UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

Index set splitting based on the polytope model

Diplomarbeit

Verfasser:

Bernhard Lehner

lehner@fmi.uni-passau.de

Aufgabensteller:

Prof. Christian Lengauer, Ph. D.

Lehrstuhl für Programmierung

Universität Passau

Betreuer:

Martin Griebl

Passau, den 3. Februar 1999

Acknowledgments

This thesis was made possible by theoretical and moral support of many people.

In the first place I would like to thank Dr. Martin Griebl who always offered his help and supported me with corrections, suggestions, his English knowledge and motivating words!

Prof. Christian Lengauer Ph.D. was proofreading this thesis and provided many corrections and improvements of my English formulations!

Andreas Dischinger, Sven Anders, Peter Faber, Nils Ellmenreich and Christian Böck for their suggestions and proposals!

And finally my wife Katarzyna who supported me not only morally but gave also suggestions and improved my English; and my parents who made my studying possible and who supported me without any pressure!

I am very thankful to all of them!

Abstract

For parallelizing sequential programs, several methods have been developed. In this diploma thesis, a method for splitting the iteration domain of nested loops is presented. The aim of the splitting is to extract single parts of the iteration domain, to which some parallelization methods can be applied, even if these methods are not applicable to the whole iteration space. We show the original idea of index set splitting and we expand it on the basis of the polytope model for loop parallelization. Further parallelism is gained by considering how the splitting of one index set affects other index sets.

Contents

Chapter 1

Motivation

Parallelization is an important way of speeding up programs. Some constructs of a programming language have good requisites for a parallelization. E.g. loop nests have a lot of potential parallelism, for two reasons: their statements are executed repeatedly and loops are regular, so their parallelization can be done statically.
During the iteration of a loop, we look at the single executions of the loop statements, which are called the instances of a statement. The set of all instances of a statement is called the index set of the statement. Two statement instances can be executed in parallel if there are no dependences between them. A dependence exists if an instance of a statement S must be executed before the instance of a statement T, where it is permitted that $S = T$.

Standard parallelization methods consider only program parts, i.e. statements and their whole index set. The fact that dependences exist only for a part of the index set limits the effectiveness of these methods. The parts without dependences could be executed in parallel. The limitation of standard parallelization methods is shown in the example program in Figure 1.1.

Is there any method which is able to identify potential parallelism? The standard methods can be divided into two groups. We distinguish methods based on for $i := 1$ to $2n$

$A[i] := A[2n - i]$

Figure 1.1:
No parallelization without splitting

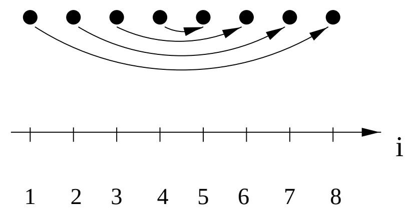


Figure 1.2: Dependences for $n = 4$

statements and methods based on instances. Instance methods cannot parallelize this program, as there are dependences whose length depends on the iteration of the loop. Statement methods cannot parallelize it either, as there is only one statement.

But parallelization would be easy if the program were replaced by two loops, each with the statement above; one loop: for $i := 1$ to n, and a second loop: for $i := n + 1$ to $2n$. Then the loops must be executed one after another, but each loop internally can be executed in parallel. The changed program needs only two logical time steps, instead of the $2n$ steps for the original program.

The idea of preparing programs in that way is to perform a dependence analysis and for every dependence d to partition the set of all instances into subsets whose instances contain d, and subsets whose instances are free from d. In that way, each subset may be executed in parallel, but the subsets must be sequenced with respect to each other.

Programs, which can be parallelized as dramatically as the one above, are rare, but there are still many programs for which index set splitting is possible and effective.

Chapter 2

Basics and definitions

In this chapter, we introduce the terms and notations which we will use within this thesis.

2.1 Loops

This thesis is about an adaption of loop programs for their parallelization. We assume loops of the structure shown in Figure 2.1.

$$
\begin{array}{lll}
\text{for } i = l \text{ to } u & & \\
S_1\text{:} & A_1[x_1, \cdots, x_{n_1}] := \cdots & \\
\vdots & \vdots & \vdots \\
S_n\text{:} & A_m[y_1, \cdots, y_{n_2}] := \cdots &
\end{array}
$$

Figure 2.1: Structure of a simple loop program

A loop consist of a *for*, followed by the *loop index* (i). The *index space* of a loop is defined by the *lower bound* (l) and the *upper bound* (u). S_1 to S_n are *statements*. In statements expressions of *arrays* ($A_1[i_1, \cdots, i_k], \cdots, A_k[\cdots]$) and constants are assigned to *arrays* $(A_l[\cdots], \cdots, A_n[\cdots])$, where $A_1, \cdots, A_k$ do not need to be different from $A_l, \cdots, A_n$, and the index expressions are affine.

A *loop program* has one or more loops, which can be *perfectly nested*, i.e., every loop body contains only another loop and the most inner loop contains only statements, or which are *imperfectly nested*, i.e., at least one loop body contains statements and another loop.

The *index set* of a statement, notation: $I(S)$, is the set of index spaces of all surrounding loops.

The array indices and the loop bounds can contain constants, which are unknown at compile time. They are called *structure parameters*.

If a statement S is surrounded by z loops $l_1, \cdots, l_z$, the *index vector* is a tuple $(i_1, \cdots, i_z) \in I(S)$, where $l_1 \leq i_1 \leq u_1, \cdots, l_z \leq i_z \leq u_z$, and $l_1, \cdots, l_z$ and $u_1, \cdots, u_z$ are the lower and upper bounds of the surrounding loops.

The *instance* of a statement S, notation: $\langle S,\ (i_1, \cdots, i_z)\rangle$, is the execution of S for the index vector $(i_1, \cdots, i_z)$.

A *program part* is a statement with its index set.

2.2 Dependences

Def.: Dependence An instance j of a statement depends on another instance i of a statement, if:

1. in both instances the same memory cell is referenced,
2. i is executed before j,
3. i or j writes to the memory cell.

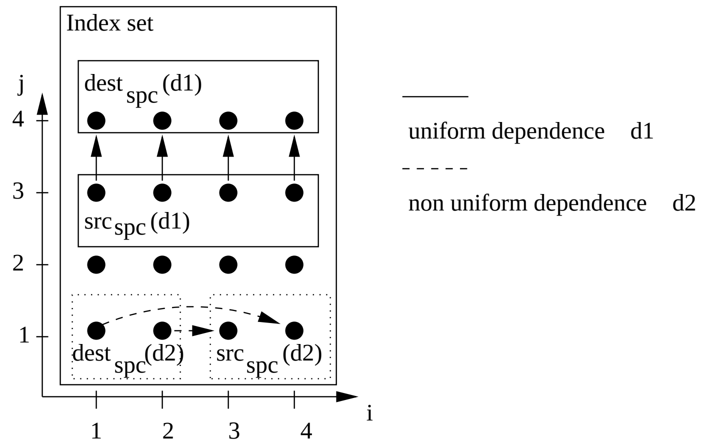
A dependence is denoted as

$$d\colon \langle S_1,\ (i_1, \cdots, i_n)\rangle \rightarrow \langle S_2,\ (j_1, \cdots, j_z)\rangle$$

It expresses that, for semantic reasons, instances $(i_1, \cdots, i_n)$ of S_1 must be executed before instances $(j_1, \cdots, j_z)$ of S_2. The statements S_1 and S_2 can be identical.

S_1 is the *source statement*, src(d), and S_2 the *destination statement*, dst(d).

The set of all $(i_1, \cdots, i_n) \in I(S_1)$, for which d exists, is called the *source space* concerning S_1, $src_{spc}(d)$, and the set of all $(j_1, \cdots, j_z) \in I(S_2)$, for which d exists, is called the *destination space* concerning S_2, $dst_{spc}(d)$.
The difference $\vec{j} - \vec{i}$ is called *dependence vector*. A dependence d is *uniform* if the dependence vector is constant. If the dependence vector is constant but contains structure parameter, then the dependence is not uniform [2]. We call it *nearly uniform*.
d is *loop-carried* $\iff$ the dependence vector $\neq \vec{0}$.



We distinguish several types of d:

- d is a *true* dependence if S_1 writes to an array A and S_2 reads A;
- d is a *anti* dependence if S_1 reads A and S_2 writes A;
- d is an *output* dependence if S_1 and S_2 write A.

We call d *direct* if A was not written between $\langle S_1, (i_1, \cdots, i_n)\rangle$ and $\langle S_2, (j_1, \cdots, j_n)\rangle$.
d is called a *flow* dependence if d is a direct true dependence.

Further information about dependences and dependence analysis can be found in the literature [1], [2].

2.3 Graphs

The *statement dependence graph* is a directed graph (N,E), whose nodes represent the statements and whose edges represent the dependences between the statements.

A *path* (a,b) is a chain of directed edges, which connects a with b.

A *cycle* is a path, whose start node and end node are identical.

A *strongly connected component* within a graph is a set of nodes and edges, in which every node can be reached from every node.

The *acyclic condensation* of a graph is given by replacing the strongly connected components by single nodes.

2.4 Sets and Relations

The necessary mathematical definitions are taken from [9].

Def.: reflexive

$R \subseteq M \times M$ is reflexive, if $\forall x \in M : xRx$.

Def.: transitive

A relation $R \subseteq M \times M$ is transitive, if $\forall x,y,z \in M : xRy \wedge yRz \Rightarrow xRz$

Def.: reflexive, transitive closure

For $A \subseteq M$, the closure of A is defined as the smallest closed set, which fully contains A and all points of M, which are reachable from A by the relations defined on M.

2.5 Polytopes

Def.: convex

A set M is *convex*, if $\forall x, y \in M : 0 \leq \alpha \leq 1 : y + \alpha * (x - y) \in M$.

Def.: polytope

A polytope is the convex hull of a finite number of points.

2.6 The Polytope model

The polytope model gives a mathematical framework for parallelizing nested loops, whose bounds must be affine expressions in outer loop indices or structure parameter. With this constraint, the index set of every body statement in the loop nest is a polytope.

For parallelization, every instance of every body statement is mapped to space, i.e. onto virtual processors (allocation), and to time, i.e. virtual execution spaces (scheduler). Technically, this space-time-mapping can be interpreted as a change of the basis, so that the target dimensions either are fully in time or fully in space. Finally, the space dimensions are enumerated by parallel loops, the time dimensions by sequential loops.

LooPo is an automatic parallelizer for sequential loop programs. It is an implementation of the polytope model. The parallelization of loop nests within LooPo consists of several steps:

1. Scanning/Parsing

2. Tiling

3. Dependence analysis

4. Scheduling

5. Allocating

6. Generation of abstract target loops

7. Generation of concrete target code

The scanner/parser prepares a source code of a language which provides constructs for loops. It generates a parse tree and a symbol table. Its output is consumed by several further modules.

Tiling is optional. It partitions index sets in order to get coarser granularity in the parallelized program.

The dependence analysis identifies all dependences within the program. It provides further information about dependences like the involved statements, the index set I of these statements and the subsets of I, to which the dependence really influences, i.e., for every dependence d: $src_{spc}(d)$ and $dst_{spc}(d)$.

The scheduler uses the dependences to create a schedule, which contains for each instance the logical time step at which it must be executed (see [3, 4]). A program using the *free schedule* executes every instance as early as possible.

The allocator also uses the dependence analysis to find the location, i.e., the processor on which the instances shall be executed.

Generation of abstract/target code Finally the output of the parser, the scheduler and the allocator are used to create the target loops, some of which are in parallel. According to the chosen destination language target code is generated.

Chapter 3

Theory of Index Set Splitting

Programs can be parallelized in several ways. There are instance methods and statement methods.
The method of index set splitting is a good preparation for both types of methods, as it will change index sets of loop programs and create new statements, which - hopefully - will enable better schedules.

3.1 The basic split

First we introduce the way of splitting, which Jemni and Mahjoub [11] found, showing the basic idea and effects of splitting index sets. Next the idea is based on the expanded polytope model [10]. This extends its usability and applicability.

3.1.1 Jemni's and Mahjoub's way of Index Set Splitting

What is the condition for finding a parallel schedule?
There must not exist a dependence between instances of the parallel executed statements!
Jemni's and Mahloub's observation [11] is that there are dependences, which are restricted to a subset I_s of the index set I of a statement. Instances belonging

to $I \backslash I_s$ could be executed in parallel. But no parallelization method will find single instances of an index set. So Jemni and Mahjoub present a method, which changes the source program in such a way that there are only statements, which are without any dependence, and statements, which have one or more dependences whose destination space is the whole index set of the statement.
Jemni and Mahjoub[11] use a dependence analysis, e.g., described in [6], which provides $src_{spc}(d)$ and $dst_{spc}(d)$.

$$\begin{array}{ll} & \text{for } i := 0 \text{ to } n \\ S_1 : & \quad A[2*i-2] := \cdots \\ S_2 : & \quad A[i+2] := \cdots \end{array}$$

Figure 3.1: Source Program

We demonstrate the method with the source program in Figure 3.1. The example is slightly different from the examples shown in [11]. Jemni and Mahjoub use their method to parallelize loops which contain if-statements with static conditions. As we parallelize more general loop programs, we simplify the example in order to point out the principle of their splitting.

The source code contains two output dependences which result from the conflict equation: $2i_1-2 = i_2+2 \quad \Rightarrow \quad i_1 \geq i_2 \Leftrightarrow i_1 \leq 3$:

1. output dependence : $S_2 \to S_1$, for $i_1 \leq 3$
2. output dependence : $S_1 \to S_2$, for $4 \leq i_1 \leq n$

Hence, there is an output dependence which changes its orientation. This causes a splitting of the index set into two parts:

$$0 \leq i \leq 3 \text{ and } 4 \leq i \leq n$$

Figure 3.2 shows these two dependences for $n = 6$. The vertical line shows where the splitting must take place.

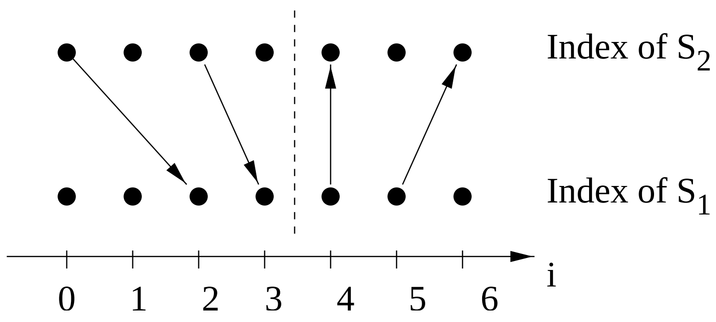


Figure 3.2: The dependence graph to Figure 3.1

In the left part, i.e., $0 \le i \le 3$, S_2 must be executed before S_1, while the opposite order must be applied in the right part. A consequence of splitting is that we must generate as many copies of the statement as the number of parts generated by the split. In Figure 3.1 there are two statements and the index set is split into two parts. Therefore we get four loops:

$$
\begin{array}{ll}
 & \text{forall } i := 0 \text{ to } 3 \\
S_2: & \quad A[i+2] := \cdots \\
 & \text{forall } i := 0 \text{ to } 3 \\
S_1: & \quad A[2*i-2] := \cdots \\
 & \text{forall } i := 4 \text{ to } n \\
S_1: & \quad A[2*i-2] := \cdots \\
 & \text{forall } i := 4 \text{ to } n \\
S_2: & \quad A[i+2] := \cdots
\end{array}
$$

Figure 3.3: The destination program to Figure 3.1

The second and third loop can be fused to a single loop.

This example could be parallelized very effectively. But the abilities of Jemni's and Mahjoub's method are limited. It is restricted to nested loops, whose array indices satisfy the condition:

$$A[a_1x_1 + c_1, .., a_nx_n + c_n] := A[b_1y_1 + d_1, .., b_ny_n + d_n]$$
$$\text{where } a_i,\ b_i \text{ are constants and } \forall i : x_i = y_i$$

This means that loop programs, such as

$$\begin{array}{ll} & \text{for } i := 0 \text{ to } n \\ & \quad \text{for } j := 0 \text{ to } n \\ S_1: & \qquad A[i, j] := A[j, i] \cdots \end{array}$$

cannot be handled.

The limit of their method can be formalized:

A hyperplane defining the border between two program parts may contain at most one loop index.

As we shall see, the hyperplane which usefully splits $I(S_1)$, is $i = j$. There are already two loop indices in this expression. With this method, no parallelization is possible. In the next section, we try to overcome this limitation by using a mathematical model.

3.1.2 Index Set Splitting within the Polyhedron Model

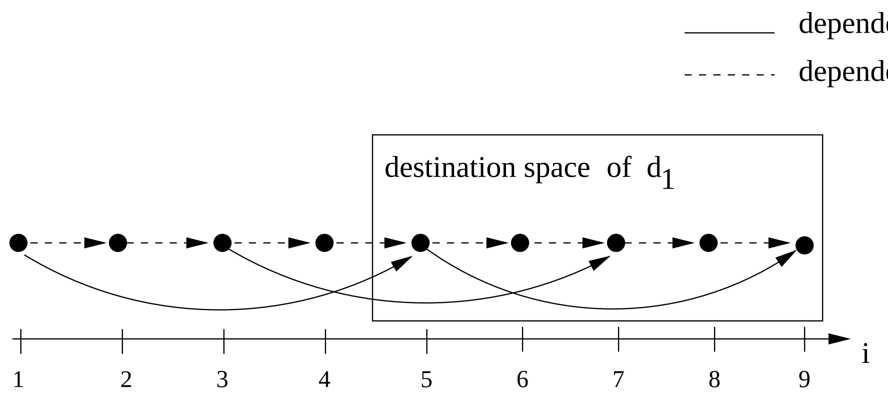
As splitting changes and creates dependences, the aim is to allow the scheduler to find as much parallelism within a program as possible. For doing the splitting, we need the information, which the dependence analysis provides, as there are the dependences with their source and destination statement, and their source and destination space. So, we plug index set splitting into the polytope model between dependence analysis and scheduler.

The simplified way of index set splitting consists of three steps.

1. Finding the subset I_{split} of the index set, which should be partitioned.
2. Splitting the index set into I_{split} and the remaining set.
3. Adding the statement, to which the former index set belonged, to the resulting splits.

Now we analyze, how the single steps can be realized.

Finding the place for splitting



The aim of splitting is to partition the whole index set according to a dependence d into a part, whose instances depend on any earlier execution of any instances, and into a part, whose instances do not depend on d. Both parts may still contain other dependences.

As described in Section 2.6, the dependence analysis gives for every dependence d a subset of the index set I, which is affected by d: $dst_{spc}(d)$. This is just the part of I, which needs to be distinguished from the rest of I.

The destination space can contain more points than instances which are involved, e.g., when there is a stride of 2.

Now, as we have the position of the split, we may think about splitting the index set.

The splitting Partitioning an index set I, if there is only one single dependence d, is performed by using the $dst_{spc}(d)$ to get two parts:

$$dst_{spc}(d) \text{ and } I \backslash dst_{spc}(d)$$

If there are more dependences d_1, d_2, where $dst_{spc}(d_1)$ is a subset of I and $dst_{spc}(d_2)$ is a subset of I , we have several options.
If $dst_{spc}(d_1) \cap dst_{spc}(d_2) = \emptyset$, we split like before; this time we get three parts.
If $dst_{spc}(d_1) \cap dst_{spc}(d_2) \neq \emptyset$, either we can choose, whether we split only according to one dependence – either d_1 or d_2 –,
or we split according to the join of both destination spaces,
or we split according to the intersection of the splits and the remaining destination spaces.
The advantage of the intersection is, that we use the information of both splits and that every partition can be executed as early as possible.
Consider the example in Figure 3.4. The instances within $dst_{spc}(d_1) \cap dst_{spc}(d_2)$

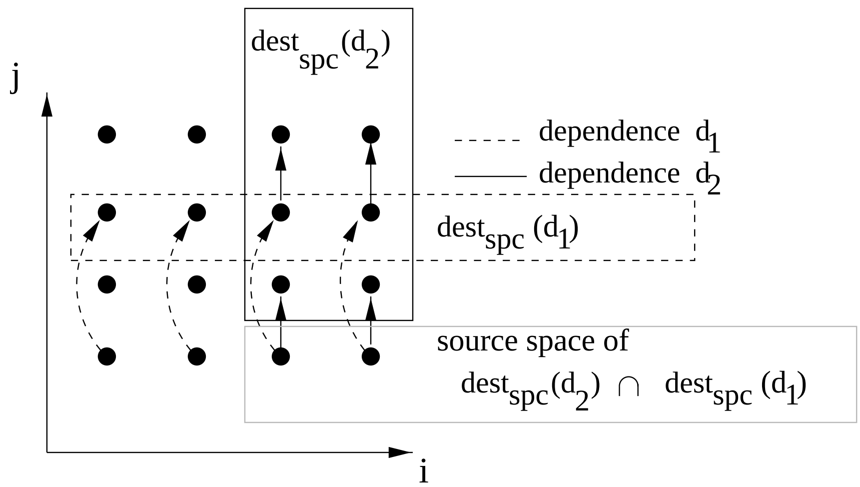


Figure 3.4: Intersection of splits

could be executed as soon as the instances within the source space of $dst_{spc}(d_1) \cap dst_{spc}(d_2)$ have been executed. But the scheduler can find this option only if we split I into the intersection, the remaining parts of the split, and the remaining subset of I, because the intersection would be within the sequential order of $dst_{spc}(d_2)$.

As we have a new set of index sets, now, we need to convert them to program parts.

The program parts Splitting the index set of any statement S according to a dependence d generates a set of index sets I_s, and S must be executed within every I_s. Therefore every I_s must get its own copy of S: new program parts are created.

In this way, finally S will be executed only within an index set, where d exists, and copies of S within the remaining index set, where d does not exist.

As the splitting is done with respect to single dependences, all newly created program parts may still contain dependences d', where $d' \neq d$.

The example in Figure 3.5 shows a program whose analysis results in an affine function with two indices, is not treatable by the method of Jemni and Mahjoub.

for $i := 0$ to n

for $j := 0$ to n

$S_1 :$ $A[i, j] := A[j, i];$

Figure 3.5: Source program

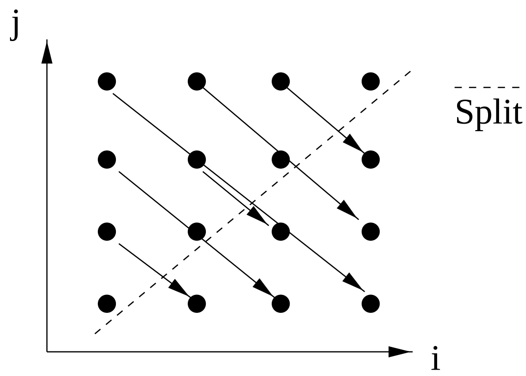


Figure 3.6: Index set with dependences

Analysis of the program yields a flow dependences and an anti dependence. Each of them has a destination space given by the affine function $i - j \geq 1$. So the index set should be split as shown in Figure 3.6.
The resulting index sets are $\{(i,j) \mid 0 \leq i \leq j \leq n\}$ and $\{(i,j) \mid 0 \leq j < i \leq n\}$.

Now the sequential schedule of n^2 steps can be reduced to a parallel schedule of two steps:

$$\theta(i,j) = \begin{cases} 0, & \text{for } 0 \leq i \leq j \leq n \\ 1, & \text{for } 0 \leq j < i \leq n \end{cases}$$

Finally we want to point out a small problem during the splitting in some cases.

Convex index sets The index set, which remains after splitting, can be non-convex. As the polyhedron model requires convexity of iteration spaces, the non-convex index set must be split further into convex polytopes. But we will neglect this problem for now (see Section 3.3.5).

3.1.3 Conditions when to split

As shown, we can split every index set I with respect to every dependence, which affects (a part of) I. But there are situations in which splitting is useless. We will now analyze these situations and try to find criteria, which allow us to exclude them from splitting.

There are several possible conditions when a splitting is useful. Partially, conditions which were added by the user can improve the run time.

Uniform dependences. Consider the program in Figure 3.7.

The destination space I_{dst} of the only dependence in this program is $\{i \mid 3 \leq i \leq n\}$. This means that the first point of the index set I could be executed in parallel with the second point, and the remaining points must be executed sequentially after the second point, because every point in I_{dst} is destination of two

$$\begin{array}{ll} & \text{for } i := 1 \text{ to } n \\ S_1 : & \quad A[i] := A[i+2] + A[i+3] \end{array}$$

Figure 3.7: Shall it be split?

predecessors. So, no different splitting than into the sets $\{i \mid 1 \leq i \leq 2\}$ and $\{i \mid 3 \leq i \leq n\}$ can be found. The parallelism gained does not outweigh the overhead incurred in executing the first two points in parallel. So, what we need is a threshold for m at which a splitting becomes useful! The number p of available processors can be a useful threshold. But also a smaller number can be useful if the overhead of the parallel execution is smaller than the time, which we gain by avoiding the sequential execution of the program part.
As we see, there are several factors, e.g., the hardware, which change the threshold for a reasonable split but which the algorithm cannot gather. Some of these factors can be known by the user. So, user interaction can help. Without a users help, we avoid splitting uniform dependences.

Nearly uniform dependences As shown above, it is generally not useful to split uniform dependences. But the example in Figure 3.8 shows that we can miss possible parallelism by not considering nearly uniform dependences. Analyzing the whole index set, only the last three instances of S_1 depend on any previous execution of instances! The biggest part of the index set, $[1, 4n]$, could be executed in parallel. On the other hand, executing $[1, 4n]$ in parallel is not effective for small n, it might even increase execution time. As the value of n is not known at compile time, it is not decidable, whether the split should be executed or not. We simply could decide optimistically for splitting.

$$\begin{array}{ll} & \text{for } i := 1 \text{ to } 4n+3 \\ S_1: & \quad A[i] := A[4n+i]; \\ & \text{end for} \end{array}$$

Figure 3.8: nearly uniform dependences

Index set size. For program parts whose number of parallel executable instances is not known, the method of splitting if the index size is bigger than a given threshold does not work. In Figure 3.9 are examples, in which only the user can know whether splitting makes sense.

a)

$$\begin{array}{ll} & \text{for } i := m \text{ to } n \\ S_1: & \quad A[i] := A[n-i] \end{array}$$

b)

$$\begin{array}{ll} & \text{for } j := 1 \text{ to } n \\ & \quad \text{for } i := j \text{ to } 2j \\ S_1: & \qquad A[i] := A[2j-i] \end{array}$$

Figure 3.9:Unknown number of points

Figure 3.9a): the index set in this example could be split into $\{(i) \mid m \leq i \leq \frac{m+n}{2}\}$ and $\{(i) \mid \frac{m+n}{2} \leq i \leq n\}$. But if the index set spawned by $n-m$ is small, the execution of the split set will be slower than executing the original index set sequentially, as the overhead for executing something in parallel needs more time than executing it sequentially.

If the variables n and m are known, the decision is easy. But, in general, n and m will be calculated during the execution of the program, and not known at compile time. We choose the optimistic approach and allow the split.

Figure 3.9b): At the start of the execution of the j-loop, the index set of the inner loop is very small. So splitting the inner index set would result in longer execution times, because of the overhead of parallelized parts. But with

growing j, the inner index set will reach a size, for which a splitting promises a good speed up. So if n is known to be large, the split should be permitted.
In both cases the user could interact helpfully, as he could know the limits. In the absence of directions from the user a splitting should be done (optimistic approach).

Communication. Splitting can slow down programs, if we take allocation into account. Figure 3.10 contains a program which can be split into two parts executable in parallel, with respect to the only dependence between the arrays $A[i]$ and $A[n-i]$. Executing every point of each half of the index at a different processor would cause that, for one short calculation, plenty of communication has to be done. But, the allocation is not part of index set splitting. So, the program will be split to give the allocator the chance to find the optimal allocation. The program with its split program parts still can be executed sequentially on one processor.

$$\begin{array}{ll} & \text{for } i := 1 \text{ to } n \\ S_1: & \quad A[i] := A[n-i] + B[i] + C[i] + \cdots + Z[i] \end{array}$$

Figure 3.10: Communication influences?

3.1.4 The basic algorithm

The basic algorithm includes all conditions and strategies described above. Its complexity is of O($|D|$), where D is the set of all dependences.

The splitting algorithm

Input: - all dependences
- every statement with its index set $I(S)$

Output: each statement either with unsplit $I(S)$ or with $I(S)$ split into a set of index sets.

1. **FOR** every dependence d **DO**
   IF conditions for a split are satisfied **THEN**
   (a) Take the polyhedron of the destination statement, which represents the subset of $I(S)$, where the dependence really exists, i.e., $dst_{spc}(d)$.
   (b) Add a copy of $dst_{spc}(d)$ to a list $L_{Split}(S_{dst})$ for the later execution of the split.
2. **FOR** all statements S **DO**
   Split $I(S)$ into all $I_s \in L_{Split}(S_{dst})$ and additionally into all intersections of I_s, and the remaining set.
   Add a copy of S to every I_s

Figure 3.11: A first split algorithm

To show how the algorithm works let us consider an example:

$$\begin{array}{ll} & \text{for } i := 1 \text{ to } 2n \\ & \quad \text{for } j := 1 \text{ to } 2n \\ S_1: & \qquad A[i,j] := A[2n-i+1, n+j]; \end{array}$$

Figure 3.12: Source Program

The index set I of the only statement S_1 is $\{(i,j) \mid 1 \le i \le 2n \ \wedge \ 1 \le j \le 2n\}$ (see Figure 3.13).

The dependence analysis yields two dependences for S_1:

d_1: a true dependence: $\langle S_1, (i,j)\rangle \to \langle S_1, (2n-i+1, j+n)\rangle$

d_2: an anti dependence:$\langle S_1, (i,j)\rangle \to \langle S_1, (2n-i+1, j-n)\rangle$

Figure 3.13 shows I for $n = 2$ with all dependences.

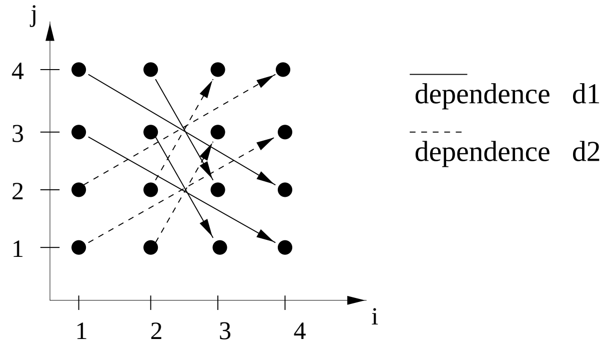


Figure 3.13: Index set and dependences

Let us apply the algorithm.

1. The first d is d_1:

   (a) The conditions for the split are satisfied, as it is not a uniform dependence.

(b) The subset P_{dst} of I, where d_1 really exists, is $\{(i,j) \mid n{+}1 \le i \le 2n \ \wedge \ n{+}1 \le j \le 2n\}$.

(c) We add P_{dst} as P_1 to $L_{Split}(S_1)$.

2. Now for $d = d_2$:

   (a) Like for d_1 the conditions are satisfied.

   (b) $P_{dst} = \{(i,j) \mid n{+}1 \le i \le 2n \ \wedge \ 1 \le j \le n\}$, that is the right lower quadrant.

   (c) We add P_{dst} as P_2 to $L_{Split}(S_1)$.

3. Finally all splits of I must be merged; i.e., we need to split I into the parts found above, into the intersections of it and the remaining subset. At the end I has been split into three parts, consisting of:

   P_1: $\{(i,j) \mid n{+}1 \le i \le 2n \ \wedge \ n{+}1 \le j \le 2n\} = P_{dst}$,

   P_2: $\{(i,j) \mid n{+}1 \le i \le 2n \ \wedge \ 1 \le j \le n\} = P_{dst}$ and the rest

   P_3: $\{(i,j) \mid 1 \le i \le n \ \wedge \ 1 \le j \le 2n\}$.

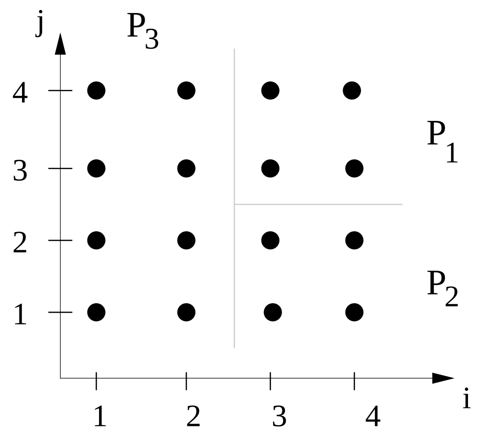


Figure 3.14: The split index set

3.2 Propagation of splits

The splitting of an index set can have positive effects on other statements, which are reachable in the statement dependence graph. If we use the information that some split parts of a statement S_1 can be executed earlier than others, we observe that we can also execute instances of statements earlier, which depend (transitively) on S_1. We call the splitting of the indices of reachable statements according to split parts of any source statement, *propagation*.

3.2.1 The idea of the splitting propagation

The goal of propagating the split of a dependence is both to gain more parallelism and to approximate the free schedule better, and, thus, to execute statements as early as possible.

Better schedules For illustration of split propagation and its positive effect on the obtained schedule, consider the example from Section 3.1.2.

$$
\begin{array}{ll}
 & \text{for } i := 0 \text{ to } n \\
 & \quad \text{for } j := 0 \text{ to } n \\
S_1: & \quad\quad A[i,j] := A[j,i]; \\
 & \text{for } k := 0 \text{ to } n \\
 & \quad \text{for } l := 0 \text{ to } n \text{ step } 2 \\
S_2: & \quad\quad B[k,l] := A[k,l];
\end{array}
$$

Figure 3.15: Better schedules through propagation

Analyzing the second statement, we discover that it is not influenced by any dependence, so the $k-$ and $l-$loop could be executed in parallel without any changes, immediately after the $i-$ and $j-$loop are executed. Any splitting seems to be unnecessary. The schedule would be:

$$\theta\langle S_2, (k,l)\rangle = 2, \text{ for } 0 \le k \le n \text{ and } 0 \le l \le n \ .$$

Using now our knowledge from the splitting of S_1, we see that almost half of the index set of k and l can be executed one step earlier. According the two parts of the first index set by propagating we get two parts also for the second index set: $\{(k,l) \mid 0 \le k \le l \le n\}$ and $\{(k,l) \mid 0 \le l < k \le n\}$.
So the optimal schedule is:

$$\theta\langle S_2, (k,l)\rangle = \begin{cases} 1, \text{ for } 0 \le k \le l \le n \\ 2, \text{ for } 0 \le l < k \le n \end{cases}$$

Let us compare the example before and after the propagation. Both programs need 3 logical steps. So, it seems that the propagation did not improve anything. But looking at a real parallel computation, we may suppose that the n processors of the parallel machine are not only available for this program, but at the same time they are used also for different programs. Comparing the split version to the propagated version, certainly the propagation will not cause any longer run time, not depending on the number of available processors, but there are many possible constellations in which it will finish earlier, e.g., during the first and the second step n processors are available, but during the third step only $n/2$ processors. Then the propagation already saved one step. (see Figure 3.16)

Expanding the program in Figure 3.15 by adding a surrounding loop of n steps, the propagated split could save n steps! Changing the amount of usable processors after time step 2 decreases or increases the number of gained steps.

real time steps	number of usable processors	executed stmts unsplit	executed stmts split
0	n	S_1	S_1
1	n	S_1	S_1, S_2
2	$n/2$	S_2	S_2
3	$n/2$	S_2	

Figure 3.16: Example of effects of propagation

More parallelism Independently of the number of processors, we may gain by propagation as much parallelism as during a basic split. Consider the example in Figure 3.17.

$$
\begin{array}{ll}
 & \text{for } i := 0 \text{ to } 2n \\
S_1 : & \quad A[i,0] := B[i-n-1,m]; \\
 & \quad \text{for } j := 1 \text{ to } m \\
S_2 : & \qquad B[i,j] := B[i,j-1] + A[i,0];
\end{array}
$$

Figure 3.17: More parallelism through propagation

The index set I_1 for S_1 is $\{i \mid 0 \le i \le 2n\}$.
The index set I_2 for S_2 is $\{(i,j) \mid 0 \le i \le 2n \wedge 1 \le j \le m\}$.
There are three true dependences (for $n = 2$ shown in Figure 3.18):

d_1: from $B[i,j]$ in S_2 to $B[i-n-1,m]$ in S_1

d_2: from $B[i,j]$ in S_2 to $B[i,j-1]$ in S_2

d_3: from $A[i,0]$ in S_1 to $A[i,0]$ in S_2

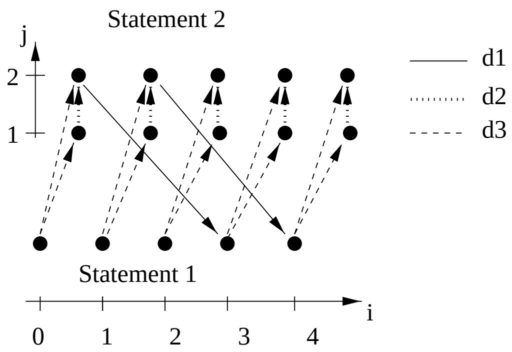


Figure 3.18: Index sets with dependences

Before splitting:

- d_3, d_2 and d_1 together prohibit the parallel execution of S_1 for all i, as some instances of S_1 must be executed before some instances of S_2 and some instances between.
- d_1 together with d_3 prohibits the parallel execution of S_2 for all i of a single j, for the same reason.
- d_2 prohibits the parallel execution of S_2 for all j of a fixed i.

The basic splits:

- d_1 causes a split of the index set of S_1 into:
  - $\{i \mid 0 \leq i \leq n\}$ with S_3 as a copy of S_1 and
  - $\{i \mid n+1 \leq i \leq 2n\}$ with S_4 as a copy of S_1.
- d_2 is a uniform dependence; therefore it does not trigger a split.
- d_3 reaches the whole index set of S_2, so the index set of S_2 stays unsplit.

The split changes the dependences d_1 and d_3. As S_1 was replaced by S_3 and S_4, dependences are now from S_3 to S_4.

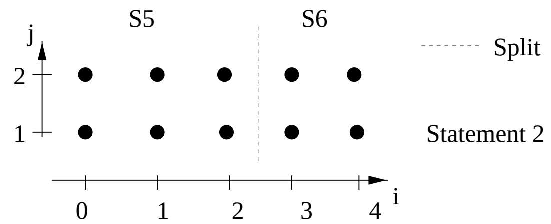
The advantage of the split is that the former statement S_1 can be executed in parallel in two steps! But S_2 is still executed sequentially as there are still instances of S_2 executed before S_4 and some after S_4.

The run time of this program is square of the one before!

<u>The propagation:</u>

We split I_2 along d_3 according to S_5 into:

- $\{(i,j) \mid 0 \leq i \leq n \wedge 1 \leq j \leq m\}$ with S_5 as a copy of S_2 and
- $\{(i,j) \mid n+1 \leq i \leq 2n \wedge 1 \leq j \leq m\}$ with S_6 as a copy of S_2.



The propagation changes d_1 into a dependence from S_5 and S_4.

Two new dependences coming from d_2 are between instances of S_5 and between instances of S_6.

Similarly, d_3 changes into two dependences from S_3 to S_5 and from S_4 and S_6.

In summary:

- First S_3 is executed in parallel.
- Next S_5 is executed for all i for every fixed j, with respect to the only dependence in S_5.
- Next S_4 is executed in parallel.
- Next S_6 is executed for all i for every fixed j, with respect to the only dependence in S_6.

Now the run time is linear as the execution of S_5 and S_6 takes together only $|2 * m|$ steps (before S_2 was the square of this)!

3.2.2 Paths for propagation

Figure 3.15 shows only a simple example, which contains only one statement, to which a propagation can be applied. But, in general, there can be various statements, which are reachable from a split statement. Searching for them, we build a statement dependence graph according to the list of all dependences. It is a multigraph as there can be several dependences from statement a to b.
An initial split must be propagated to every statement on the path to its actual destination statement. Therefore, we need an algorithm which produces for every statement a list of reachable statements containing all paths to those statements.

Kleene [5] has given an algorithm, which computes all paths between every pair of statements within the statement dependence graph. Its complexity is cubic in the number of statements (see Figure 3.19). The algorithm produces regular expressions, i.e. cycles within paths are superscripted with a '*', choices are denoted with '|'. An edge between a and b is denoted with: $a - b$

Kleene uses a three-dimensional array *paths*, where each dimension is of size of the number of statements. First he inserts the direct paths between statements. Then, for every pair (a, b) of nodes, he tests whether there is another node n which could connect a with b, directly or indirectly.
If there is a path from n to n, a cycle is found and also added to the paths from a to b, e.g., $(a - n.(n - x_1.x_1 - x_2 \cdots x_n - n)^*.n - b)$.
How often is this cycle added? As n is part of the cycle, this cycle was not detected before. So it was not added for any previous n. And, as these nodes will not be considered anymore, it is added only once.

Kleenes path algorithm

m = number of statements;

Array p[m,m,m];

1. **FOR** every start node a **DO**

   FOR every reached node b **DO**

   IF (there exists an edge from a to b)

   THEN

   add the edge to p[a, 0, b];

2. **FOR** every node n **DO**

   FOR every start node a **DO**

   FOR every reached node b **DO**

   IF (there exists a path from a to n and from n to b)

   THEN

   IF (there exists a path from n to n)

   THEN

   add to p[a,n,b]:

   (p[a,$n-1$,b] | p[a,$n-1$,n]. p[n,$n-1$,n]*. p[n,$n-1$,b])

   ELSE

   add to p[a,n,b]:

   (p[a,$n-1$,b] | p[a,$n-1$,n]. p[n,$n-1$,b])

Figure 3.19: Finding paths within a statement dependence graph

3.2.3 The method of propagation

As shown in Section 3.2.1, we gain parallelism by propagating a basic split to any statement which is reachable following the dependences.
Figure 3.20 shows a first propagation algorithm.

Computing the image The dependence analysis computes dependences, which can be interpreted as relations between two statements. The relation maps points of an index set I_1 to points of the index set I_2. Applying a relation to a polytope within I_1, we get its image within I_2.
To compute P_{image} we need to compose all dependences along the path from S_{source} to S_{reach}. Applying P_{basic} to the composition, the result is P_{image}.

Cycles The algorithm in Figure 3.20 works correctly if there is no cycle in the statement dependence graph. In case of a cycle, the algorithm will not terminate, since the propagation should be applied to every statement reachable along every possible path; i.e., traversing the cycle once results in a different path than traversing it repeatedly. So there are three possibilities to avoid an infinite loop:

1. Every reachable statement will be considered a finite number of times.

2. Every reachable statement will be treated until there is an index set left for this statement which is of a size or dimensionality ξ, e.g. $\xi = 0$.

3. We compute the reflexive, transitive closure, which represents all possible paths through the cycle.

The first way is the simplest, but a big amount of parallelism would stay unused. Consider the program in Figure 3.21.

The propagation-algorithm

Input: - all statements with their list of (basic) splits

- the statement dependence graph

Output: all statements with their list of splits, expanded by the splits generated during the propagation

FOR every statement S_{source} **DO**

FOR every statement S_{reach}
which is reachable in G from the split statement S_{source} **DO**

FOR every $I_s \in L_{Split}(S_{source})$

1. Compute the image P_{image} of I_s along all paths from S_{source} to S_{reach}

2. Add P_{image} to a list $L_{Prop}(S_{reach})$ for the later execution of that split.

FOR all statements S **DO**

Split $I(S)$ into all $I_s \in L_{Prop}(S_{reach})$ and additionally into all intersections of I_s, and the remaining set.

Add a copy of S to every I_s

Figure 3.20: A first propagation algorithm

```
        for i := 1 to n
          for j := 1 to 2n
S1 :        B[i,j] := A[i,j]
          end for
          for k := 1 to 2n
S2 :        A[i+1,k] := B[i,2k]
          end for
        end for
      end for
```

Figure 3.21: Source program for repeated propagation

There are two true dependences, shown in Figure 3.22 and Figure 3.23.

The dependence from S_1 to S_2 causes a split of $I(S_2)$: the image α is roughly half of $I(S_2)$. Then, following the dependence from S_2 starting in α to S_1 and again to S_2, we obtain as image only one half of the previous half (except for a shift in dimension i)! This means a further split into the new half and the rest would gain more parallelism, which really yields a speedup. In general, for this example, $log(n)$ splittings would yield the optimal parallel schedule.
However, searching for the optimal schedule raises another problem. How to determine the appropriate a certain number of splittings, like in this example $log(n)$, if n is unknown during the computation of the split? The possibility to repeat propagating according to a finite number can be realized by an interactive definition by the user. It is a simple method and the parallelism is higher than propagating only once, but the optimal schedule will not be reached in general.

The realization of propagating until an index set of size ξ is left is only possible if the loop bounds of all surrounding loops are known. But mostly there

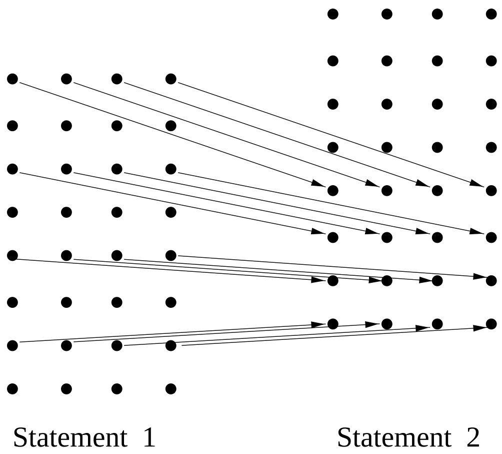


Figure 3.22: Dependence 1

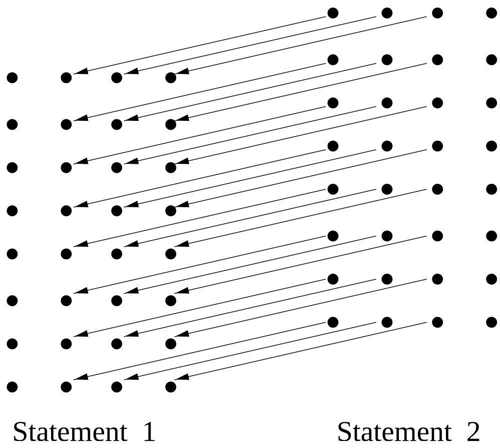


Figure 3.23: Dependence 2

are structure parameters within the expressions which define loop bounds. The unknown value of the structure parameter(s) covers the real size of the index set.

Using the mathematical method of computing the reflexive, transitive closure d^*, we can avoid handling the problems with termination of the algorithm and the problems of an optimal amount of repeated propagation. But using the closure means also less precision. And the closure is not always computable and resulting in an affine expression (see the example in Figure 3.21).

The computation can be done by any stand-alone tool like, e.g., Omega [8] (see also Section 4.3)

3.2.4 Optimizing Kleenes output

We can optimize the calculation of the paths at two places:

Optimizing the output interpretation Consider the example in Figure 3.24:

$$
\begin{array}{ll}
 & \text{for } i := 1 \text{ to } 4n \\
S_1: & \quad A[i] := A[4n - i + 1]; \\
 & \text{end for} \\
 & \text{for } j := 1 \text{ to } 3n \\
S_2: & \quad B[j] := A[j + n]; \\
 & \text{end for} \\
 & \text{for } k := 1 \text{ to } 4n \\
S_3: & \quad C[k] := A[k]; \\
 & \text{end for} \\
 & \text{for } l := 1 \text{ to } 4n \\
S_4: & \quad D[l] := C[l]; \\
 & \text{end for} \\
 & \text{for } m := 1 \text{ to } 3n \\
S_5: & \quad E[m] := B[m] + D[m]; \\
 & \text{end for}
\end{array}
$$

Figure 3.24: Source program for optimizing output

The statement dependence graph is given in Figure 3.25, and the dependences are shown in Figure 3.26.

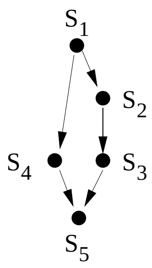


Figure 3.25: Two paths from S_1 to S_5

For propagating the split of S_1 to S_5 we need to identify the paths in the statement dependence graph.
The set of paths, which Kleene's algorithm computes, is: $S_1.(S_2|S_3.S_4).S_5$.
The choice operator '|' is used to separate subpaths. The effect is, that the index sets of the subpaths are joined. Using the result as it is will reduce precision and, thus parallelism. Consider the dependences in Figure 3.26:

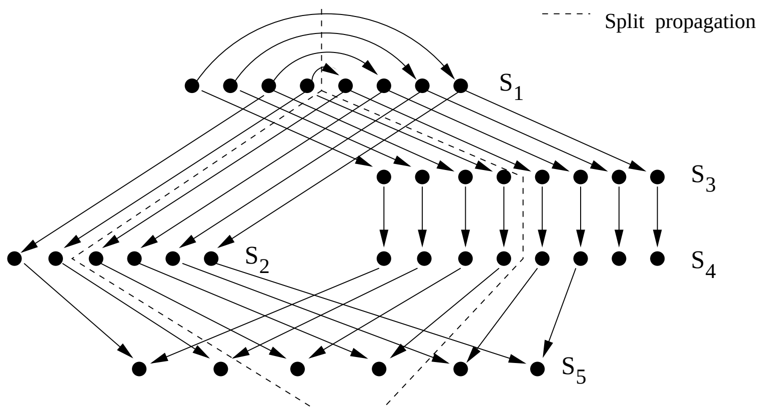


Figure 3.26: The dependences

The dependence $d : \langle S_1, (i)\rangle \rightarrow \langle S_1, (i)\rangle$ causes a split of $\mathrm{I}(S_1)$: $[2n+1, 4n]$.
Now consider the propagated split of each index set:

1. S_2: Its index set split is $[n+1, 3n]$.

2. S_3: Its index set split is $[2n+1, 4n]$.

3. S_4: Its index set split is also $[2n+1, 4n]$.

4. S_5: Its index set split is $[n+1, 3n]$, as the index sets of S_2 and S_4 are combined with the choice operator, which is interpreted as union. So, the split which reaches S_5 is: $[n+1, 3n] \cup [2n+1, 3n] = [n+1, 3n]$.

Better is using the choice operator to separate complete, single paths. So, the propagation would be performed along every single path: $S_1.S_2.S_5|S_1.S_3.S_4.S_5$. It would cause two splits of $I(S_5)$: $[n+1, 2n]$ and $[2n+1, 3n]$, and the scheduler can find the optimal schedule.

$$\theta\langle S_1, (i)\rangle = \begin{cases} 0 & \text{for } 0 \le i \le 2n \\ 1 & \text{for } 2n+1 \le i \le 4n \end{cases}$$

$$\theta\langle S_2, (j)\rangle = \begin{cases} 1 & \text{for } 0 \le j \le n \\ 2 & \text{for } n+1 \le j \le 3n \end{cases}$$

$$\theta\langle S_3, (k)\rangle = \begin{cases} 1 & \text{for } 0 \le k \le 2n \\ 2 & \text{for } 2n+1 \le k \le 4n \end{cases}$$

$$\theta\langle S_4, (l)\rangle = \begin{cases} 2 & \text{for } 0 \le l \le 2n \\ 3 & \text{for } 2n+1 \le l \le 4n \end{cases}$$

$$\theta\langle S_5, (m)\rangle = \begin{cases} 2 & \text{for } 0 \le m \le n \\ 3 & \text{for } n+1 \le m \le 2n \\ 4 & \text{for } 2n+1 \le m \le 3n \end{cases}$$

Optimizing execution time Kleene's algorithm is of cubic complexity in the number of statements. If we use an algorithm for finding the acyclic condensation of the statement dependence graph, which has a lower complexity, and we use Kleene's only for the strongly connected components, we improve the execution time, although the complexity is still cubic.
Tarjan [12] provides such algorithm with a complexity of O($\max(|N|, |E|)$).

Furthermore, we can reduce the execution time when we compute the transitive closures of the strongly connected components only once for every node of the cycle. We need it for every node of the cycle, as every node is a start of a propagation.

Remark: The final version of this algorithm (see Section 3.3.3) integrates the optimization of the propagation along single paths, if we compute the list of single paths before performing the propagation algorithm. But in our implementation, we decided to neglect the computation of single paths, as it would increase the execution time by an exponential factor (see Section 3.3.3).

3.2.5 When to propagate

Propagating a split means to split a reachable index set according to a previous split. Therefore, all conditions for splitting are also valid for propagation (see Section 3.1.3).

There are special conditions for propagating through cycles.
As structure parameters frequently influence the optimal number of propagations, we propose not to run through a cycle a fixed number of times, but to use the reflexive, transitive closure of this cycle:

1. to find a good approximated split of a program part

2. to find out whether a propagation is useful.

It is possible that the calculation of the desired reflexive, transitive closure has no fixed point or that the closure is not affine. This would mean that a propagation would result in an "unbounded" number of parts of the index set. So, if during the calculation of the closure, we discover that there is no fixed point, we stop propagating through this cycle.

3.3 The final algorithms

In this chapter we connect the optimized versions of the split algorithm and the propagation algorithm. This allows further optimization.

3.3.1 Optimizing the algorithms

The splitting algorithm discovered possible splits for each statement. A side effect occurs if the index sets are changed before the splits were propagated:

Increasing run time Splitting a program part means creating more index sets with their statements. As the execution time of the propagation algorithm depends on the number of statements, we would increase the execution time!

This problem is avoided, when the algorithms only compute the splits and just collect them for every statement. Only when all separate splits are computed, we commence the creation of the program parts for each statement.

3.3.2 The splitting algorithm

The algorithm is shown in Figure 3.27:

The splitting algorithm

FOR every dependence d **DO**

IF (conditions for a split are satisfied) **THEN**

1. Take the polyhedron P_{dst} of the destination statement S_{dst}, which represents the subset within its index set I_{dst}, where the dependence is valid.

2. Add a copy of P_{dst} to a list $L_{Split}(S_{dst})$ for the later execution of the split.

Figure 3.27: The final splitting algorithm

The complexity of the splitting algorithm depends on the number of dependences.

Before computing the complexity of the algorithm, we must analyze the complexity of the description of the polytopes representing the index sets or the source and destination spaces. A polytope is represented by a set of inequalities, one for each face of the polytope. So, the complexity for copying a polytope is linear in the number of inequalities n_i.

Now, we can analyze the splitting algorithm:

1. The outer FOR-loop (1), which is linear: $\mathrm{O}(n_d)$, where n_d is the number of dependences.

2. Adding a copy of P_{dst} to $L_{Split}(S_{dst})$ is linear: $\mathrm{O}(n_i)$, where n_i is the number of inequalities describing the polytope.

Combining the complexity, we get: $\mathrm{O}(n_d) * \mathrm{O}(n_i) = \mathrm{O}(n^2)$, where $n = \max(n_d, n_i)$.

3.3.3 The propagation algorithm

The propagation for every split can be done using the algorithm below.

The propagation-algorithm

FOR every statement S_{source}

  FOR every statement S_{reach}

   which is reachable in G from the split statement S_{source} **DO**

   FOR every $I_s \in L_{Split}(S_{source})$ **DO**

    FOR every path p from S_{source} to S_{reach} **DO**

     IF (conditions for a propagation are satisfied) **THEN**

1. **IF** (p contains no cycle) **THEN**
   Compute the image P_{ST} of I_s along p from S_{source} to S_{reach} within $I(S_{reach})$.
   ELSE
   Compute the image P_{ST} of I_s along p from S_{source} to S_{reach} within $I(S_{reach})$. For every cycle on the way, the reflexive, transitive closure d^* must be added.
2. Add a copy of P_{ST} to a list $L_{Prop}(S_{reach})$ for a later execution of the split

     ELSE

      Stop the further propagation of I_s along p.

Before we can start with the propagation, we need to compute the statement dependence graph G and compute its strongly connected components. P_{source} of the split index set will be used to split reachable statements. Further, we need to compute the reflexive, transitive closure $d^*_{S_i}$ of all dependences in every strongly connected component.

The complexity of the propagation algorithm is polynomial or exponential:

1. The first loop is linear: $\mathrm{O}(n_s)$,
   where n_s is the number of statements.

2. The second loop is linear: $\mathrm{O}(n_s)$

3. The third loop is linear: $\mathrm{O}(n_d)$,
   where n_d is the number of dependences.

4. By the fourth loop there are two cases: If we optimize according to Section 3.2.4, there is an exponential number of paths between S_{source} and S_{reach}; then this loop is $\mathrm{O}(2^{n_d})$. If we neglect the optimization there is maximal only one path combination between two statements; then this loop is constant: $\mathrm{O}(1)$.

5. The inner IF-THEN-part is square, as it depends on the number of statements and, for every statement S_i along the path, the incoming P_{ST} must be mapped to the image inside the index set of S_i. The mapping requires $\mathrm{O}(n_i)$, where n_i is the number of inequalities describing the polytope. So the mapping for n_s statements requires $\mathrm{O}(n_s) * \mathrm{O}(n_i)$.

6. The inner IF-ELSE-part is square like the IF-THEN-part. As the reflexive, transitive closure was computed before by another algorithm, we can add it in linear time.

7. Adding a copy of P_{ST} to $L_{prop}(S_{reach})$ is linear: $\mathrm{O}(n_i)$,
   where n_i is the number of inequalities describing the polytope.

8. Stopping the propagation is constant: O(1).

In combination:
The complexity of the four perfect nested loops is O(n_s) $*$O(n_s) $*$ O(n_d) * cost of the body.
Within this loop body we can summarize the inner if-statement to O(n_s), because both parts of the if-statement require O(n_s). The else-part of the outer if-statement can be neglected as it requires only O(1).
Without approximating the free schedule the complexity is:
O(n_s) * O(n_s) * O(n_d) * O(1) * (O(n_s) + O(n_i)) = O(n^4).
With approximating the free schedule the complexity is:
O(n_s) * O(n_s) * O(n_d) * O(2^{n_d})) * (O(n_s)) + O(n_i)) = O(2^{n_d}).

3.3.4 The program part generation

The two previous algorithms return two lists $L_{split}(S)$ and $L_{prop}(S)$. They contain the subsets of the index set $I(S)$, which it shall be the result of the split.

The program part generator

FOR every statement S **DO**

S_1**:** $I_{set} = I(S)$;

  FOR every $P_l \in L_{split}(S) \cup L_{prop}(S)$ **DO**

    FOR every element I_i of I_{set} **DO**

S_2:   Split I_i into $I_i \cap P_l$ and further convex polyhedra P_i, with

$$\forall P_{i1}, P_{i2} \in \bigcup_i P_i : \; P_{i1} \neq P_{i2} : \; P_{i1} \cap P_{i2} = \emptyset$$

and

$$(\bigcup_i P_i) \cup (I_i \cap P_l) = I_i$$

  FOR every $I_s \in I_{set}$ **DO**

S_3:   Add a copy of S to I_s

$L_{split}(S) \cup L_{prop}(S)$ consists only of splits, but not of the convex, inverse parts of each split. Therefore, we still need to split every I_i into $I_i \cap P_l$ and the convex polyhedra representing $I_i \backslash P_l$, to have finally only convex subsets of $I(S)$.

The complexity:

1. The first loop is linear: $O(n_s)$,
   where n_s is the number of statements.

2. Statement S_1 is constant

3. The second loop is square: $O(n_d * n_s)$,
   as $L_{split}(S) \cup L_{prop}(S)$ increases during the splitting and the propagation algorithm to $n_d(= |dependences|) * n_s(|\text{statements}|)$ parts.
   The splitting algorithm can produce for each single statement maximal as many splits as there are dependences. During the propagation maximal every statement can propagate its splits to S.

4. The third loop is exponential: $O(2^{n_l})$.
   For each iteration l, the number of elements of I_{set} is doubled, so finally we must split $O(2^{n_l})$ parts, where $n_l = |L_{split}(S) \cup L_{prop}(S)|$.

5. Statement S_2 is linear: $O(n_i)$.
   Splitting $I_i \cap l$ is constant.
   The remaining index set can be non-convex. Splitting it into convex parts results in the worst case in as many parts as there are inequalities describing the split I_i.

6. The fourth loop is exponential: $O(2^{n_l})$,
   as I_{set} consists of 2^{n_l} parts.

7. Statement S_3 is constant.

Summarizing the complexity, we get:
$O(n_s) * (\ O(1) + ((\ O(n_d * n_s) * O(2^{n_l}) * O(n_i)\ ) + (\ O(2^{n_l}) * O(1)\ ))) = O(2^{n_l})$

3.3.5 Non-convex index sets

In Section 3.1.2, we found that splitting an index set can result in a non-convex remaining set. Our splitting and propagation algorithm only computes and propagates the image of the destination space, as this is enough to make a further parallelization possible.
If our main goal is the free schedule, we also must propagate the image(s) of the remaining index set. The free schedule executes instances as early as possible. Therefore, the index set of a statement S_{reach}, which is reachable from a statement S_{source}, should be split into all images of the splits of $I(S_{source})$. Only then is it possible to start the execution of a split of $I(S_{reach})$ at once, as soon as all instances of the belonging split of $I(S_{source})$ are executed.

Why did we not include this in our algorithms, if it would approximate the free schedule more tightly?
As we have seen in Section 3.3.4, computing a split and its remaining convex polyhedra results in an exponential loop and $L_{Split}(S)$ grows to a set of exponential parts. So, already the splitting algorithm is exponential.
Further, every element of $L_{Split}(S)$ must be propagated. The loop on $L_{Split}(S)$ would become exponential. Contrary to the splitting algorithm, we do not need to compute the remaining convex polyhedra here, as they are not necessary for further propagations. Therefore, the propagation algorithm becomes exponential.
Finally, the program part generator would have to handle $L_{Split}(S)$. For every element of the exponential list we would need to compute the splits and its remaining convex polyhedra, as the propagation could have been creating splits, which again leave a non-convex index set. So the program part generator would be doubly exponential!
Performing the splitting algorithm and propagating those splits, before the index sets are changed, makes a parallelization possible. To increase our algorithms by an exponential factor, only to be a bit closer to the free schedule is not our goal.

3.3.6 Connecting the algorithms

Summarizing the single algorithms we get the following schedule for splitting the index sets to allow high parallelism or even to come close to the free schedule:

Input: - all dependences
- every statement with its index set $I(S)$

1. Perform the splitting algorithm.
2. **IF** the splitting algorithm found any split **THEN**
   (a) Compute the statement dependence graph.
   (b) Compute all paths within the statement dependence graph using Tarjan [12] and Kleene [5].
   (c) For coming close to the free schedule optimize every path combination to a list of single paths (see Section 3.2.4).
   (d) Perform the propagation algorithm.
   (e) Generate the program parts.

All in all, the combination of all algorithms is of a exponential complexity; for approximating the free schedule even doubly exponential.

Chapter 4

Implementation

In this chapter we sketch a possible implementation of index set splitting within the polyhedron model.

4.1 Refining the algorithm

For now, the splitting and the propagation algorithm make use of the dependence analysis within polyhedron methods, and they are changing the source code when they split the index sets. Plugging the algorithms into LooPo, our implementation of the polytope model, we have the problem of keeping LooPo's internal structure consistent. As we change the source code, we also have to update the existing dependences and some new statements need to have copies of updated dependences.

There are two ways for obtaining correct dependences: Either we write the changed source code to a file, and we run the scanner, parser and the dependence analysis again or we update the dependences during the program part generation, when the splitting and propagating algorithms have terminated. Surely, the updating is more effective, and we decided for it.

A first attempt expands our program part generator as follows:

The expanded program part generator

Input: - The set of all dependences D_{set}

- every statement with its index sets I_{dst}

- $L_{split}(S), L_{prop}(S)$

FOR every statement S **DO**

  I_{set} = **Index** $I(S)$ **of** S**;**

  FOR every $l \in L_{split}(S) \cup L_{prop}(S)$ **DO**

    Split every I_i **of** I_{set} **into**

    $I_i \cap l$ **and** $I_i \backslash l$**.**

  FOR every $I_s \in I_{set}$ **DO**

    Add a copy of S to I_s

    FOR every $d \in D_{set}$ **DO**

      IF dst(d) is S **THEN**

        IF ($dst_{spc}(d) \cap I_s \neq \emptyset$) **THEN**

          Add a copy d_{copy} of d to a new set

          of dependences D_{new}, where

          $dst_{spc}(d_{copy}) = dst_{spc}(d) \cap I_s$.

        ELSE

          Add d to D_{new}.

    $D_{set} = D_{new}$.

Therefore its complexity changes as follows:

1. The added loop is exponential: $O(2^{|I_s|})$.
   As for every I_s we add maximal $|D_{set}|$ copies of dependences to D_{set}, D_{set} doubles its elements for every I_s.

2. The statements within the new loop are constant

Summarizing: we have an exponential loop inside an exponential loop, we are doubly exponential: $O(2^{2^n})$.

4.2 Improving the algorithm

The reason, that the dependence loop is exponential, is that we always consider all dependences, although we need only a small subset of them.
As newly created dependences belong only to the actual statement S, we collect them separately, because S will not be considered anymore. The new list of dependences D_{new} does not need to contain unused dependences anymore, as was necessary before. But, if a statement was not split, we add d without copying and changing.
Now the new FOR-loop is linear in the number of the original dependences found by the dependence analysis. So the whole algorithm is only exponential (see Figure 4.1).

4.3 The reflexive, transitive closure

The propagation algorithm needs to know all possible paths from any statement to any other. As shown in Section 3.2.3, we need to compute the reflexive, transitive closure when there are cycles in the statement dependence graph.
There are already tools for computing closures. So, we only need to plug such a tool into LooPo. This way, improved versions of mathematical help programs can easily be inserted.
For computing the reflexive, transitive closures we used the Omega Library [8], a set of C++ classes for manipulating integer tuple relations and sets. We chose it for its simplicity of generating relations, representing dependences, combining them and computing closures of relations. The results are unions of relations, which is already a representation of convex polytopes.

The improved program part generator

FOR every statement S **DO**

$\quad I_{set} = $ **Index** $I(S)$ **of** S**;**

$\quad$ **IF** $L_{split}(S) \cup L_{prop}(S) \neq \emptyset$ **THEN**

$\quad\quad$ **FOR** every $l \in L_{split}(S) \cup L_{prop}(S)$ **DO**

$\quad\quad\quad$ **Split every** I_i **of** I_{set} **into** $I_i \cap l$ **and** $I_i \backslash l$**.**

$\quad\quad$ **FOR** every $I_s \in I_{set}$ **DO**

$\quad\quad\quad$ Add a copy of S to I_s

$\quad\quad\quad$ **FOR** every $d \in D_{set}$ **DO**

$\quad\quad\quad\quad$ **IF** the dst(d) is S **THEN**

$\quad\quad\quad\quad\quad$ **IF** ($dst_{spc}(d) \cap I_s \neq \emptyset$) **THEN**

$\quad\quad\quad\quad\quad\quad$ Add a copy d_{copy} of d to a new set

$\quad\quad\quad\quad\quad\quad$ of dependences D_{new}, where

$\quad\quad\quad\quad\quad\quad$ $dst_{spc}(d_{copy}) = dst_{spc}(d) \cap I_s$.

$\quad$ **ELSE**

$\quad\quad$ **FOR** every $d \in D_{set}$, where dst(d) $= S$ **DO**

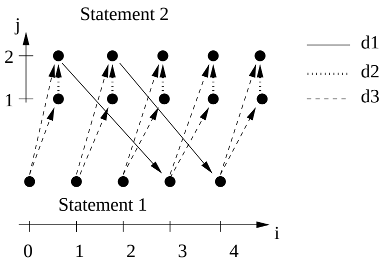
$\quad\quad\quad$ add d to D_{new}.

$\quad D_{set} = D_{new}$**.**

Figure 4.1: The improved program part generator

Furthermore, we plan to use Omega not only for index set splitting. Also different modules of LooPo, e.g. the dependence analysis, will use Omega.
Let us use the example in Figure 3.17 with its dependences shown in 3.18 to demonstrate the simplicity of Omega. We use the Omega Calculator [7] for the computation:
Let us mention the source program and its dependences:

for $i := 0$ to $2n$
S_1 : $A[i, 0] := B[i-n-1, m];$
for $j := 1$ to m
S_2 : $B[i, j] := B[i, j-1] + A[i, 0];$



The three dependences are defined in Omega as:

$d_1 := \{[i, j] \rightarrow [i`] : 0 \leq i, i` \leq 2n \quad \&\& \quad i` = n+i+1 \quad \&\& \quad 1 \leq j \leq m\}$
$d_2 := \{[i, j] \rightarrow [i, j-1] : 0 \leq i \leq 2n \quad \&\& \quad 1 \leq j \leq m\}$
$d_3 := \{[i] \rightarrow [i`, j`] : 0 \leq i, i` \leq 2n \quad \&\& \quad i` = i \quad \&\& \quad 1 \leq j` \leq m\}$

The dependences d_1 and d_3 in combination define a cycle within the statement dependence graph. Starting in S_1 we combine the dependences:

$$d_{S_1} := d_1.d_3;$$

starting in S_2 we do opposite:

$$d_{S_2} := d_3.d_1;$$

The closure easily can be computed by Omega:

$$d^*_{S_1} := d_{S_1}+;$$
and
$$d^*_{S_2} := d_{S_2}+;$$

Then we can split the index set of S_1 according to $d^*_{S_1}$, which is:

$$\{[i]->[n+i+1] : 0 \le i < n \quad \&\& \quad 1 \le m\}$$
$$\Leftrightarrow \{[i]->[i] : n{+}1 \le i < 2n{+}1 \quad \&\& \quad 1 \le m\};$$

And $I(S_2)$ we split according to $d^*_{S_2}$, which is:

$$\{[i,j]->[n+i+1, j`] : 0 \le i < n \quad \&\& \quad 1 \le j, j` \le m\};$$
$$\Leftrightarrow \{[i,j]->[i, j`] : n{+}1 \le i < 2n{+}1 \quad \&\& \quad 1 \le j, j` \le m\};$$

So, $I(S_1)$ is split into:

$$\{[i] \mid 0 \le i \le n\} \text{ and } \{[i] \mid n{+}1 \le i \le 2n\}$$

$I(S_2)$ is split into:

$$\{[i,j] \mid 0 \le i \le n \ \wedge \ 1 \le j \le m\} \text{ and } \{[i,j] \mid n+1 \le i \le 2n \ \wedge \ 1 \le j \le m\}.$$

This is the identical result like we found it in Section 3.2.1.

Chapter 5

Conclusions

In this work we introduced index set splitting as a method for adapting programs in a way that parallelization methods can work more effectively.
A first method of index set splitting was published by Jemni and Mahjoub [11], who developed a method for parallelizing static conditional loops. Since their method is not based on a mathematical model, its usability is rather limited.
We adopt their method in order to plug it into the polyhedron model [10]. This expands the applicability of index set splitting on using it on loop programs, which contain loop bounds and arrays based on affine expressions.

High parallelism. Index set splitting is not a method for changing sequential source code into parallel destination code. It only divides the source code into parts, which belong either to a group whose instances all depend on some instances according to a dependence d or to the group whose instances are freely executable according to d. Now there are only index sets, which are either fully or not at all influenced by dependence d, hence, some parts can be parallelized directly as now they are without dependence, and on every remaining part different methods can be tried for a parallelization.

The limits. The effectiveness of index set splitting partially depends on information extractable by the dependence analysis. If we do not know the boundaries of the surrounding loops, there are cases in which we cannot decide whether splitting is useful or leads to more communication (see Section 3.1.3).
Another limitation is given by demanding precision. Our goal is to gain parallelism, but not to approximate the free schedule at any cost. Such approximation would expand the index set splitting to a doubly exponential complexity.
Of course, one cannot guarantee to find the optimal solution at compile time.

Expanded code. In Section 3.1.2 we have shown that the splitting is done for each statement separately. Every statement has its own index set. When we split an index set, we insert an *if* condition into the source code where both the *then* part and the *else* part contain a copy of the statement. But more code leads to longer execution times. So, in some extreme situations, index set splitting causes a slowdown rather than a speedup.

All in all, index set splitting is a useful method for preparing programs for their parallelization.

Bibliography

[1] Utpal Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, 1996.

[2] Paul Feautrier. Dataflow analysis of array and scalar references. In *International Journal of Parallel Programming*, volume 20, pages 23–53, 1991.

[3] Paul Feautrier. Some effizient solutions to the affine scheduling problem, part one, one-dimensional time. In *International Journal of Parallel Programming*, volume 21, pages 313–347, 1992.

[4] Paul Feautrier. Some effizient solutions to the affine scheduling problem, part two, multidimensional time. In *International Journal of Parallel Programming*, volume 21, pages 389–420, 1992.

[5] Robert W. Floyd and Richard Beigel. *The Language of Machines – An Introduction to Computability and Formal Languages.* CSP, 1994.

[6] Harald Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Diploma thesis, University of Passau, 1997.

[7] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Calculator and Library.* http://www.cs.umd.edu/projects/omega, November 1996.

[8] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library.* http://www.cs.umd.edu/projects/omega, April 1996.

[9] Daniel N. Lapedes. *McGraw-Hill dictionary of physics and mathematics.* McGraw-Hill Book Company, 1978.

[10] Christian 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.

[11] Zaher Mahjoub and Mohamed Jemni. Restructering and parallelizing a static conditional loop. In *Parallel Computing*, volume 21, pages 339–347, February 1995.

[12] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, 1990.