Technische Universität München
Fakultät für Informatik

Diplomarbeit

Methods for Adaptive Tiling in the Polyhedron Model

Bearbeiter:
Georg Seidel

Aufgabensteller:
Prof. Dr. Michael Gerndt
Lehrstuhl für Rechnertechnik und
Rechnerorganisation/Parallelrechnerarchitektur
Technische Universität München

Betreuer:
Dr. Martin Griebl
Universität Passau

Abgabedatum:
15. September 2004

Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Passau, den 14. September 2004

(Georg Seidel)

Contents

Abstract

Motivated by the difficulty of writing parallel code for distributed memory machines by hand and by the trend towards parallel architectures in general, automatic loop parallelization is an active field of research. Over the last years the polyhedron model, a powerful framework for loop parallelization, has been extended in different ways, broadening its applicability and refining the used algorithms. Recent work on non-linear inequality systems, which allows the efficient treatment of parameters as coefficients, makes it possible to investigate new techniques to mitigate shortcomings of existing methods.

In this thesis we investigate the field of adaptive space tiling, which aims to improve the load balance by adapting the space tiling over time. First we provide the theoretical background and solve the underlying problem of automatic static space tiling. We develop different methods of doing adaptive tiling, apply the developed methods to selected programs, and give experimental results.

Acknowledgements

I would like to thank my tutor Dr. Martin Griebl for the discussions we had, for the proofreading, and for his remarks and suggestions. Furthermore, I thank the members of the LooPo team for their ideas, feedback and discussions: Armin Größlinger, Peter Faber, Michael Claßen, Tobias Langhammer, and Thomas Wondrak. Finally I thank Thorsten Lampe for proofreading and for many valuable stylistic remarks, and the members of channel `#maulwurf` for their help with LaTeX and programming issues.

Chapter 1

Introduction

The underlying topic of this thesis is automatic program parallelization. Our research is based on the polytope model which is a mathematical framework for loop parallelization [Len93], and on LooPo [LooPo], a prototype of a parallelizing compiler developed at the University of Passau.

Without going into the details of the polytope model, which is described in section 2.4, we give a short overview: a sequential source program is first analyzed and transformed into a mathematical model. This model is transformed to expose the parallelism, and the resulting model is converted back into a parallel target program. The exact mathematical representation of the program and the transformation is not important here.

The polytope model exposes a very fine-grained parallelism on the basis of single atomic operations, for example multiplications and additions. On existing distributed memory parallel computers it would be very inefficient to perform such a fine-grained parallelism: imagine two additions are performed in parallel on different processors, and the result is needed on a third. Then the resulting two communications are far more expensive than the additions themselves. So it would have been better to perform the additions sequentially on the third processor instead of performing them in parallel (provided that the input operands of the additions are available on the third processor without communication). To address this problem tiling is applied. Tiling (described in chapter 3) encompasses techniques to coalesce the parallel operations to allow a more efficient execution. In the form of tiling we consider operations are grouped into tiles, and these tiles are executed in parallel.

Figure 1.1 is a schematic representation of the parallelism exposed by the polytope model. The black dots represent the operations of the source program (these could be the just mentioned additions). The horizontal axis represents time. The operations must be executed in the order of the time axis. The vertical axis represents the parallel dimension. All operations in a vertical line can be executed in parallel. At time 0, for example, only one operation can be executed while at time 8 there are seven parallel operations.

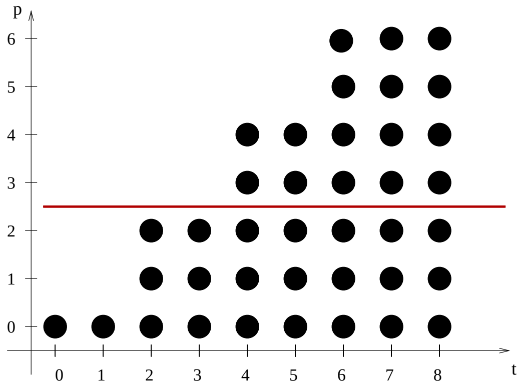


Figure 1.1: The parallelism found by the polytope model

The bold horizontal line in figure 1.1 represents a possible tiling of the operations. At every point in time all operations below the line belong to one tile, and the operations above belong to the other. When executed on a dual-processor machine the parallel program could execute all operations below the line on processor one, and all operations above on processor two. An important property of this specific tiling is that it is *static* which means that it does not change with time (i.e. the line is horizontal).

A disadvantage of such a static tiling is that the load balance between processors can be very poor. In the figure we can see that during time steps 0 to 3 only the first processor has got any work to do while the second processor is idle.

To improve the load balance in these situations we develop methods for *adaptive* tiling in this thesis. Adaptive, in contrast to static, means that the tiling can change with time. Figure 1.2 illustrates a possible adaptive tiling of our example. Again there are two tiles in each time step. But now the tiling can no longer be represented by a horizontal line because it changes at certain points in time.

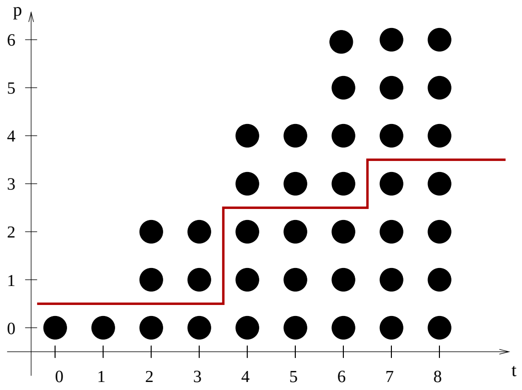


Figure 1.2: Adaptive tiling

Our approach to adaptive tiling is made possible by recent work of Armin

Größlinger [Grö03], which allows treating non-linear inequality systems with parameters as coefficients. Interestingly, before tackling adaptive tiling we had to solve automatic static space tiling, which already requires most techniques we need for adaptive tiling.

1.1 Outline

To facilitate reading this thesis we give a short outline of the next chapters.

In **chapter 2** we present the foundations that are necessary for the later chapters. We start with terminology about programs and statements followed by an introduction to program dependencies. We cover some basic definitions about linear inequalities and polyhedra, about parameters in inequality systems, and about logic.

Thereafter we describe the polytope model, with a focus on the generation of communication code. The chapter closes with the extensions that have been developed to the polytope model.

Chapter 3 is all about tiling. After an overview we concentrate on the problem of automatic static space tiling. To our surprise we encountered several unsolved problems here. While there exist techniques to deal with static space tilings for a given fixed tile shape, we did not find substantial results on automatic scaling of the tile shape and on mapping tiles to physical processors.

We investigate how to scale a tile shape, which lattice matrices are valid, and what characterizes a good lattice matrix. Because the scaling of the tile shape, and the mapping to physical processors cannot be treated independently, we evaluate different approaches to solve these two problems together, and come up with a simple algorithm.

In the **fourth chapter** we introduce adaptive tiling. After a motivation, we discuss the required tools. We investigate communication generation and propose a communication library to reduce the complexity of the generated code.

Subsequently we develop three different methods to improve the load balance of parallel programs and discuss them.

This chapter adapts the algorithm we found in chapter three and applies it in the context of adaptive tiling.

Chapter 5 introduces the test programs we used to evaluate our methods empirically and presents benchmark results, followed by a discussion.

Chapter 2

Foundations

In this chapter we first introduce an informal model of the programs we want to parallelize. Next we give definitions and terms needed throughout this thesis, before we describe the polytope model and the extensions that have been developed so far, resulting in the polyhedron model.

2.1 About Programs and Statements

2.1.1 Semantics and Costs

Parallelization of sequential code is, in its essence, program transformation. We transform a sequential program into a parallel one. To be useful the transformation has to fulfill two properties. The transformed program

(I) must be semantically equivalent to the sequential program, and

(II) should execute faster on a given parallel machine.

To formalize (I), the *correctness* of the transformation, we need to specify the semantics of the sequential programs we accept for transformation, and the parallel programs we create. To formalize (II), we need cost models, both for the sequential program and for the parallel one.

Correctness Since the complete formal specification would go beyond the scope of this thesis, we just give links to relevant literature.

The sequential input we accept is a subset of common imperative programs (the restrictions are described below). Semantics for this kind of programs can be found in [Alb88].

For the parallel programs we produce we assume a distributed memory target architecture with explicit communications. For formal semantics of parallel programs see [Be96].

We do not give a formal correctness proof for our transformation. Instead, we formulate a correctness criterion in terms of dependencies in section 2.1.5.

Cost Model Griebl and Ellmenreich developed cost models that are suitable for our purpose [Gri04, Ellm04]. We refer to our discussion in section 4.2.

2.1.2 An Informal Model for Programs

The programs we take as input to parallelization are a restricted subset of imperative programs. Our main focus is on nested loops, as they occur in many algorithms of scientific computing and promise to have most potential for parallelization.

We use a pseudo-code notation borrowed from Pascal that should be easily understandable by a reader who is familiar with one of the well known imperative programming languages (see program 1).

As a general restriction, we only allow `for` loops with unit stride and affine loop bounds. Extensions to `while` loops and rational strides are presented in section 2.6. We also do not deal with recursive functions and we treat function calls in loop bodies as atomic statements (further assuming that they are free of side-effects).

2.1.3 Statements and Operations

Remark 2.1 Throughout this thesis we use the term **structure parameter**, or simply **parameter**, in the meaning of symbolic constants in programs, which are unknown at compile time.

When working with statements in loop nests we need exact terminology in order to avoid ambiguity. We concentrate on loop nests in which the loop bounds are linear expressions in surrounding variables and parameters.

Program 1 motivates the following definitions.

```
       for i := 1 to 4
         for j := 1 to 7
S1:        C[i] := C[i] + A[j] + B[i+2*j, j]
         end
S2:      A[i] := 3*C[i]
       end
```

Program 1: A simple loop nest

The first thing that can be said about program 1 is that it contains two nested `for` loops and two statements `S1` and `S2`. We also note that the statements are in different surrounding loops.

This leads to the first definition.

Definition 2.2 A loop nest is called **perfectly nested**, or **perfect loop nest**, if and only if every loop body is either

- one perfect loop nest or

- a sequence of statements without loops.

Thus, program 1 is not perfectly nested: the `i`-loop contains one loop (which is perfectly nested) and one statement (`S2`).

Sometimes it is not enough to treat statements as atomic units. To study the interactions of different statements we analyze which **array accesses** a statement contains. Array accesses can further be classified into read and write accesses. Look at program 2, which has a single statement. This statement has five array accesses, which we count from left to right. The first array access is a write access, the remaining accesses are all read accesses.

```
...
  A[i, j, k] := A[i-1, j, k+1] - B[i]*B[j] + C[k, i]
...
```

Program 2: statement with five array accesses

The next definition helps to characterize an array access further.

Definition 2.3 Let r be an array access, let `A` be the l-dimensional array of that access, and let d be the number of surrounding loops. Then we can represent r by an affine linear function

$$a_{A,r} : \mathbb{R}^d \to \mathbb{R}^l,$$

called the **access function**. The corresponding matrix, denoted by $\mathcal{A}_{A,r}$, is called the **access matrix**.

For example, the array access `A[i,2*j,3*i + k]` with loop variables `i`, `j`, `k` has the access matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 3 & 0 & 1 \end{pmatrix}$.

So far we were concerned with the static structure of a program. But we also want to investigate dynamic aspects. In particular we want to be able to distinguish different runtime instances of statements.

Definition 2.4 Let `S` be a statement in a loop program, and let n be the number of surrounding loops. Then the **iteration vector** of `S` is an n-dimensional vector, where each dimension holds the index value of one surrounding loop, from outside in.

In program 1 the statement `S1` has iteration vector (i, j) and statement `S2` has iteration vector (i).

Definition 2.5 Let S be a statement and i be an iteration vector. Then we call the runtime instance of S at iteration i, written as $<i\,;S>$, an **operation**. We denote the **set of all operations** of a program with Ω. Let I be the set of all iteration vectors, and let Υ be the set of all statements. We use the functions $\mathrm{iv} : \Omega \to I$ and $\mathrm{stmt} : \Omega \to \Upsilon$, defined by $\mathrm{iv}(<i\,;S>) = i$ and $\mathrm{stmt}(<i\,;S>) = S$.

Definition 2.6 An iteration vector i of a statement S is called **legal**, if and only if there is a runtime instance of S with that iteration vector when actually running the program. In the same way the corresponding operation $<S\,;i>$ is called legal. For a statement S, the set of all legal iteration vectors, $\mathcal{I}(S)$, is called the **index set** of S (sometimes also called **index space**).

In our example, the index sets for `S1` and `S2`, and the set of all operations Ω are given by:

$$\begin{aligned}\mathcal{I}(S1) &= \{\, (i,j) \in \mathbb{N}^2 \mid 1 \leq i \leq 4 \wedge 1 \leq j \leq 7 \,\},\\ \mathcal{I}(S2) &= \{\, i \in \mathbb{N} \mid 1 \leq i \leq 4 \,\},\\ \Omega &= \{\, <k\,;S> \mid (k \in \mathcal{I}(S1) \wedge S = S1) \vee (k \in \mathcal{I}(S2) \wedge S = S2) \,\}.\end{aligned}$$

2.1.4 Modeling Dependencies

After looking at operations in isolation, the next step is to investigate the dependencies between operations. The knowledge of the dependency structure of a program is essential for producing a correct parallel version.

First we need to specify the sequential program order.

Definition 2.7 Let Ω be the set of all operations of a program, and let $u, v \in \Omega$ be two operations. The relation $<_{seq} \subseteq \Omega \times \Omega$ is defined by

$$u <_{seq} v \iff u \text{ is executed before } v \text{ in the sequential program.}$$

$\leq_{seq}$, the reflexive closure of $<_{seq}$, is a total order on Ω (i.e. it is reflexive, antisymmetric, transitive, and total).

Definition 2.8 Let Ω be the set of all operations of a program, and let $u, v \in \Omega$ be operations (not necessarily from the same statement). Then u and v induce a **dependency** if the following conditions hold:

1. $u <_{seq} v$ and
2. u and v access the same array cell $\mathcal{C}$ in memory. That is, they must either read from or write to $\mathcal{C}$ (since we do not allow pointers or references, there are no aliasing issues).

If these conditions are satisfied, we say that v depends on u, written as $u\ \delta\ v$, where $\delta \subseteq \Omega \times \Omega$ is the **dependency relation**. u is called **source** of the dependency, v is called **target** of the dependency.

Dependencies can be classified in the following way (using the notation of definition 2.8):

- If u reads from $\mathcal{C}$, and v reads from $\mathcal{C}$, the resulting dependency is called an **input dependency**.

- If u writes to $\mathcal{C}$, and v reads from $\mathcal{C}$, the resulting dependency is called a **true dependency**.
- If u reads from $\mathcal{C}$, and v writes to $\mathcal{C}$, the resulting dependency is called an **anti dependency**.
- If u writes to $\mathcal{C}$, and v writes to $\mathcal{C}$, the resulting dependency is called an **output dependency**.

Feautrier provides more details [Fea91].

Remark 2.9 If we assume that reads on the same memory cell can be done concurrently in our target architecture, we can ignore all input dependencies. If there is no operation w between u and v ($u <_{seq} w <_{seq} v$) that modifies $\mathcal{C}$, then the dependency is called **direct**. A true dependency that is direct is also called a **flow dependency**.

There are several ways to model dependencies, differing mainly in precision. We use the most precise one, called h-transformation [Fea91]. Griebl provides a good overview [Gri04].

Definition 2.10 Let S and T be two (not necessarily different) statements, and let d_S and d_T be the dimensionalities of their respective index spaces. An **h-transformation** is a pair $h = (f_h, D_h)$, where f_h is an affine function, $f_h : \mathbb{R}^{d_T} \to \mathbb{R}^{d_S}$, and D_h is the domain of f_h, given as a non-empty polyhedron. The function f_h maps the iteration vector of the target operation of a dependency to the iteration vector of its source operation.

Remark 2.11 h-transformations aggregate dependencies between the operations of two statements. Following Griebl, we use the term *dependency* for both dependencies between operations, and aggregated dependencies between statements. We distinguish, if necessary, by using the terms *dependency instance* for dependencies between operations, and *dependency aggregation* for dependencies that have been aggregated by an h-transformation [Gri04].

Remark 2.12 h-transformations are very precise. If all array accesses and loop bounds of our source program are affine, h-transformations are exact. If we allow non-affine accesses (e.g. an array access indexed by an array – `A[B[i]]` – or by a complex function – `A[f(i)]`), the dependency can still be conservatively overestimated with h-transformations.

An important special kind of dependency relations are so-called *uniform dependencies*. Since they play a role in the polytope model we give a definition here.

Definition 2.13 Let $h = (f_h, D_h)$ be the h-transformation of a dependency relation in a perfect loop nest. The index spaces of the source and target statement have the same dimensionality d. Then the dependency relation is called **uniform** if and only if

$$\exists\, c \in \mathbb{R}^d : \forall\, t \in D_h : t - f_h(\, t\,) = c.$$

So for a uniform dependency all dependent source and target operations have a constant distance vector. It follows that f_h must be of the form $f_h : t \mapsto t - c$ for a suitable $c \in \mathbb{R}^d$.

Remark 2.14 Uniform dependencies are defined only for perfect loop nests. It would be possible, however, to extend the definition to imperfect loop nests if we required that the source and target statement have the same surrounding loops and hence index spaces with the same dimensionality.

2.1.5 Dependencies and Correctness

Here we give a correctness criterion for program parallelization. As we already said, we don't give the exact formal semantics of sequential and parallel programs, but just assume that the sequential and parallel program are executed in terms of atomic operations.

We require that there is a unique global order relation of operations in the parallel case (for the sequential case this is obviously true).

Let Ω be the set of all operations that are executed in the sequential program, and let $<_{seq} \subseteq \Omega \times \Omega$ be the order of sequential program execution as in definition 2.7. Let further $<_{par} \subseteq \Omega \times \Omega$ be the order of the parallel program execution, and let finally $\delta \subseteq \Omega \times \Omega$ be the dependency relation as in definition 2.8.

Then the following condition must be satisfied:

$$\forall\, u, v \in \Omega : (u <_{seq} v \;\wedge\; u\, \delta\, v) \;\Rightarrow\; u <_{par} v.$$

Or, even simpler, because for $u, v \in \Omega$, $u\, \delta\, v \;\Rightarrow\; u <_{seq} v$:

$$\forall\, u, v \in \Omega : u\, \delta\, v \;\Rightarrow\; u <_{par} v. \tag{2.1}$$

In set notation this can be put down more elegantly:

$$\delta \subseteq <_{par} .$$

In other words: the dependency relation δ describes the constraints that *must* be satisfied, $<_{par}$ must respect them and can add further constraints to the execution order.

Remark 2.15 We chose the non-reflexive symbol $<$ to indicate that we want a non-reflexive relation. For two operations u and v which are executed in parallel, we have neither $u <_{par} v$ nor $v <_{par} u$. Formally $<_{par}$ is a transitive relation, but it is not reflexive and antisymmetric, as would be required for an order relation.

Based on remark 2.15, we can define a relation $\backsim_{par} \subseteq \Omega \times \Omega$:

$$\forall\, u, v \in \Omega : u \backsim_{par} v \iff (u \not<_{par} v \wedge v \not<_{par} u).$$

We *require* that this is an equivalence relation (which has consequences for the valid choices of $<_{par}$). This equivalence expresses which operations are executed in parallel: the operations inside one equivalence class.

For an operation $u \in \Omega$, we denote its representative in $\backsim_{par}$ with $[u] : \Omega \to \Omega/_{\backsim_{par}}$ (for example we could choose the least element with respect to $<_{seq}$).

From an operational perspective, the pseudo code for a parallel program that implements the order of $<_{par}$ could look like this:

```
for [u] ∈ Ω/∽par
   parfor v ∈ [u]
      v
   end
end
```

This program enumerates the equivalence classes sequentially, and executes all operations in one class in parallel. Communications that could be necessary due to distributed memory are not addressed so far.

Remark 2.16 Why do we require that $\backsim_{par}$ is an equivalence? Because we want $<_{par}$ to unambiguously order the operations into equivalence classes. If we dropped this requirement, $<_{par}$ might not put any constraints on an operation that has no dependencies on other operations. With this requirement, we force $<_{par}$ to assign this independent operation to one equivalence class, so we know when it will be executed. The reader can easily verify that for $<_{par} \stackrel{\text{def}}{=} <_{seq}$, $\backsim_{par}$ is an equivalence (because in that case $\backsim_{par}$ is really equality). But, for $<_{par} \stackrel{\text{def}}{=} \delta$, this is not true in general. As a counter example, let $\Omega \stackrel{\text{def}}{=} \{\, u, v, w \,\}$ and $\delta \stackrel{\text{def}}{=} \{\, (u, w)\}$.

Remark 2.17 Equation 2.1 is, of course, only a necessary condition for correctness, not a sufficient one: We did not deal with communications, which are necessary in the general case of distributed memory machines.

2.2 Linear Inequalities and Polyhedra

We introduce polyhedra via linear inequalities, because this is their most common representation in the polyhedron model: the loop bounds of the source program describe the inequalities that bound the index space polyhedron.

In the following, for a matrix $M \in \mathbb{R}^{k \times l}$, we use the notation $M^T \in \mathbb{R}^{l \times k}$ to denote the *transposed* of this matrix. We also identify a vector $x \in \mathbb{R}^k$ with a matrix $x \in \mathbb{R}^{1 \times k}$ that has one row and k columns. This allows us to use x^T to denote the corresponding matrix $x^T \in \mathbb{R}^{k \times 1}$, and, for two vectors $x, y \in \mathbb{R}^n$, to express the standard scalar product in $\mathbb{R}^n$ by a matrix product $x \cdot y^T$.

2.2.1 Linear Inequalities

Definition 2.18 Let $a, x \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$. Then

$$a \cdot x^T \geq \alpha$$

or equivalently

$$\sum_{i=1}^{n} a_i \cdot x_i \geq \alpha$$

is called a **linear inequality**. $a_1, \ldots, a_n$ are called the **coefficients**, $x_1, \ldots, x_n$ are called the **unknowns**, and α is called the **additive term**. Analogously, if we use $=$ instead of $\geq$, we get the definition for **linear equations**.

Remark 2.19 In the above definition, a linear inequality is represented by a vector in $\mathbb{R}^n$ and a scalar in $\mathbb{R}$. If we deal with multiple linear inequalities, represented by $a_1, \ldots, a_m \in \mathbb{R}^n$ and $\alpha_1, \ldots, \alpha_m \in \mathbb{R}$, it is convenient to store this **linear inequality system** in a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $\alpha \in \mathbb{R}^m$ such that $A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$ and $\alpha = (\alpha_1, \ldots, \alpha_m)$. With this convention, we can represent the conjunction of all inequalities by a simple matrix-vector product. Let $x \in \mathbb{R}^n$, then:

$$\bigwedge_{i=1}^{m} a_i \cdot x^T \geq \alpha_i \iff A \cdot x^T \geq \alpha^T.$$

Definition 2.20 Let ℓ be a non-zero linear function from $\mathbb{R}^n$ to $\mathbb{R}$ (also called a linear form), and let $\alpha \in \mathbb{R}$. For suitable $a \in \mathbb{R}^n \setminus \{0\}$, we have $\ell : x \mapsto a \cdot x^T$. Then we call

$$A(\ell, \alpha) \stackrel{\text{def}}{=} \ell^{-1}(\{\alpha\})$$

a **hyperplane**. Further,

$$H(\ell, \alpha) \stackrel{\text{def}}{=} \ell^{-1}([\alpha, \infty[)$$

is called a **half-space** of $\mathbb{R}^n$.

It is easy to see that hyperplanes correspond directly with linear equations, and that half-spaces correspond with linear inequalities. For example in $\mathbb{R}^2$, given $a = (3, 4)$, $\alpha = 2$ and $\ell : \mathbb{R}^2 \to \mathbb{R}$, $\ell : (x_1, x_2) \mapsto (3, 4) \cdot (x_1, x_2)^T = 3x_1 + 4x_2$, we can see that

$$A(\ell, \alpha) = \{\, (x_1, x_2) \in \mathbb{R}^2 \mid (3, 4) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = 2 \,\}.$$

That is, the hyperplane is the set of all points that satisfy the linear equation defined by a and α. Analogously, the half-space

$$H(\ell, \alpha) = \{\, (x_1, x_2) \in \mathbb{R}^2 \mid (3, 4) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq 2 \,\}$$

is the set of all points that satisfy the linear inequality defined by a and α.

2.2.2 Polyhedra

Now we are ready to introduce polyhedra as finite intersections of half-spaces.

Definition 2.21 Let $m \in \mathbb{N}$, and let $H_1, \dots, H_m$ be half-spaces in $\mathbb{R}^n$. Then

$$P \stackrel{\text{def}}{=} \bigcap_{i=1}^{m} H_i$$

is called a **polyhedron**. If P is bounded, we also call it a **polytope**.

As we have already seen half-spaces closely correspond with linear inequalities, so it not surprising that we can give a different but equivalent characterization of polytopes in terms of linear inequalities: Let $m \in \mathbb{N}$, and let $(a_i, \alpha_i) \in \mathbb{R}^n \times \mathbb{R}$ be the coefficients and additive terms of linear inequalities for $1 \leq i \leq m$. Then the set

$$P \stackrel{\text{def}}{=} \left\{ x \in \mathbb{R}^n \mid \bigwedge_{i=1}^{m} (a_i \cdot x^T \geq \alpha_i) \right\}$$

is a polyhedron in $\mathbb{R}^n$. Or, using the matrix representation of linear inequality systems (see remark 2.19), with $A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$, $\alpha = (\alpha_1, \dots, \alpha_m)$:

$$P = \left\{ x \in \mathbb{R}^n \mid A \cdot x^T \geq \alpha^T \right\}.$$

Definition 2.22 Let $P = \left\{ x \in \mathbb{R}^n \mid A \cdot x^T \geq \alpha \right\}$ be a polyhedron. A set $F \subseteq P$ is called a **face** of P, if

$$F = \left\{ x \in P \mid \hat{A} \cdot x^T = \hat{\alpha} \right\},$$

where $\hat{A}$ is a matrix whose rows are a subset of the rows of A, and $\hat{\alpha}$ is a vector which contains the elements of α that correspond with the rows in $\hat{A}$. When $\hat{A}$ has only one row, F has dimensionality $n-1$ and is called a **facet** of P.

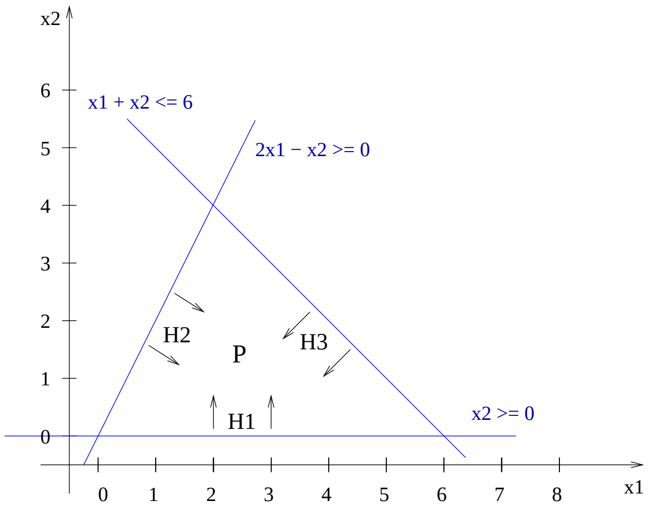


Figure 2.1: A simple polytope defined by three half-spaces

Figure 2.1 illustrates the above concepts in $\mathbb{R}^2$. We give three linear inequalities:

$$\begin{aligned} x_2 &\geq 0 \\ 2 \cdot x_1 - x_2 &\geq 0 \\ -x_1 - x_2 &\geq -6 \end{aligned}$$

Another way to write this inequality system is using the matrix representation of remark 2.19:

$$\begin{pmatrix} 0 & 1 \\ 2 & -1 \\ -1 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ -6 \end{pmatrix}.$$

In figure 2.1 you can see the three inequalities, represented by the lines that satisfy the corresponding equation. The half-spaces that are defined by the inequalities, H_1, H_2, and H_3, are indicated by the arrow pairs that are orthogonal to these lines.

Finally, the polyhedron P, the intersection of H_1, H_2, and H_3, is the triangle between the intersections of the three lines. P is clearly bounded, so it is a polytope.

Remark 2.23 We use polyhedra to describe the index sets of statements. Each surrounding loop of a statement spans one dimension, and each loop bound defines a half-space.

For example, in program 1, the index set for statement `S1` can be expressed with a polyhedron defined by the following inequality system:

$$\begin{aligned} 1 \leq i \leq 4 \\ 1 \leq j \leq 7. \end{aligned}$$

Polyhedra have been defined over the real numbers (definition 2.21). When talking about index sets, the implicit assumption is that we are only interested in the *integral points* inside these polyhedra. We denote the integral points of a polyhedron $P \subseteq \mathbb{R}^d$ with $\text{int}(P) \stackrel{\text{def}}{=} P \cap \mathbb{Z}^d$.

2.2.3 Homogeneous Coordinates

As it makes the work inside the polytope model easier, we give an alternate representation of inequality systems.

Definition 2.24 Let $x \in \mathbb{R}^n$, then the **homogeneous coordinates** $\hbar(x) \in \mathbb{R}^{n+1}$ of x are given by

$$\hbar : (x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_n, 1).$$

Conversely, for $y \in \mathbb{R}^{n+1}$, where $y = (y_1, \ldots, y_{n+1})$, $y_{n+1} \neq 0$ is in homogeneous coordinates, we get a vector $\hbar^{-1}(y) \in \mathbb{R}^n$, defined by

$$\hbar^{-1} : (y_1, \ldots, y_{n+1}) \mapsto \left(\frac{y_1}{y_{n+1}}, \ldots, \frac{y_n}{y_{n+1}} \right).$$

Homogeneous coordinates make it possible to represent an inequality system

$$A \cdot x^T \geq \alpha^T,$$

given by the coefficient matrix $A \in \mathbb{R}^{n_1 \times n_2}$ and the additive terms $\alpha \in \mathbb{R}^{n_1}$ for all $x \in \mathbb{R}^{n_2}$ by the matrix

$$A' \stackrel{\text{def}}{=} \begin{pmatrix} A & -\alpha^T \\ 0 \cdots 0 & 1 \end{pmatrix} \in \mathbb{R}^{(n_1+1)\times(n_2+1)}. \tag{2.2}$$

It is easy to see that the following condition holds:

$$\forall\, x \in \mathbb{R}^{n_2} : A' \cdot \hbar(x)^T \geq 0 \iff A \cdot x^T \geq \alpha^T.$$

We call this the **homogeneous representation** of an inequality system. Analogously, A and α define an affine linear function $f : \mathbb{R}^{n_2} \to \mathbb{R}^{n_1}$, defined by $f : x \mapsto A \cdot x - \alpha$. A' is then the homogeneous representation of that function, and we get

$$\forall\, x \in \mathbb{R}^{n_2} : f(x) = \hbar^{-1}\left(A' \cdot \hbar(x)^T\right)^T.$$

Remark 2.25 The main advantage of homogeneous coordinates is the representation of affine linear functions as linear functions in the higher dimensional homogeneous coordinate system. For example, the translation $t : \mathbb{R}^2 \to \mathbb{R}^2$, defined by $t : x \mapsto x + (1, 0)$ can be represented as a matrix $T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. We get $t(x) = \hbar^{-1}\left(T \cdot \hbar(x)^T\right)^T$. Note that it is impossible to represent t with a matrix $T' \in \mathbb{R}^{2\times 2}$ such that $t(x) = T' \cdot x$ for all $x \in \mathbb{R}^n$, because t is not linear in $\mathbb{R}^2$.

2.2.4 Sorting Inequalities

Sorting is an operation that we often perform on linear inequality systems. The input is a linear inequality system and a sorting order for its variables. Output is a new inequality system with the same solution space but with the bounds of each variable only depending on variables that precede it in the sorting order.

Here is a simple example.

$$\begin{aligned} 0 \leq x_2 \leq 3 \\ 0 \leq x_1 + x_2 \leq 3. \end{aligned} \tag{2.3}$$

The input is inequality system 2.3 and the sorting order (x_1, x_2). Inequality system 2.4 is the resulting sorted system. The first line gives the bounds of x_1, which do not depend on x_2, and the rest gives the bounds of x_2, which still depend on x_1.

$$\begin{aligned} -3 \leq x_1 \leq 3 \\ -x_1 \leq x_2 \leq 3 - x_1 \\ 0 \leq x_2 \leq 3 \end{aligned} \tag{2.4}$$

Remark 2.26 In a sorted inequality system with variables $x_1, \dots, x_n$ and sorting order $(x_{k_1}, \dots, x_{k_n})$, the bounds for the variable x_{k_1} are the *projection* of the solution space of A to the dimension x_{k_1}. Another intuition about sorting is that we get a set of lower and upper bounds for each variable x_{k_i}, which only depend on values of variables x_{k_j} for $j < i$. This intuition is especially useful when we use the sorted inequality system to create loop nests, because the bounds can be directly for the nested loops.

Remark 2.27 If an inequality system describes a polytope, sorting always gives bounds for every variable. If it describes a polyhedron, some variables might not be bounded.

The algorithm we use for sorting is called **Fourier-Motzkin elimination** [Ban93, Xue00, Grö03].

2.3 Parameters

In this section we look at techniques for treating parameters in inequality systems. After an overview we provide some basic definitions from logic, before we extend the definitions for inequalities and polyhedra.

2.3.1 Overview

There are different ways in which parameters can be used in linear inequalities. In the simplest case, the case of additive parameters, we allow no parameters in the coefficients of an inequality, but only in the constant additive term. Let p_1 be a structure parameter, then the following inequality is an example:

$$7 \cdot x_1 - 3.5 \cdot x_2 \geq 3 + p_1$$

Although this is already very useful (and can be implemented relatively easily with the existing algorithms for inequality systems and polyhedra), we want parameters in the coefficients of the inequality. For example, we want to work with inequalities of the following form:

$$p_1 \cdot x_1 - 17 \cdot p_2 \cdot x_2 \geq p_1 + p_2,$$

where p_1 and p_2 are parameters.

The big problem is that these inequalities are no longer linear. This means that all algorithms we have for linear inequalities cannot be applied. Luckily, it turns out that the algorithms we need (e.g. Fourier-Motzkin elimination) can be extended to inequalities with parameters in their coefficients [Grö03].

This has its cost, however. We have to accept case distinctions in our results. For example with Fourier-Motzkin we no longer get one sorted inequality system as a result, but a set of possible solutions, separated by case distinctions in the parameters.

For example, look at the following inequality system:

$$\begin{aligned} 0 \le p_1 \cdot x_1 \le x_2 \\ 1 \le x_2 \le 11 \end{aligned}$$

Assume we want to sort it in the order (x_1, x_2).

The result is a case distinction on the sign of p:

$p < 0$	$p = 0$	$p > 0$
$0 \le x_1 \le \frac{11}{p}$ $p \cdot x_1 \le x_2$ $1 \le x_2 \le 11$	$1 \le x_2 \le 11$	$\frac{11}{p} \le x_1 \le 0$ $p \cdot x_1 \le x_2$ $1 \le x_2 \le 11$

2.3.2 Logic

For the formal introduction of case distinctions, we need some basic terminology from logic. What we are especially interested in are *quantifier formulae* from first-order logic.

These formulae are needed to make case distinctions based on the values of parameters. For example, we want to be able to express the following condition in a mathematical manner:

> If L is less than M added to N, and Z is not negative, ...

Formulae from first-order logic provide a good model for that purpose. The condition from the sentence above can be expressed by the very simple formula

$$(L < N + M) \wedge (Z \ge 0).$$

We cover the definitions needed to introduce first-order logic here, but don't give a lot of detail or examples. A more detailed introduction is provided by Größlinger [Grö03] or any introductory book on logic [Sch00].

Before we can introduce formulae we must specify which symbols may be used in them. In other words, the formulae of first-order logic are a language, and we want to specify the alphabet for that language. This is exactly what a *signature* does.

Definition 2.28 A **signature** Σ is a triple $(\mathcal{F}, \mathcal{R}, ar)$ where $\mathcal{F}$ is the set of function symbols, $\mathcal{R}$ is the set of relation symbols, and $ar : \mathcal{F} \cup \mathcal{R} \to \mathbb{N}$ gives the arity (number of arguments) of each function symbol and relation symbol.

Of course we want to use variables in our formulae, so we assume that $\mathcal{V}$ is an infinite set of variables. Now we can define terms, the building blocks of our formulae:

Definition 2.29 The set of all **terms** in $\mathcal{V}$ and $\Sigma = (\mathcal{F}, \mathcal{R}, ar)$, $\mathit{Tm}(\mathcal{V}, \Sigma)$, is defined inductively by:

- $\mathcal{V} \subseteq \mathit{Tm}(\mathcal{V}, \Sigma)$,
- $f \in \mathit{Tm}(\mathcal{V}, \Sigma)$ where $f \in \mathcal{F}$ and $ar(f) = 0$,
- $f(t_1, \ldots, t_n) \in \mathit{Tm}(\mathcal{V}, \Sigma)$, where $f \in \mathcal{F}$, $ar(f) = n > 0$ and $t_1, \ldots, t_n \in \mathit{Tm}(\mathcal{V}, \Sigma)$.

Note that we use the usual infix notation for function symbols like $+$, $-$, $*$, $/$. So instead of $+(17, x_1)$ we write $17 + x_1$. We also use parentheses to disambiguate expressions, and the usual precedence rules (e.g. $*$ binds stronger than $+$) to save parentheses.

We can now introduce *atomic formulae*, which combine terms using relation symbols.

Definition 2.30 The set of all **atomic formulae** over $\mathcal{V}$ and $\Sigma = (\mathcal{F}, \mathcal{R}, ar)$, $\mathit{At}(\mathcal{V}, \Sigma)$, is defined by:

- truth constants r where $r \in \mathcal{R}$ and $ar(r) = 0$,
- equations $t_1 = t_2$ where $t_1, t_2 \in \mathit{Tm}(\mathcal{V}, \Sigma)$,
- predicates $r(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in \mathit{Tm}(\mathcal{V}, \Sigma)$, $r \in \mathcal{R}$, and $ar(r) = n > 0$.

For some relation symbols like $<$, we use infix notation (e.g. $1 < 2$ instead of $< (1, 2)$).

Now we are ready to define the quantifier-free formulae.

Definition 2.31 The set of **quantifier-free first-order formulae**, $\mathit{Qf}(\mathcal{V}, \Sigma)$, is inductively defined by:

- $\mathit{At}(\mathcal{V}, \Sigma) \subseteq \mathit{Qf}(\mathcal{V}, \Sigma)$,
- $(\neg\phi) \in \mathit{Qf}(\mathcal{V}, \Sigma)$ where $\phi \in \mathit{Qf}(\mathcal{V}, \Sigma)$,
- $(\phi\,\rho\,\mu) \in \mathit{Qf}(\mathcal{V}, \Sigma)$ where $\phi, \mu \in \mathit{Qf}(\mathcal{V}, \Sigma)$ and $\rho \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

To save parentheses, we use the logical operator symbols in the following order of precedence (from highest to lowest): $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

Using this convention, the following two formulae are equivalent.

- $(((\neg x_1) \wedge x_2) \vee x_3)$
- $\neg x_1 \wedge x_2 \vee x_3$

So far we have defined the syntax of a subset of first order logic. The semantics we use is the usual one, as for example defined by Schöning [Sch00].

The next definition is an application of the formulae just defined. We assume that we have structure parameters, which are used in the loop bounds of our source programs. We want to represent case distinctions in these parameters in a way that they are both space efficient and easy to evaluate.

Therefore, we use *decision trees*. A decision tree is a binary tree. The leafs are elements from some set, in our case polyhedra. The nodes are labeled with quantifier-free formulae in the parameters. The intuition behind a decision tree is that, depending on the actual values of the parameters, one of the leafs is "chosen", meaning that the formulae of all nodes that are on the path from the root to this leaf evaluate to true if the leaf node is in the left subtree and to false if it is in the right subtree.

Viewed from a different, more operational perspective, the formulae on a path from the root to a leaf can be seen as `if`-statements guarding this leaf.

Here is the formal definition.

Definition 2.32 The set of all **decision trees** over a set X and the parameters $p_1, \ldots, p_r$, written $DT_X(p_1, \ldots, p_r)$, is defined inductively as follows:

$$x \in DT_X(p_1, \ldots, p_r) \quad \text{for all } x \in X \qquad \text{(DT.leaf)}$$

$$(p, t_{true}, t_{false}) \in DT_X(p_1, \ldots, p_r) \quad \text{for all } p \in \mathcal{Qf}(\{p_1, \ldots, p_r\}, \Sigma) \ \wedge \ t_{true}, t_{false} \in DT_X(p_1, \ldots, p_r). \qquad \text{(DT.cond)}$$

$\mathcal{Qf}(\{p_1, \ldots, p_n\}, \Sigma)$ is the set of all quantifier-free formulae in $p_1, \ldots, p_n$ (see definition 2.31) and Σ is the signature of the common relations and functions in first order logic ($<$, $>$, $+$, $-$, $\ldots$).

Größlinger provides examples and an implementation of decision trees in Haskell [Grö03].

2.3.3 Parameters in Inequalities

As we already mentioned, we will encounter inequalities which contain structure parameters as coefficients. In this section we provide the necessary mathematical foundation. Größlinger gives details and a rationale for the choice of polynomials [Grö03].

Definition 2.33 Let R be a field and let $x_1, \ldots, x_n$ be unused symbols. Then we call

$$M_R \stackrel{\text{def}}{=} \left\{ c \cdot \prod_{i=1}^{n} x_i^{e_i} \mid c \in R \wedge e_i \in \mathbb{N} \text{ for all } i \in \{1..n\} \right\}$$

the set of **monomials**. The symbols x_i are also called *indeterminates*. Further, the **polynomial ring** $R[x_1, ..., x_n]$ over R is defined as:

$$R[x_1, \ldots, x_n] \stackrel{\text{def}}{=} \left\{ \sum_{i=1}^{r} m_i \mid r \in \mathbb{N} \wedge m_i \in M_R \text{ for each } i \in \{1..r\} \right\}.$$

Definition 2.34 Let R be a field. Then the **quotient field of a polynomial ring**, $R(x_1, \dots, x_n)$ is defined as:

$$R(x_1, \dots, x_n) \stackrel{\text{def}}{=} \left\{ \frac{f}{g} \mid f, g \in R[x_1, \dots, x_n] \wedge g \neq 0 \right\}.$$

An important property of this quotient field is that it is closed under division by non-zero elements (hence the name field).

The following definitions are rather technical. What they do is simple: they generalize the terms of chapter 2.2 to use the quotient field $\mathbb{Q}(p_1, \dots, p_r)$ instead of the real numbers, allowing us to speak about polyhedra with parameters, which we need for our tiling methods.

Definition 2.35 Let $p_1, \dots, p_r$ be parameters and let ℓ be a function of type $\mathbb{Q}(p_1, \dots, p_r)^n \to \mathbb{Q}(p_1, \dots, p_r)$. ℓ is called a **parametrized linear form** if it is linear.

Definition 2.36 In generalization of definition 2.20, let $p_1, \dots, p_r$ be parameters, ℓ a non-zero parametrized linear form, and $\alpha \in \mathbb{Q}(p_1, \dots, p_r)$. Then

$$A(p_1, \dots, p_r, \ell, \alpha) \stackrel{\text{def}}{=} \ell(p_1, \dots, p_r)^{-1}(\{\alpha\})$$

is called a **parametrized hyperplane**. Analogously,

$$H(p_1, \dots, p_r, \ell, \alpha) \stackrel{\text{def}}{=} \ell(p_1, \dots, p_r)^{-1}(\,[\alpha, \infty[\,)$$

is called a **parametrized half-space** of $\mathbb{Q}(p_1, \dots, p_r)^n$.

Definition 2.37 Let $p_1, \dots, p_r$ be parameters, ℓ be a non-zero parametrized linear form, $\alpha = (\alpha_1, \dots, \alpha_m) \in \mathbb{Q}(p_1, \dots, p_r)^m$, and let $a = (a_1, \dots, a_m) \in \mathbb{Q}(p_1, \dots, p_r)^n \setminus \{0\}$, such that $\ell(x) = a \cdot x^T$. Then

$$P \stackrel{\text{def}}{=} \bigcap_{i=1}^{m} H(p_1, \dots, p_r, \ell, \alpha_i)$$

is called a **parametrized polyhedron**. Analogously, we call

$$\begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot x^T + \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{pmatrix} \geq 0$$

a **parametrized inequality system**.

One possible intuition about parametrized polyhedra is that they represent a whole set of polyhedra, indexed by the parameters. So if you choose values for the parameters, you get one specific polyhedron by plugging these values into the polynomial coefficients.

This can be easily seen for parametrized inequalities. Given the parameters p_1 and p_2, the following is a parametrized inequality:

$$(p_1 + \frac{1}{2} \cdot p_2) \cdot x_1 - \frac{p_2}{p_1} \cdot x_2 + p_2 \geq 0.$$

For $p_1 = 3$ and $p_2 = 1$, for example, we get the inequality

$$3\frac{1}{2} \cdot x_1 - \frac{1}{3} \cdot x_2 + 1 \geq 0.$$

Another important fact can be seen by looking at this example: not every mapping of parameters to values results in a valid inequality. If you choose $p_1 = 0$ you can see that the result is undefined, because there is a division by p_1 in the coefficient of x_2.

Definition 2.38 A **generalized polyhedron** is a decision tree in which the leafs are parametrized polyhedra.

We will encounter generalized polyhedra later on, where they appear as output of algorithms that work on parametrized polyhedra (e.g. Fourier-Motzkin elimination).

2.4 The Polytope Model

Now that we have covered the basic mathematical machinery we get to the interesting part: the polyhedron model. We start with an older, more restrictive model and introduce the extensions later. This older model is called the **polytope model**, and was first presented by Lengauer [Len93].

We only give a short introduction to the main concepts of the polytope model, which already has been described in many papers. For a more in-depth treatment and examples see [Len93, Gri04].

2.4.1 Overview

The basic structure of parallelization with the polytope model is best seen in figure 2.2.

```
for i := 0 to 3
 for j := 0 to i+2
  A[i,j] := A[i-1, j] + A[i, j-1]
 end
end
```

```
for t := 0 to 8
 forall p := max(0,t-n) to min(t,t/2+1)
  A[t-p,p] := A[t-p-1,p] + A[t-p,p-1]
 end
end
```

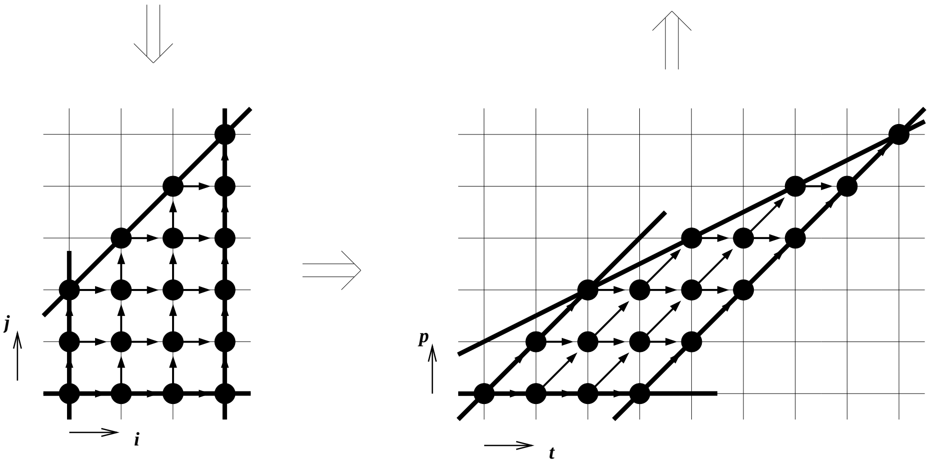


Figure 2.2: The polytope model

1. We start with the source program in the upper left. Based on this program we build a mathematical model that captures all information about the program that is necessary for parallelization. As the name suggests, the most important part of the model is a polytope, which is shown in the lower left of the figure. This polytope describes the index space of the source program. The dots represent the operations, and the arrows between them represent the dependency information.

2. After the model is built, we are ready for the actual parallelization. This phase works solely within the model, the source program text can be forgotten at that point.

3. After parallelization, we have a transformed polytope, shown in the lower right of the figure. The most important property of that polytope is that the dimensions are either strictly sequential (i.e. in time), or parallel (i.e. in space). The exact meaning of this will become clear later, for now it is enough to know that the transformed polytope exposes the parallelism that the transformation has found.

4. In the last step, we create the target program from the transformed polytope as shown in the upper right of the figure.

2.4.2 Restrictions

The polytope model [Len93] is applicable to only a very restricted set of programs.

1. Only one control structure: for loops with affine linear bounds (in surrounding variables and parameters) and with unit stride.
2. Only data type: array (no pointers allowed).
3. All array accesses are linear in the surrounding loops.
4. Loops must be perfectly nested.
5. All dependencies must be uniform.

We stick to this restricted model, because the presentation of the core concepts is more perspicuous this way. At the end of this section we mention the extensions of the model that have been developed so far.

2.4.3 The Space-Time Mapping

The core task of the parallelization phase in the polytope model is the computation of the **space-time mapping**.

The space-time mapping is a function that transforms the source index space to the target index space; in figure 2.2 this is the step from the lower left to the lower right. The space-time mapping honors the dependencies of the program, and lays out the operations in the target space in a way that space and time dimensions are separated. This means that if two operations in the target index space have the same time coordinates, they can be computed in parallel.

Remark 2.39 In the following, we use the term **logical time** or **virtual time** for a point of the projection of the transformed index space to the sequential dimensions. Similarly, we use the term **logical processor** or **virtual processor** for a point in the projection to the parallel dimensions of the transformed index space. This way, every point in the transformed index space is uniquely defined by a virtual time and a virtual processor. For convenience, we will use the term **parallel index space** as a synonym for the projection of the target index space to the parallel dimensions.

The space-time mapping can be decomposed into two independent components: The **schedule** and the **allocation**, also called **computation placement**. We assume that our source index space has dimensionality $d_S \in \mathbb{N}$, and that the target index space has dimensionality $d_T \in \mathbb{N}$.

Schedule The schedule θ is a function that assigns a logical execution time to each operation in the source index space. For technical reasons, there can be more than one time dimension (which is then translated to multiple nested sequential `for` loops in the target code). Let d_T^θ denote the number of time dimensions ($1 \leq d_T^\theta \leq d_T$).

Then θ is an affine linear function from $\mathbb{R}^{d_S}$ to $\mathbb{R}^{d_T^\theta}$, which can be represented by a matrix $\Theta \in \mathbb{R}^{(d_T^\theta+1)\times(d_S+1)}$ in homogeneous representation such that

$$\forall\, i \in \mathbb{R}^{d_S} : \theta\,(\,i\,) = \hbar^{-1}\left(\Theta \cdot \hbar(i)^T\right)^T.$$

It is interesting to note that the schedule implicitly defines the relation $<_{par}$ and the equivalence relation $\backsim_{par}$ from section 2.1.5. Let Ω be the set of all operations of a program, then it is easy to check that with

$$\begin{aligned}\forall\, u, v \in \Omega : (u <_{par} v &\iff \theta(\mathrm{iv}(u)) < \theta(\mathrm{iv}(v)) \wedge \\ u \backsim_{par} v &\iff \theta(\mathrm{iv}(u)) = \theta(\mathrm{iv}(v))),\end{aligned}$$

$\backsim_{par}$ is indeed an equivalence relation. Equation 2.1 then translates to

$$\forall\, u, v \in \Omega : u \;\delta\; v \;\Rightarrow\; \theta(\mathrm{iv}(u)) < \theta(\mathrm{iv}(v)). \tag{2.5}$$

Equation 2.5 is called the *causality condition* by Feautrier [Fea92].

We do not require that our schedules are integral. If a schedule θ satisfies the causality condition, so does the rounded schedule $\lfloor\theta\rfloor$ (for $a, b \in \mathbb{R}$, we have $a < b \Rightarrow \lfloor a\rfloor < \lfloor b\rfloor$), so we can always move to an integral schedule if necessary.

Definition 2.40 An important quality criterion of a schedule is the **latency**. For a one-dimensional schedule $\theta \in \mathbb{R}^{d_S} \to \mathbb{R}$, the latency $L(\theta)$ is defined by

$$L(\theta) \stackrel{\text{def}}{=} \max_{u\in\Omega} \theta(\mathrm{iv}(u)) - \min_{u\in\Omega} \theta(\mathrm{iv}(u)).$$

One schedule plays a prominent role, because it minimizes the latency.

Definition 2.41 The schedule that schedules every operation at the earliest possible time (as determined by the dependencies) is called **free schedule**. The free schedule is unique, and it has minimal latency.

Allocation The allocation π is a function that assigns a logical processor to each operation in the source index space. Let d_T^π denote the number of parallel dimensions ($1 \leq d_T^\pi \leq d_T$).

Then π is an affine linear function from $\mathbb{R}^{d_S}$ to $\mathbb{R}^{d_T^\pi}$, which can be represented by a matrix $\Pi \in \mathbb{R}^{(d_T^\pi+1)\times(d_S+1)}$ such that

$$\forall\, i \in \mathbb{R}^{d_S} : \pi\,(\,i\,) = \hbar^{-1}\left(\Pi \cdot \hbar(i)^T\right)^T.$$

Given a schedule matrix Θ and a placement matrix Π, they can be combined to a matrix

$$T = \begin{pmatrix} \Theta' \\ \Pi \end{pmatrix} \in \mathbb{R}^{(d_T+1)\times(d_S+1)},$$

where Θ' is Θ without the last row. This matrix T, or rather the affine linear function it defines is called the **space-time mapping**.

Definition 2.42 For $n \in \mathbb{N}_1$, a matrix $M \in \mathbb{R}^{n\times n}$ is called **unimodular** if all elements are integral (i.e. $M \in \mathbb{Z}^{n\times n}$) and $|\det(M)| = 1$. The inverse of a unimodular matrix exists and is unimodular, too.

Although schedule and allocation can be computed independently, some care must be taken to make them compatible. Obviously, they must have correct dimensionality. The number of time dimensions d_T^θ and parallel (or space-) dimensions d_T^π must sum up to the dimensionality d_T of the target index space.

Secondly, because we want the space-time matrix T to be unimodular, schedule and allocation must be at least linearly independent, because otherwise T would not have full rank. Note that this also means that T is a square matrix, thus $d_S = d_T$.

2.4.4 Transforming the Index Space

We assume that schedule and allocation have been computed, resulting in a space-time mapping matrix T in homogeneous representation. Furthermore, we have an inequality system D that describes the source index space, also given in homogeneous representation.

Now we are ready to transform the source index space by applying the space-time mapping. Since we required that T is unimodular, we know that T^{-1}, the inverse of T, exists.

Let i be an iteration vector of the source index space, and let (t,p) be an iteration vector of the target index space such that $(t,p)^T = T \cdot i^T$. Then we know:

$$D{\cdot}i^T \geq 0 \iff D{\cdot}(T^{-1}{\cdot}T){\cdot}i^T \geq 0 \iff (D{\cdot}T^{-1}){\cdot}(T{\cdot}i^T) \geq 0 \iff D'{\cdot}(t,p)^T \geq 0,$$

where $D' \stackrel{\text{def}}{=} D \cdot T^{-1}$ is the inequality system that describes the target index space.

2.4.5 Generating the Target Code

The last step is to convert the target index space given by D' to code. First thing to note is that each operation in the target index space has a coordinate in time (given by the schedule), and one in space (given by the allocation). We know that two operations with the same time coordinate do not depend on each other, allowing parallel execution.

We split code generation into three steps. First we show how to compute the *loop bounds* of the target program, then we show how the *array indices* of the target statements can be computed from the array indices of the source statements. Finally we discuss the generation of *communication code* in a separate section, because it is important for the rest of this thesis.

Loop Bounds

Definition 2.43 A parallel loop nest is called **synchronous** if the outer loops are sequential and the inner loops are parallel. Conversely, it is called **asynchronous** if the outer loops are parallel and the inner loops are sequential.

In order to compute the loop bounds of our parallel program we determine an order of loops and use the Fourier-Motzkin algorithm to sort D' in that order (see section 2.2.4). For a variable i_k, we look at all rows of the sorted inequality system where the column belonging to i_k is the rightmost non-zero entry. The inequalities described by these rows are the bounds for i_k. They have the form

$$\sum_{\nu=1}^{k} a_\nu \cdot i_\nu \geq \alpha.$$

If a_k is positive, this is a lower bound:

$$i_\nu \geq \frac{\alpha - \sum_{\nu=1}^{k-1} a_\nu \cdot i_\nu}{a_k},$$

otherwise it is an upper bound:

$$i_\nu \leq \frac{\alpha - \sum_{\nu=1}^{k-1} a_\nu \cdot i_\nu}{a_k}.$$

Note that $a_k \neq 0$, because of the way we chose the rows above.

Let D' be sorted in the order $(k_1, \ldots, k_d)$, where d is the dimensionality of the target space. Further, d^θ is the number of sequential dimensions, and d^π is the number of parallel dimensions, $k_1, \ldots, k_{d^\theta}$ are the sequential dimensions, and $k_{(d^\theta+1)}, \ldots, k_d$ are the parallel dimensions. With the technique described above, we get a set of lower and upper bounds for each i_ν, annotated by L_ν and U_ν, $1 \leq \nu \leq d$.

With this information we can create the loops of the target program (cf. program 3).

The sorting makes sure that for each $\nu \in \{ 1, \ldots, d \}$, the bounds for i_{k_ν} only depend on variables i_{k_j} where $j < \nu$, which means that they are from surrounding loops and hence well-defined.

Regarding the order of the loops we have the freedom of choice. We can either produce a synchronous or an asynchronous target program.

```
for $i_{k_1}$ := $\max(L_{k_1})$ to $\min(U_{k_1})$
  ⋮
  for $i_{k_{d^\theta}}$ := $\max(L_{k_{d^\theta}})$ to $\min(U_{k_{d^\theta}})$
    parfor $i_{k_{(d^\theta+1)}}$ := $\max(L_{k_{(d^\theta+1)}})$ to $\min(U_{k_{(d^\theta+1)}})$
      ⋮
        parfor $i_{k_d}$ := $\max(L_{k_d})$ to $\min(U_{k_d})$
          ⋮
        end
      ⋮
    end
  ⋮
end
```

Program 3: Loop bounds generated from the sorted inequality system

Array Indices

Now we have the loop bounds and the order of the loops for the target code. Additionally, we can compute the original iteration vector i from the target iteration vector (t, p) with the space-time mapping matrix T:

$$(i, 1) = \left(T^{-1} \cdot (t, p, 1)^T\right)^T.$$

This means that we have all parts ready to produce the target loops, insert the statements, and compute the original source index variables from the target loop index variables.

For an illustration, look at figure 2.2. The target code is synchronous, because the outer loop is sequential and the inner loop is parallel. The inverse space-time mapping is $T^{-1} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

If we produced code for a shared memory architecture, we would be finished by now. But we assume that we create code for a distributed memory architecture with explicit communications.

2.5 Generation of Communications

We assume that our target platform supports MPI [MPI].

We consider two basic approaches: **ownership driven** versus **dependency driven** communication. When working with ownership driven communication, every array cell has a fixed owner (a virtual processor), and whenever the value of an array cell changes, the new value has to be communicated to the owner.

Similarly, whenever the value of a cell is needed by a virtual processor that is not the owner, it has to be communicated.

With dependency driven communication, the values of the array cells are directly sent from the producer to the consumer.

We look at the two approaches in more detail, because we encounter the problem of communication generation in our adaptive tiling methods.

We split the generation of communication statements into two steps. First, we create *communication polyhedra*. The integral points inside these polyhedra describe all communications in our program. In a second step, we create code from these polyhedra.

2.5.1 Duplication of variables

Duplication of variables is a basic operation we frequently apply to inequality systems. Since it is used in the following sections we give a description here. We choose some indeterminates of our inequality system, and replace each with two new variables. At the same time, we copy the inequalities that involve the replaced variable, resulting in a duplication of those inequalities.

Definition 2.44 Let $A \in \mathbb{R}^{(k+1)\times(l+1)}$ be the coefficient matrix of an inequality system in homogeneous representation with l indeterminates and k inequalities. Let further be $(x_1, \ldots, x_l)$ the indeterminate symbols, let $r \in \mathbb{N}_1$, $r \leq l$. For the sake of simplicity we assume that the variables we want to replace are $(x_r, \ldots, x_l)$. This can always be accomplished by reordering A first. Let $(y_r, \ldots, y_l)$ and $(z_r, \ldots, z_l)$ be the new indeterminate symbols. Then we can **duplicate the variables** $(x_r, \ldots, x_l)$ in A by performing the following steps:

1. For each $r \leq i \leq l$ replace the dimension represented by x_i with two new dimensions represented by y_i and z_i.
2. For each inequality (i.e. row of A) $(\alpha_1, \ldots, \alpha_l, \beta) \in \mathbb{R}^{l+1}$ create two new inequalities:
   $(\alpha_1, \ldots, \alpha_{r-1}, \alpha_r, \ldots, \alpha_l, 0, \ldots, 0, \beta)$, $(\alpha_1, \ldots, \alpha_{r-1}, 0, \ldots, 0, \alpha_r, \ldots, \alpha_l, \beta) \in \mathbb{R}^{(2\cdot l-r+2)}$.
3. Remove all duplicate lines (if $\alpha_r = \ldots = \alpha_l = 0$, we get two identical inequalities).

Let $A' \in \mathbb{R}^{(k'+1)\times(2\cdot l-r+2)}$ be the coefficient matrix obtained by the above construction, $k \leq k' \leq 2 \cdot k$. Then our new inequality system is

$$A' \cdot (x_1, \ldots, x_{r-1}, y_r, \ldots, y_l, z_r, \ldots, z_l, 1)^T \geq 0.$$

2.5.2 Ownership Driven Communication

For each array `A` from our source program, we require a *data placement* that tells us for every element of `A` on which logical processor it is stored.

Definition 2.45 Let `A` be an l-dimensional array, and let d be the dimensionality of the allocation. Then a **data placement** is an affine linear function $\pi_A : \mathbb{R}^l \to \mathbb{R}^d$. For each array cell $\mathcal{C} = \mathtt{A}[i_1,\ldots,i_l]$, the logical processor $\pi_A(i_1,\ldots,i_l)$ is called the **owner** of $\mathcal{C}$.

How do we use the information about the allocation, the data placement and the array accesses to produce the communication statements?

The answer is: whenever the processor that computes an operation is different from the processor that holds an array cell that is accessed by the operation, we need a communication for that array cell.

Formally we might express this in the following way. Let u be an operation with iteration vector $\mathrm{iv}(u) = (i_1,\ldots,i_k)$, let r be an array access of $\mathrm{stmt}(u)$ and the l-dimensional array A, let $a_{A,r} : \mathbb{R}^k \to \mathbb{R}^l$ be the corresponding access function, and let $(i'_1,\ldots,i'_l) \in \mathbb{R}^l$, and finally let $\pi : \mathbb{R}^k \to \mathbb{R}^d$ be the allocation, and $\pi_A : \mathbb{R}^l \to \mathbb{R}^d$ be the data placement of `A`. We need a communication whenever

$$\begin{aligned} (i'_1,\ldots,i'_l) &= a_{A,r}(i_1,\ldots,i_k) \ \wedge \\ \pi(i_1,\ldots,i_k) &\neq \pi_A(i'_1,\ldots,i'_l). \end{aligned} \tag{2.6}$$

This can be rewritten in matrix notation. Let $\Pi \in \mathbb{R}^{(d+1)\times(k+1)}$, $\Pi_A \in \mathbb{R}^{(d+1)\times(l+1)}$, and $\mathcal{A}_{A,r} \in \mathbb{R}^{(l+1)\times(k+1)}$ be the respective (homogeneous) matrix representations of the affine linear functions from above. Then equation 2.6 can be written as

$$\begin{aligned} (i'_1,\ldots,i'_l,1)^T &= \mathcal{A}_{A,r} \cdot (i_1,\ldots,i_k,1)^T \ \wedge \\ \Pi \cdot (i_1,\ldots,i_k,1)^T &\neq \Pi_A \cdot (i'_1,\ldots,i'_l,1)^T. \end{aligned} \tag{2.7}$$

Now we create the communication polyhedra. We express it in target space coordinates. We start by duplicating all variables (t,p) with (t^s,p^s) and (t^r,p^r) (for sender and receiver process) in the inequality system describing the target index space (see definition 2.44). We call the result the **basic communication matrix**.

Let T be the space-time matrix. Then the transformed access matrix $\mathcal{A}'_{A,r} \in \mathbb{R}^{(l+1)\times(k+1)}$ is defined by

$$\mathcal{A}'_{A,r} \stackrel{\text{def}}{=} \mathcal{A}_{A,r} \cdot T^{-1}.$$

For each array A, and each array access r, we create a communication polyhedron based on the basic communication matrix. For read accesses, the polyhedron has the additional constraints

$$\begin{aligned} &(p^s_1,\ldots,p^s_d,1) = (\Pi_A \cdot \mathcal{A}'_{A,r}) \cdot ({}^T t^r_1,\ldots,t^r_{k-d},p^r_1,\ldots,p^r_d,1), \\ &(t^s_1,\ldots,t^s_{k-d}) = (t^r_1,\ldots,t^r_{k-d}). \end{aligned}$$

For write accesses, we get the same constraint, but with switched sender and receiver: we simultaneously substitute x^s_i for x^r_i and vice versa for $1 \leq i \leq d,\ x \in \{\,p,t\,\}$.

Note that we did not model the constraint that sender and receiver processor must differ, because we would need the union of $2 \cdot d$ polyhedra to model the inequality with linear inequalities, where d is the dimensionality of the allocation. We will deal with the problem of these *local communications* later.

If we have k array accesses, our output is the union of k communication polyhedra. The integral points inside these polyhedra, denoted by $(t^s, t^r, p^s, p^r) \in \mathbb{Z}^{k+d}$ represent communications from logical processor p^s to logical processor p^r. The data is sent at logical time t^s and received at logical time t^r.

2.5.3 Dependency Driven Communication

In contrast to ownership driven communication there is no concept of ownership when doing dependency driven communication. Instead, data is always sent from the producer to the consumer.

To create the communications for this approach we look at the dependencies or, more precisely, the h-transformations. For each operation that is in the target index set of an h-transformation we check whether the source is on the same processor.

Let $h = (f_h, D_h)$ be the h-transformation of a flow dependency, let $u \in \Omega$ be an operation with iteration vector $i = \mathrm{iv}(u)$, and let π be the allocation. Then we need to communicate data if

$$i \in D_h \ \wedge \ i' = f_h(i) \ \wedge \ \pi(i) \neq \pi(i')$$

Next we create the communication polyhedra. Let T be the space time matrix, and let $h_1 = (f_{h_1}, D_{h_1}), \ldots, h_r = (f_{h_r}, D_{h_r})$ be the h-transformations of all flow dependencies. Then the transformed h-transformations are defined in the following way: Let $\mathcal{T} : \mathbb{R}^d \to \mathbb{R}^d$ be the affine function defined by the space-time mapping matrix T, and let $\mathcal{T}^{-1}$ be its inverse. Then the transformed domains are defined by the inequality system $D'_{h_\nu} \stackrel{\text{def}}{=} D_{h_\nu} \cdot T^{-1}$, and the transformed functions by $f'_{h_\nu} \stackrel{\text{def}}{=} \mathcal{T} \circ f_{h_\nu} \circ \mathcal{T}^{-1}$ for all $1 \leq \nu \leq r$.

We now create one polyhedron C_ν for each h-transformation h_ν. We start with D'_{h_ν} and duplicate all parallel variables (t, p), once for the sender $((t^s, p^s))$ and once for the receiver $((t^r, p^r))$. Then we add the equation

$$(t^s, p^s) = f'_{h_\nu}(t^r, p^r).$$

Again, we did not model the constraint that excludes local communications. The union of the C_ν (or more precisely: of their integral points) describes all communications.

2.6 The Polyhedron Model

The polytope model has been extended in different ways over the last ten years. We just give a short overview to give the reader a feeling for the applicability of the techniques we use.

- Non-uniform dependencies
- Imperfectly nested loops
- Statement-individual, piecewise affine schedules and placements
- `for` loops with arbitrary bounds [GGL97]
- `for` loops with non-unit strides [Xue94]
- `while` loops [Gri97, GGL99, Col95]
- if-statements [BCF97]

We do not discuss these extensions further, but only state that the name *polyhedron model* stems from the treatment of `while` loops: because the loop bounds are unknown at compile time, the resulting index space is not bounded.

Chapter 3

Tiling

In this chapter we introduce tiling. After giving an overview we have a short look on existing tiling techniques and on our view of tiling. Subsequently we give basic definitions and investigate the problem of automatic parametric space tiling. It turned out that there are unsolved problems in this field. In fact, after solving the problem of automatically finding a static space tiling, we realized that adaptive tiling is not much more difficult to achieve. The same techniques can easily be adapted. For this reason we treat the problem of automatic space tiling in detail.

3.1 Overview

Tiling, in general, is an umbrella term for techniques that coalesce operations in space and/or time.

The term *tiling* is really intuitive if you look at examples in two dimensions. Figure 3.1 shows an index space, the dots represent the integral points. The lines show borders of tiles. The whole index space is covered by non-overlapping tiles.

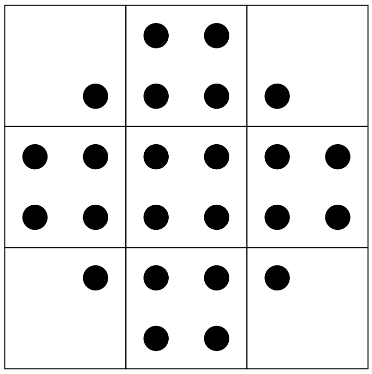


Figure 3.1: Rectangular tiling of a two-dimensional index space

Traditionally tiling has one goal: to improve execution speed. Tiling can help in different ways in achieving this goal. Examples are

- improving locality of reference in sequential programs and
- improving the ratio of computation versus communication in parallel programs.

In addition to the second item we also use tiling to map the parallelism that is found with the help of the polyhedron model onto real parallel computers.

Tiling is always a *partitioning* of the index space. For an index space $\mathcal{I} \subseteq \mathbb{Z}^d$, and a set J, we have a family $(\mathcal{T}_j)_{j \in J}$ with:

- $\mathcal{T}_j \subseteq \mathbb{Z}^d$ for all $j \in J$,
- $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ for all $i, j \in J$ with $i \neq j$, and
- $\mathcal{I} \subseteq \bigcup_{j \in J} \mathcal{T}_j$.

Traditionally the $\mathcal{T}_j$ are modeled with congruent, non-intersecting polyhedra, called *tiles* (or more precisely with their integral points). For many methods tiles are even constrained to be parallelepipeds (see definition 3.1).

3.2 Existing Methods

Tiling is an active field of research with many publications. On a closer look, however, most of the existing techniques differ in their premises and goals from what is needed in our case.

A great fraction of the research deals with tiling to improve the utilization of caches. Griebl discusses the similarities and differences [Gri04].

In the field of automatic parallelization traditional tiling techniques are used to *enable* parallelization [Xue00]. They are applied before a space-time mapping (or similar transformation). So they can be seen as a parallelization technique.

This is quite different from what we are aiming at. We already have a powerful framework for parallelization and are looking for tiling techniques to efficiently map a parallel program to real target machines. In other words: we want to be able to control the granularity of the discovered parallelism in order to adapt it to specific target machines.

Martina Wieninger deals with space tiling in the polytope model [Wie97]. Since the tools of Armin Größlinger [Grö03] for dealing with structure parameters in inequalities were not available back then the methods developed in this work can deal with structure parameters in the loop bounds of the source program only in a restricted way.

3.3 Our View of Tiling

Like Wieninger [Wie97] and as motivated by Griebl [Gri04] we examine techniques that are applied *after* parallelization to improve performance and map the target program to a particular parallel architecture.

As we have seen in the last section this is in contrast to traditional tiling methods, which mostly tile the source index space and apply parallelization techniques to the tiled source index space.

We just summarize the advantages of the first approach as given by Griebl [Gri04]:

- Good handling of multiple, imperfectly nested statements, because we have one unique target coordinate system which contains all statements.
- Better exploitation of parallelism, because the statements can be treated individually before being coalesced into tiles.

We are mostly concerned with tiling space dimensions, but we also discuss tiling of time dimensions where applicable.

3.4 Separation of Space and Time

The problem of finding a good tiling can be significantly simplified by decomposing it into two subproblems: tiling space and tiling time.

First, what does it mean for a tiling to be *in space only* (or *in time only*)? It means that all the facets of the tile shape are orthogonal to all time dimensions (or to all space dimensions, respectively). It follows that at every step in logical time we have the same space tiling. This is why we call this kind of tiling *static tiling*.

At first it is not obvious why space and time can be treated separately. Isn't it possible that there is a better tiling (with respect to communications) that is in space *and* time?

The answer is: yes there *could* be such a tiling. But we suppose that the allocation puts operations onto the same processor in a way that minimizes communication. So it is sound to choose a tiling that *respects* the allocation, which means that it must keep operations that are placed on the same logical processor on the same physical processor. Under this premise however, it turns our that we can always find a tile shape that can be decomposed into two orthogonal parts, one in space and one in time, and causes the same number of communications [Gri04].

But this argument only holds with respect to the number of communications. If we take other factors into account, for example the load balance of the tiled program, we might come to a different result. The load balance is not considered by the allocator, which implies that it might be necessary to either change the allocator to consider the load balance, or to not respect the allocation in certain cases, if we want to improve the load balance. The adaptive tiling approaches developed in this thesis are an example of the latter approach: we allow a (controlled) violation of the allocation in order to improve the load balance.

3.5 Important Terms and Definitions

First we introduce some basic definitions that we need later on.

Definition 3.1 A **parallelepiped** in $\mathbb{R}^d$ is a d-dimensional polytope which can be described by an inequality system of the following form:

$$\begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} b^T \\ c^T \end{pmatrix}, A \in \mathbb{R}^{d \times d},\ x, b, c \in \mathbb{R}^d,$$

where A is a matrix with full rank d (otherwise it would contain redundant inequalities).

Parallelepipeds are generalizations of parallelograms from $\mathbb{R}^2$ to higher dimensionalities. For each facet they have a parallel one (differing only in the translation).

A different characterization of parallelepipeds is by their *spanning vectors*. Think of a parallelogram: it is fully defined by two non-parallel vectors. More generally, if we have d linearly independent vectors $v_1, \ldots, v_d \in \mathbb{R}^d$, they define a parallelepiped (located in the origin). To convert one representation into the other it helps to remember that the rows of the matrix A are the normals of the facets of the parallelepiped (which are $(d-1)$ – dimensional faces). The spanning vectors, on the other hand, lie on one–dimensional faces (i.e. edges) of the parallelepiped.

The following lemma shows how to change from one representation to the other.

Lemma 3.2 *Let $A \in \mathbb{R}^{d \times d}$, $det(A) \neq 0$, and $b = (b_1, \ldots, b_d) \in]0, \infty[^d$ define a parallelepiped*

$$P \stackrel{def}{=} \left\{ x \in \mathbb{R}^d \mid \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ b^T \end{pmatrix} \right\}$$

(located in the origin) and let $V = ({v_1}^T, \ldots, {v_d}^T) \in \mathbb{R}^{d \times d}$ be the matrix of the spanning vectors of P. Then

$$-A \cdot V = \begin{pmatrix} b_1 & & 0 \\ & \ddots & \\ 0 & & b_d \end{pmatrix}.$$

Proof In the following $a_1, \ldots, a_d \in \mathbb{R}^d$ are the rows of A or equivalently the facet normals of the parallelepiped. We know that each spanning vector is parallel to a one-dimensional face of P. That means that it is parallel to $(d-1)$ facets of P, which implies that the scalar product with $(d-1)$ facet-normals is 0. W.l.o.g. we therefore can assume that

$$-a_j \cdot {v_i}^T = 0 \quad \text{for } 1 \leq i, j \leq d, i \neq j.$$

And since $-A$ has full rank and $v_i \neq 0$ we know that $-a_i \cdot {v_i}^T \neq 0$ for $1 \leq i \leq d$. So we know that

$$-A \cdot V = \begin{pmatrix} \beta_1 & & 0 \\ & \ddots & \\ 0 & & \beta_d \end{pmatrix},$$

where $\beta_i \stackrel{\text{def}}{=} -a_i \cdot v_i \in \mathbb{R} \setminus \{0\}$, $(1 \leq i \leq d)$. To prove the lemma we still have to show that $\beta_i = b_i$, $(1 \leq i \leq d)$.

Each point in $x \in P$ can be uniquely described by a linear combination of the spanning vectors, where all coefficients are in $[0, 1]$:

$$\forall\, x \in P : \exists\, \lambda_1, \ldots \lambda_d \in [0, 1] : x = \sum_{i=1}^{d} \lambda_i \cdot v_i.$$

This implies that for $x \in P$, $-a_i \cdot x^T$ is fully determined by the coefficient of v_i, because if $\lambda_1, \ldots, \lambda_d \in [0, 1]$ are the coefficients of $x \in P$, we get

$$-a_i \cdot x^T = -a_i \cdot \left(\sum_{j=1}^{d} \lambda_j \cdot v_j \right)^T = -a_i \cdot (\lambda_i \cdot v_i)^T = \lambda_i \cdot (-a_i \cdot {v_i}^T). \quad (*)$$

Further we know that for each $x \in P$ we have $-a_i \cdot x^T \leq b_i$. For each spanning vector v_i we know that $-a_i \cdot {v_i}^T > 0$, because otherwise v_i would not be inside P. Together with equation (*) it follows that $-a_i \cdot x^T \leq -a_i \cdot {v_i}^T$ for $x \in P$. Together we now have

$$0 \leq -a_i \cdot x^T \leq -a_i \cdot {v_i}^T \leq b_i$$

for all $x \in P$. It follows that $-a_i \cdot {v_i}^T = b_i$ since there are (by definition) $x \in P$ with $-a_i \cdot x^T = b_i$. □

Definition 3.3 In this thesis the **tile shape** is a parallelepiped located in the origin. It specifies the orientation of the facets of the tiles but leaves open the scaling of the facets. To guarantee that different tiles don't overlap we exclude the facets of the parallelepiped that do not contain the origin. This is equivalent to replacing

$$\left\{ x \in \mathbb{R}^d \mid \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ b^T \end{pmatrix} \right\}$$

with

$$\left\{ x \in \mathbb{R}^d \mid (A \cdot x^T \leq 0) \wedge (-A \cdot x^T < b^T) \right\}.$$

When we use the term spanning vector in the context of a tile shape we mean the spanning vector of the corresponding parallelepiped.

Definition 3.4 A **lattice matrix** is an integer matrix $L \in \mathbb{Z}^{d \times d}$ with full rank d. The columns of L are d *translation vectors* $l_1, \ldots, l_d$:

$$L = \begin{pmatrix} {l_1}^T & \ldots & {l_d}^T \end{pmatrix},\ l_i \in \mathbb{Z}^d,\ 1 \leq i \leq d.$$

A lattice matrix $L \in \mathbb{Z}^{d \times d}$ generates a **lattice** $\mathcal{L}(L)$ in the following way:

$$\begin{aligned}\mathcal{L}(L) &\stackrel{\text{def}}{=} \{\, L \cdot \lambda^T \mid \lambda \in \mathbb{Z}^d \,\} \\ &= \left\{ \sum_{i=1}^{d} \lambda_i \cdot l_i \mid \lambda_i \in \mathbb{Z}\ (1 \le i \le d) \right\}.\end{aligned}$$

A lattice matrix defines a linear function that transforms tile-coordinates into our regular coordinate system. Since it has full rank the inverse matrix L^{-1} exists and defines the inverse function from regular coordinates into tile coordinates.

To fully specify a tiling we still need more information: we need a scaling factor for each dimension of the tile shape.

Definition 3.5 A **scaling matrix** is a matrix

$$H = \begin{pmatrix} H_1 & & 0 \\ & \ddots & \\ 0 & & H_d \end{pmatrix} \in \mathbb{R}^{d \times d},$$

which consists of d scaling factors. Let $P \subseteq \mathbb{R}^d$ be a parallelepiped that is located in the origin. The **scaled parallelepiped** is defined by

$$P_H \stackrel{\text{def}}{=} \{\, V \cdot H \cdot V^{-1} \cdot x \mid x \in P \,\},$$

where V is the matrix of the spanning vectors of P (each column v_i is a spanning vector that lies in all but the i-th facet of P). Analogously we use the term **scaled tile shape** for the tile shape that results from scaling the corresponding parallelepiped.

At first sight the definition of the scaled parallelepiped looks too complicated. Why is it not simply defined by scaling each vector in the parallelepiped P with H, which would mean to scale the tile shape along the coordinate axes? The reason is that we want to preserve the direction of the spanning vectors when scaling the tile shape. We assume that the direction of the spanning vectors has been chosen considering the dependency structure of the program in order to minimize inter-tile communication. So instead of scaling along the coordinate axes we scale along the spanning vectors. This is accomplished by first doing a coordinate transformation into a coordinate space where the spanning vectors define an orthonormal basis (the matrix V^{-1} does this), scaling with H and transforming back to our standard coordinate system with V.

Remark 3.6 Traditionally a tile is defined by three components:

- the tile shape which traditionally only defines the direction of the spanning vectors (in our definition 3.3, the spanning vectors themselves describe the tile shape) [HCF99],
- the **tile form** which is the ratio of the length of the spanning vectors, and

- the **tile size** which is a scaling factor that is applied to all spanning vectors.

In contrast to this we describe a tile by its shape and scaling. The scaling determines the tile form and tile size in one step.

As the scaled parallelepiped P_H is scaled along the spanning vectors of P the facets of P_H have the same orientation as the facets of P. Lemma 3.7 exploits this property and shows how a parallelepiped given as an inequality system can be easily scaled.

Lemma 3.7 *If the parallelepiped $P \subseteq \mathbb{R}^d$ is given by the inequality system*

$$P \stackrel{def}{=} \left\{ x \in \mathbb{R}^d \mid \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ b^T \end{pmatrix} \right\},$$

where $A \in \mathbb{R}^{d \times d}$, $b = (b_1, \ldots, b_d) \in \mathbb{R}^d$, and $H \in \mathbb{R}^{d \times d}$ is a scaling matrix, then P_H (as defined in definition 3.5) is given by the inequality system

$$P_H = \left\{ x \in \mathbb{R}^d \mid \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ H \cdot b^T \end{pmatrix} \right\}$$

(assuming that b_i corresponds to the i-th spanning vector).

Proof We have to show

$$\begin{pmatrix} A \\ -A \end{pmatrix} \cdot (V \cdot H \cdot V^{-1}) \cdot x^T \leq \begin{pmatrix} 0 \\ H \cdot b^T \end{pmatrix} \iff \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ b^T \end{pmatrix} \qquad (*)$$

for all $x \in \mathbb{R}^d$, where V is the matrix of the spanning vectors as in definition 3.5.

Let $B \stackrel{\text{def}}{=} \begin{pmatrix} b_1 & & 0 \\ & \ddots & \\ 0 & & b_d \end{pmatrix}$. With the help of lemma 3.2 we can derive

$$A \cdot V = -B \quad \text{and} \quad V^{-1} = -B^{-1} \cdot A.$$

Using this we can transform $A \cdot (V \cdot H \cdot V^{-1})$:

$$A \cdot (V \cdot H \cdot V^{-1}) = (A \cdot V) \cdot H \cdot V^{-1} = -B \cdot H \cdot (-B^{-1} \cdot A) = (B \cdot H \cdot B^{-1}) \cdot A = H \cdot A$$

The lemma follows by simply plugging this equality into the left-hand side of equation (*), for H is a positive scaling matrix. □

Remark 3.8 If our lattice matrix L equals the matrix of spanning vectors of our tile shape, and H is our scaling matrix, we can express the scaled lattice matrix L_H simply by $L_H = L \cdot H$. We show how to compute the scaled lattice matrix in a more general setting in section 3.7.1.

Definition 3.9 Let $P \subseteq \mathbb{R}^d$ be a polyhedron, $L \in \mathbb{Z}^{d \times d}$ a lattice matrix, $S \subseteq \mathbb{R}^d$ a tile shape, and $H \in \mathbb{R}^{d \times d}$ a scaling matrix. We denote the scaled tile shape with S_H, the scaled lattice matrix with L_H, and the integral points of a set $W \subseteq \mathbb{R}^d$ with $\text{int}(W) \stackrel{\text{def}}{=} W \cap \mathbb{Z}^d$.

Then (L, S, H) is a **valid tiling** of P, if the following conditions hold:

- $\mathrm{int}(P) \subseteq \bigcup_{t\in\mathbb{Z}^d} \mathrm{int}(S_H + L_H \cdot t)$ and
- $\mathrm{int}(S_H + L_H \cdot t_1) \cap \mathrm{int}(S_H + L_H \cdot t_2) = \emptyset$ for $t_1, t_2 \in \mathbb{Z}^d,\ t_1 \neq t_2$.

(For $W \subseteq \mathbb{R}^d$ and $w \in \mathbb{R}^d$ we use the notation $W + w$ as a shorthand for $\{\, x + w \mid x \in W \,\}$.)

In other words, $(\mathrm{int}(S_H + L_H \cdot t))_{t\in\mathbb{Z}^d}$ is a partitioning of $\mathrm{int}(P)$.

3.6 Space Tiling and Parameters: A Case Study

Here we investigate the role of parameters in space tiling. We first present an example without parameters and then change it by introducing a structure parameter n. We show how we can compute a static parametric space tiling in this example. The example will motivate why we need parameters as coefficients in inequalities.

To get a feeling for what space tiling means we look at two simple examples here. In both cases we only consider the space dimensions and in both cases we have two of them.

The first example does not use parameters to describe the tile shape. The space dimensions of the target index space is given by

$$\begin{aligned} 0 \le p_2 &\le 5 \\ p_1 - 5 \le p_2 &\le p_1. \end{aligned}$$

The tile shape is given by

$$\begin{aligned} 0 \le o_2 &< 2 \\ o_1 - 2 < o_2 &\le o_1. \end{aligned}$$

A suitable lattice matrix is given by

$$L = \begin{pmatrix} 2 & 2 \\ 0 & 2 \end{pmatrix}.$$

Figure 3.2 illustrates this tiling. As usual the dots represent operations of the index space. Operations that are in the same tile are drawn in the same style (solid or shaded).

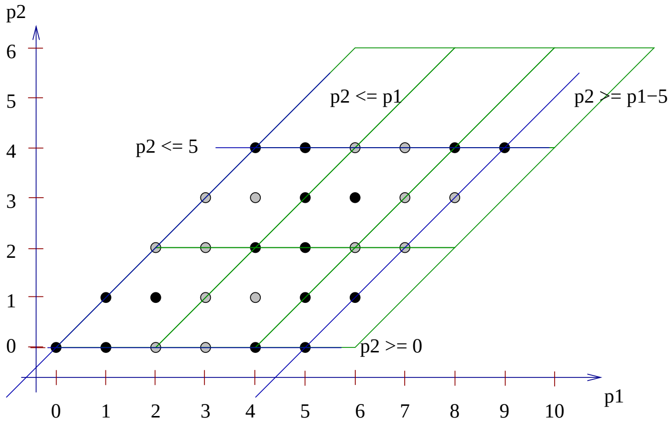


Figure 3.2: Simple tiling without parameters

Now we extend our example to make it a little more realistic: we bound the index space by a structure parameter n:

$$\begin{aligned} 0 \leq p_2 &\leq n \\ p_1 - n \leq p_2 &\leq p_1. \end{aligned} \tag{3.1}$$

For $n = 5$ this is exactly the example from above. We further assume that our target architecture is an $N\!P_1 \times N\!P_2$ processor grid and that we want to use the space tiling to map logical processors to physical processors (i.e. for each tile there is exactly one physical processor that executes all operations of this tile).

First we choose a tile shape S:

$$\begin{aligned} 0 \leq o_2 &< 1 \\ o_1 - 1 < o_2 &\leq o_1. \end{aligned} \tag{3.2}$$

Since the tile shape is the same as above, but scaled down by $\frac{1}{2}$, we use the scaled-down lattice matrix from above:

$$L = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Next we compute a scaling matrix H for the tile size. Since we want to use the space tiling to map the tiles to physical processors, the scaling matrix $H = \begin{pmatrix} H_1 & 0 \\ 0 & H_2 \end{pmatrix}$ must be chosen in a way that the number of tiles matches the number of physically available processors.

Now we are ready to formulate the necessary inequalities. In addition to inequality system 3.1 and 3.2 we add the following inequalities:

$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = L_H \cdot \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \begin{pmatrix} o_1 \\ o_2 \end{pmatrix}, \tag{3.3}$$

where r_1 and r_2 are the tile coordinates (they determine the point of the lattice $\mathcal{L}(L{\cdot}H)$), and o_1 and o_2 are the offsets inside the tiles from inequality system 3.2.

Let $p^{max} = (p_1^{max}, p_2^{max})^T = (2 \cdot n, n)^T$ be the maximum number of logical processors in each dimension, (we can calculate it by projecting inequality system 3.1 to p_1 and p_2, respectively), let $o^{max} = (o_1^{max}, o_2^{max})^T = (H_1 + H_2 - 2, H_2 - 1)^T$ be the maximum local tile offset of the scaled tile shape S_H (again, we can compute it by suitable projections of the scaled version of inequality system 3.2), and let $N\!P = (N\!P_1, N\!P_2)^T$. Then the following inequality must be satisfied:

$$p^{max} \leq L_H \cdot \left(N\!P - \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) + o^{max}.$$

Intuitively this inequality states that the tiling must reach all logical processors when using only $N\!P_1$ tiles in the first and $N\!P_2$ tiles in the second dimension. If we plug in the values and simplify the result we get the two inequalities:

$$\begin{aligned} 2 \cdot n &\leq H_1 \cdot N\!P_1 + H_2 \cdot N\!P_2 - 2, \\ n &\leq H_2 \cdot N\!P_2 - 1. \end{aligned}$$

A possible solution is $H_1 = \left\lceil \frac{n+1}{N\!P_1} \right\rceil$, $H_2 = \left\lceil \frac{n+1}{N\!P_2} \right\rceil$.

We check this for the values of the first example: $n = 5$, $N\!P_1 \times N\!P_2 = 3 \times 3$. We get $H_1 = H_2 = \left\lceil \frac{6}{3} \right\rceil = 2$. This leads to the same space tiling as above.

Now we look at a more interesting case, where the physical processor array is not square. Let $n = 19$, $N\!P_1 \times N\!P_2 = 3 \times 2$. Then $H_1 = 7$ and $H_2 = 10$. In figure 3.3 you can see the resulting index space and lattice. The operations of one specific tile (tile coordinates $(1, 1)$) are represented by black dots. The tile coordinates directly correspond to the physical processor in the two-dimensional grid.

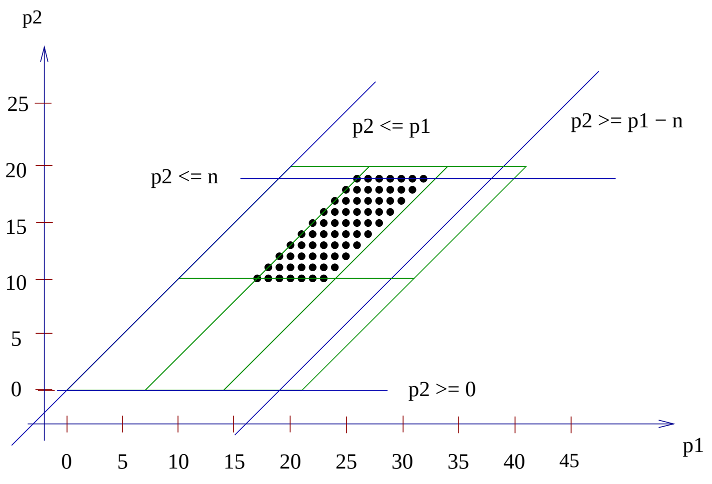


Figure 3.3: Parametric tiling for $n = 19$ and $N\!P_1 \times N\!P_2 = 3 \times 2$

So why are parameters in inequality systems, more precisely parameters as coefficients, desirable? In our second example we have several parameters: n, which is a structure parameter of our source program, $\mathcal{NP}_1$ and $\mathcal{NP}_2$, which are given by the target architecture, and H_1 and H_2, which depend on the previous parameters. All these parameters are unknown at compile time: n might depend on user input to the program, the number of physical processors might change, and with that, H_1 and H_2 will change, too. So if we cannot use parameters in inequality systems we have to compile a new version of the program whenever one of the parameters changes (or to restrict ourselves to rectangular tilings [Wie97]).

If we can deal with the parameters in our inequality systems, however, we can produce a target program that will work with different values of these parameters.

3.7 The General Parametric Space Tiling Problem

We now explore the problem of finding a parametric space-tiling in the general case of a d-dimensional parallel index space and an arbitrary tile shape.

The techniques we investigate here are relevant for our later work because the methods we develop use adaptive parametric space tiling, which is solved by a repeated application of the algorithm we develop for automatic static space tiling.

Although there is some work on parametric space tiling [Grö03], many issues, like automatically choosing a valid scaling for the tile shape, and how to perform the mapping from tiles to physical processors, have not been investigated in depth yet.

We always assume that the tile shape has already been computed. We discuss the case of one target index space and cover the necessary extensions for multiple target index spaces (from multiple statements) in the end.

We decompose the problem into subproblems. First we are concerned with valid lattice matrices for a given tile shape. The next step is to decide how to compute the scaling of the tile shape and how to map the space tiles to physical processors. It turns out that these two problems depend on each other and cannot be solved in isolation. Last of all we discuss how to create the necessary communications, how to handle multiple statements, and how to generate code for the space tiling.

3.7.1 Valid Lattice Matrices

In the second example in section 3.6 (figure 3.3) we had to make sure that the number of tiles in each dimension matches the number of physical processors. This was not difficult because the tile shape was a scaled down version of the parallel target index space.

The general problem is a little more complicated. Figure 3.4 illustrates it. On the top you see the tiled parallel index space, the two bold arrows representing the translation vectors of the lattice matrix, and on the bottom you see the tile index space (the set of all tile coordinates). The tile shape is the same as in figure 3.2, but the projection of the index space to the space dimensions is different. In the upper right corner, each tile is annotated with its tile coordinates.

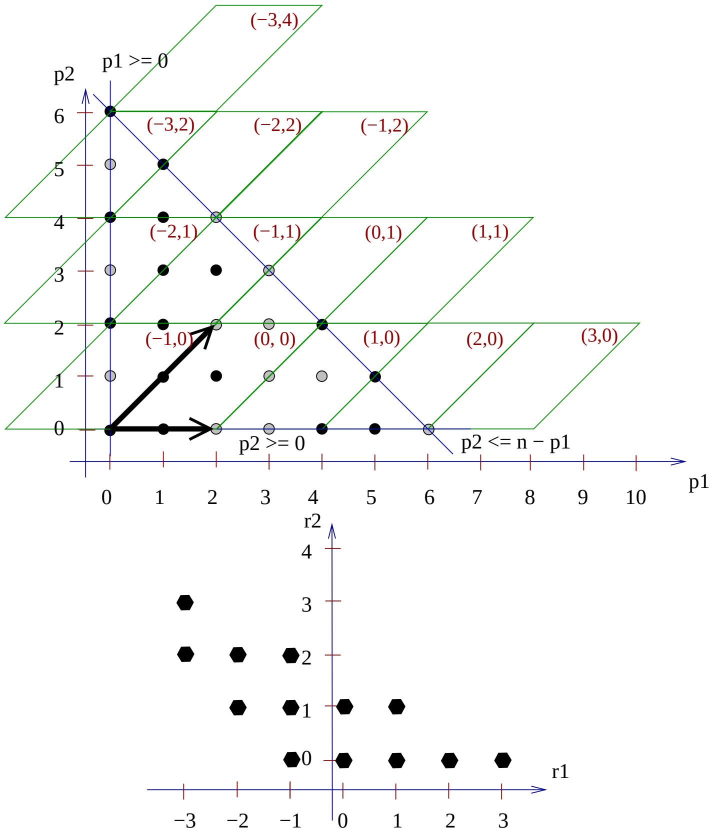


Figure 3.4: Tiling with standard lattice and corresponding tile index space

The tile coordinates in figure 3.4 range from -3 to 3 in the first dimension and from 0 to 3 in the second. If we use the tile coordinates to map tiles to physical processors, we need 7 processors in the first dimension and 4 in the second. Can we do better? Yes, if we choose a different lattice matrix. For example, the following lattice matrix

$$L' \stackrel{\text{def}}{=} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

improves the situation, as can be seen in figure 3.5. Now the first tile coordinate ranges from -1 to 3 and the second from 0 to 3. So we only need 5 physical processors in the first dimension and still 4 in the second if we use this lattice to map tiles to processors.

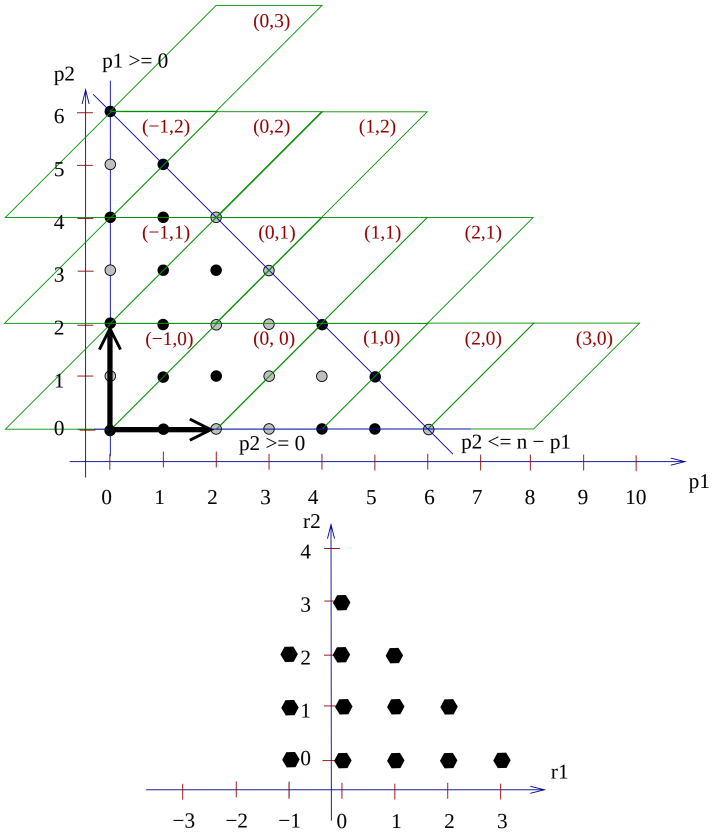


Figure 3.5: Tiling with different lattice and corresponding tile index space

We have seen that changing the lattice matrix can have an effect on the ranges of the tile coordinates. This raises the question which lattice matrices are valid for a given tile shape. The answer is simple and is given by the following definition.

Definition 3.10 Let $V \in \mathbb{R}^{d \times d}$ be the matrix of the spanning vectors of a tile shape. Then a lattice matrix $L \in \mathbb{Z}^{d \times d}$ is called **valid** if and only if $\mathcal{L}(L) = \mathcal{L}(V)$.

This definition is intuitive, since the lattice generated by a lattice matrix must "enumerate" exactly the same lattice points as the lattice generated by the spanning vectors.

Remark 3.11 The question that comes to mind here is: which lattice matrices are well-suited for our purpose? We want to improve load balance, and assume a rectangular array of physical processors. This means that a lattice matrix is well suited, if the resulting tile index space can be efficiently mapped to our physical processors.

As a measure of the quality of a lattice matrix, we propose the area of the enclosing rectangle of the tile index space, where a smaller value means higher quality. This way, a lattice matrix resulting in a nearly rectangular tile index space has a higher quality than a lattice matrix resulting in a tile index space that has the form of a skewed rectangle. For example, the standard lattice matrix in figure 3.4 results in a tile index space in which the area of the enclosing rectangle is $6 \cdot 3 = 18$, whereas the modified lattice matrix of figure 3.5 results in a tile index space in which the area of the enclosing rectangle is only $4 \cdot 3 = 12$.

The lattice matrix of figure 3.5 is a skewed version of the lattice in figure 3.4. This is no coincidence, as lemma 3.12 shows.

Lemma 3.12 *Let $V \in \mathbb{R}^{d\times d}$ be a non-singular matrix of spanning vectors. Then a lattice matrix $L \in \mathbb{R}^{d\times d}$ is valid if and only if it can be expressed as*

$$L = V \cdot U,$$

where $U \in \mathbb{Z}^{d\times d}$ is unimodular.

Proof Let $L \in \mathbb{R}^{d\times d}$ be a lattice matrix. First, let L be valid. This is equivalent to $\mathcal{L}(L) = \mathcal{L}(V)$. By definition 3.4 this is equivalent to

$$\left\{ L \cdot \lambda^T \mid \lambda \in \mathbb{Z}^d \right\} = \left\{ V \cdot \lambda^T \mid \lambda \in \mathbb{Z}^d \right\}.$$

So for every $\lambda \in \mathbb{Z}^d$ there is a $\lambda' \in \mathbb{Z}^d$ such that $L \cdot \lambda^T = V \cdot \lambda'^T$ and vice versa. Since V and L are non-singular, it follows that $\lambda' = V^{-1} \cdot L \cdot \lambda^T$ for a given $\lambda \in \mathbb{Z}^d$ and that $\lambda = L^{-1} \cdot V \cdot \lambda'^T$ for a given $\lambda' \in \mathbb{Z}^d$. Let $U \stackrel{\text{def}}{=} V^{-1} \cdot L$, then $U^{-1} = L^{-1} \cdot V$, and it follows that U defines a bijective function from $\mathbb{Z}^d$ to $\mathbb{Z}^d$, which implies that U must be unimodular [Xue00]. Since $U = V^{-1} \cdot L$ we get $L = V \cdot U$.

Let conversely $U \in \mathbb{Z}^{d\times d}$ be a unimodular matrix such that $L = V \cdot U$. We have to show that L is valid, thus $\mathcal{L}(L) = \mathcal{L}(V)$. This equation follows from the fact that for all $\lambda \in \mathbb{Z}^d$

$$L \cdot \lambda^T = (V \cdot U) \cdot \lambda^T = V \cdot (U \cdot \lambda^T) = V \cdot \lambda'^T,$$

where $\lambda'^T = U \cdot \lambda^T$, and the fact that U is unimodular (and hence defines a bijection). □

With the help of this lemma we are now able to define the *scaled lattice matrix* for any valid lattice matrix L.

Definition 3.13 Let $L \in \mathbb{R}^{d\times d}$ be a valid lattice matrix for some tile shape, and let $V \in \mathbb{R}^{d\times d}$ be the matrix of the spanning vectors of that tile shape. Let further $U \in \mathbb{Z}^{d\times d}$ be the unimodular matrix from lemma 3.12 such that $L = V \cdot U$. Then the **scaled lattice matrix** $L_H \in \mathbb{R}^{d\times d}$ is defined in the following way:

$$L_H \stackrel{\text{def}}{=} (V \cdot H) \cdot U.$$

Note that if $L = V$ as in remark 3.8, we get $U = I$ (where I is the unit matrix of the respective dimensionality) and $L_H = (V \cdot H) \cdot I = V \cdot H$. So our definition is consistent with the definition in remark 3.8.

To show that definition 3.13 is sound, we have to show that L_H defines a valid lattice for the scaled tile shape, as given in definition 3.5. So let $V \in \mathbb{R}^{d \times d}$ be the matrix of the spanning vectors of a tile shape and choose L, H, L_H as in definition 3.13. The first thing we note is that $V \cdot H$ is the matrix of the spanning vectors of the scaled tile shape. Together with lemma 3.12 this means that L_H is a valid lattice matrix, because $L_H = (V \cdot H) \cdot U$.

We have characterized valid lattice matrices, have talked about their quality, and have shown how to scale them. But an interesting question is still open: How can we find the "optimal" lattice matrix for a given index space and tile shape? This is equivalent to finding a unimodular transformation of the tile index space that minimizes the area of the enclosing rectangle. This is a non-trivial problem and seems like an interesting area for future research.

3.7.2 Scaling and Mapping

In this section we examine the problem of choosing a valid scaling for the tile shape and of mapping space tiles to physical processors. We explore different approaches, but only one of them will lead to a simple and efficient algorithm.

Our scaling matrix defines the tile form and the tile size in one step. It allows an independent scaling of the tile-shape along each spanning vector, which means that it can change the tile form. If it is not desirable to change the tile form, for example because it has been chosen to be optimal with regard to the dependency structure, we discuss how this can be achieved as a special case.

When we use tiling to map tiles to physical processors we get additional constraints on the size and number of tiles. When tiling to coalesce operations the only constraint is that the tiles cover all operations and that tiles do not overlap. This ensures that all operations will be executed exactly once in the tiled program. When we use the space tiles to map operations to processors we get the additional constraint that the number of space tiles must match (i.e. must not be greater than) the number of physical processors (either in each dimension or in total as we will see). If this constraint cannot be satisfied we have to provide an extra mechanism to make sure that tiles are only mapped to existing physical processors.

These additional constraints cause some problems, however. They are not easily expressed as linear inequalities, because they involve the projection of the inequality system (for example the range of the tile coordinates in different dimensions). Furthermore, the computation of the scaling matrix cannot be handled independently, but must be considered at the same time (since the scaling directly influences the number of needed physical processors).

We assume that the target machine is a d-dimensional processor array, where d is also the dimensionality of our allocation. The vector $N\!P = (N\!P_1, \ldots, N\!P_d) \in \mathbb{N}_1^d$ holds the number of physical processors in each dimension. In inequalities, we use the variables $r_1, \ldots, r_d$ for the space-tile index in each dimension. As above, we use the variables $p_1, \ldots, p_d$ for the virtual processors and $o_1, \ldots, o_d$ for the offsets inside the tiles. The scaling matrix H is a parametric matrix of the following form:

$$H = \begin{pmatrix} H_1 & \dots & 0 \\ & \ddots & \\ 0 & \dots & H_d \end{pmatrix},$$

where $H_1, \dots, H_d$ are the scaling parameters for each spanning vector.

The following simple example (similar to that of figure 3.2) is used throughout this section. The parallel index space is described by

$$\begin{aligned} &0 \leq p_1 \\ &0 \leq p_2 \leq n - p_1 \end{aligned} \tag{3.4}$$

The tile shape is given by inequality system 3.2, and the lattice matrix is

$$L = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}. \tag{3.5}$$

A First Attempt: Mapping Tiles Sequentially

In this first attempt, we try to map the tiles sequentially in the order in which they are enumerated by the loop nest of the target program. We develop this attempt based on our example.

Our goal is to compute a scaling that ensures that the number of tiles resulting from this scaling is less than the total number of available physical processors.

Assuming that we have found a valid scaling we propose to build a mapping from tiles to physical processors by enumerating all tiles and storing the resulting mapping. This enumeration is performed before the actual computation starts. Claßen developed this approach in his work on code generation in LooPo [Cla04].

We now investigate the problem of computing the scaling. The approach we take is to express the total number of tiles as a function of the scaling parameters H_i and to choose the H_i in such a way that the total number of tiles is less than the total number of physical processors.

Equation 3.3 is generalized to d-dimensional space (where $L \in \mathbb{Z}^{d \times d}$ is again the lattice matrix and H is a (parametric) scaling matrix):

$$\begin{pmatrix} p_1 \\ \vdots \\ p_d \end{pmatrix} = L_H \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_d \end{pmatrix} + \begin{pmatrix} o_1 \\ \vdots \\ o_d \end{pmatrix}. \tag{3.6}$$

We call equation 3.6 the **space-tiling equation**. It specifies how to enumerate the tiles (with the lattice matrix L_H) and how to enumerate the virtual processors inside each tile (with the tile-offset variables o_i).

We introduce a new parameter $\sharp_r$, the total number of tiles. Ideally, we would like to know the exact value of $\sharp_r$ for a given tiled target index space. However, this value is difficult to compute, because in order to determine it we have to count the number of integral points in a parametrized polyhedron. An alternative solution is to approximate $\sharp_r$. This problem will be addressed below.

Consider the following example. We build an inequality system from the following parts:

- inequality system 3.4 for the parallel index space,
- a scaled inequality system 3.2 for the scaled tile shape,
- the constraint that the total number of tiles is not greater than the total number of processors ($\sharp_r \leq N\!P_1 \cdot N\!P_2$),
- the space-tiling equation 3.6 with the lattice matrix from equation 3.5.

$$\begin{aligned} 0 &\leq p_1 \\ 0 &\leq p_2 \leq n - p_1 \\ 0 &\leq o_2 < H_2 \\ o_1 - H_1 &< o_2 \leq o_1 \\ \sharp_r &\leq N\!P_1 \cdot N\!P_2 \\ p_1 &= H_1 \cdot r_1 + H_2 \cdot r_2 + o_1 \\ p_2 &= H_2 \cdot r_2 + o_2 \end{aligned} \tag{3.7}$$

To compute $\sharp_r$, we project system 3.7 to r_1 and r_2 and estimate the number of integral points inside the projected polytope. As long as we assure that our estimate is not less than the true value, the result will be correct (although the load balance will become worse if the error of our estimate increases).

It is important that we can express our estimate of $\sharp_r$ in terms of the structure parameters H_1, H_2, $N\!P_1$, and $N\!P_2$, because only then we can hope to solve system 3.7 for H_1 and H_2.

A very conservative estimate for the number of space tiles can be obtained by computing the smallest bounding rectangle of the tile index space, which again can be computed from the projections to each space-dimension. In that case, however, we cannot achieve a good load balance, because the total number of tiles will be much less than the number of processors (cf. figure 3.4).

Another estimate is the volume of the polytope that describes the parallel dimensions of the target index space. This is not an upper bound for the number of integral points, however. Also, the volume of a parametrized polytope is still not easy to compute, so we don't investigate this approach any further.

None of these estimates promises to give a simple term for $\sharp_r$, which we need to efficiently solve system 3.7. So we abandoned the sequential mapping. It seems well-suited only in the case of a fixed, non-parametric tiling, because in that case the scaling is already given by the tile form and size [Cla04].

Mapping with Tile Coordinates

This approach uses the tile coordinates to index the physical processors. But there is a problem: as we have seen in figure 3.4 and figure 3.5, tile coordinates can be negative. The physical processor indices, however, must be positive.

A First Attempt to Avoid Negative Tile Coordinates We decided to address this problem by splitting the tile index variable r_i into a sum of two new arbitrary variables r^+ and r^- which are constrained by the new inequalities

$$0 \leq r_i^+ \leq \mathcal{NP}_i - 1 \qquad \text{and} \qquad -\mathcal{NP}_i + 1 \leq r_i^- \leq 0$$

and by

$$r_i = r_i^- + r_i^+$$

for each dimension $1 \leq i \leq d$. Note that for a given value of r_i the values of r_i^- and r_i^+ are not necessarily uniquely defined. We call the r_i^- the **negative offset variables**.

In order to find a valid tiling for which all tiles can be enumerated with the available processors we need to compute values for the negative offset variables and for the scaling matrix. The correct choice of the negative offset variables is crucial: they must be chosen to ensure that all tiles can be reached with the positive part (r_i^+), and that all tiles can be covered with the physical processors.

Figure 3.6 illustrates our example with $n = 6$, $\mathcal{NP}_1 = 3$, $\mathcal{NP}_2 = 2$. The tile shape is drawn at the origin, and the tiles that have a non-empty intersection with the index space are displayed with dashed borders, labeled with their tile index. The solution for this example is easy to see: choosing $r_1^- = -1$, $r_2^- = 0$, $H_1 = H_2 = 4$ results in the displayed tiling and allows to enumerate all tiles with the positive offset variables.

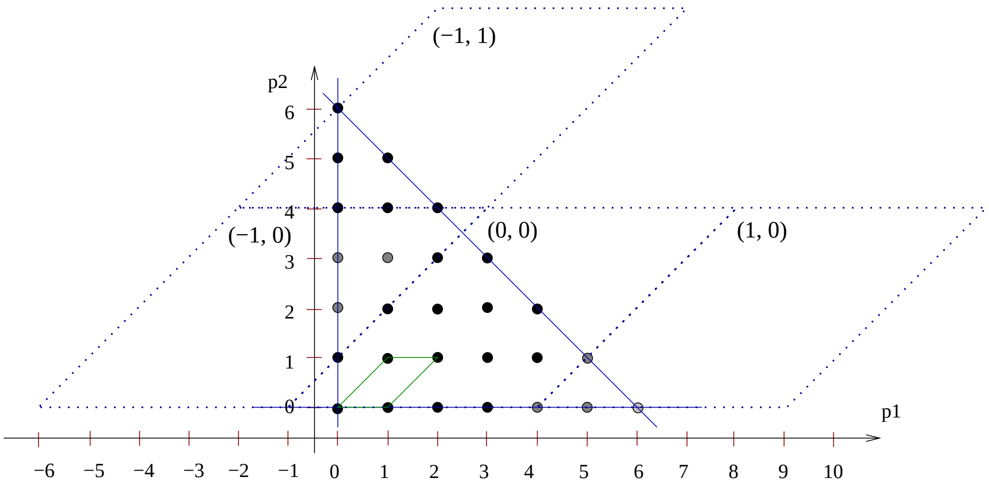


Figure 3.6: Mapping tiles to processors

We now formally show how to solve this mapping problem for arbitrary values of $\mathcal{NP}_1$, $\mathcal{NP}_2$, and n. The space-tiling equation 3.6 has to be modified to the following form (where $L \in \mathbb{Z}^{d \times d}$ is again the lattice matrix and H is a (parametric) scaling matrix):

$$\begin{pmatrix} p_1 \\ \vdots \\ p_d \end{pmatrix} = L_H \cdot \begin{pmatrix} r_1^- + r_1^+ \\ \vdots \\ r_d^- + r_d^+ \end{pmatrix} + \begin{pmatrix} o_1 \\ \vdots \\ o_d \end{pmatrix}. \tag{3.8}$$

The combined inequality system of the tile shape, the parallel dimensions of the target index space, and the new space-tiling equation 3.8 enumerates the same virtual processors as the parallel dimensions of the target index space. Or more precisely, the projection of the combined inequality system to the parallel dimensions $p_1, \ldots, p_d$ is equal to the parallel index space.

The difficult bit is that we treat the negative offset variables r_i^- as constants. It helps to think about it the following way: We sort the combined inequality system in the order

$$(r_1^-, \ldots, r_d^-, p_1, \ldots, p_d, r_1^+, \ldots, r_d^+, o_1, \ldots, o_d),$$

the outermost variables being the negative offset variables. We do not create loops that enumerate them but choose appropriate values for them at compile time (these values can include parameters).

The difficulty is that it is not obvious for which choices of the outer negative offset variables we can enumerate all logical processors of the target index space.

For an example look at figure 3.4. Here, $d = 2$. If we choose $r_1^- = r_2^- = 0$, we can not hope to enumerate the logical processor with coordinates $(0, 1)$, because it lies inside the tile with tile coordinates $(-1, 0)$, but the tile coordinates we can enumerate are all non-negative (the negative tile offsets are zero).

One solution for this example is to set $r_1 := -3$ and $r_2 := 0$. The point $(-3, 0)$ is the lower left corner of the smallest enclosing rectangle of the polytope of all tile coordinates (see figure 3.4 on the right).

In general, for a valid choice of the negative tile offsets, we have to find the smallest enclosing d–dimensional rectangle of all tile coordinates. At the same time we have to choose a valid scaling for the tile shape (which was not the case in the example above). This scaling, however, also influences the polytope of all tile coordinates.

We solve our simple parametrized example to demonstrate the difficulties of this approach.

We first give the combined inequality system together with the space-tiling equation and the bounds for the negative and positive tile coordinates (note that the tile shape and the lattice matrix have been scaled with the usual scaling matrix $H = \begin{pmatrix} H_1 & 0 \\ 0 & H_2 \end{pmatrix}$):

$$
\begin{aligned}
0 &\le p_1 \\
0 &\le p_2 \le n - p_1 \\
0 &\le o_2 < H_2 \\
o_1 - H_1 &< o_2 \le o_1 \\
0 &\le r_1^+ \le N\!P_1 - 1 \\
0 &\le r_2^+ \le N\!P_2 - 1 \\
-N\!P_1 + 1 &\le r_1^- \le 0 \\
-N\!P_2 + 1 &\le r_2^- \le 0 \\
p_1 &= H_1 \cdot (r_1^- + r_1^+) + H_2 \cdot (r_2^- + r_2^+) + o_1 \\
p_2 &= H_2 \cdot (r_2^- + r_2^+) + o_2
\end{aligned}
\tag{3.9}
$$

After sorting in the order $(r_1^-, r_2^-, p_1, p_2, r_1^+, r_2^+, o_1, o_2)$, we get the following bounds for p_1 and p_2:

$$
\begin{aligned}
0 &\le p_1 \\
H_1 \cdot r_1^- &\le p_1 \\
H_1 \cdot r_1^- + H_2 \cdot r_2^- &\le p_1 \\
p_1 &\le n \\
p_1 &\le \left\lfloor \frac{n + H_1 \cdot r_1^- + H_1 \cdot N\!P_1 - 1}{2} \right\rfloor \\
p_1 &\le H_1 \cdot r_1^- + H_2 \cdot r_2^- + H_1 \cdot N\!P_1 + H_2 \cdot N\!P_2 - 2 \\
p_1 &\le n - H_2 \cdot r_2^- \\
0 &\le p_2 \\
H_2 \cdot r_2^- &\le p_2 \\
1 - H_1 \cdot r_1^- - H_1 \cdot N\!P_1 &\le p_2 \\
p_2 &\le n - p_1 \\
p_2 &\le p_1 - H_1 \cdot r_1^- \\
p_2 &\le H_2 \cdot r_2^- + H_2 \cdot N\!P_2 - 1
\end{aligned}
\tag{3.10}
$$

Our next task is to find expressions for H_1, H_2, r_1^-, and r_1^+ (in the remaining structure parameters $N\!P_1$, $N\!P_2$, and n) such that the polytope defined by the inequality system 3.10 is equal to the polytope defined by inequality system 3.4.

By comparing system 3.10 with system 3.4, it can be seen that the following inequalities must be satisfied to make the polytopes equal:

$$
\begin{aligned}
H_1 \cdot r_1^- &\leq 0 \\
H_1 \cdot r_1^- + H_2 \cdot r_2^- &\leq 0 \\
n &\leq \left\lfloor \frac{n + H_1 \cdot r_1^- + H_1 \cdot N\!P_1 - 1}{2} \right\rfloor \\
n &\leq H_1 \cdot r_1^- + H_2 \cdot r_2^- + H_1 \cdot N\!P_1 + H_2 \cdot N\!P_2 - 2 \\
n &\leq n - H_2 \cdot r_2^- \qquad (3.11) \\
H_2 \cdot r_2^- &\leq 0 \\
1 - H_1 \cdot r_1^- - H_1 \cdot N\!P_1 &\leq 0 \\
n - p_1 &\leq p_1 - H_1 \cdot r_1^- \\
n - p_1 &\leq H_2 \cdot r_2^- + H_2 \cdot N\!P_2 - 1
\end{aligned}
$$

After some simplifications (where we exploit that $H_1 \geq 1$ and $H_2 \geq 1$, $r_1^- \leq 0$ and $r_2^- \leq 0$, and $p_1 \geq 0$) we get the following equivalent system:

$$
\begin{aligned}
n &\leq H_1 \cdot r_1^- + H_1 \cdot N\!P_1 - 1 \\
n &\leq H_1 \cdot r_1^- + H_2 \cdot r_2^- + H_1 \cdot N\!P_1 + H_2 \cdot N\!P_2 - 2 \\
1 - H_1 \cdot r_1^- - H_1 \cdot N\!P_1 &\leq 0 \qquad (3.12) \\
n + H_1 \cdot r_1^- &\leq 0 \\
n - H_2 \cdot r_2^- - H_2 \cdot N\!P_2 + 1 &\leq 0
\end{aligned}
$$

Intuitively, these inequalities state that the additional bounds from system 3.10 are always weaker than the bounds from system 3.4. The problem with system 3.12 is that, if we regard H_1, H_2, r_1^-, r_2^- as variables, it is not linear.

A possible solution is: $r_1 = \left\lceil \frac{-n}{H_1} \right\rceil$, $r_2 = 0$, $H_1 = \left\lceil \frac{n+1}{N\!P_1 - N\!P_2 + 1} \right\rceil$, $H_2 = \left\lceil \frac{n+1}{N\!P_2} \right\rceil$, if $N\!P_1 \geq N\!P_2$. This can be verified by plugging these values into system 3.12.

For the example in figure 3.4, where $n = 6$, $N\!P_1 = 7$ and $N\!P_2 = 4$ we get $r_1^- = -3$, $r_2^- = 0$, $H_1 = \lceil \frac{7}{4} \rceil = 2$, and $H_2 = \lceil \frac{7}{4} \rceil = 2$.

To sum up, we saw that the approach of splitting the tile index into a negative and a positive component yielded a quadratic, parametric inequality system, of which we needed an integral solution. In our example we could find a solution of this system. But it is unclear if it is possible to develop a general solution for arbitrary tile shapes, index spaces, and lattice matrices.

A Better Method Since splitting up the tile coordinates into positive and negative offset variables turned out to be difficult to handle, we found a different solution. The key idea is to use a modified space-tiling equation. We introduce new parameters $p_i^{\min}$ and $p_i^{\max}$ for $1 \leq i \leq d$. For the case of a rectangular tile shape these are simply the projection of the index space to the respective space dimension (or more precisely the minimum and maximum value of that dimension). In the general case, however, they have a different meaning as explained later. Now we can formulate the new space-tiling equation:

$$\begin{pmatrix} p_1 \\ \vdots \\ p_d \end{pmatrix} = L_H \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_d \end{pmatrix} + \begin{pmatrix} o_1 \\ \vdots \\ o_d \end{pmatrix} + \begin{pmatrix} p_1^{\min} \\ \vdots \\ p_d^{\min} \end{pmatrix}. \tag{3.13}$$

The parameter $p^{min} = (p_1^{\min}, \dots, p_d^{\min})$ is called the **virtual processor offset**. We show how to choose the virtual processor offset to ensure that the constraint $r_i \geq 0$ is satisfied.

To compute the scaling in each dimension, we first note that for the special case of a rectangular tile shape, where each spanning vector has unit length, it is easy to find a valid scaling for each dimension $1 \leq i \leq d$:

$$H_i \stackrel{\text{def}}{=} \left\lceil \frac{p_i^{\max} - p_i^{\min} + 1}{N\!P_i} \right\rceil,$$

where we set $p_i^{\max}$ and $p_i^{\min}$ to the least upper and greatest lower bound (rounded up and down, respectively, to the next integral value) of the projection of the parallel index space to the i-th space dimension.

In figure 3.7, the bold lines define a polytope whose integral points are displayed by black dots (the points that are in the same tile are displayed in the same fill style). The tile shape is displayed at the origin. It is rectangular and has spanning vectors of unit length. The rectangle that intersects the polytope is defined by the corner points $(p_1^{\min}, p_2^{\min}) = (2, 1)$ and $(p_1^{\max}, p_2^{\max}) = (6, 5)$. For the example we assume $N\!P_1 \times N\!P_2 = 3 \times 3$. This results in $H_1 = H_2 = 2$. The scaled tiles that cover the polytope are displayed with dotted lines.

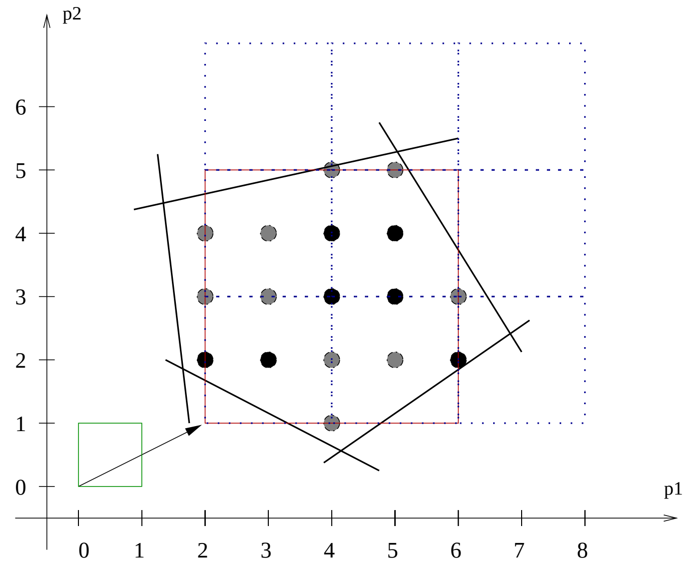


Figure 3.7: Rectangular tiling of a two-dimensional index space

We now reduce the general case, where the tile shape is an arbitrary parallelepiped, and the spanning vectors can be of arbitrary length to the case of a rectangular tile shape with spanning vectors of unit length. Let the tile shape be defined by the parallelepiped

$$\left\{ x \in \mathbb{R}^d \mid \begin{pmatrix} A \\ -A \end{pmatrix} \cdot x^T \leq \begin{pmatrix} 0 \\ b^T \end{pmatrix} \right\},$$

where $A \in \mathbb{Z}^{d \times d}$ and $b = (b_1, \ldots, b_d) \in \mathbb{Z}^d$.

Luckily, we know a coordinate system in which this tile shape is rectangular and of unit size, or more precisely, we know the matrix that describes the coordinate transformation to this coordinate system:

$$\begin{pmatrix} \frac{-1}{b_1} & & 0 \\ & \ddots & \\ 0 & & \frac{-1}{b_d} \end{pmatrix} \cdot A.$$

We call this coordinate system the **tile-shape coordinate system**. To see that this holds, consider the matrix $V \in \mathbb{Z}^{d \times d}$ of the spanning vectors of the tile shape. Lemma 3.2 tells us that

$$V = (-A)^{-1} \cdot \begin{pmatrix} b_1 & & 0 \\ & \ddots & \\ 0 & & b_d \end{pmatrix}.$$

By inverting both sides of the equation we get

$$V^{-1} = \begin{pmatrix} \frac{1}{b_1} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{b_d} \end{pmatrix} \cdot -A = \begin{pmatrix} \frac{-1}{b_1} & & 0 \\ & \ddots & \\ 0 & & \frac{-1}{b_d} \end{pmatrix} \cdot A.$$

The proposition follows because V^{-1} maps the i-th spanning vector to the i-th unit vector.

To compute the scaling we transform the parallel index space to the tile-shape coordinate system, similar to the approach Andonov [And00] developed for two-dimensional tiling. When we talk about the transformed index space in the remainder of this section we do not mean the space-time mapping of the polytope model, but the transformation to the tile-shape coordinate system.

If the parallel index space is given by $D \cdot x^T \geq c^T$, with $D \in \mathbb{Z}^{d \times d}$ and $c \in \mathbb{Z}^d$, then after transforming to the tile-shape coordinate system it is given by $(D \cdot V) \cdot x^T \geq c^T$. We now determine the scaling and offset as described above in the rectangular case. All we have to do is to transform our solution back. The bounding rectangle is transformed into a parallelepiped that is a scaled version of the tile-shape. We call it the **bounding tile** of the parallel index space. Figure 3.8 illustrates the algorithm.

To sum up, our algorithm for calculating the scaling and the tile offset variables has the following steps:

1. Transform the parallel index space into tile-shape coordinate system.
2. Calculate the projection to each coordinate axis. This results in the bounding rectangle of the transformed index space. Compute the scaling in each dimension (see figure 3.8(b)).
3. Transform the bounding rectangle back to the parallel index space, which results in the bounding tile, defined by the parameters $p_i^{\min}$ and $p_i^{\max}$ (see figure 3.8(c)).

Algorithm 1: Computing the scaling and offset of a tiling

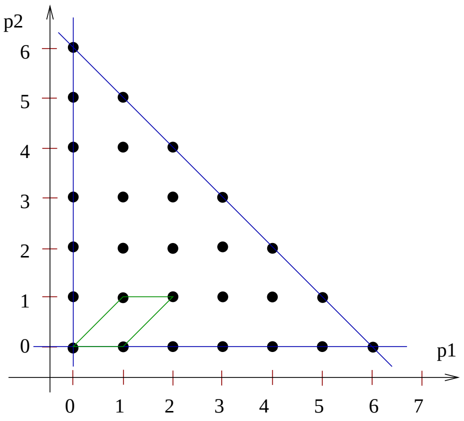
Lemma 3.14 *Algorithm 1 generates a valid tiling in which all tile coordinates r_i satisfy the constraint $0 \leq r_i \leq N\!P_i - 1$ if the lattice matrix $L \in \mathbb{Z}^d$ is equal to the matrix $V \in \mathbb{Z}^d$ of the spanning vectors of the tile shape.*

Proof We prove the lemma for the rectangular and unit-length case first. Here, we know that $L = I$, the unit matrix. It follows that the scaled lattice matrix L_H is simply the scaling matrix H. The parameters $p_i^{\min}$ and $p_i^{\max}$ are the projection to the i-th parallel dimension. It follows that the integral points of the parallel index space are fully contained in the bounding rectangle they define. We now show that every integral point inside this bounding rectangle can be reached with the scaled tiling (i.e. scaled tile shape and scaled lattice). This implies that every integral point of the parallel index space can be reached. So let $x = (x_1, \ldots, x_d) \in \mathbb{Z}^d$ be a point of the bounding rectangle. Let $1 \leq i \leq d$. We know that $p_i^{\min} \leq x_i \leq p_i^{\max}$. We define $r_i \stackrel{\text{def}}{=} \left\lfloor \frac{x_i - p_i^{\min}}{H_i} \right\rfloor$. This implies that $H_i \cdot r_i + p_i^{\min} \leq x_i < H_i \cdot (r_i + 1) + p_i^{\min}$. In other words, the point x lies inside a tile that is the tile shape scaled by the scaling matrix H and shifted to the point $L_H \cdot r + p^{\min}$ (where $r = (r_1, \ldots, r_d)$ and $p^{\min} = (p_1^{\min}, \ldots, p_d^{\min})$). We now show that r_i satisfies $0 \leq r_i \leq N\!P_i - 1$. Assume $r_i < 0$. Since $H_i > 0$ this implies that $x_i < p_i^{\min}$, which is a contradiction. To show the other bound, we plug in the definition of H_i:

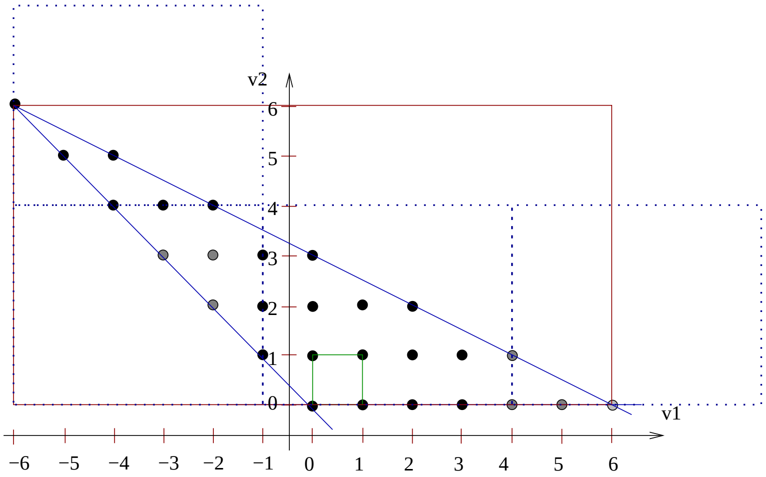
$$r_i = \left\lfloor \frac{x_i - p_i^{\min}}{\left\lceil \frac{p_i^{\max} - p_i^{\min} + 1}{N\!P_i} \right\rceil} \right\rfloor \leq \frac{x_i - p_i^{\min}}{\frac{p_i^{\max} - p_i^{\min} + 1}{N\!P_i}} \leq \frac{p_i^{\max} - p_i^{\min}}{p_i^{\max} - p_i^{\min} + 1} \cdot N\!P_i < N\!P_i.$$

Since $r_i \in \mathbb{Z}$ we get the desired bound $r_i \leq N\!P_i - 1$.

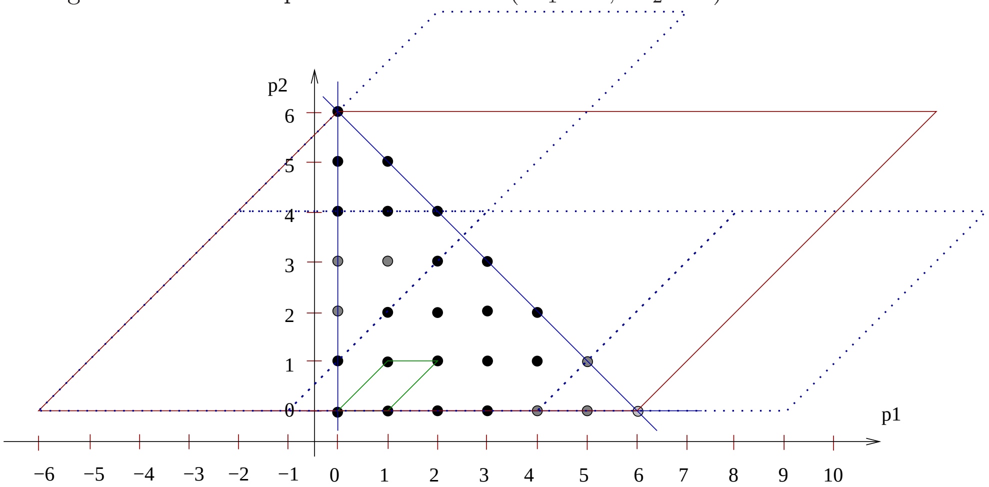
To prove the case of an arbitrary tile shape, we assume that H_i, $p_i^{\min}$, and $p_i^{\max}$ have been chosen according to the algorithm. We also know that the lattice matrix L is the matrix of the spanning vectors. The proposition follows because the parallel index space is, by construction, enclosed by the bounding tile, and every point inside the bounding tile can be reached with a tile $L_H \cdot r + p_i^{\min}$, where $0 \leq r_i \leq N\!P_i - 1$ for $1 \leq i \leq d$. $\square$



(a) Parallel index space with tile shape drawn at the origin.



(b) Index space and tile shape are transformed to the tile-shape coordinate system. The bounding rectangle is obtained by projecting the transformed parallel index space to each dimension. The dotted rectangles are the tiles, scaling is obtained as explained in the text ($NP_1 = 3$, $NP_2 = 2$).



(c) The index space, the tile shape, the bounding rectangle and the tiles are transformed back. The bounding rectangle has been transformed to a bounding parallelepiped that is a scaled version of the tile shape.

Figure 3.8: Coordinate transformation

Refining the Algorithm: Choosing a Lattice Matrix The lattice matrix can have a big impact on the load balance, as illustrated in figure 3.9. It shows the tile index spaces from figure 3.4 and figure 3.5. The greyed polygons denote processors that are idle.

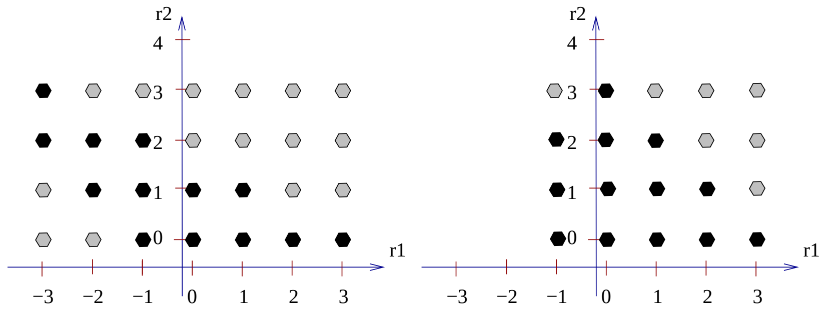


Figure 3.9: Impact of the lattice on the load balance

However, algorithm 1 assumes that the lattice matrix is the matrix of the spanning vectors of the tile shape. If we use a different valid lattice matrix, the algorithm can give incorrect results, as figure 3.10 illustrates, where the lattice matrix is $L = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. This lattice matrix is valid, because it can be written as

$$L = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

(cf. lemma 3.12). The bounding rectangle is what we called the bounding tile shape above, but it is a scaled version of the lattice vectors, not the tile shape in this case. Figure 3.10 shows that the operations with virtual processor index $(0, 1)$, $(0, 3)$, and $(0, 5)$ are not covered by any tile.

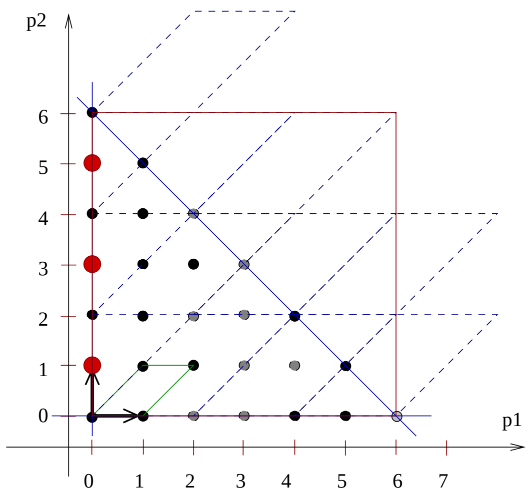


Figure 3.10: Algorithm 1 applied with a modified lattice matrix

Extending algorithm 1 to arbitrary (valid) lattice matrices seems a worthwhile direction for further research.

3.7.3 Space Tiling for Multiple Statements

Since almost any useful program contains more than one statement, we explain how to extend the presented algorithm to multiple statements. We look at algorithm 1 which uses the tile coordinates to map space tiles to physical processors.

The modification is simple: we first compute the origin of the bounding tile that contains the parallel index spaces of *all* statements. This can easily be done by computing, for each $1 \leq i \leq d$, the minimum of the parameters $p_i^{\min}$ for each statement.

In a second step we use these offsets for computing the scaling parameters in each dimension for each statement. We then use the maximum of the scalings in each dimension. This way we ensure that all parallel index spaces can be completely covered.

3.7.4 Preserving the Tile Form

We already mentioned that it might be desirable to preserve the tile-form, because some algorithms for computing the tile-shape produce a tile-form that minimizes communications [Xue97a].

We can achieve that by a simple modification of algorithm 1. Instead of using a separate scaling factor H_i for each spanning vector of the tile shape, we now only use one scaling-factor H'. It is computed as the maximum of the scaling factors that algorithm 1 computes.

It should be obvious that this can have a negative impact on the load-balance of the parallel program, so at this point we have yet another trade-off between the communication costs and the load balance.

3.7.5 Communications with Space Tiling

The generation of the communication statements is analogous to the case without space tiling in section 2.5. Instead of the virtual processor coordinates, we use the tile coordinates to enumerate the communication partners. The corresponding physical processor is determined according to the mapping we use, and local communications are eliminated at runtime.

3.7.6 Generalized Polyhedra

When calculating the projections in step 2 of algorithm 1, we will get a generalized polyhedron which describes the lower and upper bound in each dimension. To form the bounding rectangle, we have to combine the decision trees of these generalized polyhedra by building a decision tree of all possible combinations of case distinctions from each generalized polyhedron. This way we create a new decision tree whose leafs are bounding rectangles. In other words, this is a generalized polyhedron that describes the bounding rectangle. We illustrate this process based on an example.

Consider the following example. We have a two dimensional parallel index space, and the transformation of the parallel index space to the tile-shape coordinate system (see section 3.7.2) is given by the following inequality system:

$$\begin{aligned} 0 \leq v_2 \leq \frac{n}{2} + K_2 \cdot v_1 \\ K_1 \cdot v_1 \leq v_2, \end{aligned} \tag{3.14}$$

where v_1 and v_2 are variables, n, K_1, and K_2 are parameters. To ensure that system 3.14 is bounded we assume that we are in the following context: $n > 0$, $K_1 \neq 0$, $K_2 \neq 0$, $\mathrm{sgn}(K_1) = \mathrm{sgn}(K_2)$, and $|K_1| > |K_2|$.

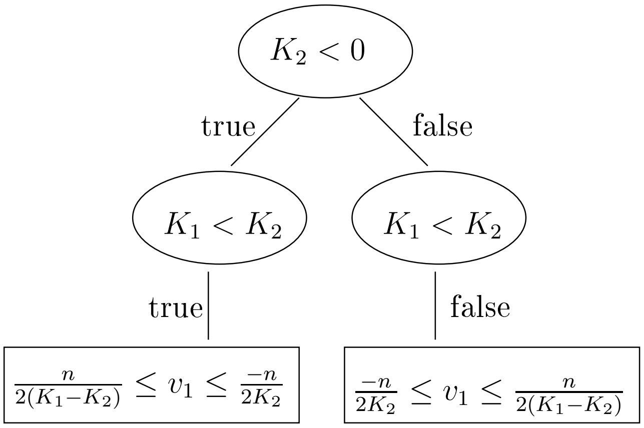


Figure 3.11: Bounds for v_1

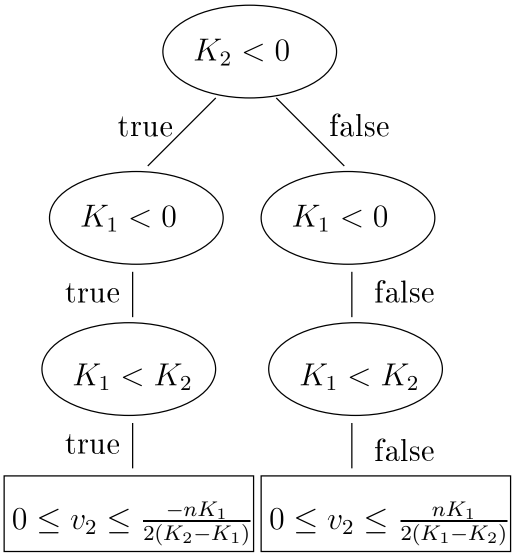


Figure 3.12: Bounds for v_2

Figure 3.11 and figure 3.12 show the result of the projections to v_1 and v_2, each represented by a generalized polyhedron. Note that we have only drawn those leafs of the decision trees whose guards are compatible with the context of our parameters.

We now create the bounding box by combining the decision trees drawn in figure 3.11 and figure 3.12. We do this by creating a new decision tree. We initialize it with the tree in figure 3.11, but without the leaf nodes. Instead of the leaf nodes we append the tree in figure 3.12. In the left copy of this tree we add the inequalities from the left leaf of figure 3.11, and in the right copy we add the inequalities from the right leaf of figure 3.11. This intermediate tree has four leafs. But after simplifying it and removing leafs with contradictory guards we get the final tree with only two leafs shown in figure 3.13.

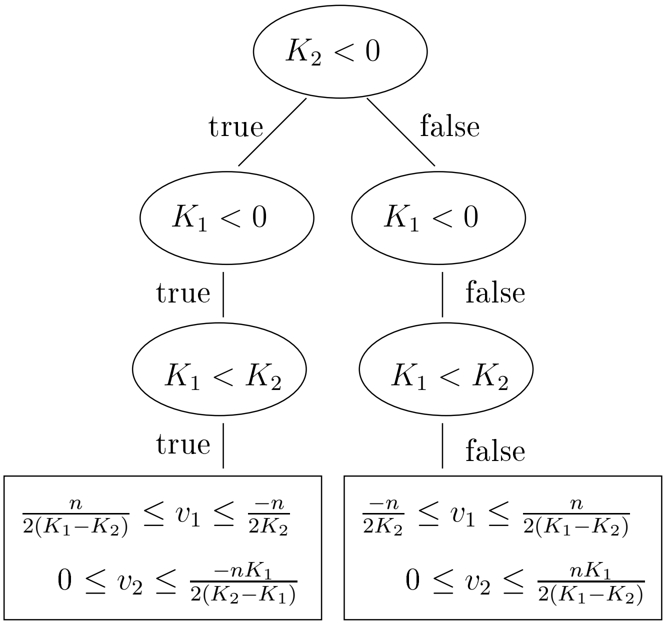


Figure 3.13: The combined bounding boxed formed from the decision trees in figure 3.11 and 3.12

For $K_1 = -1$ and $K_2 = -\frac{1}{2}$, and $n = 6$, inequality system 3.14 describes the parallelepiped drawn in figure 3.8(b). In this special case we can easily check whether the bounding box and scaling parameters are equal to those given in figure 3.8.

3.7.7 Making it Practical

We do not compute the scaling at compile-time because the parameter values are not known. Instead, we create code from the generalized polyhedron describing the bounding rectangle. This code computes the scaling at runtime The case distinctions are translated to `if`-statements, and for each leaf we create code that uses the bounds in that leaf to compute the scaling factors.

The generalized polyhedron in figure 3.13 could for example be translated to program 4.

To compute the values of the $p_i^{\min}$, the virtual processor offset variables, we create code to transform back the bounds $v_i^{\min}$ by multiplying them with the matrix of the spanning vectors of the tile shape.

Remark 3.15 The `else` branch in the code above calls an `error` function with error text “unbounded”. This is a runtime check which should never be executed. An unbounded parallel target index space simply cannot be space-tiled in the way we do it (there exist different techniques for that case, for example **L**ocal **P**arallel **G**lobal **S**equential (LPGS) tiling [Wie97]). In the next

```
if (K2 < 0 && K1 < 0 && K1 < K2)
  {
    v1min = ceilDiv(n, 2*(K1 - K2));
    v1max = floorDiv(-n, 2*K2);
    v2min = 0;
    v2max = floorDiv(-n*K1, 2*(K2 - K1));
  }
else if (K2 > 0 && K1 > 0 && K1 > K2)
  {
    v1min = ceilDiv(-n, 2*K2);
    v1max = floorDiv(n, 2*(K1 - K2));
    v2min = 0;
    v2max = floorDiv(n*K1, 2*(K1 - K2));
  }
else error("unbounded");

H1 = ceilDiv(v1max - v1min + 1, NP1);
H2 = ceilDiv(v2max - v2min + 1, NP2);
```

Program 4: Code that computes the scaling at runtime

chapter, the algorithm will be used in a context where an unbounded parallel index space cannot occur. Since that will be our main application, we do not discuss unbounded parallel index spaces any further.

Chapter 4

Adaptive Tiling

In this chapter we introduce the methods we have developed for adaptive tiling. After a motivational example we give an overview about what we want to achieve, and why we chose this approach. We talk about the design space, i.e. the many choices we face, and about the tools required for our methods. Then we give an in-depth description of the adaptive tiling methods. The chapter closes with a discussion of these methods.

4.1 Motivation

In all methods known to us tiles of the same shape, form, and size are used for the whole target index space. Figure 4.1 shows that this can lead to poor load balancing between physical processors. The circles represent the operations in the target index space, the black circles being operations executed by the physical processor P_2 and the dashed circles being operations executed by the physical processor P_1.

The arrows between operations represent instances of two uniform dependencies. The bold horizontal line is the border between the physical processors P_1 and P_2: the logical processors 0 to 2 are mapped to P_1, the logical processors 3 to 5 are mapped to P_2. This mapping corresponds to a one-dimensional tiling of the space dimension p with tile shape $0 \leq o_1 < 3$.

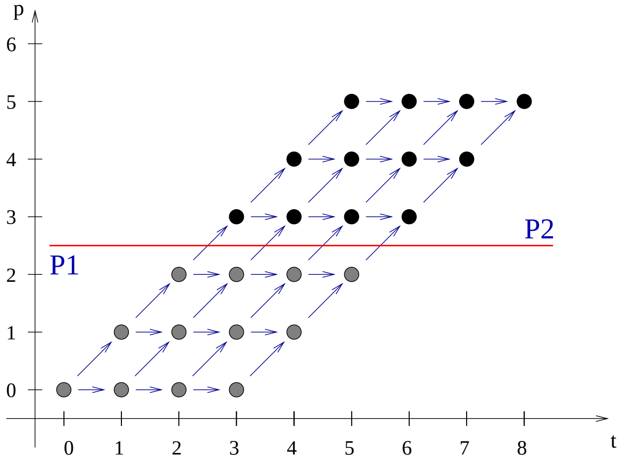


Figure 4.1: Poor load balance

We assume that the processors have the same speed and that each operation needs the same time to compute. Although both processors have the same total amount of operations, there is only one point in logical time at which the load is evenly balanced.

Ignoring all communication costs, we can do a basic estimation of computation costs: at each logical time step we count the *maximum* number of operations assigned to each physical processor. Then we sum up these counts. This gives us a rough relative estimate of the total *computation* time (for a more realistic estimate of total *execution* time, we would at least have to consider communications).

The result is:

time	0	1	2	3	4	5	6	7	8	Σ
max	1	2	3	3	2	3	3	2	1	20

Given that the total count of operations is 24, we can give an upper bound for the speedup and efficiency of the parallel program. Let t_{op} be the computation time of one operation. Then t_{seq}, the total time of the sequential program, is given by $t_{seq} = 24 \cdot t_{op}$ while t_{par}, the total time for the parallel program, is at least $20 \cdot t_{op}$. The speedup s is given by $s = \frac{t_{seq}}{t_{par}}$, leading to

$$s = \frac{t_{seq}}{t_{par}} \leq \frac{24 \cdot t_{op}}{20 \cdot t_{op}} = 1.2.$$

It follows that the efficiency is at most 0.6. While the estimate of the computation time was rather rough, this is a hard upper bound for the speedup we can achieve by this tiling.

Figure 4.2 shows a possible alternative. This time we don't have a static tiling. Instead, the tiling is adapted after each logical time step.

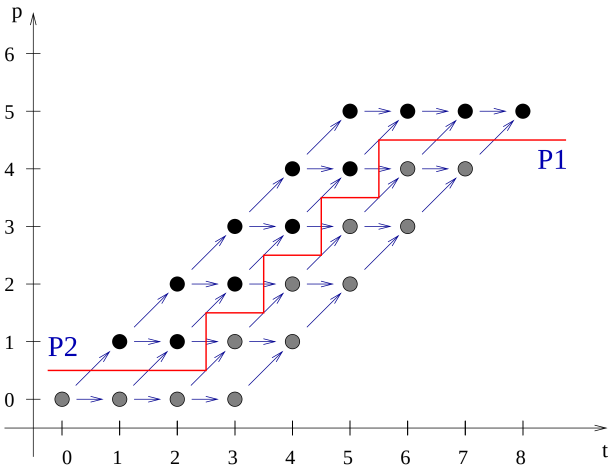


Figure 4.2: Adaptive load balance

If we do the same estimation of computation cost as before, again ignoring all communications, we get:

time	0	1	2	3	4	5	6	7	8	Σ
max	1	1	2	2	2	2	2	1	1	14

So, we only have 14 times the cost of a single operation, as compared to 20 times with the fixed mapping.

Since the adaptation of the mapping introduces additional communications, which we completely ignored with our simplistic estimation, these results can not be related to the execution speed of the two programs. But they give us the motivation to further investigate the possibilities of such an adaptive mapping.

So the question is: can tiles of different sizes be used to lower the total execution time of a parallel program? We suggest that the answer is yes, depending on the properties of the program we want to parallelize.

Remark 4.1 At first sight it might seem unwise to adapt the space tiling in almost every logical time step, as done in figure 4.2, because this way we disregard the allocation. The allocation decides which operations should be placed on the same logical processor based on the dependencies of the program.

But at closer inspection, we see that we are lucky with our example. Roughly speaking, only dependency instances that cross a physical processor border (the bold line) add to the communications, since data has to be transferred between physical processors.

We can see in figure 4.2 that between (logical) time 2 and 6 only the horizontal dependency causes communication because the adaptation "moves" into the same direction as the diagonal dependency. So we have 8 communications in total as compared to the 4 communications in the program with static tiling (figure 4.1).

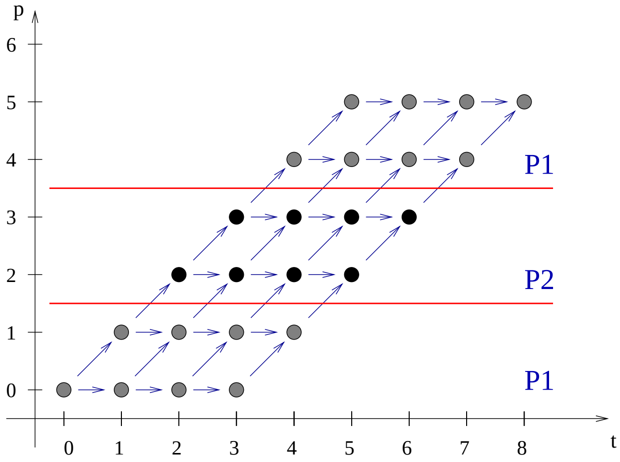


Figure 4.3: block-cyclic load balance

Remark 4.2 Since our goal is to improve the load balance one might ask if this can be accomplished by different techniques. A possible alternative that comes to mind is a block-cyclic distribution of the operations to processors. Figure 4.3 shows a block-cyclic distribution with a block size of two.

Again, we count the maximum number of computations in each time step:

time	0	1	2	3	4	5	6	7	8	Σ
max	1	2	2	2	2	2	2	2	1	16

This is worse than with adaptive tiling, and we also have a total of 8 communications. With a fully cyclic distribution (block-size 1), the load balance is as good as with our adaptive space tiling, but results in 20 communications.

Recapitulating, we see that adapting the space tiling can improve the load balance while increasing the number of communications, and that it can be compared to complementary techniques like block-cyclic distribution. We examine the trade-offs that arise due to both increased code complexity and increased communication cost.

4.2 Stepping Back: The Overall Picture

Before diving into the details of the problem it is worthwhile to look at adaptive tiling from a more abstract perspective for a moment.

The question we should ask ourselves first is: what do we actually want to optimize? The answer, of course is: the execution time of a parallel program on a given parallel machine.

As discussed in section 2.1.1 we need a cost model to estimate the execution time of parallel programs. Suitable cost models are discussed by Ellmenreich [Ellm04] and Griebl [Gri04]. They assign different costs for computations, the startup of a communication, and the volume of communications.

So we are looking for a parallel program that minimizes the cost function. The search space is the set of all parallel programs that are semantically equivalent to our sequential source program.

This is a huge search space. It is obvious that we have to restrict the search space, if we want to come up with a practical algorithm that finds a "good", or even a "best" solution (in a reasonable amount of time). So first, we restrict ourselves to programs that can be expressed inside the polyhedron model, i.e. our search space are the parallel programs that can be obtained by a space-time-mapping in combination with adaptive tiling.

Additionally we do not optimize the real runtime on a parallel machine, but we instead try to optimize several quantities of the parallel program like the latency of the schedule, the number of communication startups, the ratio of the volume of communications versus the number of communication startups, and the load balance.

The aspect we are most interested in is the load balance, because we want to mitigate the negative effects of a static space tiling with adaptive tiling. This involves a trade-off. We trade an improved load balance for a higher communication cost, because an adaptation of the space tiling means a change of the mapping of virtual processors to physical processors, and that will introduce more communications in the general case.

We can look at this trade-off from a different perspective. The static tiling methods for space and time developed by Griebl [Gri04] are based on the assumption that it is sound to treat space- and time-tiling separately (cf. section 3.4). This means that the facets of the space-tiles are orthogonal to the time dimensions. Adaptive tiling can be seen as a violation of this property. By changing the space tiles over time, we approximate tiles that have facets in space *and* time.

Consider the simple example in figure 4.2. The adaptive tiling in this figure is an approximation of the tiling in figure 4.4, which is in both space and time.

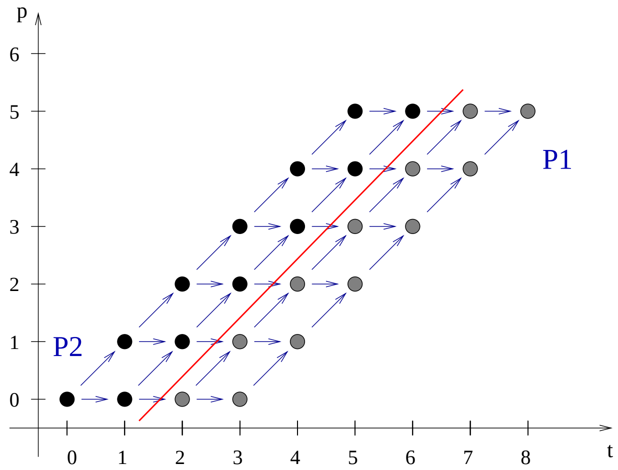


Figure 4.4: a tiling in space and time

By choosing how often we adapt the space tiling we can control by how much we violate the separation of space and time. This way we can balance the trade-off between the increase in communications and the improved load balance.

4.2.1 Input to the Adaptive Tiling Methods

As discussed in section 3.3, we decided to apply our tiling methods after the parallelization phase. In the framework of the polytope model, we start after dependence analysis, scheduler and allocator have finished. Thus we assume that the following data has been computed as input to our tiling algorithms:

- A space-time mapping T_i for every statement S_i. For an easier discussion, we assume that every schedule and every allocation has the same dimensionality.
- A target index space D_i for every statement S_i. Again, every index space has the same dimensionality.
- A set of dependencies h_j, encoded as h-transformations.
- A tile shape S for the space tiles, given as the inequality system.
- A lattice matrix L (usually the matrix of the spanning vectors).

We further assume that the target architecture is a n-dimensional processor grid where n is the dimensionality of the allocations.

4.2.2 The Design Space

In this section we enumerate all design choices we encountered when investigating methods for adaptive tiling. Although they are not fully independent, many combinations of different choices seem possible. Our methods investigate several choices that seem reasonable with respect to our goal of mitigating the problems of static tiling.

We list the most important design choices, which will be answered differently by our methods. We omit some choices that the polyhedron model leaves open, like which scheduler and allocator to use, because we regard schedule and allocation as input to our methods.

- Communications: Do we want ownership driven or dependency driven communications?
- Mapping to processors: How are space tiles mapped to physical processors?
- Scaling the tile shape: Do we want to preserve the tile form, or do we want to maximize the load balance?
- Granularity: How often do we want to adapt the tiling?
- Time tiling: How do we include time tiling?

4.2.3 Required Tool Support

Before we look at the methods we developed for adaptive tiling we want to examine which tools are required to incorporate these methods into LooPo, a prototype of a parallelizing compiler developed at the university of Passau [LooPo, GrLe96].

Some of these tools are not available yet, but there is ongoing research in the respective areas.

1. We assume that the source index spaces, the dependencies, and a space-time mapping have already been computed. LooPo does this for us.
2. We need data structures and algorithms to deal with parameters in linear inequalities. The most important are:
   - data structures for parametrized inequalities, decision trees and quantifier free formulae and
   - an algorithm to sort parametrized inequalities.

   This has been implemented by Größlinger in Haskell [Grö03].
3. We need a suitable method for code generation, that takes the generalized polyhedra our methods produce and creates the target program.

   The features we need in a code generation tool are the following:
   - it must support parametrized inequality systems
   - it must accept a set of generalized polyhedra for each statement

> We know of no tool with these features. CLooG [CLooG, Bas02] can not deal with parameters. LoopGen [LoopGen, QRW00], a predecessor of CLooG, is currently extended to deal with parameters [Amo04].
>
> The lack of such a tool prevents the fully automatic implementation of the adaptive tiling methods. It also has made the empirical evaluation more difficult, because the code generation for the examples had to be done by hand.

4.3 Deciding when to Adapt the Tiling

We use the term **re-tiling** as a synonym for the adaptation of the space tiling.

If there is only one sequential dimension, it is obvious that we do the re-tiling in that dimension. But if there are several sequential dimensions we have to determine in which dimension the re-tiling takes place.

It turns out that it is not sensible to do re-tiling in more than one sequential dimension. We assume we have several sequential dimensions $t_1, \ldots, t_k$, $k > 1$, that we do re-tiling in two dimensions t_i and t_j, $i < j$, and that the sequential dimensions are enumerated by `for` loops in the order $1, \ldots, k$. This means that the j-loop is an inner loop of the i-loop. Conceptually the sequential loops can all be enumerated by a single `for` loop (the time is sequential after all). It follows that between two consecutive re-tilings of the outer i-loop, there can be re-tilings of the inner j-loop. This is wrong because we assume a static space-tiling between two consecutive re-tilings.

This result simplifies re-tiling, because we now know that we only have to choose one sequential dimension.

In this section we use the terms **global time** and **local time** with the following meaning: We "cut" one time dimension into equally sized intervals (ignoring boundary cases for now). The width of each interval is called **re-tiling width** B.

The means by which we achieve this "cutting" is by dividing a sequential time dimension, called t here, into one global component t^g which tells us which interval we currently are in, and into one local component t^l which tells us where we are inside the global interval. Written as an equation we get

$$t = B \cdot t^g + t^l$$

(additional constraints for t^g and t^l omitted).

4.4 Terms and Definitions

As already defined in section 2.4.3 we use the term parallel index space as a synonym for the parallel dimensions of the target index space of a statement.

The equation

$$t = B \cdot t^g + t^l$$

from section 4.3 together with the inequalities

$$0 \leq t^g$$
$$0 \leq t^l \leq B - 1$$

is called the **re-tiling equation**. It divides one sequential dimension t into a global component t^g and a local component t^l.

Finally, the equation

$$p = L_H \cdot r + o + p^{\min},$$

where $p \in \mathbb{Z}^d$ is the virtual processor vector, $r \in \mathbb{Z}^d$ is tile index, $L_H \in \mathbb{Z}^{d \times d}$ is the scaled lattice matrix, $o \in \mathbb{Z}^d$ is the offset inside the tile, and $p^{\min} \in \mathbb{Z}^d$ is the virtual processor offset (cf. section 3.7.2) is called the **space-tiling equation**.

4.5 Communication

Communications are a crucial point of parallelization in general, and adaptive space tiling in particular. We already emphasized that we try to find a good trade-off between load balance and communication costs. Here we investigate how dependency driven communication can be brought together with adaptive tiling. We propose a communication library that can be implemented on top of MPI [MPI] which hides the extra complexity introduced by the adaptation of the space tiling.

When combining dependency driven communication with adaptive space-tiling, we face a fundamental problem. The mapping of virtual to physical processors changes at every re-tiling step. Dependencies that "cross" one or more re-tiling steps are problematic, because the sending processor must know which processor receives the data, but the receiving processor depends on the space tiling at the time of the target of the dependency.

Depending on the communication pattern of the input program there are at least two different possible solutions. If all dependencies are short (in logical time), which means that they are shorter than the re-tiling width, we could just precompute one space-tiling in advance. We propose to store three space-tilings at any given time: the previous one, the current one and the next one. At every re-tiling, we just compute the next space tiling and update the stored space-tilings. This algorithm can be generalized to longer dependencies, by storing more than three space-tilings.

However, if we have very long dependencies, this algorithm needs too much storage: if there were dependencies that go from the start of the logical time until the end, we would have to store all mappings. In that case, we propose to store the data to be sent instead of sending it at once. With the data, we store the virtual processor that is the target of the communication. The actual communication is then delayed until the re-tiling step that contains the time of the target of the dependency, when we know how to map the virtual target processor to a physical processor. This approach avoids the use of additional memory (the communications have to be stored anyway, even if it is implicitly in the communication network), at the cost of an increased runtime overhead at each re-tiling step. Another potential disadvantage is that because we delay the sending of messages until we know the mapping, the time between the actual sending and the receiving of messages gets shorter, which can lead to longer delays in blocking receives.

In the next section we propose to integrate the second mechanism for handling dependency driven communication into a communication library, and give some hints towards the implementation. This way, the generated target code is reduced in complexity because the communication handling is done automatically by a library.

4.5.1 Communication Library

Here we propose a simple communication library based on the experiences of Ellmenreich [Ellm04] and Claßen [Cla04].

Instead of sending and receiving individual messages, we propose to receive all required data at the beginning of a time step (or at the border of a time tile, if time tiling is used), and to send all data that is needed by other processors at the end of one time step.

This way, message coalescing can be automatically implemented (i.e. all data that is sent to another processor can be sent in one message instead of sending several messages with parts of the data), and the communication pattern is easier to understand.

The communication library also allows to hide the handling of dependency driven communication. The library can be made responsible for storing all send request until the true physical processor is known, without complicating the generated code.

Another advantage is automatic handling of local communications. We saw that it is difficult to avoid local communications, because it involves the set difference of polyhedra (cf. section 2.5). In the case of dependency driven communication, it is even impossible to avoid local communications, because the mapping at the target time of the communication is unknown. So we simply allow local communications and sort them out when sending and receiving.

The general idea for the implementation of the library is to replace the MPI commands for sending and receiving with our own primitives. Instead of actually sending (or receiving), the messages are gathered into buffers until a commit command is issued.

The implementation of these concepts is straightforward. It can, for example, be done with the help of the `MPI_Alltoall` MPI [MPI] command. For details we cite the work of Ellmenreich [Ellm04] and Claßen [Cla04], and just give a possible interface of the library in C-notation.

To deal with dependency driven communication the library is aware of the virtual time, virtual processors, and the current mapping from virtual processors to physical processors. For the presentation we concentrate on the sender side. We assume that suitable types `Matrix` and `Vector` are defined, and that the resolution of the `int`-type is big enough to represent all possible values for the virtual time.

```
// Tell the communication library which mapping should be used
// (called once at the start of each re-tiling step).
// This also sends out pending messages which have the
// receiver_time in [t_global, t_global + B - 1].
void set_current_mapping(Matrix L_H, Vector p_min,
                         int t_global, int B);

// Sends a buffer from virtual processor sender to virtual
// processor receiver. The actual send is deferred until
// commit_send().
int send(void* data, int amount, MPI_DataType type,
         Vector sender, Vector receiver, Vector receiver_time);

// Sends all data that has been issued for send since the last
// call of commit_send().
int commit_send();
```

Program 5: Interface of the communication library (sender functions only)

The design of the communication library allows to forget the problems of a mutable mapping from virtual processors to physical processors. But it is an opportunity for future research to investigate if it would be worthwhile to use a different strategy in certain cases. For example it is very well possible that a more efficient way of communicating can be found for a restricted set of programs with short dependencies, because the physical processor of the receiver can be determined at compile-time. The advantage of our proposed design is that it nicely hides complexity, simplifying the communication code generation, and that it scales well with long dependencies.

Remark 4.3 This library can also be used for ownership-driven communication. In that case the `receiver_time` will always be inside the current time interval and all data can be sent at the next commit. Of course it is possible to create an optimized version of the library for that case.

We now present three different adaptive tiling methods. The first two use algorithm 1 from chapter 3 to compute an automatic space tiling repeatedly. They only differ in the way communications are generated. The third is completely different: it uses a non-parametric space tiling and maps the tiles cyclically to the processors.

4.6 Method 1: Adaptive Space Tiling with Ownership Driven Communications

Basic Idea and Outline The basic idea behind this method is quite straightforward: At certain points in time we adapt the space-tiling in order to give each physical processor roughly equally-sized chunks of the index space (see figure 4.2).

First we choose the time dimension that should be re-tiled and the re-tiling width. This allows to control how "adaptive" we want to be. Since every re-tiling can cause communications, but fewer re-tilings can lead to more load imbalance, there is a trade-off to be made.

As the space tiles can be expressed with parametrized polytopes we use the algorithms developed by Armin Größlinger [Grö03], to create the loop bounds that enumerate the time and space dimensions.

Next are the communications. We chose ownership driven communications for this approach (for a discussion of the choices see section 4.5). Because of the adaptation of the space tiling, we face the difficulty that the data allocation changes during runtime. We show how to compute the necessary redistribution polytope.

In a last step, we combine the generalized polytopes for communications and computations to produce the target code.

We first describe these steps in detail. After that we discuss how much of this can already be done fully automatically, and where the current tools are not powerful enough.

We summarize the design decisions for this method (based on section 4.2.2) in the following table:

Communications	ownership driven
Mapping	tile coordinates
Scaling	automatic, independent along each spanning vector
Granularity	parameter B with default value N^P
Time Tiling	none

We describe only the core tasks of our method: choice of granularity, creation of the redistribution polyhedron, computation of the scaling factors, and generation of the inequality system for computations and communications. We omit the actual code generation, which is dealt with by external tools.

The template for the code we generate looks roughly like the code snippet in program 6 (where we assume a single time dimension t, split into global time t_g and local time t_l for simplicity of presentation).

As you can see, we ignore time tiling for now. The parts in angle brackets represent the core tasks enumerated above. At the beginning, we compute an initial scaling of for the space tiles. At every local time step, we first receive all data we need from different processors. Then we perform all computations for this time step, and send the computed values to the owning processors. After all logical time steps of one global time step have been processed, we compute a new scaling for the next global time step, and we redistribute all data that changes the logical processor.

In the following sections we describe how the parts in angle brackets can be generated.

```
<COMPUTE_INITIAL_SCALING>
for t_g := lower_bound_t_g to upper_bound_t_g
  for t_l := lower_bound_t_l to upper_bound_t_l
    <COMM_RECV>

    <ENUMERATE_SPACE_TILES>

    <COMM_SEND>
  end
  <COMPUTE_SCALING>
  <COMM_REDISTRIBUTION>
end
```

Program 6: Code Template for Method 1

4.6.1 Choosing the Granularity

As shown in section 4.3, we have to choose one time dimension, in which we do the re-tiling.

We propose to choose the innermost time dimension. Rationale: we try to improve the load balance, so we try to adapt the space tiling frequently enough to distribute the load roughly equally. The innermost time dimension allows the most fine-grained control.

Now that we know which time dimension to use, we still need to decide when to adapt. We introduce the re-tiling width in the form of the new parameter B, after which the tiling is adapted. This allows us to defer the choice of B until runtime. The default value for B will be the total number of processors, but it is easy to create code that allows users of the program to override the choice of B.

We use the total number of physical processors $\mathcal{NP}$ as a default value for B. Our experiments showed that for the used examples the additional communication cost of the adaptive tiling increases with the number of physical processors. So the idea is to re-tile less often if there are more physical processors to avoid excessive communication costs.

Remark 4.4 A more thorough investigation of the choice of the sequential dimension and the re-tiling width would be interesting. For example the dependency structure of the input program and the form of the target index space could be used for the decision. It is important to balance the granularity of the re-tiling with the additional communication costs.

4.6.2 Generation of the Communication Polyhedra

To make ownership driven communication work (cf. section 2.5.2), we have to perform a **redistribution** of the data at each re-tiling step because each re-tiling potentially changes the mapping of virtual processors to physical processors. And if the mapping changes, the physical processor that owns a data element changes, too. We first show in detail how to create the code for the redistribution and subsequently cover the actual communication generation.

To compute the redistribution we first compute the set of all virtual processors that change the physical processor during re-tiling.

The next step is to look at each array and compute the set of elements that must be moved to a different physical processor based on the data placement and the set of virtual processors from the first step.

We then create the corresponding send and receive statements using CLooG (for the sake of the presentation we assume that CLooG can handle parameters as coefficients, which is not yet the case).

In the following we identify tile-coordinates with physical processor coordinates.

Computing the virtual processors that change the physical processor We start with the polyhedron that describes our target index space D, with the current mapping given by the scaling matrix H and the virtual processor offset $p^{\min}$, and the mapping of the next re-tiling step given by the scaling matrix $\tilde{H}$ and the virtual processor offset $\tilde{p}^{\min}$.

First, we compute the parallel index space by projecting D to the space dimensions, creating the generalized polyhedron D_π. We introduce new variables $r, \tilde{r}, o, \tilde{o} \in \mathbb{Z}^d$ and we add two versions of the space-tiling equation, one for the current mapping:

$$p = L_H \cdot r + o + p^{\min},$$

and one for the next mapping:

$$p = L_{\tilde{H}} \cdot \tilde{r} + \tilde{o} + \tilde{p}^{\min}.$$

We call the resulting inequality system D'_π. The points of the polyhedron D'_π have the coordinates $(p, r, o, \tilde{r}, \tilde{o})$, and they satisfy the two space-tiling equations from above. D'_π contains the information which virtual processors change from processor r to processor $\tilde{r}$ during re-tiling for each pair of physical processors r and $\tilde{r}$.

Computing the array elements that must be redistributed Let A be an l-dimensional array, and let $\pi_A : \mathbb{R}^l \mapsto \mathbb{R}^d$ be its data placement. Then for a given pair of (distinct) physical processors (r_1, r_2) we have to send the array cells with index $i \in \mathbb{Z}^l$ when

1. there is a point in D'_π where the physical processor $r = r_1$, the new physical processor $\tilde{r} = r_2$, and p is the corresponding virtual processor, and
2. $\pi_A(i) = p$.

This means that for a given array A, processors (r_1, r_2) and virtual processor p that satisfy the above constraints, we have to send the set of array elements given by $\pi_A^{-1}(p)$.

Generating the code for the redistribution We create loops that enumerate for a given r all possible communication partners $\tilde{r}$ and send all data that is owned by virtual processors that change the mapping from r to $\tilde{r}$. This can be done using CLooG and the inequality system D'_π from above: we create loops that enumerate all changing virtual processors, and then send the relevant data for all existing arrays. We use CLooG to combine the communication statements for all statements. For the receiving side, this can be accomplished analogously by enumerating all communication partners r for a given $\tilde{r}$.

Generating the code for communication Because of the redistribution we can pretend that we have a static space tiling: at each step in logical time the data has been redistributed according to the current space-tiling and mapping. So we can compute the polytope that describes our communications like we did in section 2.5.2, with the difference that we add the space-tiling equation to our target index space D and group the sending and receiving code based on the processor coordinate.

4.6.3 Generation of the Code to Compute the Tile Sizes and Offsets

To adapt the tile sizes to the target index space we use a modified version of algorithm 1.

The modification is that we do not work on the parallel index space to determine the scaling and offset parameters. Instead we use a restricted form of the parallel index space. We add the re-tiling equation to the target index space and use a fixed value for the global time (and surrounding time dimensions). This fixed value is simply taken from the surrounding loops.

Next we project this restricted target index space to the parallel dimensions. We continue with the algorithm based on this index space.

This way, the calculated scaling offsets depend on the value of the global time and the surrounding time dimensions, which is just what we expected, because the tile size (and in our case also the tile form) depends on time.

Remark 4.5 As already noted in section 3.7.7 the restricted parallel index space we use is always bounded. If it were not there would be an infinite number of parallel tasks in a finite (virtual) time. This would mean non-termination of the program.

4.6.4 Generating the Code

What is left to do is to put the polyhedra describing the communications, the redistribution, and the computations together and produce code from them, following the template code in program 6. Therefore, a code generation tool that handles generalized polyhedra is needed.

4.6.5 Limits and Problems

Time Tiling The major limit of our method is that time tiling seems to be difficult to achieve. As shown by Griebl [Gri04], not every communication placement allows time tiling. He also shows that for a restricted class of placements, called **FCO** placements (**F**orward **C**ommunication **O**nly)), time tiling is always possible.

The adaptation of the space tiling effectively modifies the placement, however. And even if the original placement is FCO, the adaptation can introduce backward communications, which means that the FCO property is lost and time tiling cannot be shown to be applicable.

So we cannot apply global time tiling. One way out could be to apply time tiling only during each global time step (i.e. in between two adaptations of the space tiling). We call this local time tiling.

But local time tiling has two major disadvantages. First it greatly increases the latency and requires a very large re-tiling width to make the time tiling effective (Griebl [Gri04] gives an example program when coalescing up to 512 time steps gives best results). A large re-tiling width leads to a poorer load balance.

A second drawback is the additional control overhead of time-tiling. Adaptive tiling already considerably adds to the control overhead (e.g. the code to automatically compute a scaling at each re-tiling step). With time tiling the control overhead would become even worse.

In conclusion, time tiling and our first adaptive space tiling method do not fit together very well.

Redistribution and Communications The data redistribution at each time step can lead to a huge volume of communications. Even data that is not used for a long time is constantly shuffled around between processors because of the redistribution.

Missing tool support As discussed in section 4.2.3 we don't have tools that create code from generalized polyhedra. Although there is research in this direction this prevents a complete implementation of this method at the moment.

4.7 Method 2: Adaptive Tiling with Dependency Driven Communication

When looking at the problems of the first method, we see that redistribution causes many additional communications in some examples. Taking this into account, we propose a second method, which is conceptually nearly identical to the first one, but with the crucial difference that the communications are dependency driven rather than ownership driven.

We summarize the design decisions for this method (based on section 4.2.2) in the following table:

Communications	dependency driven
Mapping	tile coordinates
Scaling	automatic, independent along each spanning vector
Granularity	parameter B with default value $N\!P$
Time Tiling	none

The code template shown in program 7 looks similar to program 6, but the code for redistribution is missing.

```
<COMPUTE_INITIAL_SCALING>
for t_g := lower_bound_t_g to upper_bound_t_g
  for t_l := lower_bound_t_l to upper_bound_t_l
    <COMM_RECV>

    <ENUMERATE_SPACE_TILES>

    <COMM_SEND>
  end
  <COMPUTE_SCALING>
end
```

Program 7: Code Template for Method 2

Since the problem of choosing the granularity, computing the scaling, and of generating the code for the computations is identical to the first method, we will concentrate on the communication code.

4.7.1 Generation of the Communication Polyhedra

As explained in section 2.5.3, dependency driven communication uses the h-transformations to create the communication code.

Our input are the transformed h-transformations $h'_1 = (f'_{h_1}, D'_{h_1}), \ldots, h'_r = (f'_{h_r}, D'_{h_r})$. For $1 \leq i \leq r$, f'_{h_i} is the function that maps the target of a dependency to the source and D'_{h_i} is the domain of this function, both defined in the target coordinate system of the space-time transformation.

For each h'_i we create a communication polyhedron C_i. C_i is constructed in the following way:

1. start with D'_{h_i},
2. duplicate the variables t and p to t^s, t^r and p^s, p^r for the sender and the receiver,
3. add the equation $(t^s, p^s) = f'_{h_i}(t^r, p^r)$.

This is done exactly as described in section 2.5.3.

Based on the polyhedra C_i, which describe all communications, we now can create the code for sending and receiving.

Remark 4.6 To create the communication code we used a new parameter PROC. This is the physical processor index of the processor that is currently executing. Since libraries as MPI use a one-dimensional processor index, a suitable mapping from these indices to a d-dimensional processor index has to be used here. This mapping could be encapsulated in the communication library.

First we cover the receiving-code. We add the space-tiling equation for the receiver processor to each C_i :

$$p^r = L_H \cdot r + o + p^{\min}.$$

Then we set the receiver time to the current time (as given by the sequential loops) by adding the equation $t^r = t$ to each C_i. Equally we set the tile index to the current physical processor by adding the equation $r = \mathrm{PROC}$. Then we sort the C_i in the order (t^r, r, o, p^s, t^s). Since t^r and r are fixed we only need to create loops for the inner variables. Note that at this place we need a tool to generate code from a set of parametrized polyhedra.

Analogously, for the sending-code we add the space-tiling equation for the sender processor, set the sender time to the current time and set r to the sending processor.

Note that the communication library deals with the local communications, and with the mapping of logical processors to physical processors automatically. This makes communication code generation relatively simple.

4.7.2 Limits and Problems

Time tiling is as difficult as with method 1, because communications that cross re-tiling steps can be backwards, even if the placement is FCO.

Another problem is the increased complexity needed to handle the dynamic mapping of virtual processors to physical processors, which is hidden in the communication library.

Finally as in the first method we are lacking tool support for code generation.

4.8 Method 3: Block-cyclic Non-parametric Space Tiling

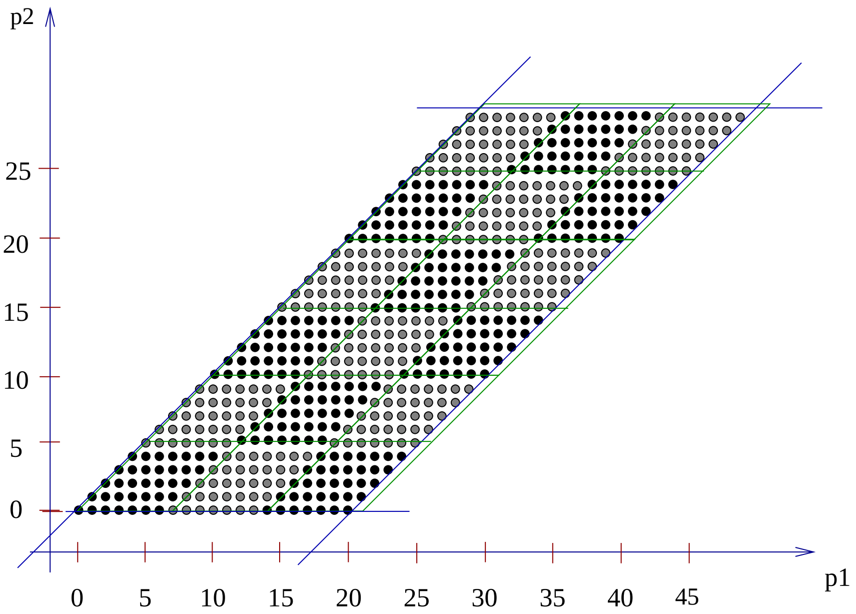
Because of the difficulties we encountered with time tiling and the missing tool support our third and last method uses a radically different approach.

Instead of computing a scaling that tries to cover the parallel index space with a minimal number of tiles, we look for a static space and time-tiling (using existing techniques as described by Griebl [Gri04]), that leads to a number of tiles that is greater than the number of physical processors. This tiling can be done without parameters: we omit the parametric scaling of the tile shape and instead choose a constant scaling. Additionally it is possible to apply existing time tiling techniques as Griebl presents them [Gri04].

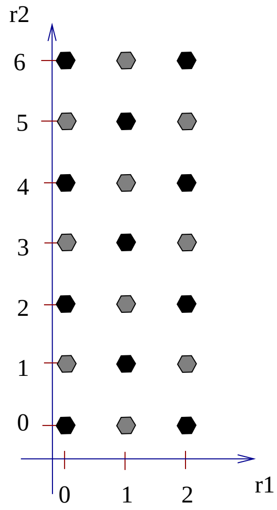
In a second step, we cyclically map these tiles to processors, resulting in a block-cyclic distribution.

We chose ownership driven communication, but it is also possible to modify this method to use dependency driven communication.

Figure 4.5 illustrates the approach. First the operations are aggregated into tiles (here only in space, but possibly also in time) and then these tiles are cyclically distributed along all physical processors.



(a) Parallel index space and space tiles



(b) Tile index space

Figure 4.5: Block-cyclic Tiling

The following table summarizes the design decisions.

Communications	ownership driven
Mapping	statically/cyclically
Scaling	constant
Granularity	depends on static tiling
Time Tiling	regular static time tiling

We omit the details on how to create the code for space-tiling and time-tiling using a constant scaling. Since the space-tiling equation does not contain parameters in this case this is partly covered by the work of Claßen [Cla04] who implemented code generation with space-tiling and time-tiling. So far communication code generation is not implemented yet, but there is ongoing research to implement it. In the following we assume that the constant space-tiling and time-tiling has already been performed.

We now discuss the modifications necessary to distribute the tiles block-cyclically. Claßen uses a sequential mapping from space-tiles to processors (cf. section 3.7.2). We modify this and use a mapping based on the tile-index instead. Since we did not do any scaling, there is no guarantee that the tile-indices are in the range $\prod_{i=1}^{d}\{0, ..., \mathcal{NP}_i - 1\}$ (where d is the dimension of the physical processor array). We make a simple modification: we take the absolute value in each dimension and then compute the remainder of dividing this value through the number of processors in that dimensions. Our new mapping function $m : \mathbb{Z}^d \mapsto \mathbb{Z}^d$ is hence defined by $m(r_1, \ldots, r_d) = (|r_1| \mod \mathcal{NP}_1, \ldots, |r_d| \mod \mathcal{NP}_d)$.

All other parts of the work of Claßen can be used without modification. It is easy to check that this mapping results in a block-cyclic distribution.

4.8.1 Limits and Problems

A problem of this method is that we must choose at compile-time a constant scaling for the tile-shape. It is difficult to find a constant scaling that works equally well for all possible combinations of structure parameters. If the scaled tile-shape is too large for example the load balance might be poor for a large number of physical processors. This means that the granularity of the load-balancing cannot be adapted very well.

Another problem are the additional communications. Every time the cyclic mapping wraps around in one dimension there are communications between the first and the last physical processor in that dimension. If the network topology of the target architecture is a grid and not a cylinder or a torus then these communications are very expensive.

4.9 Discussion

In this section we discuss our results on adaptive tiling. After a comparison of the three methods we consider the question of the applicability of adaptive tiling. Finally we list the topics we encountered that might justify further research.

4.9.1 Comparison

We have seen how several shortcomings motivated the modification of method 1. First, the poor communication behavior of ownership driven communications motivated dependency driven communications in method 2. The difficulties with time tiling and the missing tool support motivated the different approach of method 3.

But although each successive method tried to fix or mitigate problems of its successor, there is no "optimal" method. The biggest drawback of the first two methods that use automatic scaling of the space-tiling is the lack of a code generation tool. As soon as a suitable tool is developed it will be possible to investigate in detail for which input programs these methods perform best.

The third method can be implemented with existing techniques. Since it allows time tiling there probably are examples which benefit from this method.

Remark 4.7 All three methods also work with `while` loops. A problem of `while` loops is that we cannot know (in general) when they will terminate. This implies that tiling algorithms have to deal with an unbounded virtual time. Projecting to the space dimensions results in an unbounded parallel index space. For the first two methods target index space is not required to be bounded. The only requirement is that it is bounded when restricting it to one re-tiling interval. This is clearly true (otherwise the parallel program could not terminate). The third method makes no assumptions about the bounds of the parallel index space. By cyclically mapping the tiles all problems are avoided.

4.9.2 Applicability

Subsequently we discuss criteria to decide in which cases adaptive tiling is likely to improve the parallel program and which method is suitable for which kind of sequential program.

There are three cases in which adaptive tiling potentially leads to improved execution time:

1. The number of parallel tasks changes significantly as a function of time. By adaptively scaling or mapping cyclically a reasonable load balance can be achieved.

2. The "position" of the target index space changes significantly as a function of time, as shown in figure 4.2. With methods 1 and 2 the virtual processor offsets are computed at every re-tiling step. With method 3 the cyclical mapping makes sure the load balance is improved.

3. The sequential loop nest contains `while` loops. Since the parallel index space is unbounded a simple static tiling is not applicable.

Ideally, a parallelizing compiler should automatically check these criteria and choose an adaptive tiling method if appropriate. The first two criteria are also related to the choice of the granularity of re-tiling (cf. section 4.6.1). The more the number of parallel tasks or the position of the target index space changes as a function of time, the smaller should the re-tiling width be chosen in order to achieve good load balance. The development of automatic tests for criteria 1 and 2 requires further research.

Assuming the parallelizing compiler decided to apply adaptive tiling, how can it decide which of the three adaptive tiling methods is suited best for a given input program? We propose the following decision algorithm:

1. Does the program benefit a lot from time tiling? → Method 3
2. Are there many long dependencies (in virtual time) in the target index space? → Method 1
3. In all other cases → Method 2

We put the decision based on time tiling on top because of the impact that time-tiling can have on the performance of certain programs [Gri04]. Of course this algorithm must be empirically validated and refined before it can be used in practice.

4.9.3 Open Ends

During our research we discovered several open questions which justify further research. We list them here and give references to the respective chapters.

- Finding good lattice matrices. We discussed this problem and a possible approach on assessing lattice matrices in section 3.7.1.
- Modifying the scaling algorithm for arbitrary lattice matrices. Algorithm 1 only works when the lattice matrix is the matrix of the spanning vectors of the tile shape. In section 3.7.2 we discuss the problems that can occur when using different lattice matrices.
- Studying the impact of scaling along each spanning vector versus preserving the tile form (cf. section 3.7.4).
- A more efficient communication library. The library we proposed in section 4.5.1 should be empirically evaluated. In particular, the handling of dependency driven communication has potential for improvements by handling special cases (e.g. uniform dependencies) differently.
- Choosing the time dimension to re-tile and choosing the re-tiling width. To solve this problem properly we need a good cost model, also taking load balance into account, and we need to closely investigate the "shape" of the target index space.
- Automatic tests for applicability of adaptive tiling. This is related to the previous point. We proposed criteria in section 4.9.2.
- Decision algorithm to decide which adaptive tiling method is suited best for a given input program. We briefly discussed this question in section 4.9.2.

Chapter 5

Empirical Results

In this chapter we present a small empirical study of the adaptive tiling methods. After presenting the used programs we present the measured times and speedups. We close with an interpretation of these measurements. Because we do not have a tool for automatic code generation we have done the code generation by hand. Even for simple examples this can be really messy. For this reason we only used very simple examples.

5.1 The Examples

We chose two examples to test our developed methods, and to compare them with existing methods.

We tried to cover typical scenarios while keeping the problems simple enough to implement different parallel versions in the given time.

Both examples are two-dimensional and their transformed index space is not rectangular, because otherwise static tiling methods already achieve good load balance.

5.1.1 Example 1

This example is based on a one dimensional SOR kernel.

```
for i := 1 to N-1
  for j := 1 to N-1
    A[i, j] := A[i-1, j] + A[i, j-1]
  end
end
```

Program 8: Example 1

It has only uniform dependencies of length one.

5.1.2 Example 2

The second example is the backward substitution phase after a LU decomposition.

```
for k := 0 to N-1
  sum[n-k-1] := b[n-k-1]
  for l := 0 to k-1
    sum[n-k-1] = sum[n-k-1] - A[N-k-1, N-l-1] * b[n-l-1]
  end
  b[n-k-1] = sum[n-k-1] / A[n-k-1, N-k-1]
end
```

Program 9: Example 2

The dependency structure is quite different since there are many non-uniform dependencies.

5.2 Test Environment

The programs were tested on Hermes [Hermes], the Fujitsu/Siemens hpcLine cluster of the chair for programming in Passau. It consists of 32 dual-processor nodes. Each node is equipped with 512 MB of ram, the processor model used is a 1 GHz Pentium III (coppermine). The nodes are connected with SCI [SCI] as a 2D torus. The operating system is GNU/Linux [GNU, Linux] with kernel 2.4.25. The used MPI [MPI] library is ScaMPI [ScaMPI].

We implemented three versions of each program in C: a sequential version, a statically tiled version, and an adaptively tiled version.

To measure the times precisely we implemented a small experimentation frame in C. This frame first initializes the MPI library and performs a barrier synchronization to make sure all processors are ready. Before the actual algorithm is executed 10 times the first processor reads the current system time. Then, after another barrier synchronization, the first processor reads the current system time again and computes the average elapsed time per run. This frame program was run from a shell script that called it for the desired combinations of parameters. The resulting output was converted to LaTeX tables using a python script.

We compiled the test programs with gcc 2.95, compiler flags "`-O3 -ffast-math -DNDEBUG`".

The speedup sp was calculated by the formula

$$\mathrm{sp} \stackrel{\mathrm{def}}{=} \frac{t_{\mathrm{seq}}}{t_{\mathrm{par}}},$$

where t_{seq} is the time of the sequential program and t_{par} is the time of the parallel program. The efficiency was calculated by the formula

$$\mathrm{eff} \stackrel{\mathrm{def}}{=} \frac{\mathrm{sp}}{\mathcal{NP}},$$

where $\mathcal{NP}$ is the total number of processors used.

5.3 Results

5.3.1 Example 1

Our implementation of adaptive space-tiling uses method 2: dependency driven communications and mapping based on tile indices. The re-tiling width B is $N\!P$. The results are displayed in table 5.1.

5.3.2 Example 2

As in example 1 we use method 2 and re-tiling width $B = N\!P$. The results are displayed in table 5.2.

5.4 Discussion

For example 1 we can see that for large values of N ($N >= 4096$), we get a better efficiency with adaptive tiling if the number of processors is small (2, 4, or 8). For smaller N, adaptive tiling suffers from an increased constant control overhead (this can also be seen for $N\!P = 1$). For larger numbers of processors adaptive tiling suffers from the increased communications. For $N = 6144$ and 2 physical processors, the efficiency of adaptive tiling is more than 50% higher than the efficiency of static tiling.

Example 2 shows that adaptive tiling can actually perform worse than static tiling. It seems that the non-uniform dependencies cause too many additional communications when doing adaptive tiling.

As we already mentioned the code for example 1 and example 2 was created by hand (but with the help of the hsloopo tools that deal with non-linear inequality systems). Since the communication code in the adaptively tiled programs is complex we did not implement message coalescing and did not perform all possible optimizations. We think that it is possible to improve the performance of these programs.

To sum up, he data we have gathered seems to lead the following conclusions regarding adaptive tiling with automatic scaling:

- Adaptive tiling is best used for small numbers of physical processors. Since the current trend in computer-architecture is towards multi-core processor dies we expect that soon most desktop computers will have more than one CPU. Since these machines will have cheap communications (the processors have shared memory and sometimes even shared caches), adaptive tiling seems well-suited for them.
- Programs with many non-uniform dependencies can perform worse with adaptive tiling.

As soon as automatic code generation for parametrized polyhedra is possible we see several areas for further experimental studies:

- How does block-cyclic adaptive tiling perform in example 1 and example 2?
- What is the influence of the re-tiling width B on the performance of the adaptively tiled programs?
- How does adaptive tiling perform on more difficult programs?
- Can a cost-model be found to predict whether and by how much adaptive tiling improves performance?

N=1024

	static			adaptive		
NP	**time[s]**	**speedup**	**efficiency**	**time[s]**	**speedup**	**efficiency**
1	0.1250	0.6931	**0.6931**	0.1347	0.6430	**0.6430**
2	0.1155	0.7498	0.3749	0.1189	**0.7284**	0.3642
4	0.1103	**0.7853**	0.1963	0.1452	0.5963	0.1491
8	0.1339	0.6469	0.0809	0.3038	0.2851	0.0356
16	0.2203	0.3930	0.0246	0.6396	0.1354	0.0085
32	0.3909	0.2215	0.0069	1.3527	0.0640	0.0020

N=2048

	static			adaptive		
NP	**time[s]**	**speedup**	**efficiency**	**time[s]**	**speedup**	**efficiency**
1	0.7206	0.6642	**0.6642**	0.7688	0.6225	**0.6225**
2	0.5451	0.8779	0.4390	0.4740	1.0098	0.5049
4	0.3634	1.3171	0.3293	0.3740	**1.2798**	0.3200
8	0.2822	**1.6959**	0.2120	0.4573	1.0465	0.1308
16	0.3161	1.5139	0.0946	0.7669	0.6241	0.0390
32	0.4598	1.0409	0.0325	1.4539	0.3292	0.0103

N=4096

	static			adaptive		
NP	**time[s]**	**speedup**	**efficiency**	**time[s]**	**speedup**	**efficiency**
1	4.7555	0.4562	**0.4562**	4.8881	0.4438	0.4438
2	2.9499	0.7354	0.3677	2.1029	1.0316	**0.5158**
4	1.5286	1.4191	0.3548	1.2398	1.7497	0.4374
8	0.8815	2.4610	0.3076	0.9690	**2.2388**	0.2799
16	0.6518	**3.3280**	0.2080	1.0946	1.9818	0.1239
32	0.6571	3.3015	0.1032	1.7368	1.2490	0.0390

N=6144

	static			adaptive		
NP	**time[s]**	**speedup**	**efficiency**	**time[s]**	**speedup**	**efficiency**
1	14.1114	0.3573	**0.3573**	14.6750	0.3435	0.3435
2	8.4692	0.5953	0.2976	5.3539	0.9417	**0.4708**
4	3.7932	1.3291	0.3323	2.6864	1.8766	0.4692
8	1.9502	2.5852	0.3231	1.7252	2.9222	0.3653
16	1.2001	4.2011	0.2626	1.5503	**3.2519**	0.2032
32	0.9681	**5.2078**	0.1627	1.9556	2.5779	0.0806

N=8192

	static			adaptive		
NP	**time[s]**	**speedup**	**efficiency**	**time[s]**	**speedup**	**efficiency**
1	29.0982	0.3696	0.3696	29.5137	0.3644	0.3644
2	18.4106	0.5842	0.2921	12.1100	0.8881	0.4441
4	8.0682	1.3330	0.3332	5.0786	2.1177	**0.5294**
8	3.5316	3.0454	**0.3807**	2.7702	3.8823	0.4853
16	1.9910	5.4018	0.3376	2.0586	**5.2244**	0.3265
32	1.3851	**7.7646**	0.2426	2.3121	4.6516	0.1454

Table 5.1: Example 1: Static space tiling vs. adaptive space tiling

N=1024						
	static			adaptive		
NP	time[s]	speedup	efficiency	time[s]	speedup	efficiency
1	0.0985	0.6801	**0.6801**	0.1256	0.5332	**0.5332**
2	0.0887	**0.7552**	0.3776	0.1089	**0.6152**	0.3076
4	0.1229	0.5451	0.1363	0.1472	0.4552	0.1138
8	0.3997	0.1676	0.0210	0.4341	0.1543	0.0193
16	1.0113	0.0663	0.0041	1.1878	0.0564	0.0035
32	1.2634	0.0530	0.0017	1.9041	0.0352	0.0011
N=2048						
	static			adaptive		
NP	time[s]	speedup	efficiency	time[s]	speedup	efficiency
1	0.3962	0.6749	**0.6749**	0.5052	0.5293	**0.5293**
2	0.2838	0.9422	0.4711	0.3598	0.7433	0.3716
4	0.2361	**1.1325**	0.2831	0.3336	**0.8014**	0.2004
8	0.4446	0.6014	0.0752	0.6031	0.4434	0.0554
16	1.0884	0.2457	0.0154	1.2426	0.2152	0.0134
32	1.8379	0.1455	0.0045	1.9850	0.1347	0.0042
N=4096						
	static			adaptive		
NP	time[s]	speedup	efficiency	time[s]	speedup	efficiency
1	1.5857	0.6725	**0.6725**	2.0584	0.5181	**0.5181**
2	1.0506	1.0150	0.5075	1.3615	0.7832	0.3916
4	0.6748	**1.5803**	0.3951	1.0630	**1.0032**	0.2508
8	0.7545	1.4135	0.1767	1.0882	0.9800	0.1225
16	1.3632	0.7823	0.0489	1.3291	0.8023	0.0501
32	2.0956	0.5089	0.0159	1.8753	0.5686	0.0178
N=6144						
	static			adaptive		
NP	time[s]	speedup	efficiency	time[s]	speedup	efficiency
1	3.6753	0.6537	**0.6537**	4.9291	0.4874	**0.4874**
2	2.3805	1.0092	0.5046	3.0285	0.7933	0.3966
4	1.3960	1.7210	0.4302	2.2619	1.0621	0.2655
8	1.1334	**2.1197**	0.2650	1.9941	**1.2048**	0.1506
16	1.5820	1.5185	0.0949	2.0782	1.1560	0.0723
32	2.3471	1.0235	0.0320	2.4913	0.9643	0.0301
N=8192						
	static			adaptive		
NP	time[s]	speedup	efficiency	time[s]	speedup	efficiency
1	66.0013	0.1572	0.1572	60.4537	0.1717	**0.1717**
2	5.5125	1.8827	0.9413	38.8427	**0.2672**	0.1336
4	2.4522	4.2323	**1.0581**	41.2815	0.2514	0.0629
8	1.5976	**6.4962**	0.8120	39.6800	0.2615	0.0327
16	1.8615	5.5754	0.3485	44.9439	0.2309	0.0144
32	2.3384	4.4382	0.1387	44.3057	0.2342	0.0073

Table 5.2: Example 2: Static space tiling vs. adaptive space tiling

Chapter 6

Conclusion

In this thesis we gave theoretical background and hints towards a practical implementation of adaptive tiling in the polyhedron model. We started with an analysis of parametric static space-tiling and surprisingly discovered unsolved problems. Thus instead of advancing to adaptive tiling quickly this chapter became the main emphasis of this work. Furthermore the techniques we developed there could be easily adapted to adaptive space tiling.

First we established results that help when working with parallelepipeds and we showed how to scale a tile-shape along its spanning vectors. Subsequently we found a characterization of valid lattice matrices and discussed which lattice matrices are well suited for tiling. The next open problem was to automatically find a mapping from tile coordinates to physical processors. The challenge was to find a mapping that avoids negative processor coordinates, does not use more than the available processors in each dimension and covers the whole index space with the available processors. We investigated three approaches and only the last lead to a simple algorithm. The key to this algorithm was the observation that scaling and finding the virtual processor offset to avoid negative tile coordinates could be done independently.

In chapter 4 we introduced our ideas on adaptive tiling. Because we saw in our empirical studies that code complexity can be a problem we proposed a communication library that encapsulates details of communication generation. Most notably it handles message coalescing and the problem of a changing mapping of virtual processors to physical processors during adaptive tiling. We proposed three different methods for adaptive tiling, which differ in communication behavior, required tools and mapping of tiles to physical processors. The first two adapted the algorithm for automatically finding a space-tiling from chapter 3 while the last method used a different approach which combines a block-cyclic distribution with a non-parametric static space-tiling and optionally time-tiling. Finally we evaluated adaptive tiling on selected simple examples and interpreted the resulting benchmarks.

We expect that adaptive tiling can make a real difference for certain parallel programs, especially on shared memory machines with a small number of processors (e.g. 2, 4, or 8). This is a relevant class of machines, because current developments in processor design make it seem likely that the typical desktop computer will have at least two processors in the near future. But before adaptive tiling with automatic space-tiling can be implemented, further research in the field of code generation from generalized polyhedra is required. We hope that this thesis can be used to implement a fully automatic adaptive tiling algorithm in LooPo once the required tool support is in place.

Bibliography

[Alb88] Klaus Alber, Werner Struckmann. *Einführung in die Semantik von Programmiersprachen.* BI Wissenschaftsverlag, 1988.

[Amo04] Sven Amonat. *Code generation in an extended polyhedron model with non-linear parameters.* Martin-Luther-Universität Halle-Wittenberg. To appear in 2004.

[And00] R. Andonov and P. Calland and S. Niar and S. Rajopadhye and N. Yanev. *First steps towards optimal oblique tile sizing.* 8th International Workshop on Compilers for parallel computers, pages 351–366, Aussois, Jan. 2000.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.

[Bas02] Cédric Bastoul. *Generating loops for scanning polyhedra.* Technical Report 2002/23, PRiSM, Versailles University, 2002.

[BCF97] Denis Barthou, Jean-François Collard, and Paul A. Feautrier. *Fuzzy dataflow analysis.* J. Parallel and Distributed Computing, 40(2):210-226, February 1997.

[Be96] Eike Best. *Semantics of Sequential and Parallel Programs.* Prentice Hall International Series in Computer Science, 1996.

[Cla04] Michael Claßen. *Code Generierung für HS-LooPo.* Lehrstuhl für Programmierung, Fakultät für Mathematik und Informatik, Universität Passau, July 2004.

[CLooG] Cédric Bastoul. `http://www.prism.uvsq.fr/~cedb/bastools/cloog.html`

[Col95] Jean-François Collard. *Automatic parallelization of* `while`*-loops using speculative execution.* Int. J. Parallel Programming, 23(2):191-219, 1995.

[Ellm04] Nils Ellmenreich. *Comparative Parallel Programming with Abstract Parallel Machines.* Phd Thesis. Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[Fea91] Paul Feautrier. *Dataflow analysis of array and scalar references.* Int. J. Parallel Programming, 20(1):23-53, February 1991.

[Fea92] Paul Feautrier. *Some efficient solutions to the affine scheduling problem. Part I. One dimensional time.* Int. J. Parallel Programming, 21(5):313-348, 1992.

[Fi97] Gerd Fischer. *Lineare Algebra.* Reihe vieweg studium, 11. Auflage. Vieweg Verlag 1997.

[GGL97] Max Geigl. *Parallelization of Loop Nests with General Bounds in the Polyhedron Model.*
`http://www.fmi.uni-passau.de/loopo/doc/geigl-d.ps.gz`
Fakultät für Mathematik und Informatik, Universität Passau, 1997.

[GGL99] Max Geigl, Martin Griebl, and Christian Lengauer. *Termination detection in parallel loop nests with* `while` *loops.* Parallel Computing, 25(12):1489-1510, November 1999.

[GNU] GNU is not Unix.
`http://www.gnu.org`

[Gri97] Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing* `while` *Loops.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, January 1997. Technical Report MIP-9701.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures.* To appear in 2004.

[GrLe96] Martin Griebl and Christian Lengauer.
The Loop Parallelizer LooPo—Announcement. Languages and Compilers for Parallel Computing (LCPC'96), pages 603–604. SV, 1997.

[GriVl] Martin Griebl. *Skripte der Vorlesung Schleifenparallelisierung 1 und 2* by Martin Griebl. University of Passau.

[Grö03] Armin Größlinger. *Extending the Polyhedron Model to Inequality Systems with Non-linear Parameters using Quantifier Elimination.* Diploma Thesis. University of Passau 2003.

[HCF99] Karin Högstedt, Larry Carter, and Jeanne Ferrante. *Selecting tile shape for minimal execution time.* In 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99), pages 201-211. ACM Press, June 1999. Also available with proofs as UCSD Tech Report CS99-616.

[Hermes] Hermes – The hpcLine cluster at the chair of programming, University of Passau.
`http://www.infosun.fmi.uni-passau.de/infosun/`
`programmierung/hpcline/`

[Len93] Christian Lengauer. *Loop parallelization in the polytope model.* In Eike Best, editor, CONCUR'93, Lecture Notes in Computer Science 715, pages 398–416. Springer–Verlag, 1993.

[Linux] The Linux Kernel Archives.
`http://www.kernel.org`

[LoopGen] LoopGen - A Tool for Generating Loops from Polyhedra.
`http://www.irisa.fr/cosi/ALPHA/codegen/`

[LooPo] LooPo - Loop Parallelization in the Polytope Model.
`http://www.uni-passau.de/~loopo`

[MPI] MPI Forum - Home of the MPI standards documents.
`http://www.mpi-forum.org/`

[QRW00] F. Quilleré, S. Rajopadhye and D. Wilde. *Generation of Efficient Nested Loops from Polyhedra.* International Journal of Parallel Programming, vol 28, no 5, October 2000.

[Sch00] Uwe Schöning. *Logik für Informatiker.* 5. Auflage. Spektrum Akad. Verl., 2000.

[ScaMPI] Scali MPI ScaMPI.
`http://www.fujitsu-siemens.com/hpc/products/hpcline/scali_scampi.htm`

[SCI] SCI as cluster interconnect.
`http://www.fujitsu-siemens.de/loboea/hpc/products/hpcline/scali_cluster_interconnect.htm`

[Wie97] Martina Wieninger. *Partitionierung von parallelen Schleifenprogrammen.* Diploma Thesis. University of Passau 1997.

[Xue94] Jingling Xue. *Automating non-unimodular transformations of loop nests.* Parallel Computing, 20(5):711-728, May 1994

[Xue97a] Jingling Xue. *Communication-minimal tiling of uniform dependence loops.* J. Parallel and Distributed Computing, 42(1):42-59, April 1997.

[Xue00] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 2000.

Index