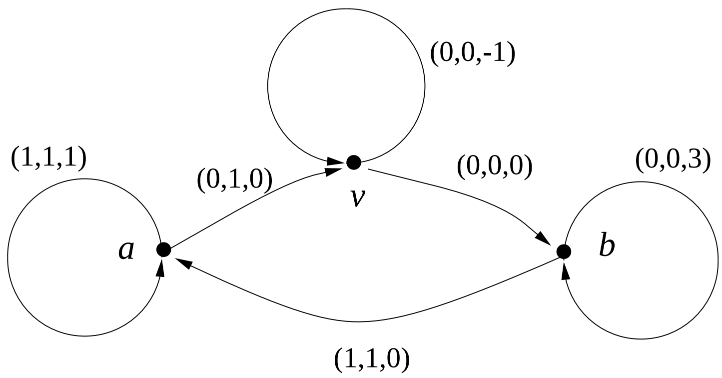


Fig. 4.2: Reduced dependence graph of Example 4.2.2

The expanded dependence graph can be obtained from the reduced one, if the index space I is known.

The dependence graph for Example 4.2.2 is depicted in Figure 4.2.

4.2.4 Remark (Differences to Loop Programs)

SUREs look very similar to perfectly nested loops with uniform dependences, as for both the notion of an index space and a reduced dependence graph exists. But we have already stated a difference: SUREs keep the execution order implicit.

There is another important difference: distance vectors in SUREs are allowed to be negative according to the lexicographic order. This does not make sense for distances in loop programs, because there we have the idea of (irreversible) time.

This allows that much more complicated programs can be modelled by SUREs. It is easy to write down an equivalent SURE for any perfect loop nest but not vice versa. There are even SUREs which are not computable at all. Karp, Miller and Winograd present a recursive decomposition algorithm of the RDG for deciding computability. Darte and Vivien show that this algorithm — in a modified version — is the key for finding multi-dimensional affine schedules for SUREs.

4.2.5 Theorem

A SURE is not solvable (not computable) iff its expanded dependence graph contains a cycle.

PROOF

A SURE with n equations is not computable if:

$$(\exists\, (k, \mathbf{i}) \ : \ 1 \leq k \leq n, \ \mathbf{i} \in I \ : \ V_k(\mathbf{i}) \text{ not computable})$$

1. The SURE is not computable $\Rightarrow$ the EDG contains a cycle.

   Assumption: The EDG is acyclic.

   Perform the following search on the EDG:

   (a) Mark the vertex $(k, \mathbf{i})$ in the EDG. $(k, \mathbf{i})$ not computable $\Rightarrow$ at least one of the instances $(k_1, \mathbf{i} - \mathbf{d}_1), \ldots, (k_m, \mathbf{i} - \mathbf{d}_m)$ on which it depends is not computable.

   (b) For each vertex in the list of not computable sources of dependence edges do:

   vertex is marked: cycle found in contradiction to the assumption.

   vertex is not marked: repeat Step (a) and (b) for all sources of incoming dependence edges.

   Since I is a bounded subset of $\mathbb{Z}^n$ and therefore finite, and in each iteration of the search either a cycle is found or the number of marked vertices is increased, the search must terminate. And it always terminates in finding a cycle which is a contradiction to the assumption.

2. The EDG contains a cycle $\Rightarrow$ the SURE is not computable.

   Then we can find a vertex $(k, \mathbf{i})$ included in the cycle. This vertex depends on itself:

   $$V_k(\mathbf{i}) \longleftarrow \cdots \longleftarrow V_k(\mathbf{i}) \longleftarrow \cdots \longleftarrow V_k(\mathbf{i}) \ldots$$

   There is an infinite chain of dependences and hence $V_k(\mathbf{i})$ is not computable. □

Remark

The proof of the theorem above is based on the fact that I is a finite set or, in other words, it is based on the polytope model. It fails for the polyhedral model, for which acyclic and not computable SUREs exist. Just look at

$$a(i) \;=\; f(a(i+1)) \qquad\qquad (I = \mathbb{N}).$$

All methods presented in this chapter are based on the fact that insolvability can be detected by cycles in the EDG. This does not mean that the scheduler cannot be used for the polyhedral model, just that additional checks are necessary.

4.2.6 Motivation

Primarily we want to find schedules, not unsolvable SUREs (for which we cannot succeed in finding a schedule). But we cannot simply expect that a given SURE has a one-dimensional affine schedule. Nevertheless, we can try to find such a one-dimensional schedule.

The method is based entirely on the reduced dependence graph (RDG) of the system. First, we reformulate the condition for a strictly separating hyperplane in graph theory. For an edge $e \in E$ of a graph $G = (V, E)$, whose weight $\mathbf{w}(e)$ is a distance vector which is oriented from source vertex $s(e)$ to target vertex $t(e)$, we get:

$$X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 1$$

A weakly separating hyperplane X satisfies:

$$X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 0$$

If a hyperplane X and constants c_v are given, we are able to check for every edge independently, if X is strictly or weakly separating (or not separating at all).

Now, the goal is to find a hyperplane X and constants c_v for each vertex, such that X is weakly separating for all edges and strictly separating for as many edges as possible:

$$\max\{\, |E_S| \;:\; E_S \subset E \;: \begin{array}{ll} (\forall\, e \;:\; e \in E_S & :\; X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 1) \wedge \\ (\forall\, e \;:\; e \in E \setminus E_S & :\; X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 0) \end{array} \,\} \qquad (1)$$

This maximization problem can be solved by linear programming. Note that it always has a solution, since $X = \mathbf{0}$ is always a weakly separating hyperplane for all edges.

But how about $E_S \subsetneq E$? No linear schedule can exist for the whole system, or else we would have found it by means of linear program (1). But we can try again to find a strictly separating hyperplane for the subset of edges $E \setminus E_S$ (which should be easier than for E). This suggests a recursive decomposition of the graph until a strictly separating hyperplane is found for all edges.

The decomposition is based on the computability check of Karp, Miller and Winograd. So we introduce this algorithm first, and then, we use it as a vehicle for the scheduling algorithm.

4.2.7 The Computability Check of Karp, Miller and Winograd

The computability check is a decomposition algorithm of the SURE's reduced dependence graph. The theoretical background is provided by means of two lemmata.

- The first lemma shows how to decide computability using the RDG rather than the EDG: the RDG must not contain a null-weight cycle.
- The second lemma is about how to find a null-weight cycle in graph:
  1. Find a null-weight multicycle (which is easier, since every null-weight cycle is a null-weight multicycle, but not vice versa). This can be done by solving a system of linear equations as we shall see in Section 4.2.8.
  2. Check if this multicycle is strongly connected, since this implies that one of its cycles has null weight.

4.2.7.1 Lemma

There is a cycle in the EDG iff there is a null-weight cycle in the RDG.

Proof

Let $\mathcal{C} = (v_1, \mathbf{j}_1) \rightarrow \ldots \rightarrow (v_{c-1}, \mathbf{j}_{c-1}) \rightarrow (v_1, \mathbf{j}_1)$ be a cycle in the EDG.

Then for each edge of $\mathcal{C}$ oriented from $(v_{i-1}, \mathbf{j}_{i-1})$ to $(v_i, \mathbf{j}_i)$ there is a corresponding edge from v_{i-1} to v_i with weight $(\mathbf{j}_i - \mathbf{j}_{i-1})$ in the RDG. These edges still form a cycle in the RDG. The weight of this cycle is the distance of the first to the last vertex of the path $\mathcal{C}$, which is $\mathbf{0}$, since $\mathcal{C}$ is a cycle in the EDG.

Let $\mathcal{C}' = v_1 \xrightarrow{\mathbf{d}_1} \ldots \xrightarrow{\mathbf{d}_{c-1}} v_{c-1} \xrightarrow{\mathbf{d}_c} v_1$ be a null-weight cycle in the RDG.

For each edge $(v_j, v_j + 1)$ of $\mathcal{C}'$ with weight $\mathbf{d}_j$ an edge

$$(v_j, \sum_{i=1}^{j-1} \mathbf{d}_i \ + \mathbf{d}_0) \ \rightarrow \ (v_{j+1}, \sum_{i=1}^{j} \mathbf{d}_i \ + \mathbf{d}_0)$$

of the EDG exists. These edges form a cycle, since the first and the last vertex are the same:

$$(v_1, \sum_{i=1}^{c} \mathbf{d}_i \ + \mathbf{d}_0) = (v_1, \mathbf{0} + \mathbf{d}_0)$$

$\square$

4.2.7.2 Lemma

G contains a strongly connected null-weight multicycle iff G contains a null-weight cycle.

PROOF [DV94, p. 10]

Example

Consider the null-weight multicycle in Figure 4.3 for example. It is strongly connected and, therefore, must contain a null-weight cycle.

Null-weight multicycle consisting of two subcycles:

$$\begin{aligned} \mathcal{C}_1 &= b \rightarrow c \rightarrow d \rightarrow e \rightarrow b \\ \mathcal{C}_2 &= b \rightarrow a \rightarrow e \rightarrow b \end{aligned}$$

Then: $\mathbf{w}(\mathcal{C}_1) = -1$, $\mathbf{w}(\mathcal{C}_2) = 1$.

For example, a null-weight cycle is:

$$\mathcal{C} = \ b \rightarrow c \rightarrow d \rightarrow e \rightarrow b \rightarrow a \rightarrow e \rightarrow b$$

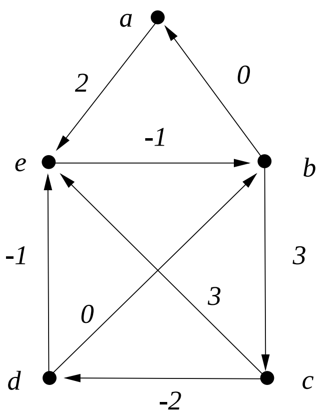


Fig. 4.3: Null-weight cycle inside a null-weight multicycle

4.2.7.3 The Algorithm

With the two preceding lemmata an algorithm for checking the computability of a SURE represented by its RDG can be formulated.

We try to find a null-weight cycle. So we look first for the maximum null-weight multicycle G' of the RDG G. A null-weight cycle can only be part of one of the strongly connected components $G'_1 \dots, G'_s$ of G'. These components may or may not contain another null-weight multicycle. So we do the same check as for G recursively to the components. If we find a strongly connected null-weight multicycle, this means that the SURE is not computable according to Lemma 4.2.7.2. If we do not find one, the SURE is computable.

Boolean CHECK (G)

1. Find the maximum null-weight multicycle G' of G.
   - If G' is empty, RETURN TRUE.
   - If G' is strongly connected, RETURN FALSE.
2. Compute the s strongly connected components of G'.
   RETURN CHECK (G'_1) $\wedge$ ... $\wedge$ CHECK (G'_s).

Algorithm CHECK will be used as the skeleton of the scheduling algorithm for SUREs (Sec. 4.2.10).

A method to compute the maximum null-weight multicycle of a graph is needed in order to implement the algorithm. This is the subject of the next section.

4.2.8 Construction of the Maximum Null-Weight Multicycle G'

Efficiency is the reason why we look for null-weight multicycles rather than for null-weight cycles. The latter is more difficult. Multicycles can be found easily by solving linear equations. The method is presented in this section.

4.2.8.1 Definition

The **connection matrix** C of a graph $G = (V, E)$ is a $|V| \times |E|$-matrix. Each row corresponds to a vertex of G and each column 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 $C_{k,j} = 1$ and set all other entries in the column j to 0. For a self loop j, the complete column j is set to 0.

The **weight matrix** W of a graph $G = (V, E)$ is an $(n, |E|)$-matrix for n-dimensional weights. Each column of W equals the weight of one edge.

Example

For our Example 4.2.2, we get as connection and weight matrix:

$$C = \begin{pmatrix} 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 & 0 \end{pmatrix} \qquad W = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & 0 & 3 & 0 \end{pmatrix}$$

4.2.8.2 System of Equations for a Null-Weight Multicycle

By means of the connection matrix, a multicycle of G can be found as follows:

$$C\mathbf{q} = \mathbf{0} \qquad \text{for a vector } \mathbf{q} \in \mathbb{N}^{|E|}$$

An entry $q_i \geq 1$ of $\mathbf{q}$ signals that the edge is part of the multicycle.

With the weight matrix W, the weight of a cycle represented by a vector $\mathbf{q}$, as defined above, can be checked. To be a null-weight multicycle, $\mathbf{q}$ must meet the following conditions:

$$C\mathbf{q} = \mathbf{0} \ \wedge \ W\mathbf{q} = \mathbf{0}$$

or, if we combine both matrices to a single matrix B:

$$B\mathbf{q} = \mathbf{0} \qquad \text{for the matrix } B = \begin{pmatrix} C \\ W \end{pmatrix}$$

Rational solutions for $\mathbf{q}$ can always be made integral by scaling them with the common denominator. Then each entry q_i tells how many times edge i is traversed in the null-weight multicycle.

Example

Example 4.2.2 contains a null-weight multicycle, since we find a solution $\mathbf{q} \neq \mathbf{0}$ for $B\mathbf{q} = \mathbf{0}$: $\mathbf{q} = (0, 0, 3, 0, 1, 0)^T$. The multicycle is depicted in Figure 4.4.

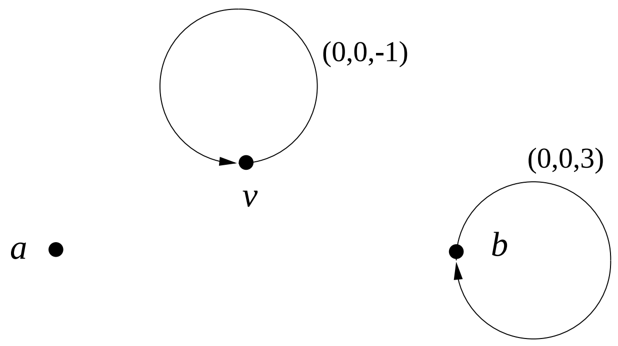


Fig. 4.4: Null-weight multicycle in the RDG of Example 4.2.2

4.2.8.3 Construction of the Optimization Problem

Now it is possible to formulate the problem as a linear program. A new variable $\mathbf{v}$ is introduced in order to change the maximization problem into a minimization problem. This is necessary, as we cannot maximize $\mathbf{q}$, because there is no upper bound for $\mathbf{q}$ (for a solution $\mathbf{q}$, also $n\mathbf{q}$ is a solution for $n \in \mathbb{N}$):

$$\min \{ \sum_{i=0}^{|E|} v_i \ : \ \mathbf{v}, \mathbf{q} \in (\mathbb{Q}^+)^{|E|} \ : \ \mathbf{q} + \mathbf{v} \geq \mathbf{1}, \ B\mathbf{q} = \mathbf{0} \ \} \tag{2}$$

The next lemma states the correctness of this linear program and gives an interpretation of vector $\mathbf{v}$.

4.2.8.4 Lemma

For any optimal solution $(\mathbf{q}, \mathbf{v})$ of linear program (2):

$$(\forall\, i \ : \ i \in \{1, \ldots, |E|\} \ : \ \begin{array}{l} v_i = 0 \Leftrightarrow e_i \in G' \\ v_i = 1 \Leftrightarrow e_i \notin G' \ ) \end{array}$$

i.e., $\mathbf{v} \in \{0,1\}^{|E|}$. An entry $v_i = 0$ of $\mathbf{v}$ marks edge e_i as an element of the null-weight multicycle, whereas $v_i = 1$ indicates that it does not belong to this multicycle.

Proof

- Let $v_i = 0$. $v_i + q_i \geq 1 \ \Rightarrow \ q_i \geq 1 \ \Rightarrow \ e_i \in G'$.
- Let $v_i = 1$. Then $q_i = 0$ (since $v_i + q_i \geq 1$ and $\mathbf{v}$ optimal) $\Rightarrow \ e_i \notin G'$

- Let $e_i \in G'$. Then $q_i > 0$.

  Assumption: $v_i > 0 \quad \Rightarrow \quad q_i < 1. \quad q_i = \frac{q_i^n}{q_i^d}$

  Scale $\mathbf{q}$ with the denominator of q_i : $\tilde{\mathbf{q}} = \mathbf{q} * q_i^d$

  $\Rightarrow \tilde{q}_i \geq 1 \;\Rightarrow\; \tilde{v}_i = 0 \;\Rightarrow\; \tilde{v}_i < v_i$.

  $\Rightarrow \mathbf{v}$ not optimal and hence the assumption is wrong.

- Let $e_i \notin G'$. Then $q_i = 0 \;\Rightarrow v_i = 1$. □

Remark

Now we have everything ready to implement the decomposition algorithm Check. The next theorem states that checking computability is closely related to finding a schedule. This theorem is the heart of the method.

4.2.9 Theorem

The set of edges of the maximum null-weight multicycle G' of an RDG G is equal to the minimum set of edges for which only a weakly separating hyperplane can be found.

Proof

The proof is based on the construction of the dual to linear program (2). First, we transform (2) to a normal form containing only equations and constraints for the variables to be nonnegative. We do so by introducing a vector $\mathbf{s}$ of slackness variables:

$$\min\{\ \sum_{i=0}^{|E|} v_i \ :\ \mathbf{v},\mathbf{q},\mathbf{s} \in (\mathbb{Q}^+)^{|E|}\ :\ \mathbf{q}+\mathbf{v}-\mathbf{s}=\mathbf{1},\ B\mathbf{q}=\mathbf{0}\ \} \tag{3}$$

We multiply the system of equations by (-1) which does not affect the solution:

$$\min\{\ \sum_{i=0}^{|E|} v_i \ :\ \mathbf{v},\mathbf{q},\mathbf{s} \in (\mathbb{Q}^+)^{|E|}\ :\ \mathbf{q}+\mathbf{v}-\mathbf{s}=\mathbf{1},\ -B\mathbf{q}=\mathbf{0}\ \}$$

For a better understanding of the construction of the dual, we combine all variables to a single vector $\mathbf{y}$ and normalize the objective function as the scalar product with a vector $\mathbf{b}$:

$$\min\{\ \mathbf{by}\ :\ \mathbf{b},\mathbf{y} \in \mathbb{Q}^{3|E|}\ :\ \mathbf{y} \geq \mathbf{0},\ A\mathbf{y}=\mathbf{d}\ \}$$

Taking a closer look at A, $\mathbf{d}$, $\mathbf{b}$ and $\mathbf{y}$ reveals ($I_{|E|}$ denotes the $(|E|,|E|)$ identity matrix):

$$A \;=\; \left(\begin{array}{c|c|c} I_{|E|} & I_{|E|} & -I_{|E|} \\ \hline -C & \mathbf{0} & \mathbf{0} \\ \hline -W & \mathbf{0} & \mathbf{0} \end{array}\right) \qquad \mathbf{d} \;=\; \left(\begin{array}{c} \mathbf{1} \\ \hline \mathbf{0} \\ \hline \mathbf{0} \end{array}\right)$$

$$\mathbf{b} \;=\; (\underbrace{0,\ldots,0}_{|E| \text{ times}}, \underbrace{1,\ldots,1}_{|E| \text{ times}}, \underbrace{0,\ldots,0}_{|E| \text{ times}})$$

$$\mathbf{y} \;=\; (q_1,\ldots,q_{|E|}, v_1,\ldots,v_{|E|}, s_1,\ldots,s_{|E|})^T$$

The dual of this optimization problem is [Sch86, p. 90]:

$$\max\{\ \mathbf{d}^T\mathbf{x} \ :\ \mathbf{d},\mathbf{x} \in \mathbb{Q}^{|E|+|V|+n} \ :\ A^T\mathbf{x} \leq \mathbf{b}^T\ \}$$

(n: dimensionality of the weights, $|V|$: number of vertices)

in detail:

$$A^T \;=\; \left(\begin{array}{c|c|c} I_{|E|} & -C^T & -W^T \\ \hline I_{|E|} & \mathbf{0} & \mathbf{0} \\ \hline -I_{|E|} & \mathbf{0} & \mathbf{0} \end{array}\right) \qquad \mathbf{b}^T \;=\; \left(\begin{array}{c} \mathbf{0} \\ \hline \mathbf{1} \\ \hline \mathbf{0} \end{array}\right)$$

Provided that there is a solution for both, the primal and the dual problem, we can conclude that $\mathbf{by} = \mathbf{d}^T\mathbf{x}$. This is true here. Set $\mathbf{v} = \mathbf{1}$, $\mathbf{q} = \mathbf{0}$ and $\mathbf{s} = \mathbf{0}$ for a solution of the primal and $\mathbf{x} = \mathbf{0}$ for a solution of the dual.

Next we divide the solution vector $\mathbf{x}$ again:

$$\mathbf{x} \;=\; (z_1,\ldots,z_{|E|}, X_1,\ldots,X_n, c_1,\ldots c_{|V|})^T$$

With these new variables the problem looks like:

$$\max\{\ \sum_{i=0}^{|E|} z_i \ :\ \mathbf{z} \in \mathbb{Q}^{|E|} \ :\ \mathbf{0} \leq \mathbf{z} \leq \mathbf{1} \ \wedge\ -W^TX - C^T\mathbf{c} + I_{|E|}\mathbf{z} \leq \mathbf{0}\ \} \tag{4}$$

Remember that the columns of C and W represent edges of G. If we pick the i-th inequality (representing an edge e), we get something quite familiar:

$$X\mathbf{w}(e) - c_{t(e)} + c_{s(e)} \geq \ z_i$$

The dual program calculates separating hyperplanes. So we are almost finished. Now recall linear program (1):

$$\begin{array}{llll} \max \{ \ |E'| : \ E' \subset E : & (\forall\, e \ : \ e \in E' & : \ X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 1) \ \wedge & \} \\ & (\forall\, e \ : \ e \in E \setminus E_S & : \ X\mathbf{w}(e) + c_{t(e)} - c_{s(e)} \geq 0) & \end{array}$$

which, based on (4), can be rewritten to

$$\begin{array}{l} \max \{ \sum_{i=0}^{|E|} z_i \ : \ \mathbf{z} \in \{0,1\}^{|E|} \ : \\ \qquad (\forall\, e_i \ : \ e_i \in E \ \wedge \ 1 \leq i \leq |E| \ : \ X\mathbf{w}(e_i) + c_{t(e_i)} - c_{s(e_i)} \geq z_i) \ \} \end{array} \qquad (5)$$

In order to prove that linear program (4) is really equivalent to (1), it is essential that $\mathbf{z} \in \{0,1\}^{|E|}$. For the completion of this proof it remains to show that

$$\mathbf{z} = \mathbf{v} \qquad \text{(which implies } \mathbf{z} \in \{0,1\}^{|E|}\text{).}$$

This equation is proved by the complementary slackness theorem [Sch86, p. 95]. Please refer to [DV94, p. 16] for the proof.

Remark

Theorem 4.2.9 shows that the "critical" edges are the same for checking computability and for looking for a schedule. Therefore, the decomposition algorithm CHECK of Section 4.2.7.3 can be used as a skeleton for a scheduling algorithm. G' contains all edges, for which no strictly separating hyperplanes can be found. We determine G' by solving linear program (5). Simultaneously, a hyperplane X and constants $c_1, \ldots, c_{|V|}$ are obtained.

During the recursive calls of CHECK, a sequence of weakly separating hyperplanes $X^1, \ldots, X^{d-1}$ followed by a strictly separating hyperplane X^d and constants $c_v^1, \ldots, c_v^d$ are computed for each vertex. Together, they form a d-dimensional affine schedule for that vertex. The depth d of the decomposition and, therefore, the dimensionality of the schedule may vary for different vertices.

We are now able to present (a first version of) the scheduling algorithm due to Darte and Vivien.

4.2.10 Scheduling Algorithm for SUREs

This is basically the same algorithm as CHECK, only the method to compute the null-weight multicycle has changed from linear program (2) to (5). This linear program yields not only the decomposition, but also a separating hyperplane and constants for all vertices.

Note that — in contrast to [DV94, DV96] — the RDG is not required to be strongly connected. Refer to Section 4.2.13 for a detailed discussion of this change.

DARTE-VIVIEN (G, k) (SURE version)

1. Find the maximum null-weight multicycle G' of G.
2. Select a vector X^k and for each vertex $v \in G$ a constant c_v^k such that:
$$(\forall\, e \,:\, e \in G \setminus G' \quad : \; X^k \mathbf{w}(e) + c^k_{t(e)} - c^k_{s(e)} \geq 1)$$
$$(\forall\, e \,:\, e \in G' \quad : \; X^k \mathbf{w}(e) + c^k_{t(e)} - c^k_{s(e)} \geq 0)$$
3. If G' is empty, RETURN.
4. If G' is strongly connected, G is not computable. STOP.
5. Decompose G' to its strongly connected components G'_i and call DARTE-VIVIEN$(G'_i, k+1)$ for each of them.

Before we give an example in Section 4.2.12, we present a method for choosing the hyperplane X in the previous algorithm.

4.2.11 Optimizing the Hyperplane X

Linear program (5) provides a hyperplane X, yet the quality of X is unknown. X does not appear in the objective function and, therefore, is not subject to optimization. There is some degree of freedom in choosing X. A criterion for a hyperplane to be optimal is needed. As we want to speed up the execution of the input program, we choose the latency of the target program as this criterion.

To find a schedule with optimum latency, the index space I must be considered. The latency l is defined as follows:

$$l = \max \{ \ t \ : \ t \in \mathbb{N}, \ \mathbf{i}, \mathbf{j} \in I, \ v_i, \ v_j \in V \ : \ t \ = \ |\Theta(v_i, \mathbf{i}) - \Theta(v_j, \mathbf{j})| \ \}$$

Minimizing l is a complex task which is performed, for example, by the Feautrier scheduler [Fea92a, Fea92b]. In order to keep the scheduling algorithm fast, we use a simpler method.

It is much easier not to consider the index space. If we assume a cubic index space, it is a good heuristic to minimize the norm of X, since this will minimize the range of the scheduling function. For a single hyperplane we are looking for:

$$\min \{ \ \|X\| \ : \ X \in \mathbb{Q}^n \ : \ X \text{ is valid hyperplane } \} \tag{6}$$

Here, valid values for X are solutions of linear program (5). Using the Manhattan norm, a linear program can find a vector of minimum norm [DV95b]. Each variable is divided into an additive and a subtractive part and both are minimized:

$$\begin{aligned}
& \min \{ \ \|\mathbf{a}\| \ : \ \mathbf{a} \in \mathbb{Q}^n \ \} \\
= \ & \min \{ \ \sum_{i=0}^{n} |a_i| \ : \ \mathbf{a} \in \mathbb{Q}^n \ \} \\
= \ & \min \{ \ \sum_{i=0}^{n} (a_i^+ + a_i^-) \ : \ \mathbf{a} \in \mathbb{Q}^n, \ \mathbf{a}^+, \mathbf{a}^- \in (\mathbb{Q}^+)^n \ \wedge \ \mathbf{a} = \mathbf{a}^+ - \mathbf{a}^- \ \}
\end{aligned}$$

We can combine linear programs (5) and (6) to a single, nested linear program. We optimize according to the objective function of (5) and then according to (6). So we can be sure to find the maximum set of edges for which strictly separating hyperplanes can be found, and then to look for the best hyperplane consistent with this set of edges:

$$\begin{aligned}
\min \{ \ \|X\| \ : \ X \in \mathbb{Q}^n \ : \ \max \{ \ \sum_{i=0}^{|E|} z_i \ : \ \mathbf{z} \in \{0,1\}^{|E|} \ : \\
(\forall \, e_i \ : \ e_i \in E \ : \ X\mathbf{w}(e_i) + \mathbf{c}_{t(e_i)} - \mathbf{c}_{s(e_i)} \geq z_i) \ \} \ \}
\end{aligned} \tag{7}$$

4.2.12 Example

Let us demonstrate the algorithm on Example 4.2.2. Its RDG G_0 can be found in Figure 4.2. We know already that it contains a null-weight multicycle (Fig. 4.4) and expect, therefore, at least a two-dimensional schedule.

1. Call DARTE-VIVIEN $(G_0, 1)$

   $\mathbf{z}$, X^1 and c_a^1, c_b^1, c_v^1 must satisfy the following constraints:

$$\begin{array}{rrrrrrrrrrl} X_1^1 & + & X_2^1 & + & X_3^1 & - & c_a^1 & + & c_a^1 & \geq & z_1 \\ & + & X_2^1 & & & - & c_a^1 & + & c_v^1 & \geq & z_2 \\ & & & - & X_3^1 & - & c_v^1 & + & c_v^1 & \geq & z_3 \\ & & & & & - & c_v^1 & + & c_b^1 & \geq & z_4 \\ & & & & 3X_3^1 & - & c_b^1 & + & c_b^1 & \geq & z_5 \\ X_1^1 & + & X_2^1 & & & - & c_b^1 & + & c_a^1 & \geq & z_6 \end{array}$$

   Optimal solutions according to linear program (7) are:

$$\mathbf{z} = (1,1,0,1,0,1), \quad X^1 = (0,1,0), \quad c_a^1 = 0, \quad c_v^1 = 0, \quad c_b^1 = 1$$

   G_0 contains a multicycle of null-weight consisting of the self loops through v and b (as already depicted in Figure 4.4).

   Both self loops form a strongly connected component. Let us call them G_1 and G_2.

   (a) Call DARTE-VIVIEN $(G_1, 2)$
   One inequality has to be considered:

$$- \; X_3^2 \; - \; c_v^2 \; + \; c_v^2 \; \geq z$$

   The optimal solution is $z = 1$, $X^2 = (0,0,-1)$ and $c_z^2 = 0$.

   (b) Call DARTE-VIVIEN $(G_2, 2)$
   Again, just a single inequality has to be considered:

$$3X_3^2 \; - \; c_b^2 \; + \; c_b^2 \; \geq z$$

   The optimal solution is $z = 1$, $X^2 = (0,0,\frac{1}{3})$ and $c_b^2 = 0$.

We have obtained a one-dimensional schedule for vertex a and two-dimensional schedules for b and v:

$$\begin{array}{rcl} \Theta(a,(i,j,k)) & = & j \\ \Theta(b,(i,j,k)) & = & (j+1, \frac{1}{3}k) \\ \Theta(v,(i,j,k)) & = & (j, -k) \end{array}$$

The schedule of b does not mean that we allow rational time steps. It says that three instances of b can be computed at the *same* time step.

4.2.13 Not Strongly Connected RDGs

In [DV94] the RDG is required to be strongly connected. This requirement is a consequence of the decomposition algorithm Check of Karp, Miller and Winograd [KMW67]. We have introduced Check differently, because it is an optimization than rather a requirement. Lemma 4.2.7.2 shows that a null-weight cycle can only be part of a strongly connected component. So it is sufficient for a computability check to divide the graph into strongly connected components and to look at these components separately.

In order to obtain a valid schedule, either the complete dependence graph must be considered, or, if only schedules for components can be derived, the partial solutions must be linked together afterwards. So two variants are presented below: the first handles not strongly connected RDGs differently for the first scheduling dimension and the second applies Darte-Vivien directly to it.

4.2.13.1 Scheduling the Strongly Connected Components (Hypergraph Method)

In [DV94] Darte and Vivien suggest to condense each strongly connected component to a single hypervertex. The resulting graph is definitely acyclic (each cycle disappears inside a strongly connected component) and can be scheduled by means of graph algorithms. So the first scheduling dimension consists of constant schedules for all vertices, depending on the strongly connected components to which they belong. Scheduling the components with Darte-Vivien then starts with the second dimension.

The simplest method is a topological sort of the vertices which is always possible for acyclic graphs [Jun94]. But the resulting schedule is purely sequential: it does not exploit any parallelism contained in the hypergraph.

For our purpose, a specially tailored algorithm is more appropriate. The main idea of the following algorithm Schedule-Acyclic is to wait, for each vertex, until all predecessors are scheduled and then schedule that vertex at the maximum of the times of all predecessors plus 1. It is equivalent to the proof of [Ban93, Theorem 1.1].

SCHEDULE-ACYCLIC(G)

    SCHEDULE-VERTEX(v)
        if all predecessors of v are scheduled **then**
            $c_v = \max\{\, c_x \;:\; x \text{ predecessor of } v \,\} + 1$
            for all not scheduled successors y of v **do**
                SCHEDULE-VERTEX(y)

while a not scheduled vertex v of G can be found **do**
    SCHEDULE-VERTEX(v)

The **while** loop calls SCHEDULE-VERTEX(v) in the worst case $|V|^2$ times, because at any time at least one vertex is ready to be scheduled (otherwise a cycle has been found). A call of SCHEDULE-VERTEX() causes at most $|V|$ recursive calls, since a recursive call only occurs if a vertex has been scheduled before. Thus, the algorithm always terminates. Note that the schedule property holds during the entire run of the algorithm for the subgraph of scheduled vertices. All vertices are scheduled, otherwise the algorithm would not terminate.

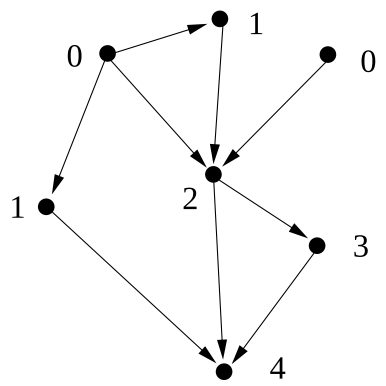


Fig. 4.5: Example of scheduling an acyclic graph

4.2.13.2 Direct Scheduling of the RDG

If we compute the maximum null-weight multicycle G' with linear program (7), the hyperplane X is correct for all vertices, regardless of whether the graph is strongly connected or not. The recursive calls of DARTE-VIVIEN at level

1 get the strongly connected components of G' as input. Thus, a problem can only occur at the top level ($k = 0$).

G' may contain multicycles that have null weight only across different strongly connected components. Thus, G' is a superset of the union of all null-weight multicycles of the strongly connected components.

So it may seem that the direct method is not as good as the hypergraph method, because more edges are assumed to be critical than is really necessary. But mind that the other variant does not even try to find a linear schedule for the whole graph.

A further optimization of the direct method is to divide the RDG into its connected components (Def. 2.3.2), as these are clearly independent and can therefore be handled separately:

SCHEDULE-RDG(G)

1. Decompose G to its connected components $G_0, \ldots, G_c$.
2. call DARTE-VIVIEN (G_i) for each of them.

Remark

This variant often yields more compact schedules than the variant of Section 4.2.13.1, though latency may be equal or even worse. Of course, a two-dimensional schedule with a constant in the first dimension has linear latency. But technically it uses two dimensions which complicates the merging with the allocation.

The question of which variant results in a better schedule remains to be answered. Examples can be found for each. But the latency of the direct method is often only a fraction of the latency of the hypergraph method (Ex. 4.6). On the other hand we found constant delays only if the direct method is slower (Ex. 4.8). Unfortunately this only holds for SUREs as introduced so far. In the next section, we will distinguish between two types of vertices. Then, the direct method sometimes results in significantly slower schedules (Ex. 4.3.5).

4.2.13.3 Example

To put the variants to the test, three examples should be considered ($n(i)$ denotes the number of loop traversals for index i).

Schedule for hypergraph method:

$$\begin{aligned}\Theta(a,(i,j)) &= (0,i)\\ \Theta(b,(i,j)) &= (1,i)\end{aligned}$$

Schedule for direct method:

$$\begin{aligned}\Theta(a,(i,j)) &= i\\ \Theta(b,(i,j)) &= i+1\end{aligned}$$

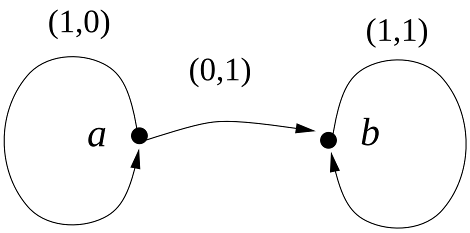


Fig. 4.6: Direct method better than hypgergraph method

For Example 4.6 the direct method yields a one-dimensional schedule which is indeed faster than the one obtained by the hypergraph method ($n(i)+1$ time steps compared with $2n(i)$ time steps).

Schedule for hypergraph method:

$$\begin{aligned}\Theta(a,(i,j)) &= (0,-i)\\ \Theta(b,(i,j)) &= (1,i)\end{aligned}$$

Schedule for direct method:

$$\begin{aligned}\Theta(a,(i,j)) &= (0,-i)\\ \Theta(b,(i,j)) &= (1,i)\end{aligned}$$

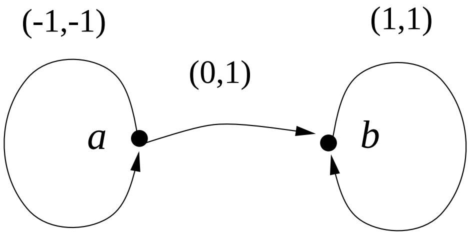


Fig. 4.7: Direct method equal to hypergraph method

Example 4.7 contains a null-weight multicycle across strongly connected components. The hyperplane found for the first dimension is $X = \mathbf{0}$ and, therefore, the result is identical to that of the hypergraph method.

Schedule for hypergraph method:

$$\begin{aligned}\Theta(a,(i,j)) &= (0,i)\\ \Theta(b,(i,j)) &= (1,j)\end{aligned}$$

Schedule for direct method:

$$\begin{aligned}\Theta(a,(i,j)) &= i+j\\ \Theta(b,(i,j)) &= i+j+1\end{aligned}$$

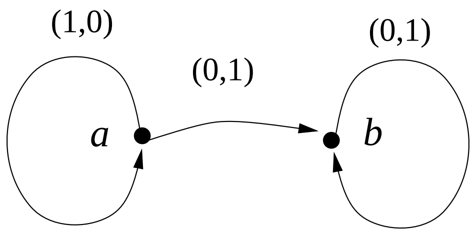


Fig. 4.8: Hypergraph method better than direct method

In Example 4.8, both schedules have almost the same latency $(n(i)+n(j))$, though their appearance is quite different.

4.3 Application to Perfectly Nested Loops

Now we present the missing link for scheduling loop programs: the transformation of a polyhedral reduced dependence graph to the reduced dependence graph of a SURE. It will turn out that, in the general case of arbitrary dependence polyhedra, we must modify the scheduling algorithm, too.

4.3.1 Transformation Principle

No work has to be done for a polyhedron consisting of a single point. It represents a single distance vector which becomes the edge weight in the RDG. For other polyhedra one can try to add a weighted edge for each element of the distance set, but this is only practical for (small-size) finite polyhedra.

One fundamental property of dependence graphs is transitivity. If a vertex s_2 depends on s_1 and a third vertex s_3 depends on s_2, it depends also on s_1. Formally:

$$(\forall\, s_1, s_2, s_3 \;:\; s_1 \in G,\; s_2 \in G,\; s_3 \in G \;: \\ s_1 \xrightarrow{\mathbf{w}_1} s_2 \;\wedge\; s_2 \xrightarrow{\mathbf{w}_2} s_3 \;\Rightarrow\; s_1 \xrightarrow{\mathbf{w}_1+\mathbf{w}_2} s_3)$$

Thus, the distance vector $\mathbf{w}_1+\mathbf{w}_2$ is modelled in the graph, but there is no explicit edge for it, only a path.

Now, the idea is to model each element of the dependence polyhedron $P(e)$ by a corresponding path in the reduced dependence graph:

$$(\forall\, \mathbf{d} \;:\; \mathbf{d} \in P(e) \;:\; (\exists\, \text{path } e_1, \ldots, e_m \;:\; e_1, \ldots, e_m \in E \;:\; \mathbf{d} = \sum_{i=1}^{m} \mathbf{w}(e_i)))$$

4.3.2 Transforming Direction Vectors

Direction vectors are only special dependence polyhedra, as we have seen in Section 3.1.3. Their structure makes it easy to model them by paths of weighted edges. They are cones with all orthogonal rays and lines. As stated in the proof of Theorem 2.4.6, a line $\mathbf{l} \in \mathcal{L}$ can be replaced by two rays $\mathbf{l}$ and $(-\mathbf{l})$. Therefore, it is sufficient to handle only points and rays. All rays of a direction vector are equal to the canonical base vectors of $\mathbb{Z}^n$. So all integral points inside the cone $P(e) = (\{\mathbf{p}\}, \{\mathbf{r}_1, \ldots \mathbf{r}_m\}, \emptyset)$ can be expressed by the following formula:

$$(\forall\, \mathbf{d} \;:\; \mathbf{d} \in P(e) \cap \mathbb{Z}^n \;:\; (\exists\, \mathbf{a} \;:\; \mathbf{a} \in \mathbb{N}^n \;:\; \mathbf{d} = \mathbf{p} + \sum_{i=1}^{m} a_i \mathbf{r}_i)) \qquad (*)$$

To model this formula by paths in the dependence graph, we first add a new vertex v to the graph, which we call a **virtual vertex**. Then we add the following edges:

- an edge (s_i, v) with weight $\mathbf{p}$,
- for each ray $\mathbf{r}_i$ a self loop (v, v) with weight $\mathbf{r}_i$,
- an edge (v, s_j) with weight $\mathbf{0}$.

Each path from s_i to s_j traverses once an edge with weight $\mathbf{p}$ to the virtual vertex, then maybe many times any sequence of self loops around the vertex. This way, any integral combination of rays is modelled. The edge from v to s_j completes the path. Thus, each possible coefficient $\mathbf{a}$ in formula $(*)$ has a corresponding path.

Example

The direction vector $(2+, *)$ has the following polyhedral representation:

$$\mathcal{P} \;=\; \{(2,0)\}, \qquad \mathcal{R} \;=\; \{(1,0)\}, \qquad \mathcal{L} \;=\; \{(0,1)\}$$

For the line in $\mathcal{L}$ two rays are added to $\mathcal{R}$:

$$\mathcal{R} \;=\; \{(1,0),(0,1),(0,-1)\}, \qquad \mathcal{L} \;=\; \emptyset$$

The corresponding subgraph is shown in Figure 4.9.

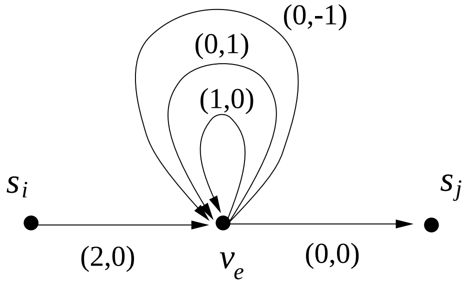


Fig. 4.9: Example for modelling a direction vector: $(2+, *)$

Remark

Note that for a line of $P(e)$ two anti-directed self loops with weights $\mathbf{r}$ and $(-\mathbf{r})$ are inserted. This adds a null-weight self loop in the reduced graph which is clearly a self loop in the expanded graph. According to Theorem 4.2.5, this means that the SURE is not solvable anymore. In order to avoid this effect, Darte and Vivien proposed in their first paper on their scheduler [DV94] the insertion of two virtual vertices: one for lexicographically negative and one for lexicographically positive self loops. But the scheduling algorithm must be adapted to the presence of virtual vertices anyway. We will treat virtual and real vertices differently allowing, for example, null-weight cycles on virtual vertices.

4.3.3 Transforming Dependence Polyhedra

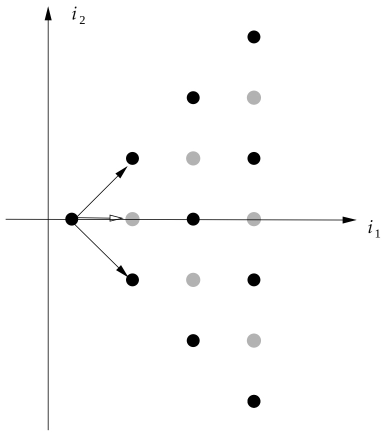
Transforming arbitrary dependence polyhedra is a bit more of a challenge. The distance vectors of the polyhedron $P(e) = (\mathcal{P}, \mathcal{R}, \emptyset)$ can be described in the following way (Theorem 2.4.6):

$$(\forall\, \mathbf{d} :\ \mathbf{d} \in P(e) \cap \mathbb{Z}^n :\ (\exists\, \mathbf{a}, \mathbf{b} :\ \mathbf{a} \in (\mathbb{Q}^+)^{|\mathcal{P}|},\ \mathbf{b} \in (\mathbb{Q}^+)^{|\mathcal{R}|},\ \textstyle\sum_{\nu=1}^{|\mathcal{P}|} a_\nu\ = 1 :\ \mathbf{d} = \displaystyle\sum_{\nu=1}^{|\mathcal{P}|} a_\nu \mathbf{p}_\nu + \sum_{\varrho=1}^{|\mathcal{R}|} b_\varrho \mathbf{r}_\varrho\ ))$$

4.3.3.1 Remark

Although the formula looks very similar to the one for direction vectors, we must allow rational coefficients $\mathbf{a}$ and $\mathbf{b}$ now in order to reach all integer points covered by the polyhedron. As Figure 4.10 shows, even for very simple polyhedra purely integral coefficients do not enumerate all points of the integer lattice.

Since edges of the RDG can either be traversed or not, there is no way to model rational coefficients. One possible solution is to extend the set of rays and points to reach all integer points by integer combinations of them. This is a well-known problem in number theory. The extended sets are called the **Hilbert base** of $P(e) \cap \mathbb{Z}^n$ [Sch86]. There are algorithms to find such a base for polyhedral cones [Fio72]. By homogenization of the dependence polyhedron they could be applied in the general case, too. Though possible, this is an expensive operation and the sets $\mathcal{P}$ and $\mathcal{R}$ may become very large.



$$
\begin{aligned}
\mathcal{P} &= \{(1,0)\} \\
\mathcal{R} &= \{(1,1),(1,-1)\} \\
\mathcal{R}_{\text{Hilbert}} &= \{(1,1),(1,-1),(1,0)\}
\end{aligned}
$$

Fig. 4.10: Dependence cone with holes indicating nonintegral combinations of the rays and an extension to a Hilbert base

Darte and Vivien choose a different approach in [DV96]. They do not fix the flaw in the transformation. Only the scheduling algorithm is modified in order to ensure correct results for the original polyhedral dependence graph.

4.3.3.2 The Construction of the Subgraph

They use a transformation for a polyhedral edge e with polyhedron $P(e)$ which is almost the same as for direction vectors. The only difference is that now more than one point has to be modelled. Thus, multiple edges from s_i to v_e are inserted. The resulting subgraph is shown in Figure 4.11.

- Add a virtual vertex v_e.
- Add an edge (s_i, v_e) with weight $\mathbf{p}_\nu$ for every point in $\mathcal{P}$.
- Add for each ray $\mathbf{r}_\varrho$ a self loop (v_e, v_e) with weight $\mathbf{r}_\varrho$.

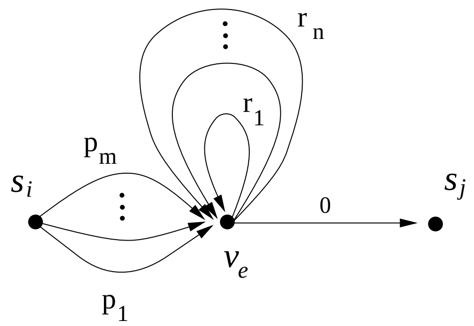


Fig. 4.11: Transformation of a polyhedral edge

- Add an edge (v_e, s_j) with weight $\mathbf{0}$.

The transformation does *not* preserve equivalence in the sense that, for each integer point of an edge's polyhedron $P(e)$ of the PRDG G_0, a corresponding path in the transformed RDG G_u exists. Nevertheless, we shall see that the schedules for G_0 and G_u are equivalent.

Remark

For a better understanding of the differences between G_0 and G_u, we introduce two lemmata. The first lemma states that the transformation preserves null-weight multicycles. This is essential, since the decomposition algorithm is based on the existence of null-weight multicycles. The second lemma states that the subset of distance vectors modelled in G_u is sufficient to ensure equivalence of the schedules for G_0 and G_u.

4.3.3.3 Lemma

Let $\mathcal{C}$ be a dependence cycle of G_0, i.e., a cycle of polyhedral edges. Then, for some integer m, there is a cycle $\mathcal{C}_u$ in G_u with the same structure as $\mathcal{C}$ and of weight $m\mathbf{w}(\mathcal{C})$.

Proof [DV96, p. 16]

4.3.3.4 Lemma

The following implications hold for a polyhedron $P(e)$ of an edge e of G_0 with points $\mathcal{P}$ and rays $\mathcal{R}$ ($\mathcal{L} = \emptyset$):

1. A hyperplane X is weakly separating for all points contained in $P(e)$ if it is weakly separating for all elements of $\mathcal{P}$ and $\mathcal{R}$. Formally:

$$\left(\forall\,\mathbf{p},\ \mathbf{r}\ :\ \mathbf{p}\in\mathcal{P},\ \mathbf{r}\in\mathcal{R}\ :\ \begin{array}{ll} X\mathbf{p}+c_v-c_{s(e)} & \geq\ 0 \\ X\mathbf{r}\ +c_v-c_v & \geq\ 0 \end{array}\right)$$
$$\Rightarrow\ (\forall\,\mathbf{d}\ :\ \mathbf{d}\in P(e)\ :\ X\mathbf{d}+c_v-c_{s(e)}\geq 0)$$

2. A hyperplane X is strictly separating for all points contained in $P(e)$ if it is strictly separating for all points of $\mathcal{P}$ and weakly separating for all rays of $\mathcal{R}$. Formally:

$$\left(\forall\,\mathbf{p},\ \mathbf{r}\ :\ \mathbf{p}\in\mathcal{P},\ \mathbf{r}\in\mathcal{R}\ :\ \begin{array}{ll} X\mathbf{p}+c_v-c_{s(e)} & \geq\ 1 \\ X\mathbf{r}\ +c_v-c_v & \geq\ 0 \end{array}\right)$$
$$\Rightarrow\ (\forall\,\mathbf{d}\ :\ \mathbf{d}\in P(e)\ :\ X\mathbf{d}+c_v-c_{s(e)}\geq 1)$$

Proof

Let $\mathbf{d}\in P(e)$. Then exist $\mathbf{a},\ \mathbf{b}\in(\mathbb{Q}^+)^n,\ \sum_{i=1}^{|\mathcal{P}|}a_i=1$:

$$\begin{aligned}
X\mathbf{d}+c_v-c_{s(e)} &= X\left(\sum_{i=1}^{|\mathcal{P}|}a_i\mathbf{p}_i\ +\ \sum_{i=1}^{|\mathcal{R}|}b_i\mathbf{r}_i\right)\ +\ c_v-c_{s(e)} \\
&= \sum_{i=1}^{|\mathcal{P}|}a_i\,X\mathbf{p}_i\ +\ c_v-c_{s(e)}\ +\ \sum_{i=1}^{|\mathcal{R}|}\underbrace{b_i\,X\mathbf{r}_i}_{\geq 0}\ +\ \underbrace{c_v-c_v}_{=0} \\
&\geq \sum_{i=1}^{|\mathcal{P}|}a_i\,X\mathbf{p}_i\ +\ c_v-c_{s(e)} \\
&= \sum_{i=1}^{|\mathcal{P}|}\left(a_i\,X\mathbf{p}_i\ +\ a_i(c_v-c_{s(e)})\right) \qquad (\text{since } \textstyle\sum_{i=1}^{|\mathcal{P}|}a_i=1)
\end{aligned}$$

For a weakly separating hyperplane X (item 1 of the lemma), we continue as follows:

$$= \sum_{i=1}^{|\mathcal{P}|}\underbrace{a_i}_{\geq 0}\underbrace{(X\mathbf{p}_i+c_v-c_{s(e)})}_{\geq 0}\quad\geq\ 0$$

For a strictly separating hyperplane X (item 2 of the lemma), we continue:

$$= \sum_{i=1}^{|\mathcal{P}|}a_i\underbrace{(X\mathbf{p}_i+c_v-c_{s(e)})}_{\geq 1}\quad\geq\ \sum_{i=1}^{|\mathcal{P}|}a_i\quad=\ 1$$

□

4.3.3.5 The Modified Version of the Scheduling Algorithm

Lemma 4.3.3.3 guarantees that the decomposition of G_u is correct for G_0, too, as both graphs have corresponding null-weight multicycles. Lemma 4.3.3.4 shows how to find a strictly or weakly separating hyperplane for a polyhedral edge: all edges of G_u which are derived from the same polyhedral edge of G_0 must be kept together during the decomposition, since only the complete set is a correct representation of the dependence polyhedron.

So the adapted scheduling algorithm is:

DARTE-VIVIEN (G, k)

1. Find the maximum null-weight multicycle G' of G.
2. Add all edges derived from a polyhedral edge $P(e)$ of G_0 to G', if at least one edge of G' was derived from $P(e)$.
3. Let G'' be G' plus all virtual vertices and all edges which originates at a virtual vertex.
4. Select a vector X^k and, for each vertex $v \in G$, a constant c_v^k such that:
$$\begin{aligned} &(\forall\, e \;:\; e \in G \setminus G'' &&:\; X^k\mathbf{w}(e) + c_{t(e)}^k - c_{s(e)}^k \geq 1) \\ &(\forall\, e \;:\; e \in G'' &&:\; X^k\mathbf{w}(e) + c_{t(e)}^k - c_{s(e)}^k \geq 0) \end{aligned}$$
5. If G' is empty or has only virtual vertices, RETURN.
6. If G' is strongly connected, G is not computable. STOP.
7. Decompose G' into its strongly connected components G'_i and call DARTE-VIVIEN$(G'_i, k+1)$ for each of them.

There are two differences to the first version of Section 4.2.10.

- Step 2 ensures that all edges derived from the same polyhedral edge are kept together. If one of them is part of the null-weight multicycle, for all others only weakly separating hyperplanes are computed. This is necessary, because only the complete set of edges is a correct transformation of the original polyhedron. For direction vectors, this step can be omitted.

- For edges starting at virtual vertices only weakly-separating hyperplanes are computed. This is sufficient, as virtual vertices are not scheduled and it is necessary in order to be able to deal with null-weight cycles around virtual vertices.

Now two linear programs have to be solved, one for finding G' and one for finding (and optimizing) X and $c_1, \ldots, c_{|V|}$. For the first linear program (4) or (2) can be used. For the second, linear program (6) can be adapted.

4.3.4 Example

The following program matches the SURE of Example 4.2.2. One virtual vertex must be inserted for the nonuniform dependence from Statement s_1 to Statement s_2 (direction vector $(0, 1, 0-)$). The transformed dependence graph for it is depicted in Figure 4.2.

for $i := 0$ **to** n **do**
    for $j := 0$ **to** n **do**
        for $k := 0$ **to** n **do**
            $a[i, j, k] := b(i-1, j-1, k) + a(i-1, j-1, k-1)$ $(*\ s_1\ *)$
            $b[i, j, k] := a(i, j-1, k+i) + b(i, j, k-3)$ $(*\ s_2\ *)$
        od
    od
od

Applying the modified Darte-Vivien-algorithm yields a slightly different (better) schedule than the one obtained for the equivalent SURE (Ex. 4.2.12). The reason is that no schedule for the virtual vertex is calculated. Therefore, it is no longer necessary to shift the schedule for statement s_2 to get a strictly separating hyperplane for the edge from v to b with weight $\mathbf{0}$.

$$
\begin{aligned}
\Theta(s_1, (i, j, k)) &= j \\
\Theta(s_2, (i, j, k)) &= (j, \tfrac{1}{3}k)
\end{aligned}
$$

4.3.5 Dependence Polyhedra versus Direction Vectors

There are two advantages of using arbitrary dependence polyhedra as opposed to direction vectors.

1. As the distance set is modelled more precisely, some null-weight multicycles are avoided. This may result in scheduling functions of lower dimensionality.
2. Schedules may have better latency if some distances not appear in the RDG.

Example

The following example demonstrates the first advantage.

for $i := 0$ **to** n **do**
  for $j := 0$ **to** n **do**
    for $k := 0$ **to** n **do**
      $a[i,j,k] := a[i-1, i+j, i+k]$ $(*\ s_1\ *)$
      $b[i,j,k] := b[i, j-1, k-2] + a[i,j,k]$ $(*\ s_2\ *)$
    od
  od
od

Three dependences are contained in the loop nest (two of them uniform):

Dep.	$P(E_\delta)$	direction vector
$s_1\ \delta_1\ s_1$	$\mathcal{P}_1 = \{(1,0,0)\},\ \ \mathcal{R}_1 = \{(0,-1,-1)\}$	$(1, 0-, 0-)$
$s_1\ \delta_2\ s_2$	$\mathcal{P}_2 = \{(0,0,0)\},\ \ \mathcal{R}_2 = \emptyset$	$(0,0,0)$
$s_2\ \delta_3\ s_2$	$\mathcal{P}_3 = \{(0,1,2)\},\ \ \mathcal{R}_3 = \emptyset$	$(0,1,2)$

The dependence graph derived from the direction vectors contains a null-weight multicycle including the self loops through virtual vertex v and statement vertex s_2 (Fig. 4.12). The dependence graph derived from the dependence polyhedra does not contain this null-weight multicycle, since no combination of self loops around v with weight $(0,-1,-2)$ exists.

Therefore, the polyhedral method results in a one-dimensional schedule, whereas the direction vector method results in a two-dimensional schedule:

direction vectors:
$$\Theta(s_1,(i,j,k)) = (i,0)$$
$$\Theta(s_2,(i,j,k)) = (i+1, \tfrac{1}{2}k)$$

dependence polyhedra:
$$\Theta(s_1,(i,j,k)) = i-j+k$$
$$\Theta(s_2,(i,j,k)) = i-j+k+1$$

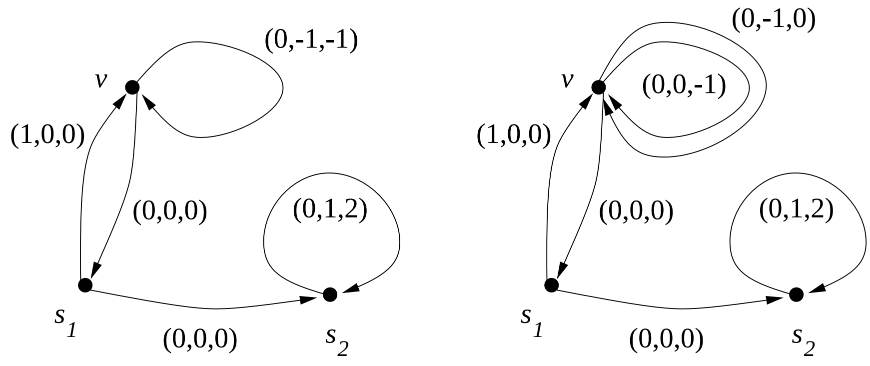


Fig. 4.12: Dependence graphs derived from dependence polyhedra (left side) and direction vectors (right side)

Another remarkable aspect of this example is the fact that the RDGs are not strongly connected. The above schedules are obtained by the direct method (Sec. 4.2.13.2). As we mentioned already in Section 4.2.13.2, this method may result in significantly slower schedules for some loop programs. The schedule for the hypergraph method is (for both dependence graphs):

$$\begin{array}{lrcl} \text{hypergraph method:} & \Theta(s_1,(i,j,k)) & = & (0,i) \\ & \Theta(s_2,(i,j,k)) & = & (1,\frac{1}{2}k) \end{array}$$

The latency of the latter schedule is $\frac{3}{2}n$. The latency of the direction vector schedule is $n * \frac{1}{2}n$, which is worse. The latency of the polyhedral method schedule is $3n$ and, therefore, within the latency of the other two schedules. But this schedule has the advantage of being also technically one-dimensional.

4.4 Extension to Imperfectly Nested Loop Programs

In order to schedule arbitrary loop programs, some extensions to our methods are necessary. One problem is that distance vectors may be of different lengths. The length depends on the number of loops common to source and target statement. But even if they are of the same length, the entries may refer to different loops.

A possible solution is to transform the loop program into an equivalent perfectly nested program and to schedule this program.

In [DV94] a (simple) transformation is presented and it is stated that the schedule for the perfect loop nest can be used as schedule for the original program without change. This is important, as the transformed programs are very inefficient and lengthy.

In [DV96] more sophisticated transformation techniques are suggested. For that reason, the transformed programs are faster and more compact and the schedule is applied to the transformed rather than to the original program. But no method is presented to do this transformation automatically.

In the LooPo environment, the scheduler is not allowed to change the source program. So our method is based on [DV94]. There, only a very short description is given. Moreover, the presented method is not fully correct. Thus, some corrections and additional theoretical background should be given in the following sections.

We modify the transformation of [DV94] slightly. Our programs have a (marginally) greater latency, but the transformation is more regular. Therefore, it is easier to prove some properties of it.

The basic idea is to enlarge the index range of each loop. Everything executed prior to a loop k is moved into an extra iteration with index $l(k) - 1$. Everything that will be computed after the loop has terminated is moved to an extra iteration with index $u(k) + 1$. So all statements are surrounded by all loops; they form the body of a perfect loop nest.

Since we are going to calculate with loop bounds, we have to normalize them in order to avoid problems arising from empty loops:

$$u'(k) \ = \ \max(l(k) - 1, u(k))$$

Every statement must be guarded by an **if** construct to make sure that it is executed only if the index lies within the original bounds.

For a loop program L with l loops and $|\Omega_L|$ statements, the transformed loop nest L' with index space I' is:

$$
\begin{array}{l}
\textbf{for } i'_1 := l(i'_1) - 1 \textbf{ to } u'(i'_1) + 1 \textbf{ do} \\
\quad \ddots \\
\quad \textbf{for } i'_l := l(i'_l) - 1 \textbf{ to } u'(i'_l) + 1 \textbf{ do} \\
\qquad \textbf{if} \;\; \textsc{Execute}\,(s_1, \mathbf{i}') \quad \textbf{then} \;\; s_1 \\
\qquad \quad \vdots \\
\qquad \textbf{if} \;\; \textsc{Execute}\,(s_{|\Omega_L|}, \mathbf{i}') \quad \textbf{then} \;\; s_{|\Omega_L|} \\
\quad \textbf{od} \\
\textbf{od}
\end{array}
$$

The guarding predicate Execute which ensures that each statement is executed only at the point, when it would have been executed in the original program, can be defined as follows:

$$
\begin{array}{l}
\textsc{Boolean Execute}\,(s, \mathbf{i}) \\
\quad \textbf{return} \;\; \begin{array}{lllll}
(\forall k : & k \in A(s) & : & i_k = u'(k) + 1 &) \wedge \\
(\forall k : & k \in P(s) & : & i_k = l(k) - 1 &) \wedge \\
(\forall k : & k \in C(s) & : & l(k) \le i_k \le u'(k) &)
\end{array}
\end{array}
$$

using the following sets:

$$
\begin{array}{lcl}
A(s) & = & \{\, k \,:\, 0 \le k \le l \,:\, \text{loop } k \text{ textually before } s \,\} \\
C(s) & = & \{\, k \,:\, 0 \le k \le l \,:\, \text{loop } k \text{ carries } s \,\} \\
P(s) & = & \{\, k \,:\, 0 \le k \le l \,:\, \text{loop } k \text{ textually after } s \,\}
\end{array}
$$

$A(s)$ represents the past of s, $C(s)$ the present and $P(s)$ the future.[2]

[2] A is for “ante”, C for “current” and P for “post”.

There is no need to actually perform the transformation. It just helps to extend the distance set to the new dimensionality. Since the depth of the perfectly nested program is equal to the number of loop indices of the whole program, this is also the size of the iteration and distance vectors in L'.

4.4.1 Definition

For each statement s, we define e_s to be the function that returns the entry k of $\mathbf{i}' \in I'$, which corresponds to an entry m of $\mathbf{i} \in I_s \subset \mathbb{Q}^d$.

$$e_s \;:\; \{1,\ldots,d\} \longrightarrow \{1,\ldots,l\} \;:\; m \longrightarrow k$$

i.e., e_s maps the entries of the short index vectors to their places in the expanded index vectors.

Next we define a transformation for the iteration vectors of a statement s of L to those of L':

$$t_s \;:\; I_s \longrightarrow \mathbb{Q}^l \;:\; \mathbf{i} \longrightarrow \mathbf{i}', \quad i'_k = \begin{cases} u'(k)+1 & k \in A(s) \\ l(k)-1 & k \in P(s) \\ i_m & k \in C(s) \;\wedge\; k = e_s(m) \end{cases}$$

Remark

e_s is injective and monotone, since all loops of L appear in L' in their original textual order.

Example

An illustration of the entry mapping e_S and the index vector transformation t_s is given in Figure 4.13 for an index vector of the statement in the following loop program. It consists of seven imperfectly nested loops (numbered $0\ldots6$). Let us divide the loops into the sets:

$$A(s) = \{0,1,4\}, \quad C(s) = \{2,3,5\}, \quad P(s) = \{6\}.$$

$$\left.\begin{array}{l}\textbf{for } a := 1 \textbf{ to } n \textbf{ do}\\ \quad \textbf{for } b := 1 \textbf{ to } m \textbf{ do}\\ \quad\quad \vdots\\ \quad \textbf{od}\\ \textbf{od}\end{array}\right\} A(s)$$

$$\left.\begin{array}{l}\textbf{for } i := 1 \textbf{ to } n \textbf{ do}\\ \quad \textbf{for } j := 1 \textbf{ to } n \textbf{ do}\\ \quad\quad \left.\begin{array}{l}\textbf{for } c := 1 \textbf{ to } 2 \textbf{ do}\\ \quad \vdots\\ \textbf{od}\end{array}\right\} A(s)\\ \quad\quad \textbf{for } k := 1 \textbf{ to } n \textbf{ do}\\ \quad\quad\quad (*\ \text{Statement } s\ *)\\ \quad\quad \textbf{od}\\ \quad \textbf{od}\\ \textbf{od}\end{array}\right\} C(s)$$

$$\left.\begin{array}{l}\textbf{for } d := 0 \textbf{ to } n \textbf{ do}\\ \quad \vdots\\ \textbf{od}\end{array}\right\} P(s)$$

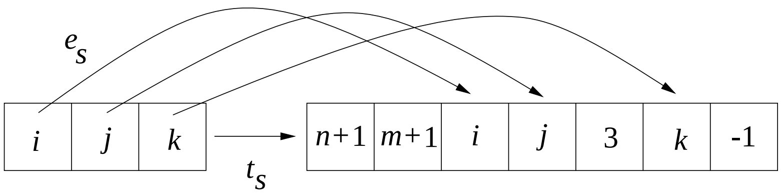


Fig. 4.13: Transformation of an index vector

4.4.2 Theorem

The original program L and the transformed program L' are equivalent, i.e., for all statements, the transformation t_s is an injective function with the following image:

$$(\forall\, s \ : \ s \in \Omega_L \ : \ t_s(I_s) \ = \ \{\, \mathbf{i}' \ : \ \mathbf{i}' \in I' \ : \ \text{EXECUTE}(s, \mathbf{i}') = \text{TRUE} \,\})$$

which preserves the execution order of all instances of statements. Formally:

$$(\forall\, s_i, s_j \ : \ s_i, s_j \in \Omega_L \ : \ (\forall\, \mathbf{i}, \mathbf{j} \ : \ \mathbf{i} \in I_{s_i}, \ \mathbf{j} \in I_{s_j} \ : \\ (s_i, \mathbf{i}) <_E (s_j, \mathbf{j}) \ \Rightarrow \ (s_i, t_{s_i}(\mathbf{i})) <_E (s_j, t_{s_j}(\mathbf{j})) \ ))$$

$(s_i, \mathbf{i}) <_E (s_j, \mathbf{j})$ means that instance $\mathbf{i}$ of s_i is executed before instance $\mathbf{j}$ of s_j.

PROOF

• t_s injective:
Let $\mathbf{i},\ \mathbf{j} \in I_s,\ \mathbf{i} \neq \mathbf{j},\quad \mathbf{i}' = t_s(\mathbf{i}),\ \mathbf{j}' = t_s(\mathbf{j})$.
$\Rightarrow\ (\exists\, m\ :\ 0 \le m \le d\ :\ i_m \neq j_m)\ \Rightarrow\ i'_{e_s(m)} \neq j'_{e_s(m)}\ \Rightarrow\ \mathbf{i}' \neq \mathbf{j}'$

• Image of t_s:
Define $I'_x = \{\ \mathbf{i}'\ :\ \mathbf{i}' \in I'\ :\ \text{EXECUTE}(s, \mathbf{i}') = \text{TRUE}\ \}$.

Let $\mathbf{i}' \in I'_x$. $\Rightarrow\ (\exists\, \mathbf{i}\ :\ \mathbf{i} \in I_s\ :\ \mathbf{i}' = t_s(\mathbf{i}))$
(according to the definitions of EXECUTE and t_s).

Let $\mathbf{i} \in I_s,\quad i' = t_s(\mathbf{i})\ \Rightarrow\ \text{EXECUTE}(s, \mathbf{i}') = \text{TRUE}$
(according to the definitions of EXECUTE and t_s).

• Preservation of the execution order:
Let $s_i,\ s_j \in \Omega_L,\quad \mathbf{i} \in I_{s_i},\ \mathbf{j} \in I_{s_j},\quad \mathbf{i}' = t_{s_i}(\mathbf{i}),\ \mathbf{j}' = t_{s_j}(\mathbf{j})$.

We assume that s_i is textually before s_j (otherwise just rename them).
Let d be the length of the distance vectors between s_i and s_j.

1. $\text{dist}(\mathbf{i}, \mathbf{j}) \neq \mathbf{0}\quad \Rightarrow (\exists\, m\ :\ 1 \le m \le d\ :\ i_m \neq j_m)$
   Let $k = e_{s_i}(m)\quad (= e_{s_j}(m))$
   $A(s_i) \subset A(s_j)$ and the first $m-1$ entries of $\mathbf{i}$ and $\mathbf{j}$ belong to the same loops and e_s is a monotone function
   $\Rightarrow$ the first $k-1$ entries of $\mathbf{i}'$ and $\mathbf{j}'$ are equal.
   (a) $\text{dist}(\mathbf{i}, \mathbf{j}) < \mathbf{0}\quad \Rightarrow (s_j, \mathbf{j}) <_E (s_i, \mathbf{i})$
   $$i'_k > j'_k\ \Rightarrow\ \mathbf{i}' > \mathbf{j}'\ \Rightarrow\ (s_j, \mathbf{j}') <_E (s_i, \mathbf{i}')$$
   (b) $\text{dist}(\mathbf{i}, \mathbf{j}) > \mathbf{0}\quad \Rightarrow (s_i, \mathbf{i}) <_E (s_j, \mathbf{j})$
   $$i'_k < j'_k\ \Rightarrow\ \mathbf{i}' < \mathbf{j}'\ \Rightarrow\ (s_i, \mathbf{i}') <_E (s_j, \mathbf{j}')$$
2. $\text{dist}(\mathbf{i}, \mathbf{j}) = \mathbf{0}$
   i.e., $(s_i, \mathbf{i})$ and $(s_j, \mathbf{j})$ share the same iteration of the loops common to both of them. $\Rightarrow\ (s_i, \mathbf{i}) <_E (s_j, \mathbf{j})$

   Let d_i be the depth of s_i and d_j that of s_j.

   $A(s_i) \subset A(s_j)$ and the first d entries of $\mathbf{i}$ and $\mathbf{j}$ belong to the same loops and e_s is a monotone function

   $\Rightarrow$ the first d entries of $\mathbf{i}'$ and $\mathbf{j}'$ are equal.

(a) $d_i > d \qquad$ let $k = e_{s_i}(d+1)$

$\Rightarrow\ i'_k < j'_k, \quad$ since $k \in A(s_j) \ \Rightarrow\ \mathbf{i}' < \mathbf{j}' \ \Rightarrow\ (s_i, \mathbf{i}') <_E (s_j, \mathbf{j}')$

(b) $d_j > d \qquad$ let $k = e_{s_j}(d+1)$

$\Rightarrow\ i'_k < j'_k, \quad$ since $k \in P(s_i) \ \Rightarrow\ \mathbf{i}' < \mathbf{j}' \ \Rightarrow\ (s_i, \mathbf{i}') <_E (s_j, \mathbf{j}')$

(c) $d_i = d_j$
$\Rightarrow\ \mathbf{i}' = \mathbf{j}' \ \Rightarrow\ (s_i, \mathbf{i}') <_E (s_j, \mathbf{j}'), \quad$ since the textual order of the statements is the same in L and L'. □

4.4.3 Distance Sets in L'

All distance vectors of L' are of the same length l. Starting with a distance set E_δ for L, the vectors are only of length less or equal to l. A method to extend E_δ to a distance set E'_δ of L' is needed:

$$E'_\delta \ = \ \left\{ \ \mathbf{d}' \ : \ \mathbf{d}' \in \mathbb{Z}^l \ : \ \mathbf{d}' = \mathrm{dist}(\mathbf{i}', \mathbf{j}') \ \wedge \ s_i(\mathbf{i}') \ \delta \ s_j(\mathbf{j}') \ \right\}$$

As L' is perfectly nested, the distance $\mathbf{d}'$ between two instances of statements is the difference of the iteration vectors:

$$\mathbf{d}' \ = \ \mathrm{dist}(\mathbf{i}', \mathbf{j}') \ = \ \mathbf{j}' - \mathbf{i}' \ = \ t_{s_j}(\mathbf{j}) - t_{s_i}(\mathbf{i})$$

For each entry d'_k of $\mathbf{d}'$, nine different cases may apply, since k can either be in $C(s)$, $A(s)$ or $P(s)$ for statement s_i or s_j. Refer also to Figure 4.14 for an illustration.

1. $k \in A(s_i) \ \wedge \ k \in P(s_j) \ : \quad d'_k \ = \ l(k) - 1 \quad - \quad u'(k) + 1$
2. $k \in A(s_i) \ \wedge \ k \in A(s_j) \ : \quad d'_k \ = \ u'(k) + 1 \quad - \quad u'(k) + 1$
3. $k \in A(s_i) \ \wedge \ k \in C(s_j) \ : \quad d'_k \ = \ j_m \quad - \quad u'(k) + 1$
4. $k \in P(s_i) \ \wedge \ k \in P(s_j) \ : \quad d'_k \ = \ l(k) - 1 \quad - \quad (l(k) - 1)$
5. $k \in P(s_i) \ \wedge \ k \in A(s_j) \ : \quad d'_k \ = \ u'(k) + 1 \quad - \quad (l(k) - 1)$
6. $k \in P(s_i) \ \wedge \ k \in C(s_j) \ : \quad d'_k \ = \ j_m \quad - \quad (l(k) - 1)$
7. $k \in C(s_i) \ \wedge \ k \in P(s_j) \ : \quad d'_k \ = \ l(k) - 1 \quad - \quad i_m$
8. $k \in C(s_i) \ \wedge \ k \in A(s_j) \ : \quad d'_k \ = \ u'(k) + 1 \quad - \quad i_m$
9. $k \in C(s_i) \ \wedge \ k \in C(s_j) \ : \quad d'_k \ = \ j_m \quad - \quad i_m$

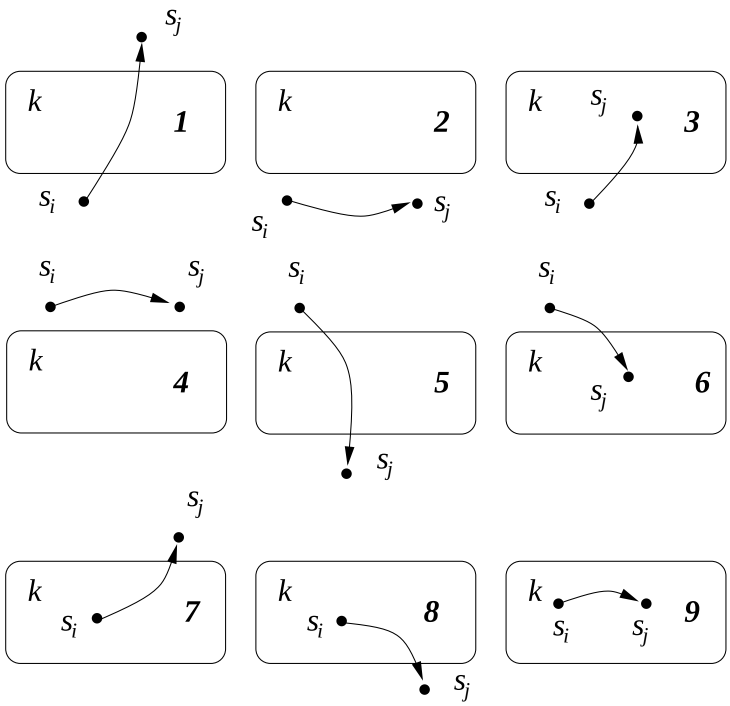


Fig. 4.14: Illustration of the different positions of an edge relative to a loop

4.4.3.1 Simple Extension

In most cases, d'_k is not just a single value but a range of values. Without considering the bounds $l(k)$ and $u'(k)$, entries for a direction vector can be given immediately:

Cases	entry of direction vector
2, 4	0
5, 6, 8	1+
1, 3, 7	$(-1)-$
9	d_m

For dependence polyhedra, corresponding rays have to be added to $\mathcal{R}$. Existing points and rays are extended with the functions e_{s_i}. New dimensions of points are filled with values of 0 (Cases 2,4), (-1) (Cases 1, 3, 7) or 1 (Cases 5, 6, 8). New dimensions of rays are filled with 0.

Remark

Cases 1 and 5 are exceptional for several reasons. They are missing in [DV94] and can, indeed, be ignored under some circumstances. We refer to the corresponding edges as **bypass edges**. Have a look at Figure 4.14 for the reason why.

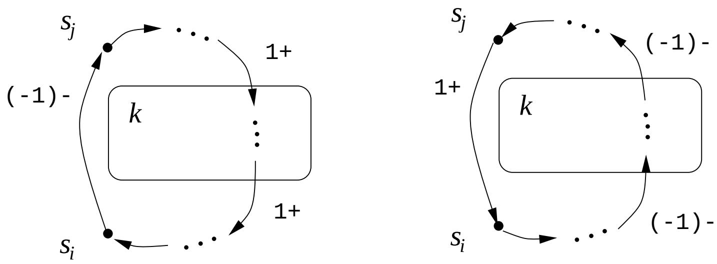


Fig. 4.15: Cycles through a loop traversing a bypass edge

Only if there is a cycle that uses a statement vertex inside the loop k *and* the bypass edge, it is necessary to set d_k to $1+$ or $(-1)-$ (shown in Figure 4.15). We can set $d_k = 0$ for all other bypass edges. Though inconsistent with the transformation, this does not affect the correctness of the schedule because the bypass edge is independent of loop k.

4.4.3.2 Optimized Extension

Since $l(k)$ and $u'(k)$ are defined by systems of inequalities, the extent of the distance set for dimension k can be calculated similarly to the dependence conversion for perfectly nested loops (Sec. 3.2.1, which is now Case 9 of our enumeration).

Let us demonstrate this for Case 8. The input is an h-transformation with source polytope $P(S_\delta)$ and target polytope $P(T_\delta)$. Then lower bound l_k and upper bound u_k of the extent are defined as follows ($m = e_{s_j}(k)$):

$$\begin{array}{rcl} l_k & = & \inf \lceil \{ \; u'(k) - i_m \; : \; \mathbf{i} \in P(S_\delta) \; \} \rceil + 1 \\ u_k & = & \sup \lfloor \{ \; u'(k) - i_m \; : \; \mathbf{i} \in P(S_\delta) \; \} \rfloor + 1 \end{array}$$

Note that $\{l(k), \ldots, u'(k)\}$ is the extent of I_{s_i} in direction m. $P(S_\delta) \subset I_{s_i}$ implies $l_k \geq 1$. Set entry k of each point to that value:

$$(\forall \, \mathbf{p} \; : \; \mathbf{p} \in \mathcal{P} \; : \; p_k = l_k)$$

The upper bound u_k may or may not exist: if it does not exist the extent is modelled as entry p_k of the points and ray e_k. If it does exist, the bounds of the extent are both modelled by the k-th entry of a point. Therefore, the point set must be doubled:

- $u_k = \infty$: add a new ray: $\mathcal{R} = \mathcal{R} \cup \{e_k\}$.
- $u_k < \infty$: let $\mathcal{P}_l = \mathcal{P}$ and $\mathcal{P}_u = \mathcal{P}$:

$$(\forall\, \mathbf{p} \;:\; \mathbf{p} \in \mathcal{P}_u \;:\; p_k = u_k)$$

  set $\mathcal{P} = \mathcal{P}_l \cup \mathcal{P}_u$.

If both bounds exist, the point set $\mathcal{P}$ is doubled to model them. Thus, $\mathcal{P}$ may become very large, if many indices are treated this way.

4.4.4 Adapting the Schedule to the Original Program

A schedule is a function of surrounding loop indices and structure parameters. Since in L' all statements are surrounded by all loops, a statement's schedule might depend on indices which are not in the loop nest around the same statement in L. If we want to use schedules of L' for L, we have to make sure that this case does not occur. Alternatively, we can provide a technique for eliminating these indices.

4.4.4.1 Definition

We call an index k **foreign** to s, if $k \in A(s) \cup P(s)$, i.e., if it does not carry the statement s in the original program L.

4.4.4.2 Lemma

If an RDG contains two self loops with weights

$$\mathbf{e}_k = (0, \ldots, 0, \underbrace{1}_{k}, 0, \ldots, 0) \quad \text{and} \quad -\mathbf{e}_k = (0, \ldots, 0, \underbrace{-1}_{k}, 0, \ldots, 0),$$

then $X_k = 0$ for all possible hyperplanes calculated by the Darte-Vivien algorithm.

PROOF

The hyperplane $X \in \mathbb{Q}^l$ must satisfy the following conditions in order to be weakly separating for both self loops:

$$\left.\begin{array}{rcrcrcl} X_k & - & c_v & + & c_v & \geq & 0 \\ -X_k & - & c_w & + & c_w & \geq & 0 \end{array}\right\} \Rightarrow \quad X_k = 0$$

□

4.4.4.3 Theorem

If DARTE-VIVIEN is only applied to strongly connected components of the RDG G of L' and if the non-optimized expansion of the distance set is used (Sec. 4.4.3.1), each optimal hyperplane X is free of indices foreign to statements scheduled with X.

PROOF (Sketch)

Let s be a statement vertex of strongly connected component $G' = (V', E')$ of G. Let $k \in A(s) \cup P(s)$.

1. $(\exists\, s_k \;:\; s_k \in V' \;:\; k \in C(s_k))$
   Since G' is strongly connected, there is a cycle through s and s_k. This cycle contains self loops weighted by the base vector $\mathbf{e}_k$ and its reverse $(-\mathbf{e}_k)$. Refer to Figure 4.15 for the two cases with bypass edges and to Figure 4.16 for the remaining two types of cycles.

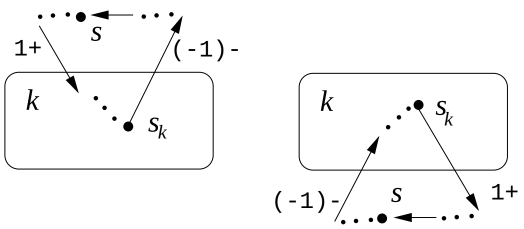


Fig. 4.16: Types of cycles through s and s_k without bypass edges

   According to Lemma 4.4.4.2 this implies $X_k = 0$.

2. $(\forall\, s' \;:\; s' \in V' \;:\; k \notin C(s'))$
   Loop k is foreign to all statements in G'. Cases 1, 2, 4, or 5 may apply for setting entry d_k
   $\Rightarrow$ $d_k = 0$ (refer to Remark 4.4.3.1 for Cases 1 and 5)
   $\Rightarrow$ for any optimum solution X: $X_k = 0$. □

4.4.4.4 Remark

The requirements of Theorem 4.4.4.3 are satisfied by the hypergraph scheduling method (Sec. 4.2.13.1) in combination with the simple extension method (Sec. 4.4.3.1). In this case, the schedule for L' can be used for L without any change (as assumed in [DV94]). If we use other variants, foreign indices may appear in a statement's schedule. Next, a method is presented to eliminate foreign indices.

4.4.4.5 Substituting Foreign Indices

A foreign index k in the schedule is not really a problem, since it always equals $l(k) - 1$ or $u'(k) + 1$ for all instances of the statement. So it is like a parameter rather than like an index. We can replace it by the minimum lower bound minus 1 respectively the maximum upper bound plus 1.

The substitution rule for a locally shifted linear schedule

$$\Theta_s \; : \; I_s \longrightarrow \mathbb{Q} \; : \; \mathbf{i} \longrightarrow X\mathbf{i} + c$$

can be defined as follows:

$(\forall\, x_k \; : \; X \in \mathbb{Q}^l, \; (x_k \neq 0 \;\wedge\; k \notin C(s)) \; :$

$$k \in A(s): \quad \begin{aligned} X_{subst} &= X \;-\; (0, \ldots, 0, x_k, 0, \ldots 0) \\ c_{subst} &= c \;+\; x_k\, (\lceil \min\{\, i_k \;:\; \mathbf{i} \in I_s \,\} \rceil - 1), \end{aligned}$$

$$k \in P(s): \quad \begin{aligned} X_{subst} &= X \;-\; (0, \ldots, 0, x_k, 0, \ldots 0) \\ c_{subst} &= c \;+\; x_k\, (\lfloor \max\{\, i_k \;:\; \mathbf{i} \in I_s \,\} \rfloor + 1)\;) \end{aligned}$$

As I_s is a parameterized polytope, the extremum in some direction might depend on parameters, too. Thus, it is possible that the result of the substitution is a piecewise affine mapping.

4.5 Implementation

In this section some comments are given on the LooPo implementation of the scheduler. The implementation does not follow the guidelines of [DV96], simply because most of the work was done before this paper appeared. Darte and Vivien try to avoid linear programming totally and thus propose an alternative method, which uses more of graph theory.

The LooPo implementation is based on linear programming (on the optimization tool PIP [FT90]), but some effort has been made to reduce the size of the optimization problems. First, we present our technique for speeding up the optimization and then we give an overview of the options available for controlling the scheduler.

4.5.1 Virtual Vertices Reconsidered

For each (nonuniform) dependence, a virtual vertex is added to the transformed dependence graph G_u. As for all other vertices, a constant is calculated for each scheduling dimension. All constraints for self-loops around a virtual vertex v with constant c_v are of the form:

$$X\mathbf{w}(e) + c_v - c_v \geq 0; \qquad \text{i.e.,} \qquad X\mathbf{w}(e) \geq 0.$$

If a hyperplane X satisfies this inequality for a self loop e with weight $\mathbf{w}(e)$, it satisfies the inequalities for all other self loops of same weight around other virtual vertices. The solution does not depend on the constant c_v. Therefore, a single virtual vertex carrying all self loops for rays of the dependence polyhedra is sufficient. We call this vertex the **generator vertex**.

We change the transformation of a polyhedral edge $P(e)$ from s_i to s_j defined by $(\mathcal{P}, \mathcal{R}, \emptyset)$ to the following procedure:

1. add an edge (s_i, s_j) with weight $\mathbf{p}$ for each point $\mathbf{p} \in \mathcal{P}$,
2. add a self loop (g, g) with weight $\mathbf{r}$ for each ray $\mathbf{r} \in \mathcal{R}$, if there is not already a loop with this weight around g.

Figure 4.17 illustrates this transformation.

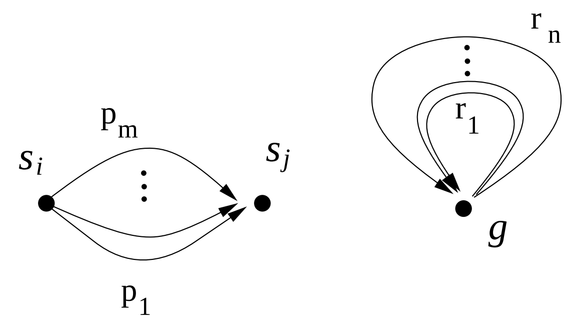


Fig. 4.17: Transformation of a polyhedral edge using the generator vertex

This transformation saves one vertex and one edge for each nonuniform dependence and possibly many equally weighted self loops. Thus, the size of the optimization problems is reduced.

Another advantage is that the transformation can be done on the fly during the decomposition. If a polyhedral edge is represented as an edge with multiple, alternative weights (one for each point in $\mathcal{P}(e)$), which is associated with a set of rays $\mathcal{R}(e)$, the PRDG can be used as scheduler input directly. This results in the following version of the algorithm:

Darte-Vivien (G, k) (PRDG version)

1. Build the ray index of $G = (V, E)$:
$$\mathcal{R}_{\text{Index}} = \bigcup_{e \in E} \mathcal{R}(e)$$
2. Build a temporary graph $G_{\mathcal{R}}$ with a single vertex g and self loops (g, g) for each element of $\mathcal{R}_{\text{Index}}$.
3. Find the maximum null-weight multicycle G' of $G \cup G_{\mathcal{R}}$.
4. Select a vector X^k and for each vertex $v \in G$ a constant c_v^k such that:
$$(\forall\, e \,:\, e \in G \cap G' \quad : \; X^k \mathbf{w}(e) + c_{t(e)}^k - c_{s(e)}^k \geq 1)$$
$$(\forall\, e \,:\, e \in G' \cup G_{\mathcal{R}} \quad : \; X^k \mathbf{w}(e) + c_{t(e)}^k - c_{s(e)}^k \geq 0)$$
5. If G' is empty, Return.
6. If G' is strongly connected, G is not computable. Stop.
7. Decompose $G \cap G'$ to its strongly connected components G'_i and call Darte-Vivien$(G'_i, k+1)$ for each of them.

4.5.2 Options Overview

In Figure 4.18, the scheduler's option box as realized in the LooPo frontend is shown (including references to the corresponding sections in this thesis). Two of the switches have not been described yet. So short explanations are given subsequently:

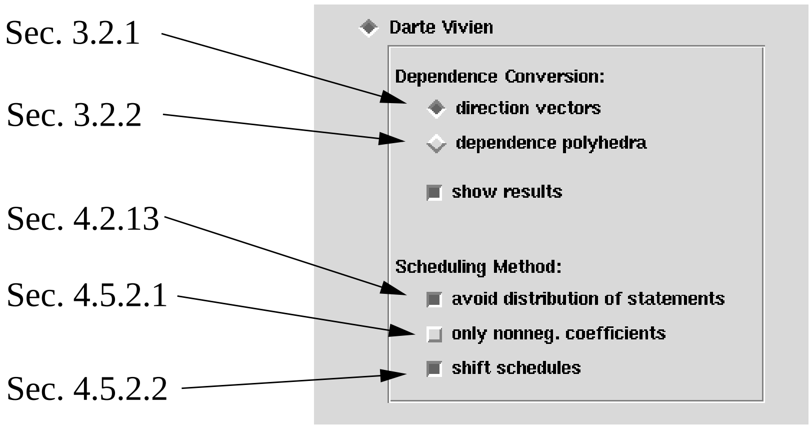


Fig. 4.18: Option box of the LooPo implementation

4.5.2.1 Only Nonnegative Coefficients

This option forces all hyperplanes to be nonnegative in each entry: $X \in (\mathbb{Q}^+)^n$. In combination with the hypergraph top level method, no parameters defining upper bounds appear in the schedule. This is useful for modules of LooPo that introduce parameters of that kind [Gei96].

The reason is that parameters appear only in the schedule if foreign indices are substituted (Sec. 4.4.4.5) or if the schedule is shifted (Sec. 4.5.2.2). The former does not occur for the hypergraph method (Theorem 4.4.4.3), the latter does not occur for parameters that appear only in upper bounds, provided all coefficients to indices are nonnegative. This option should not be enabled by default, as it is really a restriction.

4.5.2.2 Shifting the Schedules

Scheduling functions are expected to have strictly nonnegative results (Def. 4.1.1). The raw output of the Darte-Vivien scheduler does not satisfy this property, because the algorithm does not consider the index space.

As the index space is always bounded, it is possible to calculate shifting constants for each hyperplane in order to make sure that there are no negative results. A hyperplane X may be used for several statements. Thus, all occurrences of it must be shifted by the same amount.

The shifting constant c_X of a hyperplane X must satisfy the following implication:

$$(\forall\, s \;:\; s \in \Omega_L \;:\; (\exists\, k \;:\; \Theta(s,\mathbf{i}) = (\ldots, \underbrace{X}_{k\text{-th entry}}, \ldots)\,\mathbf{i} + (\ldots, \underbrace{c_s}_{k\text{-th entry}}, \ldots)) \Rightarrow (\forall\, \mathbf{i} \;:\; \mathbf{i} \in I_s \;:\; X\mathbf{i} + c_s - c_X \geq 0))$$

It can be calculated as follows (k defined as above):

$$c_X \;=\; \min\{\, t_k \;:\; s \in \Omega_L,\ \mathbf{i} \in I_{s\in\Omega_L} \;:\; \mathbf{t} = \Theta(s,\mathbf{i}) \,\}$$

Similarly to the substitution of foreign indices (Sec. 4.4.4.5), the constant c_X may depend on parameters. Thus, shifting may result in piecewise affine scheduling functions.

4.5.3 Speed

The LooPo project gives the opportunity to compare the Darte-Vivien scheduler with the Feautrier scheduler. The table below gives an impression of speed and results of both methods (Time values have been measured on a Sun SparcStation 5).

Name	Statements	Loops	Parameters	Dependences	nonuniform D.	perfectly nested	time DARTE-VIVIEN	time FEAUTRIER	same results
MATMULT	1	3	1	3	3	✓	0.3	0.6	✓
RET8D	5	3	2	11	6	–	0.4	8.7	✓
FLOYD	1	3	2	21	20	✓	5.5	59.7	✓
REFHULL	9	3	7	35	24	–	1.9	48:09.8	✓(*)

(*) up to a constant shift

Although both methods make use of linear programming, the Darte-Vivien scheduler clearly outperforms the Feautrier scheduler on complex input programs.

4.6 Comparison to Other Scheduling Algorithms

The last section has shown that the Darte-Vivien scheduler is actually faster than the Feautrier scheduler and that the results are often the same for both methods. Beside these empirical results, a comparison with other scheduling methods on a theoretical basis is of major interest. The next theorem states that the results are optimal relative to the dependence abstraction used.

4.6.1 Theorem (Vivien, Rao)

If d is the depth of the decomposition algorithm Check for an RDG G and if the index space contains an n-dimensional cube of width N, then a path of length kN^d can be found in the apparent dependence graph for some $k \in \mathbb{Q}^+$.

Proof [DV94, pp. 12–15]

4.6.2 Optimality for PRDGs

Since Check is used as a skeleton for all variants of our scheduler, the depth of the recursion corresponds to the maximum dimensionality of the affine schedules given to the statements of the program.

All instances involved in a path of the ADG must be computed sequentially. Therefore, the theorem implies that a valid schedule must have at least latency $O(N^d)$. As this is also the latency of a d-dimensional affine schedule, the Darte-Vivien algorithm yields — up to a constant factor — optimum results for a PRDG.

In [DV95a] the algorithm is compared in detail with the algorithm of Allen and Kennedy and the algorithm of Wolf and Lam. Both schedulers use a simpler dependence description and detect fewer parallelism for this reason. It is shown that Darte-Vivien generalizes these methods.

4.6.3 Limitations

Of course, the PRDG is not the most precise dependence description available. The longest path in the EDG may be much shorter than that in the ADG (Sec. 3.1.2). Therefore, a scheduler which uses precise dependence description is able to detect more parallelism in an input program. The method of Feautrier is based on h-transformations, which are as precise as the EDG. Example 4.19 demonstrates a limitation of our method:

for $i := 1$ **to** n **do**
  for $j := 1$ **to** n **do**
    $a[i, j] := b[i-1, j+i] + a[i, j-1]$
    $b[i, j] := a[i-1, j-i] + b[i, j-1]$
  od
od

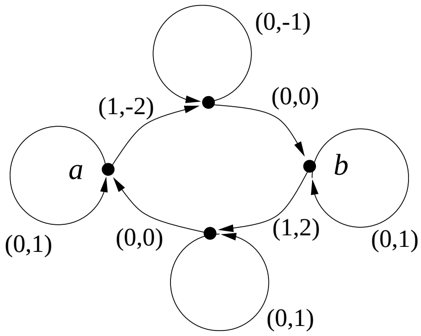


Fig. 4.19: Sample program for a limitation of the Darte-Vivien scheduler (with its transformed RDG)

The RDG contains a null-weight multicycle involving both statement vertices. Therefore, no linear schedule can be found. Darte-Vivien yields:

$$\begin{aligned}\Theta(a, (i, j)) &= (i, j)\\ \Theta(b, (i, j)) &= (i, j)\end{aligned}$$

But there is a linear schedule for the loop nest, which can be found with Feautrier's scheduler:

$$\begin{aligned}\Theta(a, (i, j)) &= \tfrac{3}{2}i + j + \tfrac{1}{2}\\ \Theta(b, (i, j)) &= \tfrac{1}{2}i + j\end{aligned}$$

5 The Dion-Robert Allocator

In the last chapter we gave an answer to the question: *when?* Now we concentrate on another question: *where?* An allocator assigns to each instance of a statement a processor of a parallel computer for executing it. In contrast to scheduling, no constraints for an allocation to be valid are imposed on the source program explicitly. Any operation may take place on any processor. But, as operations need input data and produce output, communication between processors may be necessary. Data must be requested from another processor, if it is not available locally. As such communications reduce the execution speed tremendously, the main goal of allocation is to reduce the communication volume. Therefore, variable accesses are the most import input to be considered (as dependences are for the scheduler). Thus, the *access graph* is the basic structure of the Dion-Robert allocator [DR94b, DR95]. This graph, which codes variable accesses, is an improvement of the communication graph as introduced by Darte and Robert in [DR93a].

The basic idea of this allocator is to place instances of statements *and* array elements in such a way that most data is available on the same processor or only a short distance away. But only the "most important" accesses are obliterated or made local. In general, it is impossible to make all accesses local or internal. Thus, the algorithm concentrates on those accesses, which can be made local most likely.

First, we describe the computer architecture which we assume to be the target machine. Then, after some basic definitions in Section 5.2, we formulate in Section 5.3 the *affine mapping problem* as the underlying problem of the allocator. As it will turn out that this problem is NP-complete, a heuristic is introduced in order to obtain a reasonable good solution quickly (Sec. 5.5). Some comments on the LooPo implementation (Sec. 5.7) complete this chapter.

5.1 The Architecture of the Parallel Computer

The target architecture is a **virtual processor grid** of dimensionality d. In this context *virtual* means that we do not assume a fixed amount of processors, but as many as needed in every direction. Every processor (depicted as a node in Figure 5.1) has its own local memory. Exchange of data is only possible by explicit communication between two processors. Communication may take place between any processors of the grid, but the resulting delay depends on the distance:

- **Neighborhood communication** is fast, because direct links exist between the processors.
- **Arbitrary communication** is slow, since routing across the processor field is necessary.

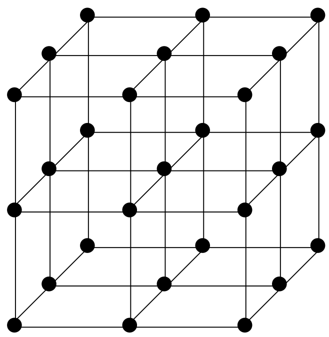


Fig. 5.1: three-dimensional processor grid

Although this computer model is somewhat artificial and simplified, the communication costs correspond with those of existing parallel computers. On a CM-5, for example, local communication is more than 30 times faster than nonlocal communication [DR95].

The advantage of this model is that it is easy to handle and (almost) independent of the concrete architecture of the target machine. It represents a class of machines called **distributed-memory parallel computers** (DMPCs).

But, as a consequence, some problems remain unsolved at allocation time. Subsequent compilation phases are necessary to map the virtual grid onto a real processor array and to establish correct communication schemes. A description of the techniques used by LooPo can be found for the first problem in [Sch97] and for the second in [Fab97].

5.2 Basic Definitions

5.2.1 Definition

A **statement allocation** is a function that assigns to each instance of each statement a place on an m-dimensional processor grid:

$$\Pi \ : \ \Omega_L \times I_{s \in \Omega_L} \longrightarrow \mathbb{Z}^m$$

We will frequently use the projection of Π to a single statement:

$$\Pi_s \ : \ I_s \longrightarrow \mathbb{Z}^m$$

A **data allocation** is a function that assigns to each element of an n-dimensional array x a place on an m-dimensional processor grid:

$$\Pi_x \ : \ \mathbb{Z}^n \longrightarrow \mathbb{Z}^m$$

As we want to deal with affine functions only, we get

$$\Pi_s(\mathbf{i}) \ = \ \lfloor M_s \mathbf{i} + \mathbf{c}_s \rfloor, \qquad M_s \in \mathbb{Q}^{m \times d}, \ \mathbf{c}_s \in \mathbb{Q}$$

as statement allocation for a statement s of depth d and

$$\Pi_x(\mathbf{a}) \ = \ \lfloor M_x \mathbf{a} + \mathbf{c}_x \rfloor, \qquad M_x \in \mathbb{Q}^{m \times n}, \ \mathbf{c}_x \in \mathbb{Q}$$

as data allocation for an array x of dimensionality n.

Remark

The Gaussian brackets in the previous definition allow several instances to be mapped to the same processor. Consider, for example, the following statement allocation for a statement surrounded by a single loop:

$$\Pi_s(i) \ = \ \lfloor \tfrac{1}{2} i \rfloor,$$

which maps pairs of subsequent instances to the same processor. We omit the Gaussian brackets in the following sections, as we already have done with those around the affine scheduling function (Def. 4.1.2). Again, subsequent modules must apply the Gaussian brackets in order to get the correct places.

5.2.2 Definition

An **access function** Γ is an affine function from a statement's index space to the data space of an array:

$$\Gamma \;:\; I_s \longrightarrow \mathbb{Z}^n \;:\; \mathbf{i} \longrightarrow F\mathbf{i} + \mathbf{f}$$

It describes a single access to an n-dimensional array in statement s (either read or write).

Remark

The definition of the access function Γ is not as general as the one in Section 2.5. Dion and Robert do not consider parametric accesses. $a[n-i]$ is, for example, not a valid access, for now. A method to integrate parameters is given in Section 5.7.1.

A loop program may contain many access functions. We could highlight the fact that Γ is the k-th access to variable x in statement s by the notation $\Gamma_{s,x}^{(k)}$. But this is wearisome to read. Thus, we simply assign index numbers to them relative to their textual order: $\Gamma_1, \Gamma_2, \ldots$.

5.2.3 Example

The following loop nest serves as an example throughout this chapter:

for $i := 0$ **to** n **do**
  for $j := 0$ **to** m **do**
    $a[i,j] := a[2i-1, j-1] + b[i-j+1, i+j, i-1]$ $(*\ s_1\ *)$
    for $k := 1$ **to** $n+m$ **do**
      $b[i,j,k] := a[i+k+2, j-k] + c[k-i]$ $(*\ s_2\ *)$
    od
  od
od

Access Functions

$$
\begin{array}{llcl}
\Gamma_1 : & F_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \mathbf{f}_1 = & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
\Gamma_2 : & F_2 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} & \mathbf{f}_2 = & \begin{pmatrix} -1 \\ -1 \end{pmatrix} \\
\Gamma_3 : & F_3 = \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} & \mathbf{f}_3 = & \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \\
\Gamma_4 : & F_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \mathbf{f}_4 = & \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\
\Gamma_5 : & F_5 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} & \mathbf{f}_5 = & \begin{pmatrix} 2 \\ 0 \end{pmatrix} \\
\Gamma_6 : & F_6 = \begin{pmatrix} 0 & -1 & 1 \end{pmatrix} & \mathbf{f}_6 = & 0
\end{array}
$$

5.2.4 Definition

With the previous definitions, a function can be defined for the **communication distance** Δ of an access Γ:

$$\Delta \ : \ I_s \longrightarrow \mathbb{Z}^m \ : \ \mathbf{i} \longrightarrow \Pi_s(\mathbf{i}) - \Pi_x(\Gamma(\mathbf{i}))$$

This is the distance in the processor grid between the place of an operation and the place of an array element which is read or written by this operation.

Remark

If we replace the allocation and access functions with their affine representations, we get:

$$\Delta(\mathbf{i}) \ = \ M_s\mathbf{i} + \mathbf{c}_s \ - \ M_x(F\mathbf{i} + \mathbf{f}) - \mathbf{c}_x$$

Now we divide the term into a linear part that depends on the iteration vector $\mathbf{i}$ and another, constant part:

$$= \underbrace{(M_s - M_xF)\mathbf{i}}_{\text{linear part}} + \underbrace{\mathbf{c}_s - M_x\mathbf{f} - \mathbf{c}_x}_{\text{constant part}}$$

The linear part is considered long-distance and irregular, as its values change for different $\mathbf{i}$. In contrast, the constant part is considered local and regular.

Therefore, the main goal is to *eliminate* the linear part, i.e., to render it a constant function of value $\mathbf{0}$. The following classification of distance functions is tailored according to this goal.

5.2.5 Definition

We distinguish three types of communications, which differ by their communication distance:

1. **internal communication:** (zero cost) $\Delta = \mathbf{0}$

$$M_s - M_x F = (\mathbf{0}) \;\wedge\; \mathbf{c}_s - M_x \mathbf{f} - \mathbf{c}_x = \mathbf{0}$$

2. **local communication:** (low cost) Δ constant

$$M_s - M_x F = (\mathbf{0}) \;\wedge\; \mathbf{c}_s - M_x \mathbf{f} - \mathbf{c}_x \neq \mathbf{0}$$

3. **affine communication:** (high cost) Δ affine

$$M_s - M_x F \neq (\mathbf{0})$$

Remark

Local communications form a superset of the neighborhood communications as defined in Section 5.1. Nevertheless, there are reasons for classifying them to be of low cost. One reason is that they are regular, i.e., location-independent (note the similarity with uniform dependences). Therefore, it is possible to treat them more efficiently than arbitrary communications.

Another reason is that short-distance communications may vanish or become neighborhood communications, when the virtual grid is folded onto the physical grid. The partitioning method supporting this is called *Locally Sequential Globally Parallel* [DR93b]: an area of neighboring virtual processors is folded onto a single real processor, and all operations that are placed to these processors are sequentialized. An example is depicted in Figure 5.2.

5.2.6 The Dimensionality of the Processor Grid

The key value for any allocation is the dimensionality m of the processor grid. $m = 0$ means that there is only a single processor. Therefore, no communication can occur but the execution order is purely sequential. As m is

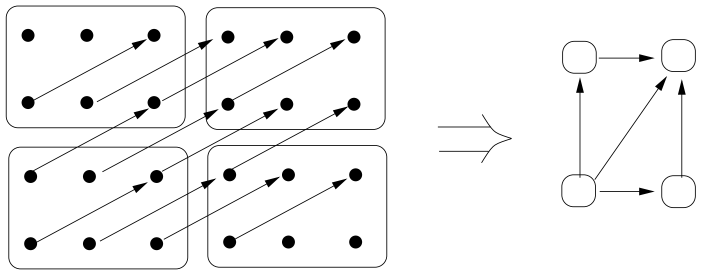


Fig. 5.2: Reducing local communications by partitioning

increased, the amount of possible parallelism grows, but also the communication volume and distance may grow. Thus, it is crucial to find the right value for m, and any communication reduction must be done according to a fixed dimensionality m. Otherwise, the result is always $m = 0$ which implies no parallelism at all.

Upper limits for m are the dimensionality of the physical target machine (m_{phys}) and the amount of detectable parallelism (π_s). The latter may vary for different statements, since it depends on a statement's nesting depth and schedule. To exploit all parallelism contained in the source program, we can choose the greatest π_s found.

The maximum meaningful dimensionality then is:

$$m_{\max} \;=\; \min(m_{\text{phys}}, \max(\pi_{s_1}, \ldots, \pi_{s_{|\Omega_L|}}))$$

5.2.7 Compatibility with the Schedule

An allocation is compatible with a schedule if no two operations, which are scheduled at the same time, are allocated to the same processor. Formally:

$$\begin{aligned}(\forall\, s_i,\ s_j \;:\; s_i,\ s_j \in \Omega_L \;:\; (\forall\, \mathbf{i},\ \mathbf{j} \;:\; \mathbf{i} \in I_{s_i},\ \mathbf{j} \in I_{s_j} \;:\\ \Theta(s_i,\mathbf{i}) = \Theta(s_j,\mathbf{j}) \;\Rightarrow\; \Pi(s_i,\mathbf{i}) \neq \Pi(s_j,\mathbf{j})\ ))\end{aligned}$$

In the general context of piecewise affine schedules, it is a complex task to achieve this. Thus, the state of the art is to calculate schedule and allocation independently and make them compatible afterwards [Wet95].

So the Dion-Robert allocator in its current implementation does not retrieve more from the schedule than its dimensionality.

5.3 The Affine Mapping Problem

Best would be a communication-free allocation. But one can show already for very simple loop programs (e.g., matrix multiplication [DR95]) that such an allocation does not exist for a grid of higher dimensionality than 0.

Thus, the Dion-Robert allocator does not concentrate on making communications internal but on minimizing the number of truly affine communications. So no distinction is made between local and internal communications. Refer to Section 5.6 for possibilities for making communications internal and not only local.

The goal of the method is to eliminate the linear parts of as many communication distances as possible for a given dimensionality m of the processor grid. Dion and Robert call this problem the *affine mapping problem* in [DR94b].

First, solution techniques for a single access are considered.

5.3.1 Solution for a Single Access

For a single access, the condition for eliminating the linear part of the distance function Δ is:

$$M_s - M_x F = (\mathbf{0}),$$

where M_s (M_x) is the matrix of the statement (data) allocation and F is the matrix of the access function.

As both M_s and M_x can be chosen freely, it is always possible to find a solution for this equation: for example set $M_s = (\mathbf{0})$ and $M_x = (\mathbf{0})$. But then the allocations are equivalent to the single processor solution $m = 0$.

We want the allocation function to use the full dimensionality of the processor grid. Therefore, the matrices M_s and M_x are required to be of rank m.

Let us summarize the requirements:

- $\text{rank}(M_s) = m, \quad M_s$ is (m, d)-matrix $\Rightarrow \; d \geq m$
- $\text{rank}(M_x) = m, \quad M_x$ is (m, n)-matrix $\Rightarrow \; n \geq m$
- $\text{rank}(F) = \min(d, n) \quad$ (F has maximum rank)

Thus, maximum rank allocation matrices cannot be found for all statements or arrays. The nesting depth d of a statement and the array dimensionality n must be at least as great as the dimensionality m of the grid.

The access matrix F is required to be of maximum rank in order to find maximum rank solutions for M_s and M_x (details follow).

Now, possibilities are considered for computing M_s or M_x, when either M_x or M_s are given. The appropriate method depends on the shape of the access matrix F and, therefore, on the ratio of nesting depth d and array dimensionality n.

Two lemmata about ranks of matrices are needed in order to understand the arguments, which are used for the different possible cases.

5.3.1.1 Lemma

Let M be an (m, d)-matrix of rank m and F be a (d, n)-matrix of rank d, where $m \leq d \leq n$. Then $\text{rank}(MF) = m$.

PROOF Appendix of [DR94b]

5.3.1.2 Lemma

Let M be an (m, d)-matrix of rank m and F be a (d, n)-matrix of rank n, where $m \leq n \leq d$. Then the equation $XF = M$ admits the following solution of rank m: $X = MF^{(l)}$.

PROOF Appendix of [DR94b]

5.3.1.3 The Different Cases

As we mentioned already, the shape of the access matrix F has great influence on how and, indeed, whether the equation can be solved. Let us list all possible cases:

- ***F* square:** (nesting depth equal to array dimensionality: $d = n$)
  In this case, F is invertible.

  M_x known: Let $M_s = M_x F$.
  M_s is of rank m due to Lemma 5.3.1.1

  M_s known: Let $M_x = M_s F^{-1}$.
  Again, M_x is of rank m, since F^{-1} is of rank m and hence Lemma 5.3.1.1 can be applied.

- ***F* flat:** (nesting depth greater than array dimensionality: $d > n$)
  In this case, F has a right inverse $F^{(r)}$.

M_x known: Let $M_s = M_x F$.
M_s is of rank m due to Lemma 5.3.1.1

M_s known: Though it is always possible to calculate $M_x = M_s F^{(r)}$, it is not assured to be of maximum rank. Consider, for example:

$$M_s = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \quad \text{and} \quad F = \begin{pmatrix} 0 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$$

$$\text{A right inverse of } F \text{ is } F^{(r)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix}$$

and hence $M_s F^{(r)} = \begin{pmatrix} 0 & 0 \end{pmatrix}$, which is of rank 0!

- ***F* narrow:** (nesting depth lower than array dimensionality: $d < n$)
In this case, F has a left inverse $F^{(l)}$.

M_x known: $M_x F$ may be of arbitrary rank and, therefore, it is not assured that a maximum rank solution for M_s exists. For example, just rename the sample matrices above:

$$M_x = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \quad \text{and} \quad F = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix}$$

Then $M_x F = \begin{pmatrix} 0 & 0 \end{pmatrix}$, which is of rank 0.

M_s known: $M_s F^{(l)}$ is a solution of rank m according to Lemma 5.3.1.2.

5.3.1.4 Summary

We summarize all the restrictions that we have to impose. First, the depth of the statement and the dimensionality of the array are both required to be greater than or equal to m. Moreover, the access matrix must be of maximum rank.

But even if these preconditions are fulfilled, it is not always possible to determine a maximum-rank matrix M_x, if M_s is known or a maximum rank matrix M_s, if M_x is known. For a flat access matrix we may not succeed in finding M_s from M_x and for a narrow access matrix we may not succeed in finding M_x from M_s.

All these restrictions are coded into the access graph, which is introduced in Section 5.4.

5.3.2 Complexity

As we have seen in the previous section, it is not easy to find the desired allocations even for a single access. Dion and Robert proved the NP-completeness for the complete set of maximum rank accesses.

5.3.2.1 Theorem

The affine mapping problem, i.e., the problem of finding maximum rank allocation matrices for a processor grid of dimensionality $m > 0$, such that the amount of local accesses is maximal, is NP-complete in the strong sense.

Proof [DR94b, pp. 16]

5.3.2.2 Remark

This theorem is not surprising, because similar results exist for other placement strategies. Darte and Robert, for example, showed the NP-completeness of the following problem: find allocations for statements and data in order to maximize the number of internal communications of a perfect loop nest with only uniform dependences [DR93a]. This seems to be simpler, since all access matrices are identity matrices, only the constant shifts differ. But the optimization goal is higher. The Dion-Robert allocator subsumes this method, as we are going to see in Section 5.6.2.

NP-complete does not mean unsolvable. An optimal solution could be found, for example, by means of linear programming. But in order to find a solution efficiently, it is necessary to develop a heuristic for the affine mapping problem. The solution obtained this way may not be optimal, but the allocation process is fast.

The main tool for our heuristic is the access graph, which is introduced in the next section.

5.4 The Access Graph

As we have seen in Section 5.3.1, it is not always possible to find a suitable maximum rank allocation matrix M_x (M_s) to localize the access distance according to a given matrix M_s (M_x).

The access graph illustrates these restrictions. As the rank of an access matrix must be greater than or equal to the dimensionality of the processor grid, the access graph also depends on m.

Its formal definition is as follows:

5.4.1 Definition

An **access graph** is a bipartite graph $G = (V, E, m)$, which represents a set of accesses of a loop program. It depends on the dimensionality m of the processor grid. The graph consists of the following vertices and edges:

- A vertex for each statement.
- A vertex for each array.
- An edge for each access Γ, provided that the rank of its matrix F is at least m and maximal. Let d be the depth of statement s and n the dimensionality of array x. Three cases must be distinguished:

  ***F* flat ($d > n$):** an edge pointing from x to s with weight F.

  $$s \xleftarrow{F} x$$

  ***F* narrow ($d < n$):** an edge pointing from x to s weighted by the left inverse of F.

  $$s \xrightarrow{F^{(l)}} x$$

  ***F* square ($d = n$):** an undirected edge between s and x with weight F.

  $$s \overset{F}{\text{———}} x$$

  This illustrates that both directions are possible either with weight F ($x \longrightarrow s$) or F^{-1} ($s \longrightarrow x$).

Remark

The number of edges in the access graph depends on the dimensionality m of grid. For $m = 0$, all accesses have a corresponding edge. Their number is reduced, as the dimensionality is increased. If m is greater than the rank of any access matrix, the access graph does not contain any edges at all. Thus, the following always holds:

$$(\forall m \ : \ m \in \mathbb{N}, \ G = (V, E, m), \ G' = (V, E', m + 1) \ : \ E' \subset E)$$

Therefore, the only difference between access graphs of different dimensionalities is the presence or absence of access edges.

5.4.2 Example

The access graphs of Example 5.2.3 are shown for grids of dimensionalities $m = 1$ and $m = 2$ in Figure 5.3. Remember that:

$$d_{s_1} = 2, \quad d_{s_2} = 3, \quad n_a = 2, \quad n_b = 3, \quad n_c = 1$$

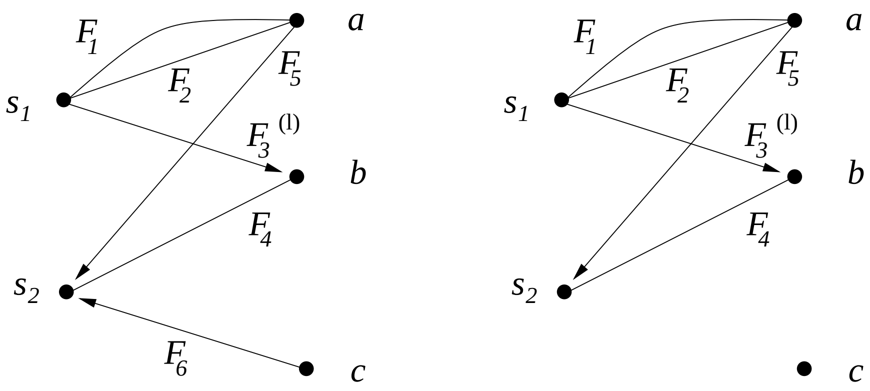


Fig. 5.3: Access graphs ($m = 1$ and $m = 2$) for Example 5.2.3

A placement to a three-dimensional processor grid is possible, too. Then Γ_4 is the only access to appear in the access graph.

5.4.3 The Structure of the Access Graph

Now we investigate the structure of the access graph and present methods for deriving allocations by means of the access graph.

Though access graphs are bipartite, all vertices can be treated the same way. The following definition is useful for abstracting from the different interpretations of vertices:

5.4.3.1 Definition

The **potential** p_a of a vertex a is either its depth (for a statement) or its dimensionality (for an array).

Remark

We know for each edge of the access graph: the target is of equal or higher potential then the source. Therefore, each path must have a similar property: The first vertex of the path is of lower or equal potential than the last vertex.

So all edge weights of edges $a \to b$ are flat (or square) matrices of size (p_a, p_b). We denote them all F, since we do not care for the individual type of weight (which can be either F, F^{-1} or $F^{(l)}$).

5.4.3.2 Definition

We redefine the **weight of a path** to the product of the edge weights rather than their sum. For a path π we get:

$$F_\pi = \prod_{e \in \pi} F_e$$

Then, the weight of a path from a to b is a (p_a, p_b)-matrix, too.

5.4.3.3 Principle for Calculating Allocations

For an edge from a vertex a to a vertex b of the access graph, it is possible to compute the allocation M_b of b from the allocation M_a of a and the edge's weight F. No distinction between statements and array vertices is necessary:

$$a \xrightarrow{F} b \qquad \Rightarrow \qquad M_b := M_a F$$

Thus, allocation matrices can be calculated along any directed path: first, assign to the start vertex a an allocation matrix M_a (e.g., the truncated identity matrix of size (m, p_a)). Then, subsequentially multiply the actual matrix with the weight of the following edge and set the allocation matrix of the target vertex to the result.

Remark

Each access whose edge is traversed in the previous way is made local. The next question is: how must the access graph be traversed in order to maximize the number of considered access edges. There are at least three structural aspects of the access graph that complicate this task: multiple, alternative paths, cycles and multiple incoming edges, even if they do not establish alternative paths. Let us discuss them in detail.

5.4.3.4 Multiple Paths

With the previous straight-forward computation of allocation matrices, only the accesses of a single path between two vertices are made local. Consider, for example, the two paths in Figure 5.4.

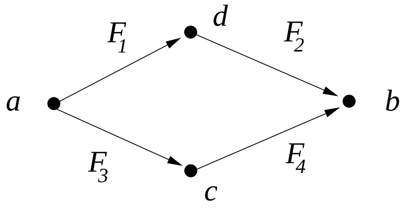


Fig. 5.4: Two alternative paths in an access graph

M_a must be set to a matrix that satisfies the following equation in order to make all accesses local:

$$\left.\begin{array}{rcl} M_b & = & M_a F_1 F_2 \\ M_b & = & M_a F_3 F_4 \end{array}\right\} \Rightarrow \; M_a F_1 F_2 = M_a F_3 F_4$$
$$\Leftrightarrow M_a(F_1F_2 - F_3F_4) = (\mathbf{0})$$

Generally, for two paths of weight F_{π_1} and F_{π_2} wet get:

$$M_a(F_{\pi_1} - F_{\pi_2}) = (\mathbf{0}) \quad \Leftrightarrow \quad (F_{\pi_1} - F_{\pi_2})^T M_a^T = (\mathbf{0})$$

The solution space for this homogeneous system of equations is the kernel of $(F_{\pi_1} - F_{\pi_2})^T$ [Str86]. The dimensionality of the kernel of a matrix is the number of its columns minus its rank. So we obtain the following condition for the possibility to find a matrix M_a of rank m which satisfies the equation above (p_a: potential of a):

$$p_a - \mathrm{rank}(F_{\pi_1} - F_{\pi_2}) \;\geq\; m$$

If this is true, the m columns of M_a^T can be set to the first m base vectors of the kernel $\ker(F_{\pi_1} - F_{\pi_2})^T$. So M_a is ensured to be of rank m.

Example

Consider the multiple paths in the main example (Ex. 5.2.3) of this section:

Path π_1: $s_1 \longrightarrow a \longrightarrow s_2 \longrightarrow b$

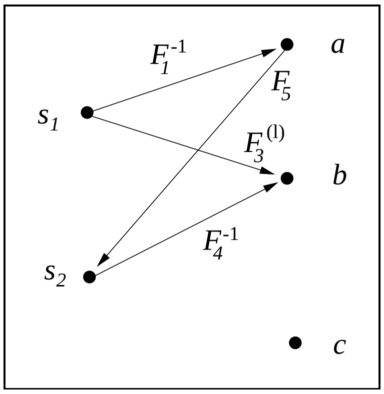
$$F_{\pi_1} = F_1^{-1} F_5 F_4^{-1} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

Path π_2: $s_1 \longrightarrow b$

$$F_{\pi_2} = F_3^{(l)} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

Rank of the difference:

$$\mathrm{rank}(F_{\pi_1} - F_{\pi_2}) = \mathrm{rank} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = 1$$



The vertex s_1 is of potential $p_{s_1} = 2$. Thus, only a one-dimensional matrix M_{s_1} can be found that localizes both accesses:

$$M_{s_1} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = (\mathbf{0}) \quad \Rightarrow \quad M_{s_1} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

5.4.3.5 Cycles

Starting at any vertex a of a cycle, the straight-forward method zeros out all distance matrices except the last one with target a (e.g., the edge with weight F_4 in Figure 5.5), because M_a was already set at the beginning.

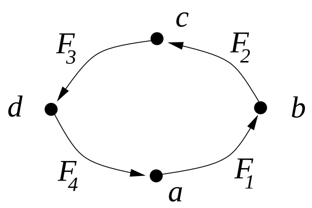


Fig. 5.5: Cycle in an access graph

The solution is quite similar to that of multiple paths: set M_a in a way to localize the preceding access in the cycle. The following equation must hold:

$$M_a = M_a F_{\mathcal{C}} \quad \Leftrightarrow \quad M_a(I - F_{\mathcal{C}}) = (\mathbf{0})$$

where $F_{\mathcal{C}}$ denotes the weight of a cycle $\mathcal{C}$ and I the identity matrix.

Remark

In fact, cycles in the access graph are only special cases of multiple paths. Since the potential of an edge's target is higher than or equal to the potential of its source, all vertices involved in a cycle must be of the same potential. Therefore, all edges of the cycle are undirected. So we can choose another vertex b of the cycle and orient the edges in such a way that there are two alternative paths between a and b (Fig. 5.6). Then, the solution technique of Section 5.4.3.4 can be applied.

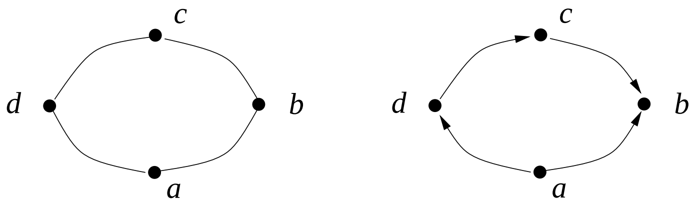


Fig. 5.6: Cycle and multiple paths traversing the same set of edges

5.4.3.6 Multiple Incoming Edges

Another problematic structure is a set of multiple incoming edges for a vertex a, even if they do not establish alternative paths (Fig. 5.7). The allocation matrix M_a is fixed when a is visited for the first time. Thus, normally only one of the edges can be considered. A possible solution is to reverse the other edge in order to take it into account.

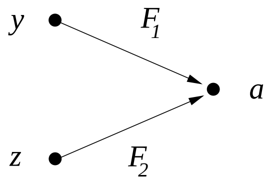


Fig. 5.7: Multiple incoming edges to a vertex

Note that it is not assured that a maximum rank allocation matrix can be found for the new target of the reversed edge, otherwise there would have been already an edge $a \to z$ in the access graph. So we cannot decide before we know the value of M_a whether it is possible to reverse the edge or not.

Let us assume that M_a of Figure 5.7 has been computed via edge $y \to a$: $M_a = M_y F_1$. In order the make Γ_2 local, the following equation must be satisfied by M_z:

$$M_z F_2 = M_a \quad \Leftrightarrow \quad F_2^T M_z^T = M_a^T$$

It is not guaranteed that an M_z exists which satisfies this equation. The condition is that the rank of the augmented matrix $(F_2^T | M_a^T)$ must not be higher than the rank of F_2^T [Usm87]:

$$\mathrm{rank}(F_2^T) \;=\; \mathrm{rank}(F_2^T | M_a^T)$$

The next lemma helps to show that — provided there is a solution for M_z^T — it has maximum rank m:

Lemma

For any two matrices $A \in \mathbb{Q}^{a\times b}$ and $B \in \mathbb{Q}^{b\times c}$ holds:

$$\mathrm{rank}(AB) \;\leq\; \min(\mathrm{rank}(A), \mathrm{rank}(B))$$

Proof [Str86, p. 201]

M_a (and hence M_a^T) is of rank m and, therefore, any valid solution M_z^T must be at least of rank m, according to the previous lemma. Since M_z is a (m, p_z)-matrix, any valid solution for M_z^T is exactly of rank m.

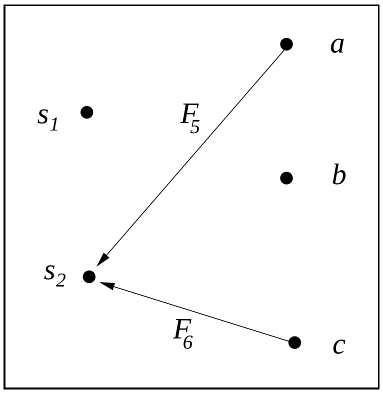
Remark

F_2^T is a narrow (p_a, p_z)-matrix of (maximum) rank p_z. The dimension of the kernel of a matrix is the number of its columns minus its rank. Therefore, F_2^T has only a null-dimensional kernel, which implies that the solution for M_z^T is unique, provided there is a solution.

Example

Consider the two edges pointing to vertex s_2 in the main example (Ex. 5.2.3) of this section. The dimensionality of the grid is $m = 1$.

Let $M_{s_2} = \begin{pmatrix} 0 & 1 & -1 \end{pmatrix}$
(derived via edge $a \rightarrow s_2$).



$$\begin{aligned} \operatorname{rank}(F_6^T) = \operatorname{rank}(F_6) &= 1 \\ \operatorname{rank}(F_6^T | M_{s_2}^T) = \operatorname{rank} \begin{pmatrix} 0 & 0 \\ -1 & 1 \\ 1 & -1 \end{pmatrix} &= 1 \end{aligned}$$

Thus, the following system of equations is solvable:

$$\begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} M_c^T = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

The solution is $M_c^T (= M_c) = (-1)$.

5.5 The Heuristic

In the previous section, only methods for partial solutions have been presented. The heuristic combines these methods in order to find allocations for a complete access graph systematically. It is a greedy heuristic, i.e., each allocation is only computed once and never changed afterwards.

As we have seen that problems occur from multiple paths or multiple incoming edges, the first step is to find a subgraph that does not contain any of these phenomena: a *branching* of G.

5.5.1 The Maximum Branching

5.5.1.1 Definition

A **branching** is an acyclic graph, in which no two edges pointing to the same vertex.

A **maximum branching** of a graph G is a subgraph that is a branching which maximizes the sum of its edge priorities (an **edge priority** is a positive number attached to an edge).

Remark

In a branching, each vertex r without incoming edges is the root of a tree. So it is possible to choose any maximum rank matrix M_r and subsequently calculate allocations for all other vertices in the graph. Thus, all accesses in a branching can be made local.

An efficient algorithm for finding a maximum branching is the one of Edmonds [EM92].

5.5.1.2 Choosing Priorities for the Edges

The edge set of the maximum branching can be controlled by means of the edge priorities. In order to get the maximum number of edges, the priority is set to 1 for each edge. A better objective is to find a branching with maximum communication volume. A simple but effective estimate of the communication volume of an access is provided by the rank of its access matrix. Therefore, we are going to use this rank as the priority for an edge.

5.5.1.3 Example

The maximum branching for our running example is shown in Figure 5.8 (showing also the priorities of the edges).

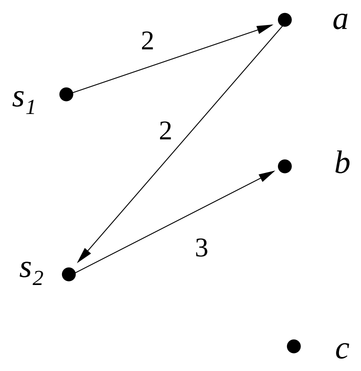


Fig. 5.8: Maximum branching for the access graph of Figure 5.3 (grid: $m = 1,\ m = 2$)

5.5.2 Remark

The number of edges in the maximum branching is limited to $|V|-1$. In order to make more accesses local, additional techniques are necessary. Therefore, the heuristic consists of several steps.

5.5.3 Root Vertex Heuristics

The allocations of the root vertices are the only remaining degree of freedom if all edges of the maximum branching are to be considered. We use the techniques discussed in Sections 5.4.3.4 and 5.4.3.6 for setting them to suitable values.

We only consider the outgoing edges of root vertices. The subset of edges which start at a root vertex and are not part of the maximum branching can be divided into two classes:

1. edges that point to a vertex of the same tree. These edges establish shortcuts in the tree.
2. edges that point to a vertex of another tree. They establish links between the separate trees.

We discuss heuristics for these two classes in the next two sections.

5.5.3.1 Shortcuts Inside a Tree

Shortcuts are alternative, single-edge paths to paths already present in the maximum branching. Thus, the techniques for multiple paths are suitable.

Let r be the root vertex of a tree of the maximum branching and let $\vec{E}$ be the set of outgoing edges of r that establish shortcuts in the tree. Let $F_{\pi_1}, \ldots, F_{\pi_k}$ be the corresponding weights of alternative paths for the k elements of $\vec{E}$. In

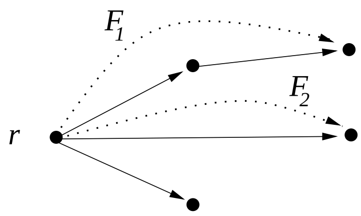


Fig. 5.9: Shortcut edges in a tree

order to consider all elements of $\vec{E}$, the following system of equations must have a maximum-rank solution:

$$\begin{bmatrix} M_r(F_{\pi_1} - F_1) & = & (\mathbf{0}) \\ \vdots & & \vdots \\ M_r(F_{\pi_k} - F_k) & = & (\mathbf{0}) \end{bmatrix}$$

It can be transformed to a single matrix equation:

$$M_r \underbrace{(F_{\pi_1} - F_1| \ldots |F_{\pi_k} - F_k)}_{:= F_{\vec{E}}} = (\mathbf{0}| \ldots |\mathbf{0})$$

We have shown in Section 5.4.3.4, that a maximum-rank solution exists if $\ker(F_{\vec{E}}^T)$ is at least of dimensionality m. Otherwise the following heuristic can be applied in order to get a subset of shortcut edges for which a maximum-rank solution exists:

1. Reduce $\vec{E}$ to a set $\vec{E}'$ which does not contain edges with equal weights. Assign a priority to each element of $\vec{E}'$ according to the number of edges in $\vec{E}$ which it represents.
2. Test for all edges $e \in \vec{E}'$ the dimensionality of $\ker(F^T_{\vec{E}'\setminus\{e\}})$.
3. Delete the edge with lowest priority out of the edges of $\vec{E}'$, for which the kernel in Step 2 has maximum dimensionality.
4. Repeat Steps 2 and 3 until the dimensionality of the kernel is at least m or $\vec{E}' = \emptyset$.

5.5.3.2 Links to Other Trees

Let $\hat{E}$ be the set of outgoing edges of root vertex r that point to readily allocated vertices of other trees. As we have seen in Section 5.4.3.6, it may be possible to reverse some of them by setting M_r to an appropriate value. We know that the allocation M_r is unique, provided there is a solution for

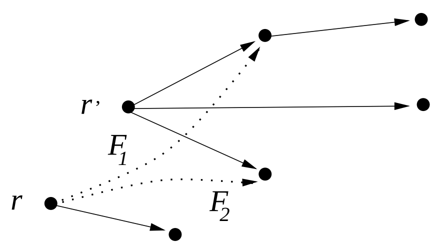


Fig. 5.10: Links to other tree

it. Therefore, it does not make sense to build a compound access matrix like $F_{\vec{E}}$ in the previous section in order to reverse more edges at once. Either the solution for M_r is the same for all reversals or the system is not solvable. In the former case it would have been enough just to consider one of the edges. So we use the following procedure:

1. Reduce $\hat{E}$ to a set $\hat{E}'$ which does not contain edges with equal weights pointing at vertices with equal allocation matrices. Assign a priority to each element of $\hat{E}'$ according to the number of edges in $\hat{E}$ which it represents.

2. Test all elements of $\hat{E}'$ by decreasing priority until a solution is obtained for one of them. Set M_r accordingly.

5.5.3.3 Summary: Allocating a Root Vertex

For each root vertex r the allocation matrix M_r is found according to the following procedure:

1. Try to localize some accesses related to shortcut edges (Sec. 5.5.3.1).

2. If M_r is not set yet, try to localize some accesses related to link edges (Sec. 5.5.3.2).

3. If M_r is not set yet and there are link edges to not allocated vertices, defer the allocation of M_r until these vertices are allocated.

4. Set M_r to a default value, if it has not been set by any of the preceding steps. We use the truncated identity matrix of size (m, p_r), where p_r is the potential of r.

5.5.4 The Algorithm

Now, we have discussed all steps of the heuristic. An overview of the complete algorithm is given in the box below:

Dion-Robert (G, m)

1. Build access graph $G = (V, E, m)$

2. Compute the maximum branching G' of G.

3. Try to find a suitable allocation matrix M_r for the root of a connected component of G'.

4. If Step 3 was successful, traverse the tree starting with r and compute allocation matrices of the visited vertices: set $M_b = M_a F$ for an edge $a \xrightarrow{F} b$.

5. Repeat Steps 3 and 4 until all vertices are allocated.

Remark

If the allocation of some root vertices is deferred, a cyclic wait condition may occur: some root vertices wait for each other to be allocated. An implementation must be able to detect and resolve such a deadlock. This can be done by a progress check in Step 4: if no vertices have been allocated during a complete scan of the list of root vertices, the first deferred vertex is set to the default allocation when Step 3 is executed the next time.

5.5.5 Example

We demonstrate the algorithm on Example 5.2.3 for grids of dimensionalities $m = 1$ and $m = 2$. The maximum branching (the same for both values of m) is depicted in Figure 5.8. Root vertices are s_1 and c.

Allocation to a grid of dimensionality $m = 1$.

- Select s_1 as the root vertex: Outgoing edges that establish shortcuts in the maximum branching are $s_1 \xrightarrow{F_3^{(l)}} b$ and $s_1 \xrightarrow{F_2^{-1}} a$. We have found in Section 5.4.3.4 a solution for M_{s_1} that makes the former access local:
$$M_{s_1} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$
This localizes $s_1 \xrightarrow{F_2^{-1}} a$, too.
- Traverse the tree of s_1 in the maximum branching:

$$M_a = M_{s_1} F_1^{-1} = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$
$$M_{s_2} = M_a F_5 = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & -1 \end{pmatrix}$$
$$M_b = M_{s_2} F_4^{-1} = \begin{pmatrix} 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & -1 \end{pmatrix}$$

- Select c as the root vertex: The only outgoing edge of c is $c \xrightarrow{F_6} s_2$, which is not in the maximum branching. It is possible to reverse it, as we have seen in Section 5.4.3.6:
$$M_c = (-1)$$

Allocation to a grid of dimensionality $m = 2$.

- Again, select s_1 as the root vertex. Neither Γ_2 nor Γ_3 can be made local, since the kernel dimensionality of the difference of the path weights is not sufficient (Sec. 5.4.3.4). Therefore, set M_{s_1} to the default:
$$M_{s_1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$
- Traverse the tree of s_1 in the maximum branching:
$$\begin{aligned}
M_a = M_{s_1} F_1^{-1} &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} &&= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
M_{s_2} = M_a F_5 &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} &&= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} \\
M_b = M_{s_2} F_4^{-1} &= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} &&= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}
\end{aligned}$$
- c has potential 1 and, therefore, cannot be placed onto a two-dimensional grid.

A summary of the results is presented in the table below:

$m = 1$	$m = 2$
localized: $\Gamma_1, \ldots, \Gamma_6$	localized: $\Gamma_1, \Gamma_4, \Gamma_5$
Allocations: $\Pi(s_1,(i,j)) = j$ $\Pi(s_2,(i,j,k)) = j-k$ $\Pi_a(x_1,x_2) = x_2$ $\Pi_b(x_1,x_2,x_3) = x_2 - x_3$ $\Pi_c(x) = -x$	Allocations: $\Pi(s_1,(i,j)) = (i,j)$ $\Pi(s_2,(i,j,k)) = (i+k, j-k)$ $\Pi_a(x_1,x_2) = (x_1,x_2)$ $\Pi_b(x_1,x_2,x_3) = (x_1+x_3, x_2-x_3)$ $\Pi_c(x)$ —

5.5.6 Comparison with the Original Heuristics

The heuristic presented is not the same as the one proposed by Dion and Robert, though the underlying ideas are taken from [DR94b]. There, the maximum branching plus multiple-path and cycle techniques are combined to the *simple heuristic*. It becomes the *refined heuristic*, when multiple incoming edges are handled, too.

For the simple heuristic, no concrete algorithm is presented. So we reduce it to the outgoing edges of root vertices in order to preserve the greedy, straight-forward character of the maximum branching.

Multiple incoming edges are treated differently than in Section 5.4.3.6. Virtual edges are inserted in the access graph in order to mark the possibility of localizing two access edges pointing to the same vertex.

5.5.7 Towards an Optimal Solution

Of course, a heuristic is never optimal. Better heuristics are likely to exist. Thus, it would be interesting to have an algorithm for the optimal solution in order to detect weaknesses of a heuristic or possible improvements. But such an algorithm has to solve large optimization problems. The most tricky problem, when linear (integer) programming should be used, seems to be that the matrices must be of maximum rank.

5.6 Internalizing Communications

Dion and Robert concentrate on making communications local. No distinction is made between local and internal communications. But as local communications may still cause overhead (of an amount that depends on the real target architecture), it is desirable to internalize as many of them as possible.

The allocations obtained by the original method [DR94b, DR95] are purely linear. No constant shifts are calculated. So one possibility is to look for constants afterwards that maximize the amount of internal communications. Another possibility is to modify the method in such a way that it internalizes rather than localizes communications. Both approaches are discussed in the next two sections.

5.6.1 Shifting Allocations

Shift constants for statement and data allocations should be determined in order to make some of the local communications really internal. We assume that all allocation matrices are computed already and that $\Gamma_1, \ldots, \Gamma_\gamma$ are the γ accesses which are now local. The communication distance for a local access Γ_α to variable x_α in statement s_α is:

$$\Delta_\alpha = \mathbf{c}_{s_\alpha} + M_{x_\alpha}\mathbf{f}_\alpha - \mathbf{c}_{x_\alpha}$$

Δ_α is a vector of size m for all α. We define D to be a (m, γ)-matrix, whose α-th column is Δ_α:

$$D = \left(\Delta_1 \,\middle|\, \cdots \,\middle|\, \Delta_\gamma \right)$$

The objective of the optimization is to make as many columns of D as possible equal to $\mathbf{0}$. To achieve this, we look for the (lexicographic) minimum of a vector $\mathbf{d}$ defined as follows:

$$\min \{ \, \mathbf{d} \; : \; \mathbf{d} \in (\mathbb{Q}^+)^\gamma \; : \; (\forall \, \alpha \; : \; 0 \le \alpha \le \gamma \; : \; d_\alpha = \sum_{i=1}^{m} |D_{i,\alpha}| \,) \, \}$$

The optimization tool PIP is able to solve a linear program of this type. We obtain a set of constants $\mathbf{c}_{s_1}, \ldots, \mathbf{c}_{s_{|\Omega_L|}} \in \mathbb{Q}$ and $\mathbf{c}_{x_1}, \ldots, \mathbf{c}_{x_a} \in \mathbb{Q}$, where a is the number of arrays in the program.

Example

We demonstrate the calculation of shift constants for Example 5.2.3 and for a one-dimensional grid. Note that $\mathbf{d} = |D|$ is this case.

- **Variable *a*:** $M_a = \begin{pmatrix} 0 & 1 \end{pmatrix}$, $\quad M_a \mathbf{f}_1 = 0, \quad M_a \mathbf{f}_2 = -1, \quad M_a \mathbf{f}_5 = 0$
- **Variable *b*:** $M_b = \begin{pmatrix} 0 & 1 & -1 \end{pmatrix}$, $\quad M_b \mathbf{f}_3 = 1, \quad M_b \mathbf{f}_4 = 0$
- **Variable *c*:** $M_c = \begin{pmatrix} -1 \end{pmatrix}$, $\quad M_c \mathbf{f}_6 = 0$

Thus, without any shift, we have the following access distances:

$$D = (0, -1, 1, 0, 0, 0)$$

Access Γ_1 is already internal. Γ_2 cannot be made internal as this would displace Γ_1, which is to the same array in the same statement. Γ_3 gets internal by setting $c_{s_1} = -1$. c_a can be adapted to keep the distance of the accesses of s_1. So we get:

$$c_{s_1} = -1, \quad c_{s_2} = 0, \quad c_a = 1, \quad c_b = 0, \quad c_c = 0$$

The optimal access distances are: $D_0 = (0, -1, 0, 0, 0, 0)$.

5.6.2 Homogeneous Accesses

For some parallel computer architectures even local communications may be too expensive. So the goal should be to really eliminate a communication and not only to make it local.

The original method can be adapted to this objective in an easy way: by using *homogeneous representations* of the affine functions rather than only the linear parts. In these functions, the constant part is integrated as the highest dimension of the linear part. So accesses, statement and data allocations look as follows:

$$\begin{array}{rcl} \Pi_s : \mathbb{Z}^{d+1} \longrightarrow \mathbb{Q}^m & : & \Pi_s(\mathbf{i}) = M'_s\mathbf{i} \\ \Pi_x : \mathbb{Z}^{n+1} \longrightarrow \mathbb{Q}^m & : & \Pi_x(\mathbf{a}) = M'_x\mathbf{a} \\ \Gamma : \mathbb{Z}^{d+1} \longrightarrow \mathbb{Z}^{n+1} & : & \Gamma(\mathbf{i}) = F'\mathbf{i} \end{array}$$

Then, the communication distance Δ simplifies to:

$$\Delta(\mathbf{i}) = M'_s\mathbf{i} - M'_xF'\mathbf{i}$$

So the allocation algorithm is looking for internal communications only:

$$M'_s - M'_xF' = (\mathbf{0}) \quad \Rightarrow \quad \Delta(\mathbf{i}) = \mathbf{0}$$

Compared to the original definitions (Def. 5.2.1), the functional matrices have the following contents:

$$M'_s = \left(\begin{array}{c|c} M_s & \mathbf{c}_s \end{array}\right), \quad M'_x = \left(\begin{array}{c|c} M_x & \mathbf{c}_x \end{array}\right), \quad F' = \left(\begin{array}{c|c} F & \mathbf{f} \\ \hline \mathbf{0} & 1 \end{array}\right)$$

As M_s and M_x are of rank m, also M'_s and M'_x are of rank m. Also the homogeneous access matrix F' is of maximum rank if F is. Thus, all access edges present in the non-homogeneous graph appear in the access graph for the homogeneous matrices.

It is even possible to consider more accesses, as the rank of F' may be higher than that of F. Consider the following example:

Example

This fragment of a loop program is to be placed onto a two-dimensional grid:

for $i := 0$ **to** n **do**
    $a[i, i-2] := \ldots$
od

$$F = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \qquad F' = \begin{pmatrix} 1 & 0 \\ 1 & -2 \\ 0 & 1 \end{pmatrix}$$

$$\mathrm{rank}(F) \ = 1 < m, \quad \mathrm{rank}(F') = 2 \geq m$$

The rank of F' is sufficient and, therefore, allocation matrices M'_s, M'_x of rank m can be found:

$$M'_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \Rightarrow M'_s = M'_x F' = \begin{pmatrix} 1 & 0 \\ 1 & -2 \end{pmatrix}$$

The allocation matrix $M_s = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is not of rank 2 (which is impossible for a statement of depth 1). Nevertheless, an allocation has been found. This is useful for imperfectly nested loops, since allocations are calculated for more statements. A more general solution for statements of insufficient depth is presented in Section 5.7.2.

We can check that the communication is really internal now:

$$\begin{aligned} \Delta(\mathbf{i}) &= M_s i + \mathbf{c}_s - M_x(Fi + \mathbf{f}) - \mathbf{c}_x \\ &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} i + \begin{pmatrix} 0 \\ -2 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \left(\begin{pmatrix} 1 \\ 1 \end{pmatrix} i + \begin{pmatrix} 0 \\ -2 \end{pmatrix} \right) + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} i \\ i-2 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ i-2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{aligned}$$

Differences to the Purely Linear Approach

At first sight, the use of homogeneous accesses seems to be as good or even superior to the use of the linear accesses. The access graph and, therefore, its maximum branching look the same or even form a superset of the original graph. But the matrices are more complex and, hence, chances of finding allocations which zero out more of them at once are reduced.

5.7 Implementation

Here, two problems are discussed which arise in the LooPo context: parametric access functions and allocations for statements with nesting depth lower than the dimensionality of the processor grid. An options overview concludes this section.

5.7.1 Parametric Access Functions

In this chapter, all affine functions have been parameter-less. But the initial definition of an affine function (Def. 2.2.5) allows parametric functions. Thus, a method for handling parameters is needed. The idea is to treat them as indices of additional loops around the statement. So matrices for statement allocations and accesses grow by p columns. But we do not generally extend the index space by the complete set of p parameters. There are two arguments against it:

1. The potential of all statements would be increased to $d + p$, which is likely to exceed the dimensionality of any array. Thus, all edges would point from an array vertex to a statement vertex and all paths would have only length 1.
2. Narrow access matrices may not be of maximum rank anymore, when they get expanded to width $d + p$.

The consequences of the first argument to the allocation are not certain. This may or may not decrease the number of localized accesses. But the second argument definitely results in allocations of reduced quality. This unwanted effect can occur, even if only the minimum set of parameters is included, as the following example demonstrates.

Example

Consider the two accesses in the following statement of depth $d = 1$. As the parameter n occurs in the second access, the depth is increased to $d' = 2$.

for $i := 0$ **to** n **do**
  $a[i, i] := a[n - i, i - 1]$ $F_1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}, \quad F_2 = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}$
od

The rank of the first access is too low for the new depth, while the second, parameterized access can be considered.

5.7.2 Statements with Insufficient Depth

Only accesses with rank at least the dimensionality m of the processor grid are considered in the access graph. Therefore, statements of depth lower than m are always represented by isolated vertices. No allocation matrix of rank m can be found for them either. But every statement must have an allocation. So one possibility is to give a default allocation to the statement which does not take any of its accesses into account.

A better method is to build a new access graph for an $(m-1)$-dimensional grid with the not allocated statements and arrays. So accesses of dimensionality $m-1$ may be considered for allocating these statements and arrays.

This procedure can be repeated with decreasing dimensionality of the grid until all statements have allocations. Arrays are permitted not to have an allocation, which means that subsequent modules are free to allocate them to wherever they want.

Allocations with lower dimensionality can always be interpreted as one layer (more generally: linear subspace) of a higher-dimensional grid. Thus, if all allocations are required to be of same dimensionality, they can be filled with zero rows if needed. The box below gives an overview of this top-level algorithm:

```
DION-ROBERT-TOP(G, m_0)

    m := m_0
    REPEAT
        DION-ROBERT (G, m)
        m := m - 1
    UNTIL all statement allocations are computed
```

Example

One-dimensional and two-dimensional allocations for Example 5.2.3 have been presented in Example 5.5.5.

Now, let us try an allocation to a three-dimensional grid. The only access of rank 3 is Γ_4. Thus, the access graph and its maximum branching have only a single edge with weight F_4. Allocations are determined only for s_2 and b:

$$M_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad M_{s_2} = M_b F_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

M_{s_1} is not determined yet, so the access graph is constructed for $m = 2$. It contains only accesses to array a, as the allocation of b is fixed already. We have already stated in Example 5.5.5 that it is not possible to localize both accesses from s_1 to a for a grid of dimensionality $m = 2$. So just Γ_1 is made local. Allocation matrices for s_1 and a are:

$$M_a = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad M_{s_1} = M_a F_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

No allocation for c is calculated.

Thus, only accesses Γ_1 and Γ_5 are localized.

5.7.3 Options Overview

In Figure 5.11, the allocator's option box as realized in the LooPo frontend is shown (including references to the corresponding sections in this thesis).

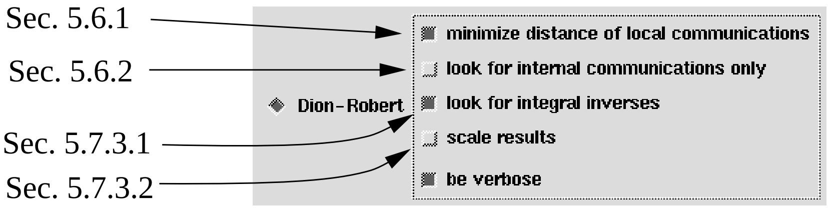


Fig. 5.11: Option box of the LooPo implementation

Short explanations are given for two options that have not been described so far.

5.7.3.1 Optimizing Inverses

We have used the following matrix as left inverse of F_3 in our example:

$$F_3^{(l)} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

As mentioned already in Section 2.2.3, left inverses are not unique. The following matrix is a left inverse of F_3, too:

$$\hat{F}_3^{(l)} = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ -\frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}$$

It is even more likely that standard techniques of linear algebra (e.g., the Gauss-Jordan algorithm) result in $\hat{F}_3^{(l)}$ rather than in $F_3^{(l)}$. As we prefer integral coefficients for allocation functions, we try to optimize left inverses by linear integer programming:

$$\min\{\ ||F^{(l)}||\ :\ F^{(l)} \in \mathbb{Z}^{n\times m},\ F \in \mathbb{Q}^{m\times n}\ :\ F^{(l)}F = I\ \}$$

If there is no solution for this minimization problem, we switch back to the Gauss-Jordan algorithm.

5.7.3.2 Scaling Allocations

Some subsequent modules may have problem with fractional results of the allocation function. Then, this option is useful, since it scales all allocations to integral coefficients.

6 Conclusion

We have presented the two partial solutions of the parallelization step of an automatically parallelizing compiler. A time step and a place is assigned to every operation, when schedule and allocation are fixed.

Basically, the two methods meet the requirements which we have postulated at the beginning. They accept imperfectly nested loop programs with affine accesses and dependences. The scheduler is reasonably fast and could even be accelerated by using the implementation strategies of [DV96]. The allocator is very fast in its current implementation. Thus, there is room for a refined, more expensive heuristic.

The scheduling results are nearly optimal for dependence polyhedra. Great improvements are only possible by the use of a more precise dependence abstraction. Small improvements are possible for the top level of the algorithm (hypergraphs or direct scheduling) and for imperfectly nested loops.

The allocation results can be improved by a better heuristic. They also should be improved in order to justify the costs required for the code generation: affine statement *and* data allocation. Possible starting points for improvements are:

- An adaption of the techniques that are used for the root vertices to every vertex visited.
- A more flexible access graph. The current graph is constructed according to worst-case predictions. Everything which may cause trouble is disregarded. So more edges might be added to the graph. Then it must be decided if an edge can be used at the time when the allocation of its source or target is known.

The Darte-Vivien scheduler is more advanced than the Dion-Robert allocator. As both are important to the quality of the target program, efforts should be made to improve the allocator.

Another desirable goal would be to coordinate allocator and scheduler in order to make their results compatible. Nevertheless, both methods are very promising for use in a practically usable compiler.

Bibliography

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

[Ban93] U. Banerjee. *Loop Transformations for Restructering Compilers: The Foundations.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[DK91] A. Darte and L. Khachiyan. Linear scheduling is nearly optimal. Research Report 91-35, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, November 1991.

[DR93a] A. Darte and Y. Robert. A graph–theoretic approach to the alignment problem. Research Report 93-20, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, July 1993.

[DR93b] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. Research Report 93-03, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, January 1993.

[DR94a] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. on Parallel and Distributed Systems*, 5(8), August 1994.

[DR94b] M. Dion and Y. Robert. Mapping affine loop nests: New results. Research Report 94-30, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, November 1994.

[DR95] M. Dion and Y. Robert. Mapping affine loop nests: New results. *Lecture Notes in Computer Science*, 919:184–189, 1995.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[DV94] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling. Research Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.

[DV95a] A. Darte and F. Vivien. A comparison of nested loops parallelization algorithms. Research Report 95-11, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, May 1995.

[DV95b] A. Darte and F. Vivien. Revisiting the decomposition of Karp, Miller and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.

[DV96] A. Darte and F. Vivien. Optimal fine and medium grain prallelism detection in polyhedral reduced dependence graphs. Research Report 96-06, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, April 1996.

[EM92] J.R. Evans and E. Minieka. *Optimization Algorithms for Networks and Graphs.* Marcel Dekker Inc., 1992.

[Fab97] P. Faber. Transformation von Shared-Memory-Programmen zu Distributed-Memory-Programmen. Diplomarbeit, Universität Passau, 1997. In preparation.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.

[Fio72] J. Ch. Fiorot. Generation of all integer points for given sets of linear inequalities. *Mathematical Programming*, 3:276–295, 1972.

[FT90] P. Feautrier and N. Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel PIP. Technical Report 90-2, Institut Blaise Pascal, UPMC, Laboratoire MASI, January 1990.

[Gei96] M. Geigl. Parallelization of general loop nests in the polyhedron model. Diplomarbeit, Universität Passau, 1996. In preparation.

[GL94] M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.

[GL96] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In *Ninth Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, 1996. To appear.

[Jun94] D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag, 1994.

[Kei96] H. Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Diplomarbeit, Universität Passau, 1996. In preparation.

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

[Lam74] L. Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[Loo96] The polyhedral loop parallelizer LooPo . http://www.uni-passau.de/~loopo/, 1996.

[MDR94] J. Philippe M. Dion and Y. Robert. Parallelizing compilers: waht can be achieved? Research Report 94-11, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, March 1994.

[MDR95] C. Randriamaro M. Dion and Y. Robert. How to optimize residual communications. Research Report 95-27, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, October 1995.

[Pug92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 8:102–114, August 1992.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.

[Sch97] M. Schumergruber. Partitionierung paralleler Schleifensätze. Diplomarbeit, Universität Passau, 1997. In preparation.

[SF89] W. Shang and J. A. B. Fortes. On the optimality of linear schedules. *J. VLSI Signal Processing*, 1(2):209–220, October 1989.

[SS94] W. Shang and Z. Shu. Data alignment of loop nests without nonlocal communications. In *Application Specific Array Processors*, pages 439–450. IEEE Computer Society Press, 1994.

[Str86] G. Strang. *Linear Algebra and its Applications.* Harcourt Brace Jovanovich, 1986.

[Usm87] R. A. Usmani. *Applied Linear Algebra.* Marcel Dekker Inc., 1987.

[Wet95] S. Wetzel. Automatic code generation in the polytope model. Diplomarbeit, Universität Passau, November 1995. http://www.uni-passau.de/~loopo/doc/wetzel-d.ps.Z.

[Wie95] C. Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Universität Passau, May 1995. http://www.uni-passau.de/~loopo/doc/wieninger-d.ps.Z.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Publication interne 785, Institut de Recherche en Informatique et Systemés Aleèatoires, Rennes, December 1993.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

[Zie95] G. M. Ziegler. *Lectures on Polytopes.* Springer-Verlag, 1995.

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, daß ich diese Diplomarbeit selbstständig und ohne Benutztung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, und alle Ausführungen , die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 27. September 1996

(Wolfgang Meisl)