Fakultät für Mathematik und Informatik

Habilitation

Automatic Parallelization of Loop Programs for Distributed Memory Architectures

Author:
Martin Griebl

2nd June 2004

Abstract

Parallel computers, especially in the form of clusters of standard PCs, have become reasonably cheap within the last few years. It is an obvious desire to use the increased computation power of such parallel hardware in order to speed up any given application. However, for that purpose, these application programs must be transformed such that they take benefit of the parallel hardware. One solution to generate the necessary parallel software is to use automatic parallelization, i.e., a parallelizing compiler. Such a tool takes a program in which nothing is specified about parallelism, and automatically transforms it to a parallel program. This idea allows to introduce parallelism easily, i.e., without much effort and, simultaneously, with guaranteed correctness with respect to the input program.

This thesis presents a way to build up such a parallelizing compiler. In order to be efficient, we restrict ourselves to arbitrarily nested loops as the only control structure causing repeated computations. We apply a mathematical model, the *polyhedron model*, that gives a unified framework for the various parallelization tasks, and that allows a directed search for optimal solutions.

We touch nearly every parallelization task and demonstrate the interactions between them. One of the main topics of this thesis is how to extract parallelism of the right granularity: too coarse-grained parallelism might not exploit the parallelism available from the hardware, and too fine-grained parallelism leads to increased overhead, especially to communication overhead. We shall derive a method that allows to precisely adapt the granularity to the given parallel architecture.

Acknowledgements

This work has only become possible due to a lot of people that helped me in very different ways.

First of all, I would like to thank my advisor Christian Lengauer, who always has had time for me when I needed him – no matter whether the reason was a technical question, some proof-reading, an administrative problem, a discussion about actual developments in the academic environment, or "just" the planning of my future. Thank you, Christian, for your support in any situation. I hope, no, I am sure that our good relationship will survive the day when I shall have to leave your team.

I am also very grateful to Paul Feautrier, with whom I had many very fruitful discussions on various topics of this thesis. Merci, Paul, for our productive collaboration and, especially, for the fact that you always reserved plenty of time for me when I visited you – both, time during the days and the evenings.

I also would like to thank François Irigoin and all members of his team: visits to you are always very agreeable and, at the same time, very instructive. Thank you for that.

A-pro-pos France: thanks to all my French friends and colleagues! Thanks, of course, for the professional benefit due to your comments on my work and the various discussions we had, but also for taking care of me every time I come to France. As a few representatives of all the friendly and helpful people, I would like to mention explicitly Denis Barthou, Cédric Bastoul, Albert Cohen, Jean-François Collard, Alain Darte, and Tanguy Risset. Although we cannot meet very often, good friendships have evolved.

In this context I am grateful to the French-German exchange programme PROCOPE *through the DAAD and APAPE, as well as to the French-Bavarian exchange programme* BFHZ/CCUFB*: without their financial support, all these contacts would never have been possible.*

Thanks also to Mohamed Jemni for discussing his index set splitting method with us in Passau, which started off work on parts of this thesis, and thanks to his advisor Zaher Majoub and all his team members for the informative and agreeable week in Tunis.

Let me now come back to Germany, where there is a huge number of people that I owe my thanks, and I know that my list must be incomplete. Anyway, I want to mention some of them explicitly.

First, I would like to thank the colleagues within our group. I am grateful to Nils Ellmenreich who shared the office with me for years: thanks for good professional and private discussions and for your tea service, to Peter Faber who acted on my behalf in several respect during my stay in Halle: thank you for that, for your activities as a co-leader of LooPo, and also for all the technical support I got from you, to Armin Größlinger, my former student and new colleague in the office, who has been willing to rock the foundations of our mathematical model, thus enabling important extensions: thanks for your courage and congratulations that you eventually convinced me of functional programming, and to Christoph Herrmann for his proof-reading and his comments on my work.

In addition, I would like to thank Siegfried Graf, Gottlieb Leha, and Niels Schwartz for

their mathematical advice and support.

Also, I am grateful to all *my colleagues in Halle, who, from the first day, gave me the feeling that I belong to the team. Such an enjoyable environment is the basis for productive work. Thank you all for that! Nominally, I want to mention Wolf Zimmermann who offered me that possibility.*

Thanks also to all the people in Passau and Halle who kept work away from me that I did not like too much. There are many helpers in the background, who, anyway, have a big influence on ones efficiency: all the secretaries, among which I want to mention Johanna Bucur as only one representative, the people involved in system administration who always have helped immediately when I had some trouble, and the helpful people in our administration. Thank you all very much for your support.

Direct support for my research activities comes from a group of students: the LooPo team. *They assist me by implementing the theoretically derived methods, and they run experiments in order to evaluate and compare different approaches, i.e., they are responsible for checking the practicability of my work. Nominally, I want to mention those with a long-term engagement with LooPo since the time when I started writing this thesis: Sven Amonat, our "remote member" from Halle, Sven Anders, Gordon Bolduan, Michael Classen, who supports me with the tricky job of communication code generation, Andreas Dischinger, Babette Eckart, Axel Krauth, Tobias Langhammer, Bernhard Lehner, Oliver Nyderle, Stefan Schuster, Michael Seibold, Georg Seidel, who actually wants to improve the methods described in this thesis, Andreas Wolf, and Thomas Wondrak. Thank you all for your wonderful collaboration.*

Last, but certainly not least, I would like to say thank you to all members of my family and to all my friends. You provide me with the right balance between motivation and energy for work on the one hand, and the necessary distance from it and the important connection to the world besides work on the other.

Especially, thank you, Gabi and Lisa, for understanding that I had to spend much time writing this thesis, instead of spending the time with you, and thank you very much for your love which has been a never-ending source of energy for me. Thanks to my parents who also had to suffer from my limited time, even though I know that they are the ones who gave me the possibility to pursue this career, and thanks to Mäx for many interesting discussions and for some proof-reading.

Thanks to God for all of you.

Contents

Chapter 1

Introduction

In the last decade, parallel computers have become widely spread. Even some of today's home computers have more than one processor. Small enterprises can buy cheap "supercomputers" by simply connecting standard PCs with standard network components; the necessary operating system support for this kind of machine is included in standard operating systems for free (e.g., Linux).

This strong presence of parallel hardware requires adequate software that can take advantage of the distributed computation power. However, as previously for sequential computers, software development is far behind. This thesis – like many other research papers – is a step towards filling this gap.

Direct approach At a first sight, one could argue that parallel programs are simpler than sequential programs because no order needs to be specified for parallel computations. However, it turns out to be quite difficult for a programmer to keep in mind all possible orders in which the computations of a parallel program might be executed, and, among these, to disable undesired orders by introducing appropriate synchronization statements. The generation of this necessary control code with its synchronization and communication statements is the central problem when writing explicitly parallel programs.

Note that this difficulty increases with an increasing number of potentially parallel computations, since the number of possible execution paths then grows exponentially.

Fully automatic approach Compared to this direct approach to parallelism, it seems to be simpler to write sequential programs, thereby over-specifying the execution order. Then, however, we need a parallelizing compiler that takes sequential programs as input, and automatically decomposes the specified work and distributes it over several processors, while also generating all necessary control, synchronization and communication code.

Hence, our global goal is to develop all necessary techniques for such a parallelizing compiler.

Note that this approach is not limited to sequential imperative programs as input, but can also be taken for functional programs or specifications. In this case, the execution order

is often less restricted than in sequential imperative programs, but usually it is still an over-specification. Again, we want to generate the explicitly parallel program automatically.

Correctness This fully automatic approach leads to a property that is considered more and more important: provable correctness of software. In our case, we can guarantee correctness w.r.t. the source program or specification (provided the compiler is proved correct). Thus, the programmer can focus on writing a correct sequential program and need not think about the (difficult) subject of correctness in parallel systems – even if his/her program is executed on a parallel computer.

Performance The price for this fully automatic approach is a reduced performance when compared to the direct approach. This is inherent with the procedure: in general it is undecidable for the parallelizing compiler whether a specified execution order in the source program is due to the problem specification or due to the over-specification in the sequential program. Thus, the parallelizer cannot find all parallelism which exists in the original problem specification.

In extreme situations, the programmer also can give hints to the parallelizer, or even write explicit parallel code, in order to improve performance at the price of a longer program development time and, maybe, correctness.

In this sense, using automatic parallelization is analogous with using high-level programming languages for sequential programming: in extreme cases, the programmer can take the direct approach and write highly optimized assembler code by hand, but most of the time the indirect but automatic way through a compiler is taken, and a small loss in performance is accepted for a safer and faster program development.

Portability This analogy carries on to the case where a program must be ported to another hardware platform (and hardware changes surprisingly often). In the direct approach we need to port or, at least, to adjust the parallelism of every application program individually; in the automatic approach, we only need to port the parallelizing compiler once.

Applicability Besides the mentioned loss in performance, the second drawback of the fully automatic approach is its limited applicability. Much work in this research area, and also this thesis, focuses on nested loops. The reason is two-fold.

- Programs spend the majority of their execution time in repeating the same computations on different data. Thus, control structures that lead to repetition are worth to be parallelized, and loops are, besides recursive functions, the most important reasons for repeated execution: a loop generates multiple run-time *instances* of the statements in the loop body.
- Loops tend to have some regularity which makes the task of parallelization simpler, if not to say solvable at all.

Conviction We think that the automatic approach has not yet reached its limits, neither at the performance nor at the applicability border. Hence, some parts of this thesis aim at generalizing the type of programs that can be treated automatically (e.g., Chapter 5). Other chapters aim at improved performance of the parallel program – the main reason why parallelism is introduced at all.

Chapter 2

Motivation: two typical examples

2.1 Gaussian elimination

Before going into technical detail, we introduce a program that can be used to illustrate many of the definitions and methods presented in this thesis. This running example is one of the most popular algorithms for solving a system $A\,x = b$ of linear equalities: Gaussian elimination [BS89].

The full algorithm works in two phases:

1. an elimination phase transforms the regular square coefficient matrix A of the equality system to an upper triangular form, and
2. a backward substitution phase computes the solution vector x.

In order to keep the example simple enough to be illustrative, we modify the algorithm as follows:

- we only show the elimination phase;
- the right hand side vector b of the equality system is appended to the coefficient matrix A as an additional last column; this extended matrix is stored in an array a;
- we assume that the diagonal entries in A are never going to become 0, which means that we can drop the case distinction and the special treatment for the other case;
- we do not explicitly store the 0 values in the lower triangle of the resulting coefficient matrix since these elements are, of course, never used afterwards;
- we use a two-dimensional array for storing temporary values so as to avoid reassignments; the same can be achieved when we start from a single scalar variable for the temporary values and then apply automatic methods for a conversion into single-assignment form [Fea91]. But since this is not in the focus of this thesis, we provide the source program in a suitable form. Note that there remain reassignments on array a.

```
        for i := 0 to n−1
          for j := i+1 to n−1
T :         tmp[i, j] := a[j, i]/a[i, i]
            for k := i+1 to n
S :           a[j, k] := a[j, k] − tmp[i, j] ∗ a[i, k]
            endfor
          endfor
        endfor
```

Figure 2.1: Source code of the simplified Gaussian elimination

The code for this simplified Gaussian elimination method is shown in Figure 2.1.

We are going to analyze the features of this code along with the definitions of the respective technical terms in the subsequent chapters.

2.2 LU decomposition

At some places, a more complex example is needed to illustrate the power – or also the limitations – of the methods presented. For this purpose, we use LU decomposition [Ger78]: the task is to take a regular square matrix A and to return a lower triangular matrix L and an upper triangular matrix U with the property $A = L\,U$, and the diagonal entries of U are all 1. A possible implementation is given in Figure 2.2.

We chose a syntactic representation of the algorithm in which every summation due to the loops with index $k1$ or k has its own accumulator; this reduces the number of dependences. As for Gaussian elimination, this LU decomposition algorithm does not fill the trivial parts of the resulting matrices, i.e., the triangular submatrices which are filled with 0, and the unit diagonal of U.

We are not going to analyze this algorithm in detail; we just focus occasionally on special aspects of this code.

```
        for i1 := 1 to n
S1:       L[i1, 1] := A[i1, 1]
        endfor
        for j1 := 1 to n
S2:       U[1, j1] := A[1, j1]/A[1, 1]
        endfor
        for j := 2 to n
          for i := j to n
S3:         SUM[j, i] := 0
            for k1 := 1 to j−1
S4:           SUM[j, i] := SUM[j, i] + L[i, k1] ∗ U[k1, j]
            endfor
S5:         L[i, j] := A[i, j] − SUM[j, i]
          endfor
S6:       U[j, j] := 1
          for i2 := j+1 to n
S7:         SUMM[j, i2] := 0
            for k := 1 to j−1
S8:           SUMM[j, i2] := SUMM[j, i2] + L[j, k] ∗ U[k, i2]
            endfor
S9:         U[j, i2] := (A[j, i2] − SUMM[j, i2])/L[j, j]
          endfor
        endfor
```

Figure 2.2: LU decomposition

Chapter 3

Basics: parallelization in the polytope model

3.1 The polytope model and its history

The basic idea of loop parallelization is that independent loop iterations can be executed in parallel. For this purpose, many parallelization methods textually modify the loop structure in the source program until some loops can be marked to be parallel. We call this approach *text-based parallelization*. Some typical techniques in this context are *peeling*, *splitting*, *fusion*, *reversal*, *interchange*, *permutation*, ... of loops [Wol95]. Further research effort in this area has developed more evolved parallelization techniques which allow more serious changes of the structure of the source program, e.g., *loop skewing*, *echelon transformations*, and *loop distribution* [Ban94, Wol95].

The main challenge of this approach is to pick the right transformations for a given source program – and the right order of these transformations.

Model-based parallelization A completely different approach is to discard the textual structure of the source program, and to do all parallelization based on a mathematical model. The expected advantage is that, if the model is general enough, the right sequence of the right parallelization steps can be represented as one single transformation, and this transformation can be discovered automatically by classical mathematical optimization methods. We call this approach *model-based parallelization*.

For some methods of this approach, the structure of the loop nest is important.

Definition 1 (nested loops). A set of nested loops is called a *perfect loop nest* iff the body of every loop consists of

- either precisely one (further nested) perfect loop nest,
- or (in the case of the innermost loop) a sequence of statements without a loop.

Otherwise, the loops form an *imperfect loop nest* (also called *nontightly nested loop nest* [Wol95]).

The *body of a perfect loop nest* L is the body of the innermost loop of L.

The number of loops surrounding a statement S is called the *nesting depth* of S.

The central task in this model-based approach is to determine, for every computation, when it should take place. This temporal distribution is called *schedule*. In this setting, parallelism is expressed by scheduling several computations at the same time step. The constraints for scheduling are all kinds of dependences between computations.

Similarly to the schedule, there exists also a spatial distribution, the *placement*, which determines, for every computation, where it should take place. Both distributions are formalized in Section 3.3.

Seminal Work The first scheduling method was proposed about 35 years ago by Karp, Miller, and Winograd, and it was designed for scheduling uniform recurrence equations [KMW67]. This seminal work is the basis of many research activities in the field of systolic array design in the late Seventies and Eighties [KL80, Kun88, Mol83, Qui87, RK88].

In 1974, Lamport transferred this approach to the field of automatic parallelization of nested loops [Lam74]. His scheduling method, known as *Lamport's hyperplane method*, can be considered as the first model-based loop parallelization method.

The original polytope model In 1993, Lengauer presented a brief description of a model-based framework for automatic parallelization of nested loops that covers all parallelization phases [Len93]. It has a mathematical basis, the *polytope model*, which imposes severe restrictions on the source program:

- It must be a perfect loop nest with only assignments in its body.
- All loop bounds must be affine expressions in indices of surrounding loops and in *structure parameters*, i.e., symbolic constants which typically represent the problem size.
- The only supported data structure is the array; the array indices must be affine functions in indices of surrounding loops and structure parameters. Scalars are viewed as 0-dimensional arrays.
- The dependences in the source program must be *uniform*, i.e., identical at all instances, i.e., iterations, of the body of the loop nest.

Parallelization procedure With these limitations, automatic parallelization in the polytope model roughly works in three steps (more formal details are given in Sections 3.3.6 to 3.3.8):

1. The textual representation of the source program is converted into a model-based representation:

- every loop in the program corresponds to one dimension in the model, i.e., the model of a program with d nested loops is a d-dimensional space, the *index set*, whose bounds in the various dimensions are given by the loop bounds;
- the dependences between instances of statements are computed and represented in the model, e.g., as dependence vectors.

2. The model-based description is used to extract parallelism:
   - the distribution of the body instances in (logical) time – the schedule – is computed (this distribution guarantees the correctness of the parallelized program);
   - the distribution of the body instances in (virtual) space – the placement – is computed (this distribution aims at efficiency, e.g., by reducing the communication volume [Fea00, DR95]);
   - schedule and placement together form the *space-time mapping*;
   - the index set is expressed in a new coordinate system of temporal and spatial dimensions, resulting in the *target index set*;
3. the model-based representation, the target index set, is converted to a target program. For this purpose, a loop nest is generated that scans the target index set, i.e., that enumerates all points inside. This loop nest consists of sequential and parallel loops, which enumerate the dimensions in time and space, respectively.

The fact that the placement distributes the computations in *virtual* space means that it assumes an unbounded number of available processors. Thus, the target index set represents which virtual processors are active at every logical time step. In this sense, it also gives us the maximal amount of parallelism that our approach can extract from the source program.

Extensions Some of the methods used for parallelization in the polytope model have been generalized in their application framework even before the comprehensive description of the polytope model. E.g., Quinton started off with systolic array design from uniform recurrence equations – which correspond to uniform dependences – but presented also a way how to deal with not fully indexed variables – corresponding to imperfectly nested loops [Qv89]. Similarly, the data flow analysis method of Feautrier was neither restricted to perfectly nested loops, nor to uniform dependences [Fea91].

These and more extensions have generalized the original polytope model, without changing the principal procedure. Hence, the polytope model nowadays allows that:

- dependences may be affine, not only uniform,
- schedules and placements may be piecewise multi-dimensional affine functions, which are defined individually for each statement and, as a consequence,
- imperfectly nested loops are allowed as input [Fea92a, Fea92b, Fea94, DV94, DR95].

Programs that satisfy these limitations are often called *static control programs* [Fea91]. They can be analyzed precisely at compile time. If we relax the constraints further, we give up this property and obtain *dynamic control programs*:

- the type of loops is generalized to for loops with arbitrary bounds [GGL99] and to while loops, i.e., loops whose number of iterations is unknown even when the loop starts iterating [Gri97, Col95];
- arbitrary conditionals are allowed [BCF97];
- complex array accesses can lead to non-affine dependences.

In all these cases, suboptimal results due to necessary approximations must be accepted.

Note that, for the integration of while loops, two different approaches have been explored. In a conservative approach, control statements ensure that the body is only executed at iterations which are also executed in the sequential program [Gri97]. In contrast, a speculative approach allows to execute the body at additional iterations [Col95]. The advantage is a reduced number of dependences, i.e., more parallelism, but the price is a complex commit protocol for computed values or a roll back for illegally executed iterations. Due to the existence of these solutions, and also because the integration of while loops into the model and possible code generation schemes for them have been the subject of a precursing PhD thesis [Gri97], we shall not focus on this topic here any further, and we refer to the original literature for technical details on these extensions.

Since the introduction of while loops leads to an index set that is unbounded at compile time, the model with these dynamic extensions is also called the *polyhedron model*. As just noted, this aspect is not important for the rest of this thesis, so we use the more traditional name polytope model further on.

3.2 Open problems

Despite these generalizations, there are issues in the different parallelization phases which must be improved in order to move the polytope model towards practical usability and acceptance. We visit some of them in the traditional order of the parallelization phases. These aspects are also the main topics of this thesis.

- **Applicability of the model** First, any correct program – not just loop nests with static control flow – should be accepted by a parallelizer. This goal leads to, apart from some implementation effort, a theoretical question for the first phase of the parallelization pipeline: how do we model and analyze programs with a control flow that is less structured than regular loop nests (e.g., that may contain break or continue statements), or even completely unstructured (e.g., by goto statements)? And how can we deal with dynamic control flow (e.g., by if statements) without losing too much information? These questions are tackled in detail elsewhere [CG99], but we review the basic ideas in Section 5.2, with a focus on the latter question.

- **Granularity control** The central challenge is to find the right granularity of parallelism, in order to balance the distribution of work and the communication overhead. The standard procedure is to apply tiling techniques to the source program in order to increase the grain size of the computations which shall be distributed among the processors. Unfortunately, when compared to traditional parallelization techniques, this procedure is usually less applicable (e.g., many tiling methods are designed for perfect loop nests), and less powerful (e.g., even in simple cases, the optimal communication amount may be missed by orders of magnitude). We focus on this aspect in Chapters 8 to 12.

- **Schedule improvement** There are cases in which even the best existing scheduling methods [Fea92a, Fea92b] miss the existing parallelism by orders of magnitude [Fea92a]. An attempt to improve this situation is presented in Chapter 6.

- **Placement** It turns out that we can support tiling further by constructing a placement which leads to some regularity for the communications. More detail, together with an appropriate placement algorithm, is given in Chapter 7.

- **Code generation** An often neglected but essential topic is the generation of efficient parallel code. A first question is how we can generate a parallel program at all, if the (independently computed) schedule and placement functions do not satisfy a necessary mathematical constraint (Section 13.1). Another problem is to check how the representation of the schedule can affect the simplicity and, hence, the efficiency of the parallel loop nest (Section 13.2). Also, there are several methods for generating the parallel loop code as well as a several possible target languages among which one must decide (Sections 13.3 and 13.5). Finally, a topic of active research is the generation of efficient communication code including the aspects of message vectorization, or buffer organization and reuse (Section 13.4).

3.3 Mathematical background

As already mentioned, in the presence of loops, every statement in the body has several instances at run time.

Definition 2 (iteration vector). Let S be a statement in a loop program. The *iteration vector* of statement S is the vector that consists of the indices of all loops surrounding S, ordered from outside in.

Definition 3 (index set). The set of all iteration vectors for a given statement S is the already mentioned index set of S. We denote it by $\mathcal{I}(S)$.

Definition 4 (operation). An *operation* is a run-time instance of a statement. It is identified by the name S of the statement and the iteration vector i. We denote an operation by $\langle i; S \rangle$.

The set of all operations of a program is denoted by Ω.

If there is only one statement of interest, we sometimes omit giving the name S and denote an operation by the iteration vector only.

Example 5. Consider our running example in Figure 2.1. Note that the iteration vector for statement T is two-dimensional, whereas the iteration vector for statement S is three-dimensional. The first operation that is computed by the sequential program is $\langle 0, 1; T\rangle$, the last operation is $\langle n-1, n-1, n; S\rangle$.

3.3.1 Mathematical constraints

In order to use efficient mathematical tools, and in order to obtain a clean presentation, we require the loop bounds in this thesis to be *affine functions* in the surrounding loop indices and *structure parameters*, i.e., symbolic constants [Len93, Fea96]. A method avoiding this restriction is given elsewhere [Gri97].

Definition 6 (linear). A d'-dimensional function f with d arguments $v_1, \cdots, v_d$ is *linear* iff it can be expressed in the following form:

$$\textit{linear} \qquad f(v) = M_f\, v, \tag{3.1}$$

where $v = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{d' \times d}$ is a matrix with d' rows and d columns.

In our context, we usually deal with rational matrices, i.e., $M_f \in \mathbb{Q}^{d' \times d}$, or even with integral matrices, i.e., $M_f \in \mathbb{Z}^{d' \times d}$.

Definition 7 (affine). A d'-dimensional function f with d arguments $v_1, \cdots, v_d$ is *affine* iff it can be expressed in the following form:

$$\textit{affine} \qquad f(v) = M_f\, v + f_0, \tag{3.2}$$

where $v = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ and $M_f \in \mathbb{R}^{d' \times d}$ as above, and $f_0 \in \mathbb{R}$.

Analogously, in most of our cases $f_0 \in \mathbb{Q}$ or even $f_0 \in \mathbb{Z}$.

Remark 8 (no parameters as coefficients). It is important to note that, in both definitions, M_f is only allowed to contain numbers, not symbolic constants. The same is true for f_0 in the affine case. This is a severe limitation that we shall meet at several places in this thesis. At some places, we can get around the limitation by formulating our problem differently (e.g., in Chapter 7), and at others we cannot (e.g., in Section 12.3).

This limitation has crafted a currently very active research field, in which one uses quantifier elimination to overcome this limitation. The central change is that, with this extension, one obtains a tree of possible solutions, depending on the values of the parameters, instead of one solution [Grö03, GGL04]. Keeping this tree manageably small is one of the greatest challenges in that context.

Remark 9 (piecewise linearity). Compared to the mathematically difficult task of allowing parameters as coefficients, it is relatively simple to break the linearity or affinity constraints by using *piecewise linearity* whenever appropriate. Chapter 6 describes one application of this idea in detail.

Notation Before we consider the relationship between linear and affine functions in more detail, let us briefly introduce some basic notation, which is used frequently throughout this thesis.

The *transpose* of matrix M is denoted by $M^\top$.

Id_m represents the identity matrix of rank m, and $Zero_{r,c}$ denotes a matrix with r rows and c columns in which all entries are 0.

3.3.2 Homogeneous coordinates

Desirable linear setting The application of a linear function can be represented as a matrix multiplication. Hence, the application of a sequence of transformations can be expressed as a sequence of matrix multiplications. Now, since matrix multiplication is associative, we can group all matrix products together, before applying the resulting one transformation matrix to the argument:

$$M_{f_r}\,(\cdots\,(M_{f_1}\,v)\cdots) = (M_{f_r}\,\cdots\,M_{f_1})\,v \tag{3.3}$$

I.e., we only need to compute a combined transformation only once, and then we can apply it to all arguments in turn.

Problem in the affine setting For an arbitrary shift vector s of dimension m, there is no matrix M_s such that

$$v + s = M_s\,v. \tag{3.4}$$

This is an undesirable situation for us, since we intend to use affine functions for any kind of transformation within the mathematical model, and we would like to deal with a single transformation matrix, as this is possible in the linear case.

Solution Fortunately, there is a way that all affine functions (including shifts) can be formalized such that Equation (3.1) holds. For this purpose, we must add a dimension for the constant 1, i.e., we treat the constant 1 as a new variable that is set to the numeric value 1 and not changed by f [Max46, FvDF$^+$94]. It turns out that a shift in the original space becomes a skewing in this higher-dimensional space – and skewing can be represented by a matrix.

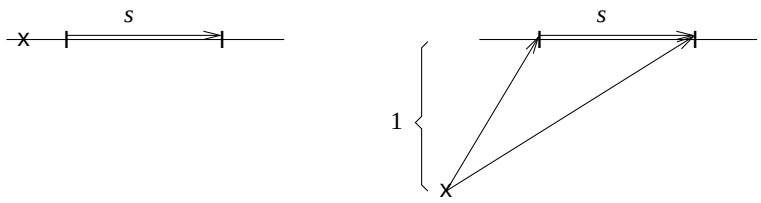


Figure 3.1: A shift in homogenous coordinates

Example 10. Let us illustrate the correspondence between shifting and skewing along the example in Figure 3.1. The left part shows a one-dimensional space; the origin of the coordinate system is indicated with x*, and the shift by a vector s is depicted as an arrow within the one-dimensional space. The right part is a two-dimensional space, in which the origin of the coordinate is 1 unit below the one-dimensional subspace. Obviously, the skewing, indicated by the two arrows starting at the origin, leads to the desired shift in the one-dimensional subspace.*

More formally, every shift can be represented as a matrix multiplication, as shown in Equation (3.1), if we use this extended coordinate system. We set

$$M_s = \begin{pmatrix} Id_m & s \\ Zero_{1,m} & 1 \end{pmatrix},$$

where s is the desired shift vector as above.

Now, let us check whether Equation (3.4) holds. First, we must extend v and s in order to adjust the dimensionality of the two sides of the equality: we must extend v by a coordinate 1, since it is one unit from the origin in the additional dimension, and s by a coordinate 0, since it has no extent in that dimension. With these conventions, it is easy to see that Equation (3.4) holds.

Formal definition We now present the formal definition, which, in principle, allows arbitrary coefficients in the additional dimension. However, this generalization is not relevant for our purpose.

Definition 11 (homogeneous coordinates). Let v be an m-vector that represents a point in an m-dimensional space. Then, the *homogeneous coordinates* of $v = (v_1, \cdots, v_m)$ are $(v_1, \cdots, v_m, 1)$. Conversely, an $(m+1)$-vector $u = (u_1, \cdots, u_m, u_{m+1})$ in homogeneous coordinates represents the point $(\frac{u_1}{u_{m+1}}, \cdots, \frac{u_m}{u_{m+1}})$ in an m-dimensional space.

General usage Hence, the usage of the homogeneous coordinates is quite simple: we just add a component 1 to every vector and, thus, can consider every affine function as a linear one, which can be represented as a matrix.

Extension In our model, we want to be able to deal with structure parameters. For this purpose, we can extend the idea of homogeneous coordinates and consider all parameters as variables – as we just did with the constant 1. As above, we only specify functions that do not change the value of these parameters.

Therefore, when dealing with, e.g., iteration vectors, we always join the l-vector i of the indices of the surrounding loops and the m-vector n of structure parameters in order to obtain the d-dimensional homogeneous index vector ($d = l+m$). Note that the m-vector of structure parameters shall always contain one entry for the constant 1.

Example 12. Consider again our running example in Figure 2.1. The iteration vector for T in standard coordinates has length 2: one entry for i and one for j. In the homogeneous representation, the iteration vector is of length 4, one entry for i, j, n, and 1.

3.3.3 Basic linear algebra definitions

Before we demonstrate how the constructs of the source program are modeled, let us briefly recall some basic definitions.

Definition 13 (regular, singular). A square matrix is called *regular* if it has full rank; otherwise, it is called *singular*. Regular matrices are *invertible*.

Definition 14 (hyperplane, halfspace). A *hyperplane* is a $d-1$-dimensional affine subspace of a d-dimensional space. Thus, it can be represented by one affine equality.

A *halfspace* consists of all points of the d-dimensional space which are on one side of a hyperplane, including the hyperplane. Thus, it can be represented by one affine inequality.

We frequently denote a system of r affine equalities by a matrix M, assuming the following normal form

$$\textit{equalities} \qquad M\, v = 0, \tag{3.5}$$

where M is an $r \times d$ matrix, and d is the length of the vector of variables and parameters v; M and v are in homogeneous representation.

Similarly, for systems of affine inequalities, we use the normal form

$$\textit{inequalities} \qquad M\, v \geq 0. \tag{3.6}$$

Polyhedra

Definition 15 (polyhedron, polytope). A *polyhedron* is the intersection of finitely many *halfspaces*. A *polytope* is a bounded polyhedron.

Representations of polytopes On the one hand, as an immediate consequence of Definition 15, a polyhedron can be denoted like an inequality system in homogeneous representation (cf. Equation (3.6)). Every row of matrix M represents one halfspace. Alternatively, every row can also be considered as an equality that defines a hyperplane bounding the polyhedron.

On the other hand, a non-homogeneous representation is more typical in books on linear programming [Sch86, NW88]. There, a non-parameterized polytope is represented in the following normal form:

$$A\,x \leq b, \tag{3.7}$$

where b is a constant vector.

Similarly, parameterized polytopes are usually represented as

$$A\,x + B\,n \leq b, \tag{3.8}$$

where x is the vector of the variables, n is the parameter vector.

Note that the representation of Equation (3.6) and (3.8) are equally expressive, since the constant vector b and the matrix B can be coded in the homogeneous extension of A.

Cones

In contrast to polyhedra, there is an important difference between homogeneous and non-homogeneous representation when considering cones:

Definition 16 (polyhedral cone). A *polyhedral cone* $\mathcal{C}$ is a set, defined by

$$\mathcal{C} = \{x \,:\, A\,x \leq 0\}.$$

In this definition, A and x are in non-homogeneous representation [Sch86]. Note that this is a limitation: as in the case of polyhedra, every row of matrix A can be interpreted as an equality which represents a hyperplane bounding the cone, but now, every hyperplane contains the origin of the coordinate system, because in the non-homogeneous representation, $x = (0, \cdots, 0)$ is always a solution to $A\,x = 0$.

This is different in the homogeneous representation. There, $x = (0, \cdots, 0)$ is not a possible solution to $A\,x = 0$, since the vector of the parameters (which now is a part of x) is a constant, not a variable. I.e., we may not solve the system by setting a parameter n to 0, or, even more obvious, by setting the constant 1 to 0.

Properties of cones Since Definition 16 is in non-homogeneous representation, the origin of the coordinate system is always contained in a cone. The origin is even the only vertex of a cone.

This indicates a dual definition of cones, which is equivalent to Definition 16 [Sch86]: any cone can be characterized by its extremal rays $r_1, \ldots, r_s$ and its lines $l_1, \ldots, l_t$:

$$\mathcal{C} = \{x \,:\, x = \sum_{k=1}^{s} x_k r_k + \sum_{j=1}^{t} y_j l_j \;\wedge\; (\forall k : x_k \geq 0)\}. \tag{3.9}$$

There are well known algorithms for finding the rays and lines of a cone, and there is at least one efficient implementation, the Polylib [Wil93].

3.3.4 Counting integer points in parameterized polytopes

Frequently in this thesis, we are interested only in the integer points inside a polytope (cf. Remark 22), and at some places, we need to know how many there are.

Ehrhart polynomials Philippe Clauss has proposed a precise method for counting integer points inside a parameterized polytope using Ehrhart polynomials, a generalization of ordinary polynomials [Cla96]. The generalization is that the coefficients are *periodic numbers*, denoted

$$[c_0, \cdots, c_{p-1}]_n$$

for a parameter n. The value of such a periodic number is the entry c_i with $i = n$ modulo p.

We do not present Clauss' method in detail, but we give a brief overview based on an example; for any technical detail we refer to the original work [Cla96].

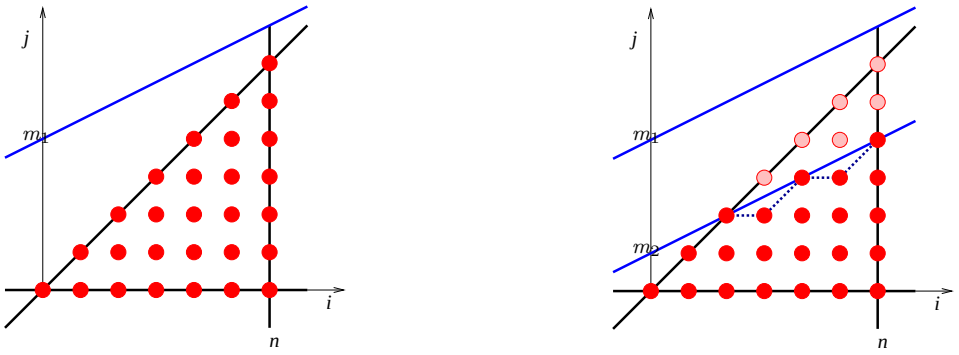


Figure 3.2: Different shapes due to different parameter values

Example 17. The three constraints $0 \leq j \leq i \leq n$ *define the triangle in Figure 3.2 (left). The additional constraint* $j \leq m + \frac{i}{2}$ *has no influence on the shape if* $m \geq \frac{n}{2} \geq 0$ *– as in the left part of the figure – or it cuts off the tip of the triangle if* $0 \leq m < \frac{n}{2}$ *– as in the right part.*

Shape of the polytope As one might expect, different shapes of a polytope, due to different parameter settings, lead to different Ehrhart polynomials. Hence, the first task when computing the number of integral points inside a parameterized polytope is to partition the parameter domain into regions that lead to different shapes of the polytope.

Example 18. When we give the constraints of Example 17 to the Polylib function that implements Clauss' method [Wil93], we obtain first a domain $2m \geq n \geq 0$ *(whose shape*

is the triangle), and the number of integer points inside the polytope is given by the polynomial $(\frac{1}{2}n^2 + \frac{3}{2}n + 1)$. *The second parameter domain* $0 \leq 2m \leq n$ *is considered in Example 19 below; there, the Ehrhart polynomial is different.*

Ragged bounds The necessity for periodic numbers as coefficients arises from polytope bounds that have an angle w.r.t. the coordinate axes that is not a multiple of 45°. If we follow the integer points inside the polytope that are closest to that bound, we obtain a ragged border (cf. the dotted line in Figure 3.2, right). Since these peaks are regularly spread, a periodic number can cover the different possible changes in the number of integer points.

Example 19. For the second parameter domain $0 \leq 2m \leq n$, *this case occurs: for increasing values of* n, *the height of the polytope changes iff we increment* n *from an odd to an even value. This leads to two polynomials for the number of points inside. These two polynomials can be expressed as one Ehrhart polynomial:*

$$\frac{1}{4}n^2 \;+\; (m{+}1)\,n \;+\; [(-m^2+m+1), (-m^2+m+\frac{3}{4})]_n$$

The value of the periodic number for the constant term is $-m^2+m+1$ *if* n *is even, and* $-m^2+m+\frac{3}{4}$ *if* n *is odd. In Figure 3.2, right, we depicted the polytope with* $m = 1$ *and* $n = 6$. *For these concrete numbers, the Ehrhart polynomial evaluates to* $9 + 12 + 1 = 22$. *If we decrease* n *to 5, we obtain* $\frac{25}{4} + 10 + \frac{3}{4} = 17$ *instead. It is guaranteed that the result is always an integer number.*

Computation of Ehrhart polynomials The computation of Ehrhart polynomials proceeds in two phases: the first phase partitions the parameter domain and computes the parametric vertices for every subdomain; the second phase computes the coefficients of the Ehrhart polynomials, i.e., the periodic numbers.

It is known that the degree of the Ehrhart polynomial is given by the dimensionality of the polytope, the dimension, i.e., the number of unknowns is given by the number of parameters, and the maximal period is given by the least common multiple of the denominators of the vertex coordinates. With this knowledge, we can craft a template for the Ehrhart polynomial in which all coefficients are unknowns. For the computation of its concrete coefficients, one sets the parameters to different values and, for every parameter setting, one counts the number of integer points in the resulting non-parametric polytope. With as many parameter settings as there are unknown coefficients in the template, one can determine the Ehrhart polynomial.

The cost of the first phase is determined mainly by the number of variables and parameters, and the number of bounds. The cost of the second phase depends heavily on the size of the integer coefficients in the polytope description.

3.3.5 Modulo functions and polyhedra

It is well known that modulo functions can be easily integrated into the linear algebra framework by expressing them as two-dimensional functions: one dimension represents the result of integer division, and the other the remainder.

If a variable i with $l \leq i \leq u$ shall be partitioned in blocks of size b, we formulate this by the following system of (in-)equalities, where d is the result of integer division and r is the remainder:

$$l \leq i \leq u \tag{3.10}$$
$$0 \leq r \leq b-1 \tag{3.11}$$
$$i = d * b + r \tag{3.12}$$

This system fully describes all variables of interest, including i. If desired, we can eliminate i by substituting Equation (3.12) in (3.10) and obtain two inequalities for the two variables d and r that, together, replace i.

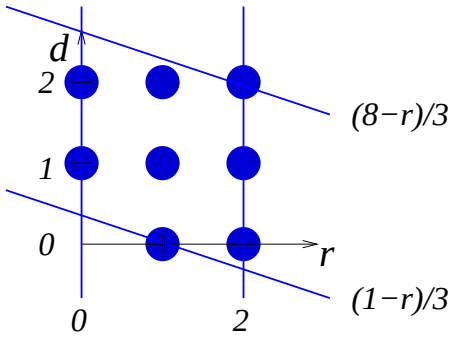


Figure 3.3: Modulo functions in the linear algebra framework

Example 20. Consider $1 \leq i \leq 8$*, which shall be partitioned in blocks of size* 3 *(Figure 3.3). We obtain the complete system*

$$1 \leq i \leq 8$$
$$0 \leq r \leq 2$$
$$i = 3 * d + r$$

Substitution and elimination leads to

$$0 \leq r \leq 2$$
$$\frac{1-r}{3} \leq d \leq \frac{8-r}{3}$$

Since d *is an integer, it must even satisfy*

$$\left\lceil \frac{1-r}{3} \right\rceil \leq d \leq \left\lfloor \frac{8-r}{3} \right\rfloor$$

Remark 21 (no modulo with parametric second argument). Note that the above system is linear only if b is given as a concrete numerical value; a symbolic constant leads out of this framework, since $d * b$ in Equation (3.12) is not a linear term, as we have already stated in Remark 8.

3.3.6 The mathematical model of a source program

Now, we are ready to present a more formal description of the parallelization framework introduced in Section 3.1. The first step of a parallelization in the polyhedron model is to transfer all necessary information contained in the program text to a mathematical, i.e., model-based description. Much of this information can be extracted directly from the program text.

Loops

The affine bounds of the loops surrounding a statement S can be read as affine inequalities and, consequently, expressed as a system of linear inequalities in homogeneous representation and, thus, represented as a $c \times d$ matrix D_S, where c is the number of affine bounds of the loops, and d is the dimensionality of the index set of S plus the number of symbolic parameters (i.e., $d = l + m$ as above). This matrix gives us a formal description of the index set (cf. Definition 3):

$$\textit{index set} \qquad D_S \begin{pmatrix} i \\ n \end{pmatrix} \geq 0 \tag{3.13}$$

where i is the iteration vector of S, and n is the vector of all structure parameters. (Aside: for consistency, we take care in our examples that the trivial inequality $1 \geq 0$ is always included in D_S; some mathematical tools, e.g., the Polylib [Wil93], require it.)

From this representation, we see that every index set is a polyhedron. Note that, in the case of non-affine loop bounds or dynamic control programs, the index set can, in general, only be a superset of the set of instances executed at runtime.

Since the original model requires all loop bounds to be affine expressions, the resulting index sets are all bounded, i.e., polytopes. Hence, the name "polytope model".

Remark 22 (linearly bounded lattices). To be precise, the index set (for loops with stride 1) is a $\mathbb{Z}$*-polyhedron* (or a $\mathbb{Z}$*-polytope*), which is the intersection of a polyhedron (polytope) and the integer lattice. If we additionally deal with affine transformations (as we do for parallelization), we obtain *linearly bounded lattices*, which have been studied in detail by Teich and Thiele [TT93]. However, in this thesis, we shall not focus on the arising complications since they can be postponed and solved during code generation (cf. Section 13.3). Thus, when clear from the context, we shall not distinguish between a polyhedron (polytope) and the integer points it contains.

Example 23. Let us come back to our example in Figure 2.1. The index set for statement T is given by $0 \leq i \leq n-1$ and $i+1 \leq j \leq n-1$. Hence, the matrix form D_T is

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

The columns represent the coefficients for i, j, n, and the constant 1, in this order; every row represents one inequality.

Analogously,

$$D_S = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & -1 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

where the columns represent the coefficients for i, j, k, n, and the constant 1, in this order.

Statements

Most of the time, we only consider assignments as statements. We also allow calls to side-effect free functions and side-effect free procedure calls, but we do no interprocedural analysis in this case. We just consider, e.g., a procedure call with several return arguments as a simultaneous assignment. Recursive calls are forbidden.

The program text of the statement is stored but not considered until target code generation; only the accesses to arrays in the statements have a correspondence in the model, as we shall see later in this section.

Loops as statements Note that loops are *not* considered statements.

However, this is different in the case of non-affine bounds. There, we use an affine approximation at compile time, i.e., an affinely bounded superset. To control the execution of the necessary iterations in the target code, we add some control statements (and some dependences between them). Hence, in this case, loops do lead to statements – but this is not the topic of this thesis.

Conditionals as statements A conditional consists of a boolean guard and a then and an optional else branch. Two types of conditionals must be distinguished.

1. Conditionals whose guards consist of affine (in-)equalities in the surrounding loop indices and structure parameters are not considered statements. Instead, for every

statement S in the then branch of the conditional, the affine constraints from the guard are added to the matrices representing the index set $\mathcal{I}$ of S. Note that this additionally restricted index set $\mathcal{I}$' is always a polyhedron if the guard is a conjunction of affine (in-)equalities.

For every statement S in the else branch of the conditional, we would like to add the negation of the affine constraints from the guard to the matrices representing the index set of S. However, if the guard contains conjunctions, the negation contains disjunctions, which, in general, leads to a non-convex set (cf. Figure 3.4).

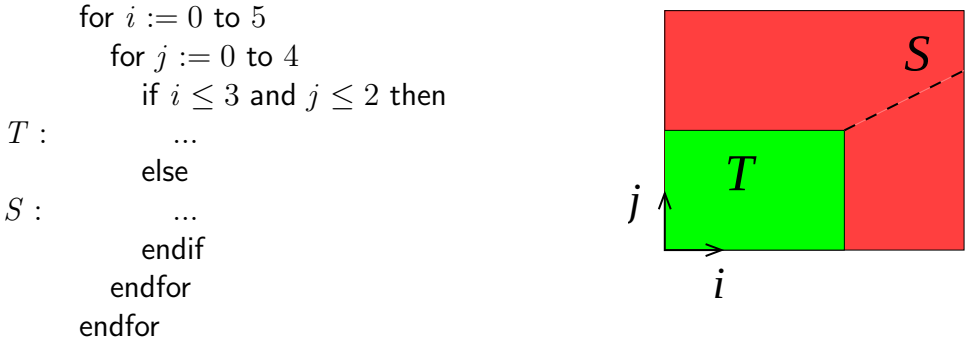


Figure 3.4: An else branch and its non-convex "index set"

In other words, the set of all iterations where the else branch is taken, cannot be described by one system of inequalities, i.e., by one index set.

A simple solution is to partition the "index set" of the else branch (more formally, the complement of the affinely guarded subset $\mathcal{I}$' w.r.t. the index set of the conditional), into disjoint polyhedra and to consider every such partition as a separate affinely guarded conditional (indicated by the dotted line in Figure 3.4).

The same solution is taken for the then branch if the guard contains disjunctions.

2. Non-affine guards cannot be modeled precisely. Hence, we approximate them conservatively. In this case, the guards are considered (especially marked) statements which compute the value of the conditional and store it in a new array C (each result in a separate cell, so as not to introduce new dependences).

   In addition, dependences from these statements to every statement in the body of the conditional are added. This guarantees that the guard is scheduled before its body statements.

   In target code generation, the statements in the body of a conditional are surrounded by new guards reading the appropriate values stored in C.

Array accesses

Let $a_{A,r}$ be the d_A-vector of array indices of an access to array A at some occurrence r in the program text, extended by entries for the parameters. We require all d_A indices to be affine expressions in the surrounding loop indices and structure parameters. Hence, the array access can be represented as an *access matrix* $\mathcal{A}_{A,r}$ of size $d_A \times d$:

$$\textit{access matrix} \qquad a_{A,r}(i,n) = \mathcal{A}_{A,r} \begin{pmatrix} i \\ n \end{pmatrix} \tag{3.14}$$

where d is the dimensionality of the index set of S plus the number of symbolic parameters, as above.

Every access to a variable is modeled by this access matrix, together with the identifier of the array, and the fact whether the access reads or writes the variable. (If it does both, as possible, e.g., in C notation, we consider it as two different accesses.)

Example 24. Let us continue our motivating example in Figure 2.1. In order to distinguish the occurrences of the seven array accesses, we denote them by the label of the statement, followed by a number that counts the array accesses in every statement from left to right, starting at 0. Then, we have the following access matrices, where the rows correspond to the first and second array dimension and to the homogeneous coordinates for n and the constant 1, in this order; the columns represent, for statement T, the coefficients for i, j, n, and the constant 1, in this order:

$$\mathcal{A}_{tmp,T0} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathcal{A}_{a,T1} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathcal{A}_{a,T2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For statement S, the columns represent the coefficients for i, j, k, n, and the constant 1, in this order:

$$\mathcal{A}_{a,S0} = \mathcal{A}_{a,S1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{A}_{tmp,S2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathcal{A}_{a,S3} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Dependences

The last step in crafting the model for a given source program is to compute and express the interaction between the operations in the index sets. This is the task of dependence analysis tools, and the subject of Chapter 5.

$$
\begin{array}{ll}
 & \mathsf{for}\ j := 0\ \mathsf{to}\ n \\
S: & \quad A[j] := A[j-3] \\
 & \mathsf{endfor}
\end{array}
$$

Figure 3.5: Non-integral schedule

3.3.7 Space-time mapping

As for the loop bounds, we also require that schedule and placements are affine in the surrounding loops' indices and structure parameters. We formalize them as follows.

Schedule

A *schedule* θ of a program is a function which maps every operation to an integer vector that represents logical time. For every statement S, its schedule θ_S can be represented by a $t \times d$ matrix Θ_S, where t is the number of time dimensions, and d is the dimensionality of the index set of S plus the number of symbolic parameters (i.e., $d = l+m$):

$$\textit{affine schedule} \qquad \theta_S(i,n) = \Theta_S \begin{pmatrix} i \\ n \end{pmatrix} \tag{3.15}$$

As long as we have no values for the entries of Θ_S, we call Θ_S a *template of a schedule*.

Example 25. For our motivating example, the templates for one-dimensional schedules are:

$$\Theta_T = \begin{pmatrix} t_T^i & t_T^j & t_T^n & t_T^1 \end{pmatrix} \quad \textit{and} \quad \Theta_S = \begin{pmatrix} t_S^i & t_S^j & t_S^k & t_S^n & t_S^1 \end{pmatrix}$$

Definition 26 (causality condition). A schedule must satisfy the following *causality condition*, which states the correlation between dependences and schedules:

$$\textit{causality condition} \qquad (\forall u, v \in \Omega : u \,\delta\, v \Rightarrow \theta(u) + 1 \leq \theta(v)) \tag{3.16}$$

Note that we can drop the condition that schedules must be integral: if θ satisfies the causality condition, then so does $\theta'(u) = \lfloor \theta(u) \rfloor$ [Qui87]. In fact, non-integral schedules do appear as results in current scheduling methods [Fea92a], and we interpret them as integral schedules by implicitly applying the floor function.

Example 27. Consider the program in Figure 3.5. The schedule by Feautrier's method [Fea92a] is $\theta_S(j,n) = j/3$, i.e., $\Theta_S = \begin{pmatrix} \frac{1}{3} & 0 & 0 \end{pmatrix}$. This means that we can execute three iterations simultaneously at every logical, integral time step: iterations 0,1,2 are executed at logical time 0, iterations 3,4,5 at time step 1, and so on.

Definition 28 (latency). For a schedule θ, the latency L of a program is defined as

$$\textit{latency} \qquad L = \max_{u \in \Omega} \theta(u) - \min_{u \in \Omega} \theta(u)$$

The latency can be interpreted either as the running time on a parallel computer with sufficiently many processors, or as the minimum number of synchronization points. Whatever the interpretation, it is easy to see that frequently – not always – one wants to minimize the latency.

Definition 29 (free schedule). A schedule which specifies that *every operation* is executed as early as allowed by the dependences is called *free schedule*.

Obviously the free schedule has minimal latency, but not every schedule with minimal latency is a free schedule.

Placements and space-time matrix

Computation placement Analogously to the schedule, a *computation placement* π is a function which maps every operation to an integer vector that represents a virtual processor. With the same affinity constraints, the placement of a statement S, π_S, can be represented by a $p \times d$ matrix Π_S, where p is the number of processor dimensions, and d is as above:

$$\textit{computation placement} \qquad \pi_S(i,n) = \Pi_S \begin{pmatrix} i \\ n \end{pmatrix} \tag{3.17}$$

For placement functions, we require that the dimensionality of the range, i.e., the number of processor dimensions is constant, i.e., independent of the statement. Note that this is not essential for model-based parallelization, but it respects the – in practice quite frequent – layout of the processors of a parallel computer as a grid of fixed dimension. All other topologies must be embedded in this grid.

Example 30. For our motivating example, the templates for one-dimensional placements are:

$$\Pi_T = \begin{pmatrix} p_T^i & p_T^j & p_T^n & p_T^1 \end{pmatrix} \quad \textit{and} \quad \Pi_S = \begin{pmatrix} p_S^i & p_S^j & p_S^k & p_S^n & p_S^1 \end{pmatrix}$$

Space-time mapping The combined matrix $\begin{pmatrix} \Theta_S \\ \Pi_S \end{pmatrix}$ is called *space-time matrix* and represents the space-time mapping. The homogeneous representation of this space-time matrix, as used in our implementation, is

$$\textit{space-time matrix} \qquad \mathcal{T}_S = \begin{pmatrix} \Theta_S \\ \Pi_S \\ \mathit{Zero}_{m,l} \;\; \mathit{Id}_m \end{pmatrix}. \tag{3.18}$$

Example 31. Let us come back to our motivating example in Figure 2.1. For the moment, we are not interested in how schedule and computation placement are computed; we just present the results of our implementation. For more detail we refer to Chapters 6 and 7 and to the literature [Fea92a, Fea92b, DV94, DV96b, Fea94, DR95].

For the schedule of T and S, we obtain $2*i$ and $2*i+1$, respectively. In matrix form:

$$\Theta_T = \begin{pmatrix} 2 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \Theta_S = \begin{pmatrix} 2 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Note that the interpretation of the columns is as in Example 23.

This schedule specifies that all j iterations (and also all k iterations of statement S) can be executed in parallel. The i iterations of T and S are mutually interleaved.

For the computation placement, we must first decide the dimensionality of our processor grid. For a 2-dimensional grid, Feautrier's algorithm [Fea94] computes (j,i) and (j,k) as placements for statements T and S, respectively. Formally:

$$\Pi_T = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \Pi_S = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Hence, the space-time matrices are

$$\mathcal{T}_T = \left(\begin{array}{cc|cc} 2 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right), \text{ and } \mathcal{T}_S = \left(\begin{array}{ccc|cc} 2 & 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right). \tag{3.19}$$

Remark 32 (dimensionality considerations). It is important to realize that, without loss of generality, the placements and the schedules have the same dimensionalities for all statements:

- For the placement, this is guaranteed by the placement method, since, in a d-dimensional processor space, every placement must determine the processor coordinates in all d dimensions.

- The schedule may, a priori, have different dimensionalities for different statements. But a schedule with insufficient dimensionality may always be extended by suitable coordinates: for every time dimension i determine its minimal value m_i (this exists and is finite) and set the time coordinate to $m_i - 1$ for all those schedules in which time dimension i is not specified explicitly.

- If the dimensionality of the space-time mapping (the combined schedule and placement) is less than the dimensionality of the index set, we can either integrate the missing index set dimensions in the model by increasing the dimensionality of all schedules in the program as just described, or we can exclude these dimensions from the model by considering the according loops as a textual part of the statement. The latter approach may be a practical alternative to tiling (cf. Chapters 12 and 14), but it is not the default in our proposed model-oriented parallelization framework. Further aspects are discussed in Section 13.1.1.

In other words: after space-time mapping, all instances of all statements can be viewed as being embedded in a perfect nest of time and space loops. Note that this is true regardless of whether the target program is indeed perfectly nested or not. The latter depends on the quality of the applied code generation algorithm.

Data placement Similarly to the computation placement, a *data placement* maps array elements to virtual processors. For each array A, we express this placement as:

$$\textit{data placement} \qquad \pi_A(a, n) = \Pi_A \begin{pmatrix} a \\ n \end{pmatrix} \tag{3.20}$$

where a is the l'-vector of the subscripts of array A, Π_A is a $p \times d_A$ matrix, with $d_A = l'+m$, and n, m, and p are as above.

Example 33. For our motivating example, the templates for one-dimensional data placements are:

$$\Pi_{tmp} = \begin{pmatrix} p_{tmp}^{1st} & p_{tmp}^{2nd} & p_{tmp}^{n} & p_{tmp}^{1} \end{pmatrix} \quad \textit{and} \quad \Pi_a = \begin{pmatrix} p_a^{1st} & p_a^{2nd} & p_a^{n} & p_a^{1} \end{pmatrix}$$

Definition 34 (owner). If a data placement is given, then the processor to which an array element is mapped, is called *owner of the array element*. I.e., with the definitions above: $\pi_A(a, n)$ is the *owner of* $A[a]$.

Remark 35 (implicit data or computation placement). Note that, instead of specifying separate placements for computations and data, they could also be defined implicitly by each other.

- **Owner computes rule** The well known *owner computes rule* (*OCR*), e.g., used in HPF [KLS+94], assumes a data placement and, based on it, determines the computation placement according to the rule that only the owner of an array cell computes the values for this cell. Thus, we have

  $$\pi_S(i, n) = \pi_A(a_{A,r}(i, n)),$$

  where r is the occurrence of the write access to array A in statement S. Note that, in this case, every data item has a fixed owner $\pi_A(a, n)$.

- **Computer owns rule** Symmetrically, we can define a *computer owns rule*, which starts from a computation placement and stores every value at the processor that computed it. In this context, if a value is recomputed on different processors, the "owner" of the array cell changes implicitly during program execution. We have a similar equality as before, but now, the data placement may depend on the iteration vector:

  $$\pi_S(i, n) = \pi_{A,i,n}(a_{A,r}(i, n)).$$

Remark 36 (communication basis). Note that the data placement determines the communications of the target program:

- **ownership-driven communication** If every data item resides on a qualified owner, then the communications must take place between the owner on the one hand and the producer or consumer of the value on the other hand, for a write or a read access, respectively. We call this approach to determining communications *ownership-driven*. This is the most prominent approach because it is the basis of the data-parallel programming method.

  A typical language of this approach is HPF, which is, *additionally*, based on the owner computes rule (Remark 35), i.e., the computation placement is specified implicitly by the data placement. Since, with that convention, only the owner of a data item computes the value, there is never a communication from the producer to the owner.

  In general, the ownership-driven approach does *not* require the owner computes rule.

- **dependence-driven communication** In the *dependence-driven approach*, the data are sent directly from the producer to the consumer. Hence, the notion of ownership becomes useless. Instead, this approach is *always* based on the computer owns rule (Remark 35).

Remark 37 (types of placement algorithms). In analogy to these two ways of determining communications, we have two ways to compute a placement, because the main goal of a placement is to minimize communications (under the constraint of generating sufficiently much parallelism):

- if the placement algorithm computes an explicit data placement, we call it an *ownership-driven placement algorithm*;
- if the data placement is implicitly given due to the computer owns rule, we call it a *dependence-driven placement algorithm*.

Comparison of the two types of placement algorithms If some array element $A[x]$ is reassigned, the new (possibly different) producer P holds the new value in the dependence-driven approach, i.e., P becomes the new temporary "owner" of $A[x]$. Since in this context, the "ownership" for an array element can change freely, the dependence-driven approach is, in this aspect, more flexible than the ownership-driven approach.

On the other hand, in the dependence-driven approach, the implicit owner is always the computer of the value; we have no possibility to define a different processor as owner, even if this would be beneficial. So, in this aspect the ownership-driven approach is more flexible [Fea00].

Definition 38 (cutting dependences). In the dependence-driven approach, we say that a dependence is *cut*, if the dependent operations are mapped to the same processor [Fea94].

3.3.8 Code generation

When switching from the model back to the code level, we must scan the target (i.e., space-time mapped) index sets, i.e., we must enumerate all points inside. For this purpose, we again want to represent the target index sets as polyhedra.

The first method for this purpose has been proposed by Ancourt and Irigoin [AI91]. Chamski extends this work by using imperfectly nested loops in order to reduce run-time overhead [Cha92]. Further approaches are considered in more detail in Section 13.3.

Definition 39 (unimodular). A matrix is *unimodular* if its determinant is ± 1 (consequently, the matrix must be square and of full rank), and all entries of the matrix are integer values.

Target index sets

If the space-time matrix $\mathcal{T}_S$ is unimodular, the target index set for every statement can easily be described in terms of the space and time variables: from Equation (3.13), we derive

$$D_S \left(\mathcal{T}_S^{-1} \mathcal{T}_S\right) \begin{pmatrix} i \\ n \end{pmatrix} \geq 0.$$

Since matrix multiplication is associative, we regroup and obtain

$$\textit{target index set} \qquad D'_S \begin{pmatrix} t \\ p \\ n \end{pmatrix} \geq 0, \tag{3.21}$$

where

$$D'_S = D_S \, \mathcal{T}_S^{-1} \tag{3.22}$$

and t and p are the time and space coordinates, respectively.

For the case of a space-time matrix $\mathcal{T}_S$ which is not unimodular but still square and of full rank, several authors have published solutions in the early Nineties [Ram92, Dar94, KPR94, LP94, Xue94, Ram95]. A non-unimodular matrix $\mathcal{T}_S$ can represent, among other transformations, a scaling of the target index set, which typically leads to regularly spread holes in the $\mathbb{Z}$-polyhedron. The idea is that non-unit strides can be used to skip the holes.

Example 40. Consider again our running example. The space-time matrix for S is shown in Equation (3.19). It is not unimodular: its determinant is 2. The inverse of $\mathcal{T}_S$ is

$$\mathcal{T}_S^{-1} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Hence,

$$D'_S = D_S \, \mathcal{T}_S^{-1} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & 0 & 1 & -\frac{1}{2} \\ -\frac{1}{2} & 1 & 0 & 0 & -\frac{1}{2} \\ 0 & -1 & 0 & 1 & -1 \\ -\frac{1}{2} & 0 & 1 & 0 & -\frac{1}{2} \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

I.e., in a more readable form, we have the following six non-trivial inequalities for the target index set, which we can read from the first six rows of D'_S:

1.:	$t \geq 1$	*3.:*	$p_1 \geq \frac{t}{2} + \frac{1}{2}$	*5.:*	$p_2 \geq \frac{t}{2} + \frac{1}{2}$
2.:	$t \leq 2 * n - 1$	*4.:*	$p_1 \leq n - 1$	*6.:*	$p_2 \leq n$

The corresponding target polytope is depicted in Figure 3.6. The left part shows how the polytope is constructed; the number at every border represents the inequality number (equivalent with the row number in D'_S) that describes this border. The right part shows the real index set.

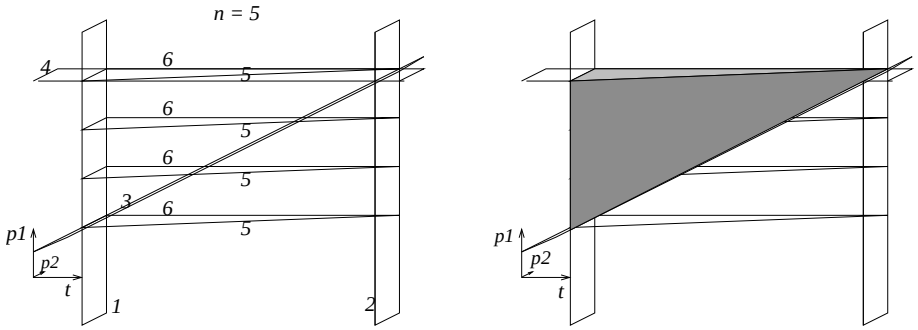


Figure 3.6: Target index set in real space

In order to illustrate the holes, we have depicted the operations that must be executed in the source and the target program as bullets in Figure 3.7.

It is easy to see that the source index set in the left part is dense, whereas the target index set at the right hand side consists only of holes for every second t index – as we could also expect from the schedule. Graphically, the space-time mapping corresponds, roughly spoken, to stretching the horizontal axes by a factor of 2. Hence, with a stride of 2 in the t dimension we can skip these holes.

The position of the origin of the coordinate system also shows that t starts at 1, in contrast to the source index i which starts at 0. Thus, the loop for t must start at 1 and iterate with stride 2.

The basic mathematical approach in the various publications on this subject is as follows [Ram92, Dar94, KPR94, LP94, Xue94, Ram95]: $\mathcal{T}_S$ is transformed into its *Hermite normal*

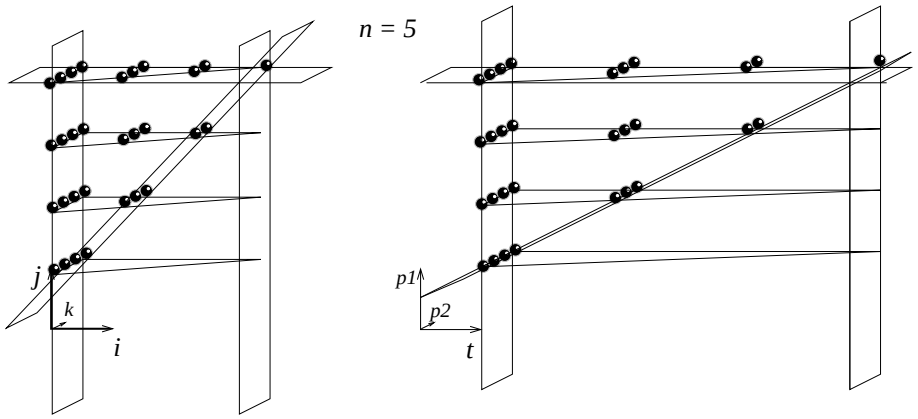


Figure 3.7: A non-unimodular space-time mapping

form [NW88], whose diagonal entries represent the strides of the target loops, and whose other entries can be combined to adjust the offset for the lower bound of the loops. A more detailed technical description of the method, including further extensions – e.g., how to deal with non-integer coefficients in the schedule – can also be found in Wetzel's diploma thesis [Wet95].

If the space-time matrix has not full rank, we refer to Section 13.1.1 for a possible solution. This situation occurs, e.g., for statement T in our running example (cf. Equation (3.19)).

Loop generation

Once the target index set of a statement is given as a polytope (as in Equation (3.21)), the loop nest scanning this polytope can be generated using some projection method: the central requirement is that the bounds of outer loops must not depend on the indices of inner loops.

The elimination method of Fourier-Motzkin [Ban93, Sch86] is one well-known technique that eliminates variables from a system of inequalities. Its doubly exponential time complexity is still acceptable since the number of dimensions (i.e., the depth of the loop nest) is typically a very small number. Note, further, that Fourier-Motzkin elimination works on real polyhedra but not on $\mathbb{Z}$-polyhedra. Since we know that our indices are integer values, we can apply implicitly floor and ceiling functions for the upper and lower bounds, respectively. But this is not sufficient to adapt Fourier-Motzkin to $\mathbb{Z}$-polyhedra. E.g., the method cannot recognize that a $\mathbb{Z}$-polytope is empty if the real polytope contains rational but no integral points. As a consequence, a loop with empty range might be generated. Alternative methods for generating loops are presented in Section 13.3.

Synchronous and asynchronous parallelism Note that the order of the target loops can be chosen freely, thus, leading to *synchronous* (*horizontal*, in terms of Banerjee

[Ban93]) or *asynchronous programs*, depending on whether the outer loops enumerate the time or space dimensions, respectively.

Note further that our asynchronous programs still allow synchronization and/or communication between the processors – in contrast to Banerjee's *vertical parallelism* [Ban94]. Typically, non-blocking sends and blocking receives are used to insert the necessary synchronizations. This difference also explains why we can always generate asynchronous programs, whereas vertical parallelism in the sense of Banerjee only exists if the dependences satisfy very strong constraints.

Example 41. Let us now assume that we want to generate an asynchronous target loop nest for statement S of our running example. In this case, we must eliminate the variable t in the bounds of p_1 and p_2, since the sequential loop on t is nested inside the two parallel loops, and, consequently, the loop index t must not occur in the borders of p_1 or p_2.

For this purpose, we now apply Fourier-Motzkin elimination. We first rewrite the six inequalities in Example 40, such that the variable to be eliminated, i.e., t, is isolated (if it occurs in the inequality):

1.:	$t \geq 1$	*3.:*	$2 * p_1 - 1 \geq t$	*5.:*	$2 * p_2 - 1 \geq t$
2.:	$t \leq 2 * n - 1$	*4.:*	$p_1 \leq n - 1$	*6.:*	$p_2 \leq n$

The inequalities that contain t can be interpreted as the bounds of the loop on t.

We then eliminate t by writing all pairs of lower and upper bounds for t:

1./2.:	$1 \leq 2 * n - 1$	*4.:*	$p_1 \leq n - 1$
1./3.:	$1 \leq 2 * p_1 - 1$	*6.:*	$p_2 \leq n$
1./5.:	$1 \leq 2 * p_2 - 1$		

Now, we have a system without variable t and we can recursively eliminate p_2.

Fortunately, there is no inequality that contains both, p_1 and p_2. Therefore, we isolate p_1 and p_2 simultaneously:

1./2.':	$1 \leq n$	*4.:*	$p_1 \leq n - 1$
1./3.':	$1 \leq p_1$	*6.:*	$p_2 \leq n$
1./5.':	$1 \leq p_2$		

From the previous calculations, we can directly write down the asynchronous loop nest for S (stride 2 for the t loop is inserted based on the above considerations):

```
          for p1 := 1 to n−1
            for p2 := 1 to n
              for t := 1 to min(2 * n − 1, 2 * p1 − 1, 2 * p2 − 1) step 2
S':             ...
              endfor
            endfor
          endfor
```

Array indices in the target program As just seen, the target loop nest does not contain the indices of the source loops. However, they still occur in the array indices. Fortunately, in the case of regular space-time matrices, the source loop indices are easily computed from the target loop indices by applying $\mathcal{T}_S^{-1}$. Since

$$\textit{source to target} \qquad \begin{pmatrix} t \\ p \\ n \end{pmatrix} = \mathcal{T}_S \begin{pmatrix} i \\ n \end{pmatrix}, \tag{3.23}$$

where i is the source index vector, t and p are the target index vectors for sequential and parallel dimensions, respectively, and n is the parameter vector (including the constant 1), we have

$$\textit{target to source} \qquad \mathcal{T}_S^{-1} \begin{pmatrix} t \\ p \\ n \end{pmatrix} = \begin{pmatrix} i \\ n \end{pmatrix}. \tag{3.24}$$

Every row of $\mathcal{T}_S^{-1}$ corresponds to one source index (or parameter), and every column contains the coefficients of one target index (or parameter).

Example 42. We can read the first three rows of $\mathcal{T}_S^{-1}$ from Example 40 as follows:

- $i = \frac{t-1}{2}$
- $j = p_1$
- $k = p_2$

Hence, statement S in the loop body is rewritten as

$$S\text{':} \qquad a[p_1, p_2] := a[p_1, p_2] - tmp[\tfrac{t-1}{2}, p_1] * a[\tfrac{t-1}{2}, p_2]$$

Merging index sets

So far, we have considered only one statement. Another challenge is to generate code that efficiently enumerates the (usually non-convex) union U of target index sets of different statements. More precisely, we must scan polyhedra $\mathcal{P}_1, \cdots, \mathcal{P}_k$, where k is the number of statements in the target program.

A naïve method is to enumerate a single polyhedron $\mathcal{P}$ that contains $\mathcal{P}_1, \cdots, \mathcal{P}_k$ [GLW98]. A typical choice for $\mathcal{P}$ is the convex or the rectangular hull of $\mathcal{P}_1, \cdots, \mathcal{P}_k$. In the body of the loop nest scanning $\mathcal{P}$, we add a guard for every statement i which restricts $\mathcal{P}$ to $\mathcal{P}_i$, for $1 \leq i \leq k$. Our performance experiments showed that the repeated evaluation of the guards is an inacceptably high control overhead: every guard typically consists of at least two tests per dimension of $\mathcal{P}$.

Several more sophisticated code generation algorithms focus on this problem [QRW00, KPR94, Col94, Bas03]. They result in the imperfectly nested target programs already mentioned. This aspect of code generation is treated in more detail in Section 13.3.

Chapter 4

Overview: all modules of a parallelizing compiler

After the presentation of the basics of the polytope model, we can now visit selected topics in more detail. This also means that, from this point on, we focus on the description of our own methods and, thus, not all parallelization phases are presented in equal detail. In order to get an overview of the methods described in the remainder of this thesis, we briefly describe all phases of a parallelizing compiler and indicate which of them are treated in more detail in which parts of this thesis. Thereby, we shall see that we contribute research results to every parallelization phase (where the contribution to scheduling is only indirect: we do not develop our own scheduling algorithm since elaborate methods exist in the literature [Fea92a, Fea92b, DV97], but we improve their power by a preceding refinement step; cf. Chapter 6).

4.1 LooPo

This thesis presents various methods to be used in a parallelizing compiler. In order to see whether these methods, which we derive on a theoretical basis, behave as expected in practical use, we implement them in a loop parallelizer called *LooPo*. LooPo is a source-to-source compiler, in development at the University of Passau, which integrates methods for all parallelization phases.

For the mentioned purpose, flexibility and easy adaptivity are more important aspects than having a complete, automatic, efficient, and user-friendly tool. Therefore, LooPo is designed as a prototypic system, which means that not everything runs fully automatic, even if this would be possible from the theoretical point of view, or the implementations of some modules require harder input constraints than the underlying theory, but the design allows to easily integrate new methods.

Its modular structure also enables us to have several methods implemented for every parallelization phase and, thus, it allows to compare different methods which have the same purpose. Furthermore, we can observe easily the effects of a new method on all subsequent

parallelization phases, i.e., we can see which other methods benefit (or suffer) most from a new method.

Beyond all methods developed in this thesis, LooPo consists of

- a scanner and parser for a subset of C and Fortran,
- two methods for dependence analysis [Ban97, Fea91],
- three scheduling methods [Lam74, Fea92b, DV94],
- two placement algorithms [Fea94, DR95],
- an extension to deal with dynamic control programs [Gei97],
- a limited implementation of a tiling method [Xue97a],
- a graphical display tool [Wüs97], and
- a lot of auxiliary tools.

Additional information about LooPo is available in the first announcement [GL97], in a dissertation [Gri97], or in the web [Leh].

In this thesis, we use LooPo

- to compute schedules for all our examples, since here, we do not present a scheduling algorithm explicitly,
- to skip occasionally the computation of some parallelization phases if details are irrelevant but the result is needed for the method to be explained,
- to illustrate how a method can be implemented,
- to assist us in the generation of the target code for the performance experiments,
- to evaluate the practical use of a derived method, and
- to explain the structure of this thesis (cf. Section 4.2).

4.2 The interplay of parallelization methods

We can use the structure of LooPo to demonstrate in which order the different parallelization methods should be applied in a parallelizing compiler, and how these methods work together. Note that the chapters of this thesis are ordered the same way.

Figure 4.1 depicts in which order the modules should be traversed. The numbers in the node labels represent the number of the section that describes the method; the text is just some mnemonic label, typically the name of the module in LooPo.

Figure 4.1 consists of three rows, which represent the classical three phases of parallelization in the polytope model:

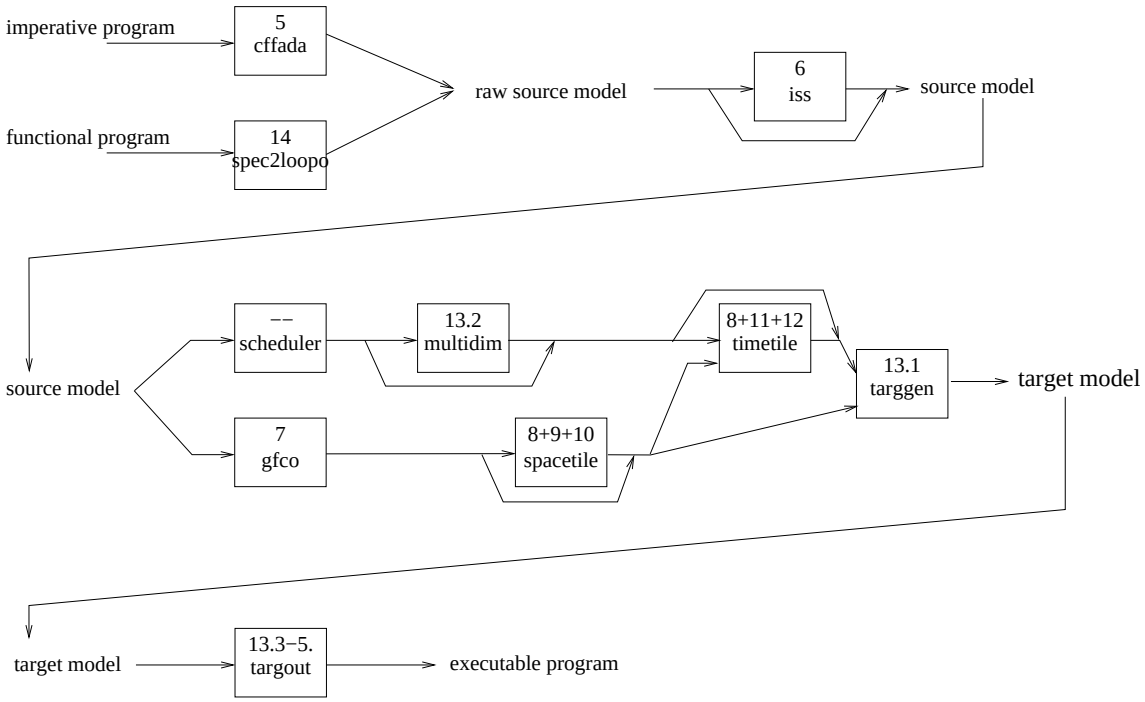


Figure 4.1: Control flow through the compiler

1. the generation of the mathematical, i.e., model-based description of the source program,
2. the model-based parallelization, and
3. the generation of a parallel program from the model-based description after the parallelization.

4.2.1 Generation of the model-based description

Input (cffada, spec2loopo)

First, we can see that there are two input arrows. It means that we can either give an imperative loop program to LooPo, which is the already described standard case, or we can give a possibly mutual recursive specification to LooPo. The second input variant is, on the one hand, more flexible in the sense that the input does not specify an execution order of the operations and, on the other hand, more restrictive as it does not allow reassignments. We come back to this second input variant in Chapter 14.

Until that point, we focus on the traditional case, which takes loop programs in C or Fortran notation (where we require that, also in the C notation, the loop index and the variables that appear in the loop bounds are not modified in the loop body, i.e., the number of iterations of the loop is known when the loop starts iterating). LooPo accepts both,

static and dynamic control programs (cf. Section 3.1); the underlying theory for dynamic control programs is partly described in this thesis (the dependence analysis module in Section 5.2), and mainly in the precursing PhD thesis [Gri97].

The most difficult part of the input module is to compute the set of all dependences of an imperative loop program. Chapter 5 presents the basics of dependence analysis and illustrates a method that is applicable for dynamic control programs – then returning approximate results – but that is precise for static control programs, i.e., that finds all existing dependences and returns no non-existing dependence.

Improvement (iss)

As we can see in Figure 4.1, the model-based description can be used directly as the source model, or it can be optionally refined. This refinement is especially helpful in the presence of non-uniform dependences: it can improve the quality of schedule and/or placement significantly. The idea is to partition the index sets of the statements into pieces that are more homogeneous w.r.t. the dependences than the original index sets are; a detailed description is given in Chapter 6. In LooPo, the user controls whether to apply this refinement or not.

4.2.2 Determining parallelism

The second row in Figure 4.1 is drawn in two different levels: the lower level deals with the space mapping, and the upper level with the time mapping; the last module is in between as it simultaneously deals with both.

Schedule (schedule)

We have not derived our own scheduling algorithm because we obtained very satisfactory results with the methods by Feautrier [Fea92a, Fea92b] and by Darte and Vivien [DV94, DV97]. Especially the availability of both methods in LooPo is beneficial, since Feautrier's method on the one hand is more powerful but, on the other hand, its execution time is longer and the resulting schedules occasionally have irregularities, even for uniform dependences, that can be avoided if one is not interested in the free schedule (Definition 29) but in minimal latency only (as is often the case).

Multi-dimensional schedules (multidim) In order to simplify the subsequent code generation, it can be profitable to rewrite a given schedule syntactically. Since this does not modify the schedule but can be considered as a preprocessing of target code generation, we present the method in the chapter dealing with code generation.

The idea is that the coefficients of the schedule are simplified, e.g., they become 1, at the price of increasing the dimensionality of the schedule. Frequently, this helps to avoid control overhead at run time; Section 13.2 points out when this rewriting useful, and how it can be done. In LooPo, the rewriting is integrated as a user-controlled option.

Placement (gfco)

Similarly to the schedule, there also exist good placement methods in the literature, e.g., one by Feautrier [Fea94], or one by Dion and Robert [DR95]. Anyway, in Chapter 7, we derive our own placement algorithm, denoted gfco in Figure 4.1. The reason is that we require an additional constraint from the placement. This constraint is, in principle, optional, but it largely extends the applicability of a subsequent parallelization phase (named time tiling below) that can be essential for achieving performance (cf. Chapter 11). LooPo contains the placement methods by Feautrier and by Dion and Robert, and also a prototypic implementation of our own placement algorithm.

At this point, we have extracted as much parallelism as our methods are able to find. In the remaining modules of this phase, we reduce the existing parallelism to a practically useful amount, i.e., we search for the right balance between control and communication overhead on the one hand, and ignoring parallelism on the other hand. Note that this procedure conforms to Foster's approach, who also suggests to first extract all parallelism and then limit it according to the parallel architecture [Fos95]. Anyway, this is still *not* the traditional setting in parallelizing compilation. Chapter 8 discusses our decision in detail.

Space tiling (spacetile)

According to our procedure, the next step after the computation of a placement is to reduce the number of processors and, at the same time, to reduce the number of communications as far as possible. The idea is to aggregate several virtual processors to one physical processor. Chapter 9 aims at a communication-minimal solution for uniform dependences, and Chapter 10 adds some heuristic extensions for the non-uniform case.

If we are interested in very fine-grained parallelism, we can skip the space tiling phase, but for today's parallel architectures this phase is usually necessary.

Time tiling (timetile)

If the parallelism is still too fine-grained after the space tiling, we also can aggregate time steps. This possibly surprising task reduces the number of communication phases. Chapter 11 explains this idea in detail and derives a way to compute the optimal number of time steps to aggregate.

Chapter 12 then discusses whether tiling time is necessary. It turns out that there is a relationship between the number of dimensions in space, computed by the placement algorithm, the dimensionality of the index set, and the necessity of tiling time.

Generating the target model (targgen)

The results of the distribution in space – with or without space tiling – and the distribution in time – with or without time tiling – are then combined to the space-time mapping (cf. Section 3.3.7). But even without any tiling, there is a technical difficulty: since schedule

and placement can be computed independently (cf. Figure 4.1), the two affine functions might be linearly dependent or might, together, have a dimensionality that is different from the dimensionality of the index set. Hence, the resulting space-time matrix (cf. Equation (3.18)) might be not invertible – which would be necessary for computing the target index set (cf. Equation (3.22)). A possible solution to this problem is derived in Section 13.1. The implementation is integrated in LooPo and cannot be switched off by the user since the proper treatment of non-invertible space-time matrices is mandatory for generating the target loop nest in LooPo.

4.2.3 Generation of the target program

Code generation (targout)

The biggest challenge in the last step is to write a single (typically imperfect) loop nest that enumerates all target operations, i.e., the target index sets of all statements simultaneously without introducing too much control overhead. Section 13.3 briefly summarizes our practical experience with some approaches to this topic.

Another important task in this context is to generate communication code. Section 13.4 discusses which data must be transferred, and how the code can be generated so that, between any two physical processors, there is at most one communication per abstract time step containing all data to be transferred. A proper implementation of this topic in LooPo is under way.

Finally, Section 13.5 presents some remarks on the choice of the language for the target program.

Chapter 5

Preparation: dependence analysis

The most interesting part of crafting the model is the computation of the dependences. In the next sections, we shall first briefly review some basic terminology and then summarize a practical algorithm to compute dependences.

5.1 General dependence analysis

5.1.1 Basic terminology

Intuitively, a dependence exists when some computed value is used or, in the imperative setting, also overwritten. Here is a more precise definition (cf. [Fea91]):

Definition 43 (dependence). A *dependence* from an operation s to an operation t exists if the following constraints hold:

conflict: s and t must reference the same memory cell;

existence: s and t must exist, i.e., they must be executed by the program;

order: s must be executed before t;

write (optional): at least one of s or t must modify the memory cell;

optimization (optional): there must not be an operation r between s and t modifying the same memory cell.

We call s the *source* and t the *destination of the dependence*. We denote a dependence by $s \longrightarrow t$.

Depending on the type of access (read or write) in s and t we have the following naming convention for dependences:

	source (s)	
destination (t)	reads	writes
reads	input	true
writes	anti	output

If the *write* constraint holds, we have a dependence in the sense of Bernstein [Ber66]. The idea behind this constraint is that we usually ignore read-read, i.e., input dependences since the ordering of two reads does not matter. Note that this implies that, at the model level, concurrent reads are assumed to be possible.

If the *optimization* constraint holds, we call the dependence *direct* (cf. [Bra88]).

If a true dependence is direct, we call it *flow dependence*, since it models the data flow in the program.

Aggregating dependences

Note that this definition refers to operations, i.e., to instances of statements, which is the granularity we need.

Example 44. Let us come back to our motivating example in Figure 2.1. Obviously, operation $\langle 0,1;T\rangle$ *writes a value to* $tmp[0,1]$ *which is read by the operation executed next in the sequential program, namely by* $\langle 0,1,1;S\rangle$*. Hence, we have a flow dependence* $\langle 0,1;T\rangle \longrightarrow \langle 0,1,1;S\rangle$*.*

Since $tmp[0,1]$ *is not overwritten as long as the loop on* k *iterates, and since* $tmp[0,1]$ *is read at every* k *iteration, there is also a flow dependence from* $\langle 0,1;T\rangle$ *to every operation* $\langle 0,1,k;S\rangle$ *with* $1 \leq k \leq n$*.*

Of course, we do not want to denote every operation to operation dependence separately. Note that this is even impossible, because the number of dependences may depend on parameters and therefore is unknown at compile time. Instead, we would like to aggregate dependences to parametric sets of dependences, i.e., to parametric *dependence relations*.

Example 45. Continuing our considerations from Example 44, we see that the same type of flow dependence exists for every combination of i *and* j*, not only for* $i = 0$ *and* $j = 1$*. Hence, we can aggregate further and, thus, obtain the following dependence relation:*

$$\{\langle i,j;T\rangle \longrightarrow \langle i,j,k;S\rangle \;:\; 0 \leq i \leq n-1 \wedge i+1 \leq j \leq n-1 \wedge i+1 \leq k \leq n\}$$

In the literature, these dependence relations are often called *dependences* as well. In the rest of this thesis, we follow this convention and, only if necessary, we distinguish between *dependence relation* and *dependence instance*.

5.1.2 A brief overview on dependence analysis

The conflict constraint in the definition usually is a set of equalities which result from the array indices; the existence constraint corresponds to an inequality system that is given by the index sets. The representation of the order constraints depends on the algorithm that computes the dependences.

Many algorithms have been proposed for computing dependences (cf. [ASU86, Ban97, Fea91, CG99]). Some are just tests that conservatively approximate whether an assumed

dependence really exists, not even considering all constraints of the definition (cf. [ASU86, Ban97]); others compute the precise flow of the data [Fea91]; some allow arbitrary control flow [ASU86, CG99], some are tailored for loop programs [Ban97], or even for static control programs [Fea91]. Before we introduce an algorithm that tries to combine the best of the properties just mentioned, we need to review briefly the precision levels of dependences.

5.1.3 Abstraction levels of dependences

h-transformations

The most precise description of dependences is the *h-transformation*: it is an affine function that takes a destination operation t and yields the source operation s [Fea91]. The domain of the function is denoted explicitly.

Example 46. The dependence of Example 45 can be written as h-transformation as follows: $h_{S2}(\langle i,j,k;S\rangle) = \langle i,j;T\rangle$, *for* $0 \leq i \leq n-1 \wedge i+1 \leq j \leq n-1 \wedge i+1 \leq k \leq n$, *where the index of the h-transformation denotes the occurrence of the read access under consideration (cf. Example 24).*

Distance vectors

If we are not interested in the dependent operations but only in the dependence distances, we can take the difference of the iteration vectors of t and s for their commonly surrounding loops. If this difference is constant for all dependent operations, we call the dependence *uniform w.r.t. commonly surrounding loops*, and the difference *distance vector*. If the loop nest is perfectly nested, then all loops are commonly surrounding both statements. In this case, and if the distance vector contains no parameters, we call the dependence *uniform*.

Example 47. In our example, the commonly surrounding loops are the loops on i *and* j. *The iteration vector of the source is* (i,j), *and the iteration vector of the destination, projected to the* i *and* j *dimensions, is also* (i,j). *Hence, the difference is* $(0,0)$ *for any values of* i *and* j, *and we have a uniform dependence w.r.t. the outer two loops, with distance vector (0,0).*

Dependence polyhedron

If the difference is not constant, we could try to list all possible dependences, but since this list is of unbounded length, we must, as before, use a parametric description: a polyhedron that subsumes all possible distances (and encloses them as tightly as possible). This is called the *dependence polyhedron*.

Example 48. In order to find a non-uniform dependence, let us again look at the code in Figure 2.1. Consider the read of $a[i,k]$ *in statement* S. *Which operation has written most recently to that array cell? The answer is given by the h-transformation:* $h_{S3}(\langle i,j,k;S\rangle) =$

$\langle i-1, i, k; S\rangle$, *with* $i \geq 1 \wedge j \geq i+1 \wedge j \leq n-1$ *and some other constraints. (Note that, for the moment, it is not important how this h-transformation is computed; an algorithm is presented in Section 5.2.)*

Starting from this h-transformation, we subtract the iteration vectors from the destination and the source, and we obtain the vector $(1, j-i, 0)$. *Since we know the bounds for* i *and* j *in the h-transformation, we know that* $j-i \geq 1$ *and* $j-i \leq n-2$. *Hence the dependence polyhedron is the set* $\{(1, x, 0) : 1 \leq x \leq n-2\}$.

Generalized distance vectors and direction vectors

From the dependence polyhedron we can abstract further in order to get rid of the parameters and have a description which is nearly as simple as distance vectors. For that purpose, we eliminate all parameterized bounds from the polyhedron and project it to every dimension separately. The result is a vector whose entries consist of a constant number (minimal and/or maximal value in that dimension), and an indicator $+$ or $-$ whether larger or smaller values can exist in that dimension, respectively. If there is no upper nor lower bound, the entry $*$ is used. We call this vector the *generalized distance vector*.

Example 49. The only parametric bound in Example 48 is the upper bound on x, *which we eliminate. Thus, we only keep the knowledge that* $x \geq 1$, *which leads to the generalized distance vector* $(1, 1+, 0)$.

From the distance vector or the generalized distance vector, we can further abstract and save only the possible signs for every component. The resulting vector is called *direction vector*. Instead of $+1$, 0, and -1, we can also find in the literature $<$, $=$, and $>$, respectively (e.g., [ZC90, Wol95]). The entry $*$ stands for any value.

Example 50. The direction vector of $(1, 1+, 0)$ *is* $(+1, +1, 0)$ *or* $(<, <, =)$.

Note that if the generalized distance vector was $(1, -1+, 0)$, *the direction vector would have lost much more information:* $(+1, *, 0)$.

Dependence level

The coarsest abstraction is the *dependence level* which is the depth of the first loop that has a non-zero entry in the direction vector. This loop is said to *carry the dependence*. If the direction vector is zero in all dimensions, the dependence is called *loop-independent* or *text-carried*, otherwise *loop-carried*. The level of a loop-independent dependence is by convention ∞ [AK87].

Example 51. The dependence in Example 50 is carried by the first loops, i.e., the dependence level is 1, and the dependence is loop-carried. In contrast, the dependence with distance vector $(0, 0)$ *in Example 47 is loop-independent.*

A similar series of dependence abstractions, including the concept of a dependence cone, but without the generalized form of distance vectors, has been proposed by Yang et al. [YAI95].

5.2 An algorithm for computing the data flow

Let us now present a method that computes the flow of the data in a program, i.e., a method for computing true or flow dependences. Modifications for other types of dependences are discussed in Section 5.2.2.

For static control programs, there exists already a data flow analysis method, in other words, a method for reaching definition analysis by Feautrier, which has full precision: for every operation that reads a value, it computes precisely the source, i.e., the one operation that wrote this value [Fea91]. The disadvantage of this method is its limited applicability to static control programs.

On the other hand, there are methods for reaching definition analysis in every standard compiler [ASU86]. The problem with these methods is that they usually consider an array as atomic, i.e., they do not distinguish between accesses to different array cells. Similarly, they usually consider only statements but not operations, i.e., different run-time instances. These abstractions are too coarse for automatic parallelization.

Example 52. Consider the program in Figure 5.1, left, and its control flow graph at the right. We assume that predicate $P(i,j)$ is unknown at compile time. Our task is to

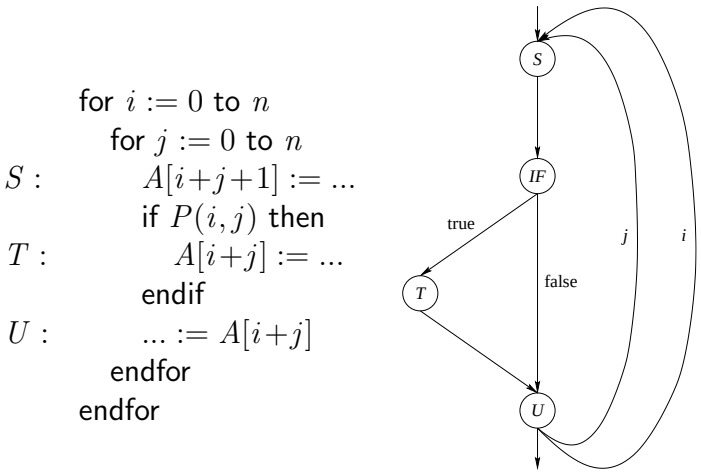


Figure 5.1: Nested loops with dynamic guards

determine which operation defines the value that is read at operation $\langle i, j; U\rangle$.

Standard techniques are not precise enough, and Feautrier's method is not applicable.

Thus, our goal is to present a data flow analysis technique, that is

- as precise as Feautrier's algorithm [Fea91] whenever it is applicable;
- as generally applicable as the typical reaching definition analysis [ASU86] in any standard compiler.

In the rest of this chapter, we summarize a method that satisfies both constraints [CG99], called *CfFada*, which stands for a "Control Flow graph based Fuzzy Array Data flow Analysis". The subterm *Fada* has been introduced by Collard et al. [CBF95, BCF97]; it expresses the unavoidable fuzziness, i.e., imprecision of data flow analysis (at compile time) when dealing with dynamic control programs. The mathematical foundation of that paper is reused in CfFada. In contrast to the original presentation, we focus only on structured programs in this thesis.

5.2.1 The basic idea

Principal procedure The essential idea is that we use a backward traversal through the control flow graph (as in the standard reaching definition analysis), starting at the node representing the read of interest.

At every node we test whether we have a write to this read array. If so, we merge the potential sources of the actual node with the already found sources, where we take the optimization constraint into account.

Instance-wise consideration In order to distinguish different run-time instances of the statements and different array cells, we take the array index vectors into account when testing for the conflict constraint (like, e.g., Feautrier [Fea91]), and we give special regard to the loop edges in the control flow graph.

For that purpose, we hold a *loop level vector* when traversing the graph. This loop level vector looks like a direction vector, and it constrains the set of dependences we are currently searching. It is initialized to a constant zero vector when starting the traversal, and, thus, we start searching for loop-independent dependences. During the iterations in the control flow graph, we update the loop level vector and, thus, successively find dependences carried by any loop.

Such an update of the loop level vector is done every time when, during our traversal of the control flow graph, we meet the head of a loop for the first time. In this case, we do not continue the traversal outside the loop, but we jump back to the end of the loop body and modify the loop level vector so that we now search for dependences which are carried by the actual loop. If we meet an already visited head of a loop, we ignore the loop edge and proceed outside the loop.

Evaluation Since we assume structured programs, we traverse, with this procedure, the loop nest inside-out. Hence, the detected sources are the older, i.e., smaller w.r.t. the execution order of the program, the later they are found. Consequently, once we find a source (and we are sure that it is executed), we may stop the traversal. Otherwise, e.g., if all sources are inside conditionals that cannot be evaluated at compile time, the analysis proceeds until it reaches the beginning of the program. Once it gets there, it reports that the variable under consideration might have no source in the given program segment (depending on the actual values of the predicates at run-time).

A more algorithmic description of the analysis is shown in Figure 5.2.

Let us now apply this algorithm to our non-trivial example in order to see the principal procedure, but also some technically interesting detail. The quality of the result of our algorithm is surprising.

Example 54. ***Initialization steps:*** *Consider the control flow graph in Figure 5.1. The read of interest is in statement U, i.e., at depth 2. Hence, we initialize our pointer p to node U, and set $llv = (0,0)$ and $s = todo = \emptyset$ (Steps 1–5).*

The first traversal with loop level vector $(\mathbf{0},\mathbf{0})$***:*** *According to Step 6, we now traverse the control flow graph backward, taking the left edge to node U (and saving the right edge to todo). This brings us to statement T at some, for the moment, undetermined indices i' and j'. Since this is an assignment to the array of interest, we enter Step 7a and consider $\langle i', j'; T\rangle$ as a possible source. The loop level vector requires $i = i'$ and $j = j'$, which is at the same time a solution to the conflict equality. The existence inequalities, resulting from the loop bounds, are sure to be satisfied for $\langle i, j; T\rangle$ if the dependence destination $\langle i, j; U\rangle$ is executed. However, the additional constraint on the existence of $\langle i, j; T\rangle$, due to its surrounding predicate is undetermined at compile time. Hence, $\langle i, j; T\rangle$ is the source for $\langle i, j; U\rangle$ iff it is executed, but we must continue searching for other sources in the case that $\langle i, j; T\rangle$ is not executed.*

By Steps 8 and 6, we reach node IF. Since it is a branch node, we enter Step 7b, and because there is one edge in todo, we traverse it, ending up again at node IF without any possible sources. Hence, the result at this node after the merge is: $\langle i, j; T\rangle$ is the source for $\langle i, j; U\rangle$ iff it is executed.

By Step 8 we reach statement S at some iteration (i', j'), which again is an assignment to the array of interest. Thus, we enter Step 7a and consider $\langle i', j'; S\rangle$ as a possible source. Here, the loop level vector still requires that $i = i'$ and $j = j'$, which contradicts the conflict equality $i'+j'+1 = i+j$. Therefore, $\langle i', j'; S\rangle$ cannot be a source of $\langle i, j; U\rangle$ for any values of i' or j'. Hence, the result so far is still: $\langle i, j; T\rangle$ is the source for $\langle i, j; U\rangle$ iff it is executed.

The next visited node is the head of the inner loop. Following Step 7c of our algorithm, we jump to the end of the loop body, i.e., node U, and reset llv to $(0,+1)$.

The second traversal with loop level vector $(\mathbf{0},+\mathbf{1})$***:*** *Node U is an assignment, but, as we assume, not to our array. Hence, we proceed as in our first traversal.*

When we now reach $\langle i', j'; T\rangle$ with $i = i'$ and $j' < j$ (due to our modified loop level vector), we find no new source, because of a contradiction with the conflict equality $i'+j' = i+j$.

Then we arrive at $\langle i', j'; S\rangle$ with $i = i'$ and $j' < j$, which is a solution to the conflict equality if we set j' to $j-1$. Hence, $\langle i, j-1; S\rangle$ is a source of $\langle i, j; U\rangle$, provided $\langle i, j-1; S\rangle$ exists, i.e., for $j \geq 1$. This means that we must continue our search, because we have not yet found a source for the case that $j = 0$ and $P(i, j)$ is false.

This time, we ignore the loop head of the inner loop and arrive at the loop head of the outer loop. Thus, we restart with loop level vector $(+1, *)$ at the end of the loop body.

Algorithm 53 (CfFada).

1. Let R be the statement containing the read access for which we search the sources, and d be the nesting depth of R;
2. let *llv* be a vector of length d, initialized with zeroes – to be used as loop level vector;
3. let G be the control flow graph of the program to be analyzed, and p a pointer to the nodes in G, initially pointing to the node representing R;
4. let s be the set of already found sources, initialized to the empty set;
5. let *todo* be a set of edges for open tasks, initialized to the empty set;
6. move p backward in G; if there are multiple possibilities, choose one, and store the others in *todo* for future consideration; if there is no predecessor, stop.
7. depending on the type of the node A where p is pointing to, do:
   (a) if A is an assignment to the array read in R, then
      i. generate the equalities c_c that result from the conflict constraint;
      ii. generate the inequalities c_e that result from the existence constraint, i.e., from the loop bounds;
      iii. generate the equalities or inequalities that result from the ordering constraint, depending on the entries of *llv* in every dimension i ($1 \leq i \leq d$):
         - an entry of 0 means that the index vector of the read operation and the potential source must be equal in dimension i;
         - an entry of $+1$ in the *llv* corresponds to the constraint that the index vector of the read operation must be strictly larger than the corresponding index of the potential source in dimension i;
         - an entry of $*$ causes no additional constraint;
      iv. put the constraints c_c, c_e, and c_o to a tool for parametric integer programming, e.g., PIP [Fea88] or Omega [PW92]
      v. if the constraint system has a solution, we have possibly new sources, which we merge with the sources in s;
   (b) if A is a branch node, then merge the sources from the different branches; if there are edges in *todo*, i.e., there might be unconsidered branches of A, then take one of them, set p to the source of this edge, and go to Step 7;
   (c) if A is the head of a loop l at depth $d' \leq d$, then if A is visited for the first time, continue the search for sources, i.e., dependences, that are carried by l:
      i. set component d' of *llv* to $+1$, and component $d'+1$ to $*$ (if it exists);
      ii. set p to the last statement in the body of l (it can be reached by following the loop's backward edge in the control flow graph in opposite direction);
      iii. go to Step 7
8. go to Step 6

Figure 5.2: Reaching definition analysis (CfFada)

The third traversal with loop level vector $(+\mathbf{1}, *)$***:*** *We find a first set of candidates at* $\langle i', j'; T\rangle$ *with* $i' < i$ *and* j' *arbitrary, if* $P(i', j')$ *holds. Thus, potential sources are all operations* $\langle i-k, j+k; T\rangle$ *with* $P(i-k, j+k)$ *for* $k < i$ *and* $k < n-j$*. Since we are only interested in sources for* $j = 0$*, the set of all sources at this step of the algorithm must be enlarged by* $\langle i-k, k; T\rangle$ *with* $P(i-k, k)$ *for* $k < i$ *if* $j = 0$ *and* $\neg P(i, j)$*. Note that the optimality constraint cannot cancel any of the sources due to the unpredictable predicate.*

We continue our backward traversal and arrive at $\langle i', j'; S\rangle$ *with* $i' < i$*. At this point, again many solutions to the conflict equality exist. However, these solutions are all executed and, thus, due to the optimality constraint, we can select the latest sources only. These are* $\langle i-1, j; S\rangle$*. Furthermore, since we already found solutions for* $j > 0$*, we obtain only one new dependence source:* $\langle i-1, 0; S\rangle$*, which only exists for* $i > 0$*.*

At this point, it is important to realize that we can (and, hence, must) again apply the optimality constraint – this time between the sources already computed and those newly found: the newly found source $\langle i-1, 0; S\rangle$ *is not guarded by any predicate and is executed later than* $\langle i-k, k; T\rangle$ *for* $k > 1$*. I.e., it overwrites any value which comes from* $\langle i-k, k; T\rangle$ *for* $k > 1$*. Thus, from this set, only the solution for* $k = 1$ *remains a source, i.e.,* $\langle i-1, 1; T\rangle$ *with* $P(i-1, 1)$ *is a source for the case* $j = 0$ *and* $\neg P(i, j)$*.*

Now, we continue our traversal, this time ignoring all loop heads and arrive at the program entry point with the following information: the source of $\langle i, j; U\rangle$ *is*

- $\langle i, j; T\rangle$ *iff* $P(i, j)$
- $\langle i, j-1; S\rangle$ *iff* $\neg P(i, j)$ *and* $j > 0$
- $\langle i-1, 1; T\rangle$ *iff* $\neg P(i, j)$ *and* $j = 0$ *and* $P(i-1, 1)$ *and* $i > 0$
- $\langle i-1, 0; S\rangle$ *iff* $\neg P(i, j)$ *and* $j = 0$ *and* $\neg P(i-1, 1)$ *and* $i > 0$
- *outside the program fragment iff* $\neg P(i, j)$ *and* $i = j = 0$

Note that this is the most precise information we can get at compile time.

Extensions in the original publication The formal merge of already determined and new possible sources is the same as in the work of Collard et al. [CBF95], but their work is not based on the control flow graph. This has two consequences:

- The method described here may stop the traversal before reaching the entry node of the control flow graph: once it has found all sources, it need not continue the traversal outside of the actual loop, because all candidates that can be found there are older than the already discovered sources and, hence, overwritten.
- The method can be applied to unstructured problems. The main difference is that in unstructured programs we must give up the loop level vector (for the unstructured

"loops", not for all loops), which means that we have fewer constraints in the algorithm, resulting in more possible sources. However, this is not due to an inferior analysis technique but is inherent in the unpredictable control flow of such programs.

All technical details can be found in the original publication [CG99].

5.2.2 Beyond true dependences

The described method is a data flow analysis: it computes directly true or, ideally, flow dependences. Adapting it to the search for output dependences is straight-forward: we start from a write and search for the last preceding write. However, the adaptation to anti dependences causes additional changes.

Anti dependences

More sources for a single destination First, it does not make sense to search for the latest read before a given write, because we do not impose an order on the reads. Thus, in Figure 5.3 there are two anti dependences: $S \longrightarrow R$ and $T \longrightarrow R$.

$$\begin{array}{ll} S\text{:} & \ldots := x \\ T\text{:} & \ldots := x \\ R\text{:} & x := \ldots \end{array}$$

Figure 5.3: Several anti dependences with the same destination

A theoretically possible way out is to impose an order on reads, i.e., to consider input dependences as any other kind of dependences. In that case, the dependence $S \longrightarrow R$ is the transitive combination of the input dependence $S \longrightarrow T$ and the anti dependence $T \longrightarrow R$, i.e., we can drop it – as we do with true dependences in order to obtain flow dependences. But this leads to a huge number of additional limitations of parallelism, which we cannot accept.

Consequently, we must modify the algorithm such that the merge of already found and potentially new sources becomes a simple union.

Adding a different optimization In addition to the just mentioned change, we must introduce a different kind of optimization. In Figure 5.4 the anti dependence from $S \longrightarrow R$ can be dropped as it is the transitive combination of the anti dependence $S \longrightarrow T$ and the output dependence $T \longrightarrow R$. However, this optimization cannot be done during the merge of already found and potentially new sources, like in the algorithm for true or output dependences, but it must be treated separately. One possible way is to use two different data structures:

- one for the source operations (containing the read accesses), and

$$\begin{array}{ll} S\text{:} & \ldots := x \\ T\text{:} & x := \ldots \\ R\text{:} & x := \ldots \end{array}$$

Figure 5.4: A different optimizing for anti dependences

- another one for the operations that cancel direct dependences (containing the write accesses).

Technical remark: Our implementation of the presented method relies on the Omega library [PW92, KMP+96], which is based on Presburger arithmetics. The result of the tool may be a union of several solutions (partly due to different sources, partly due to limited simplification of the result). In this situation, we generate separate dependences with accordingly restricted domains and ranges; they are considered as individual dependences for the remainder of the parallelization process.

5.3 Single assignment form

Dependences limit parallelism. Thus, our goal should be to eliminate as many dependences as possible. In fact, we can eliminate every anti and every output dependence in a program, because these types of dependences result from reusing the same array cell for a different value.

A program in which every array cell is written at most once is in *single assignment form*. One way to reach this form for static control programs is presented by Feautrier [Fea91]. The basic idea is as follows:

- For a write access in operation $\langle i; S\rangle$, the write is redirected to $A_S[i]$, where A_S is a new array that has as many dimensions as there are loops surrounding S. This avoids any anti or output dependences.

- For a read access in operation $\langle j; T\rangle$, we compute the source of the value, i.e., the source of the flow dependence whose destination is $\langle j; T\rangle$. Let us assume that the source is an operation $\langle i; S\rangle$, then we replace the original read by a read access to $A_S[i]$.

In the case of dynamic control programs, the source of a dependence cannot be determined uniquely at compile time, as we have seen in Example 54. Consequently, we obtain a tree of possible sources, and a function, usually named ϕ, selects the relevant branch at run time [BCF97]. Still, all anti and output dependences can be avoided that way.

Disadvantages As just noted, the case of dynamic control programs leads to run-time overhead. One suggested solution is to use maximal static expansion, i.e., to convert to single assignment for only as far as every source can be determined uniquely at compile time [BCC98].

Another problem is the amount of memory needed. Frequently, the memory consumption is inacceptably high. There are many methods that focus on reducing the amount of memory needed by allowing reassignments again [LF98, DSV03, QR00, TVSA01].

We shall not convert to single assignment form in this thesis and, therefore, we must consider also anti and output dependences during parallelization. Thus, we shall not distinguish between different types of dependences until we explore their impact on communication code generation in Section 13.4.4.

Chapter 6

Model refinement: index set splitting

Starting point After the execution of the dependence analysis algorithm, the source program is converted to a corresponding model-based description. All information that is necessary for the parallelization is extracted and represented in the model. Thus, we are ready to start with computing a space-time mapping. Much research has been done in this field, and good methods for computing schedules and placements are available [Fea92a, Fea92b, DV94, DV96b, Fea94, DR95, DRV01].

Goal However, this chapter shows that, in some cases, it is useful to refine the model-based description of the source program, before applying space-time mapping methods. In principle, this refinement improves the quality of schedules and placements in the same way. But for the presentation of the method, we focus on schedules. Hence, the goal of this chapter is to improve the quality of (already existing) scheduling methods.

Idea For the case of uniform dependences, Darte, Khachiyan, and Robert proved that there are methods which yield a schedule with (asymptotically) optimal latency [DKR91]. However, for the case of affine, non-uniform dependences, this optimum is sometimes missed by orders of magnitude. The use of different schedules for different iterations of the same statement frequently improves this situation. Thus, the idea is, to partition the index sets of all statements independently of the problem size into a fixed number of parts and compute individual schedules for each part.

6.1 Statement of the problem

It is generally not possible to find an arbitrary schedule with minimum latency. Therefore, one usually restricts the search to a subset of all possible functions, in our case the functions which are affine in the loop counters. As we have described in Section 3.3.7, one builds an affine template for every schedule (i.e., an affine function with unknown coefficients) and writes inequality (3.16) for all possible values of u and v. The unknowns are the coefficients of the scheduling functions, and it is easy to see that the resulting constraints are affine in

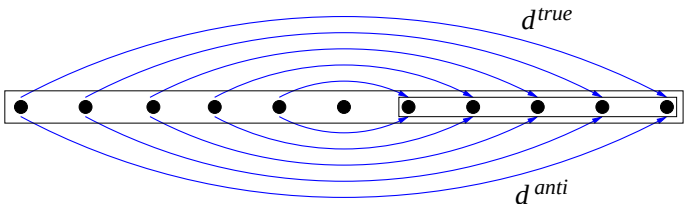


Figure 6.1: Simple example showing the necessity of splitting

these unknowns. Note that, in principle, this must be done for all values of n, yielding an infinite set of constraints. Fortunately, thanks to special properties of affine functions, this set can be shown to be equivalent to a finite set of affine constraints, which can be solved by the usual linear programming methods.

For some problems, the resulting linear program is found to have no solution. In this case, one resorts to multi-dimensional schedules. One can find a maximal subset of the dependences which still gives a feasible linear program. The resulting function is the first component of the multi-dimensional schedule. Then one applies the same algorithm to the unsatisfied dependences, obtaining the next component of the schedule, and so on until all dependences are satisfied.

One may wonder whether the schedule found in this way bears any relation to the minimum latency schedule. It has been proved that, when all dependences are uniform, the two schedules are asymptotically equivalent [DV96a]. However, there are well known counter-examples showing that this is not true for arbitrary affine dependences [Fea92a].

Example 55. Consider:

$$\begin{array}{l}\textsf{for } i := 0 \textsf{ to } 2 * n \\ \quad A[i] := ...A[2 * n - i]... \\ \textsf{endfor}\end{array}$$

The index set and its dependence graph are given in Figure 6.1. The best affine schedule is linear:

$$\textit{affine schedule} \qquad \theta_1(i) = i/2 \quad \textsf{if } 0 \leq i \leq 2 * n \tag{6.1}$$

while the minimum latency schedule is piecewise constant:

$$\begin{aligned} \textit{piecewise schedule} \qquad \theta_2(i) &= 0 \quad \textsf{if } 0 \leq i \leq n && (6.2) \\ &= 1 \quad \textsf{if } n < i \leq 2*n && (6.3) \end{aligned}$$

The time-mapped index sets, together with the true dependence are depicted in Figure 6.2.

θ_2 can be found by splitting the index set into two subsets, $I_1 = [0, n]$ and $I_2 = [n{+}1, 2{*}n]$, and postulating two separate scheduling functions one for I_1 and one for I_2. The details of the resolution method are not affected by the splitting; the number of unknowns, however, is doubled. This splitting can also be interpreted as a code transformation yielding the program:

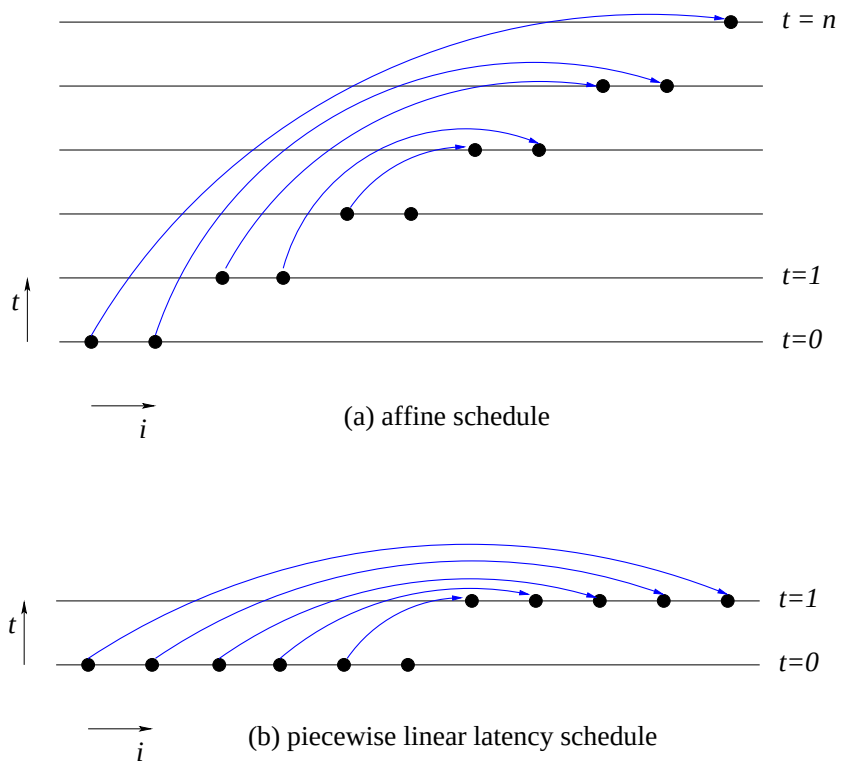


Figure 6.2: Best affine and best piecewise schedule

for $i := 0$ to n
  $A[i] := ...A[2 * n - i]...$
endfor
for $i := n + 1$ to $2 * n$
  $A[i] := ...A[2 * n - i]...$
endfor

followed by the application of any convenient scheduling algorithm.

Our aim in the remainder of this chapter is to derive an algorithm for deciding when splitting is useful, and for finding the splits.

6.2 Comparison and interaction with other parallelization methods

Tiling Our notion of index set splitting seems very similar to tiling (cf. Chapter 8): both techniques partition the index sets. However, the idea of tiling is to enumerate the given index set in a higher-dimensional space: one set of dimensions for the tiles and another set of dimensions for the points inside a tile; all tiles are treated equally. In contrast, index

set splitting does not change the number of dimensions but benefits from an individual treatment of the various partitions.

Scheduling Like index set splitting, the scheduling method by Feautrier [Fea92a, Fea92b] can also result in piecewise affine functions. However, the schedules found are minima of a finite set of affine functions, and most piecewise affine schedules cannot be cast in this form. Example 55 is a case in point.

Loose coupling with dependence analysis and parallelization Note that index set splitting is a postprocessing phase of dependence analysis; it is applicable independently of whether the analysis is more restricted but exact [Fea91] or less restricted but approximate [PW94]; it inherits the restrictions and precision from the preceding dependence analysis tool. On the other hand, it is a preprocessing step for model-based (hence automatic) parallelization; in contrast to Pugh and Wonnacott [PW94], we need not check any interference of our method with every existing parallelization technique individually; by applying index set splitting followed by some model-based parallelizer, we get the result of (a suitable combination of) unimodular transformations, loop peeling, strip mining, etc. directly and automatically.

Text-based parallelization methods The idea of index set splitting goes back to Wolfe [Wol89b], and further to Allen and Kennedy [AK87] and Banerjee [Ban79]. In the underlaying text-based parallelization methods, index set splitting is related to a single loop. Our method expands on these seminal efforts by incorporating them into the polytope model.

The work most closely related is by Jemni and Mahjoub [MJ95, MJ96] and deals with partitioning the index set at points where the type of a dependence changes, e.g., from true to anti. But also their method is not based on a model, which means that they can separate the index set only along planes parallel to the coordinate directions.

6.3 Analysis

Let us first explain why schedule (6.1) is suboptimal. An arbitrary affine function can be written as follows:

$$\theta_S(x, n) = \tau_S\, x + \sigma_S$$

and obviously has the property that the difference

$$\theta_S(x + y, n) - \theta_S(x, n) = \tau_S\, y$$

depends on y but not on x. Suppose that there is a dependence from $\langle x; S\rangle$ to $\langle x + y; S\rangle$ for some x and y. Then:

$$\theta_S(x + y, n) - \theta_S(x, n) = \tau_S\, y \geq 1$$

Iterating this result k times, we obtain:

$$\theta_S(x + k * y, n) - \theta_S(x, n) \geq k \tag{6.4}$$

The concept of latency (Definition 28) can be extended to individual statements S:

$$L_S = \max_{x \in \mathcal{I}(S)} \theta_S(x, n) - \min_{x \in \mathcal{I}(S)} \theta_S(x, n),$$

where $\mathcal{I}(S)$ is the index set of S (Definition 3). It is clear that the latency of a statement is a lower bound on the latency of the program. From (6.4) we deduce:

$$L_S \geq \max\{k \,:\, x + k * y \in \mathcal{I}(S)\} - \min\{k \,:\, x + k * y \in \mathcal{I}(S)\}$$

Since $\mathcal{I}(S)$ depends linearly on a size parameter n, we may expect that L_S also increases linearly with n.

In Example 55, we have a one-dimensional dependence vector, $y = (2)$, from instance $n-1$ to $n+1$. Since y can be iterated n times within the index set, we have a latency of at least n for the execution of statement S, which is exactly the latency of schedule (6.1).

Suppose now that $\mathcal{I}(S)$ has been partitioned into two subsets, I_1 and I_2, and that $x \in I_1$ and $x + y \in I_2$. The above reasoning no longer applies, since the schedule θ is not necessarily the same affine function in I_1 and I_2. Hence, the above estimate of the latency of S no longer holds, and we may hope for a better schedule.

If we split the index set of the target statement of a dependence d into I_1, which contains the image of d (and therefore has to satisfy the schedule constraints for d), and I_2, which contains the rest of the index set (and therefore need not satisfy constraints which are due to d), we get a constant schedule for I_2 (if d is the only dependence). If there are no dependences inside I_1, i.e., the domain of d is contained in I_2, we also get a constant schedule for I_1.

For our example, the dependence analysis gives us the information that only the instances $n+1, \cdots, 2*n$ are in the range of the existing dependences. Since iterations $0, \cdots, n$ are not images of any dependence, they need not satisfy any scheduling condition and, thus, should be given an individual schedule template. This splitting enables us to find schedule (6.2), which has a constant latency.

Application to placement The same reasoning applies to placement problems. Here the goal is to find a placement function $\pi_S(x, n) = \Pi_S\, x + \mu_S$ which specifies the number of the processor that executes operation $\langle x; S\rangle$. Two operations which access the same memory location are to be executed on the same processor. If $\langle x; S\rangle$ and $\langle x + y; S\rangle$ access the same memory cell, we must have

$$\pi_S(x, n) = \pi_S(x + y, n) \Rightarrow \Pi_S\, y = 0.$$

Consequently, by induction, all operations $\langle x + k * y; S\rangle$ will use the same processor as $\langle x; S\rangle$, whether there is a reason, e.g., it reduces the number of communications (cf. Chapter 7), or not. On the other hand, putting $\langle x; S\rangle$ and $\langle x + y; S\rangle$ in different subsets of

the index set allows to choose two different placements. This invalidates the inductive reasoning and may result in better parallelism. In the case of Example 55, the existence of a single instance of the dependence vector $y = (2)$ entails $\Pi_S = 0$, all operations are on the same processor, and there is no parallelism. After splitting, the only condition is that operations $\langle x; S\rangle$ and $\langle 2 * n - x; S\rangle$ are on the same processor. The degree of parallelism is n.

6.4 A first, naïve splitting algorithm

As we have seen in Section 6.3, suboptimal schedules can result from the fact that some parts of a statement's index set are in the image of a dependence, and some are not. If the distinguished statement is the target of several dependences, we should apparently subdivide its index set, each part corresponding to a selection of the incoming dependences. Furthermore, if there is a dependence d_1 from statement R to statement S and a dependence d_2 from S to T, we should use the composite dependence $d_2 \circ d_1$ for splitting $\mathcal{I}(T)$. Thus, a naïve approach for index set splitting proceeds as follows:

1. compute all possible paths in the dependence graph
2. split every statement according to all incoming paths.

Even though some drawbacks are obvious, let us analyze this method in more detail, in order to illustrate the limits of what we can expect from the final algorithm to be presented in Section 6.5.

6.4.1 Merging multiple incoming paths

First, we realize that a single statement can be reached via several paths. Thus, independently of the splitting algorithm chosen, we will have to merge the splits resulting from different incoming dependence relations. In principle, we split the index set according to the first incoming dependence, then split every part according to the next incoming dependence, and so on.

It is fortunate that the order of treating the incoming dependences is irrelevant: to merge splits means to compute an intersection of the image (or its complement which we call the non-image) of an incoming dependence and the parts already split, and intersection is associative and commutative.

The complexity of merging the splits of k arbitrary incoming (composite) dependences is 2^k, i.e., in general, we must expect an algorithm with at least exponential time complexity. Therefore, our main concern must be to reduce the number k of different incoming (composite) dependences as much as possible, in order to get an efficient algorithm.

Remark 56 (special case without exponential complexity). If the dependence graph is a tree, the complexity of merging is only linear in the length of the paths, i.e., in the

depth of the tree: various incoming paths can only differ in length (for two different paths reaching a tree node, one must be a postfix of the other). Therefore, the image of the longer path is a subset of the image of the shorter path, i.e., we never need to split the non-images, which avoids exponential growth.

6.4.2 Numbers of paths

We have just seen that, in a tree, the number of different paths to a given node is linear in the tree's depth. How many paths are there in a directed acyclic graph (DAG)? From graph theory we know that there can be exponentially many paths between two nodes in a DAG. Hence, if we consider all different paths from s to t, we end up with exponentially many incoming splits in t, which must then be merged. If we do so using the exponential method as explained in Section 6.4.1, we obtain a doubly exponential algorithm, which we consider impractical.

Obviously, the naïve method is not effective within strongly connected components since the number of different possible paths is unbounded if there are cycles in the statement dependence graph. As simplest example, take a single statement with one self loop: every different number of loop traversals results in a different path. Note that, in this simple situation of only one self loop, the number of splits would increase linearly rather than exponentially in the number of iterations (like in the case of the tree above), but it is still unbounded.

6.4.3 Preparing an effective algorithm

As we have just seen, using paths directly as descriptions of composite dependences leads, in general, to a doubly exponential algorithm. Hence, we suggest for our splitting algorithm to abstract from precise paths in order to be efficient.

Abstraction by Kleene's path descriptions For this purpose, we treat composite dependences systematically by associating a finite automaton with the dependence graph. The states of this automaton are the statements and the transitions are the dependences. There are well known algorithms (e.g., by Kleene [FB94]) for associating any two states S and T with a *regular expression* representing all paths from S to T. The letters in this regular expression are dependence *names*, and the operators are the dot (concatenation), the vertical bar (set union), and the Kleene star (closure). We call this regular expression the *Kleene's path description*. The composite dependence from S to T is obtained by replacing each dependence name in this expression by the dependence relation itself, the dot by relation composition, the vertical bar by relation union, and the Kleene star by transitive closure. In suitable cases, one can compute the composite dependence in closed form by using, for instance, the Omega calculator [PW92]. However, since the transitive closure of an affine relation is not always affine, the above computation does not always succeed, and the Omega calculator returns an overestimate. Our proposal is to ignore a composite dependence when a closed form cannot be computed.

```
for i := 0 to m
   for j := 0 to 2 * n
      A[i, j] := A[i, 2 * n − j] + A[i−1, j+2]
   endfor
endfor
```

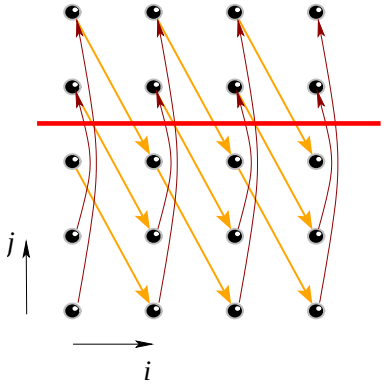


Figure 6.3: Splitting due to the initial phase

Overly high loss of precision Unfortunately, if we base our splitting algorithm on Kleene's path descriptions, we find many practical examples which cannot but should be split. This is because the use of the operators union and closure entail a loss of precision. In fact, we lose all information on the *delay* associated with the composite dependence.

Example 57. Consider the program and iteration dependence graph in Figure 6.3. The range of dependence $d_{upwards}$ is the upper half of the index set; this dependence is a typical candidate for splitting. In contrast, since $d_{downwards}$ is uniform, its range is nearly the complete index set (apart from boundary iterations). Consequently, the range of the combined dependence $d = d_{upwards} \mid d_{downwards}$ contains also nearly all iteration vectors; more formally, the range is $[0, m] \times [0, 2{*}n] \setminus [0, 0] \times [0, n]$. The desired split into $[0, m] \times [0, n]$ and $[0, m] \times [n+1, 2{*}n]$, which leads to a linear execution time (see Figure 6.3), is not found if we base the algorithm on Kleene's path descriptions only.

The difficulty in this example is that the different dependences have individual properties (e.g., different ranges), which are merged by the union operation, making the distinctions invisible. However, treating all dependences separately is too costly, in general, as shown in Section 6.4.2.

A compromise between Kleene's path descriptions and individual paths Our heuristic solution is to consider (in addition to all combined paths) every single dependence once, and split the index set of its target statement into its range and the rest. This guarantees that the properties of different dependences are considered at least once, and that complexity is increased by only a linear factor (D dependences instead of 2^D paths). In other words, by splitting the index set $\mathcal{I}(S)$ of the target statement S of every single dependence d into the range of d and the rest, we have already a conservative approximation of the range of all *paths* to S whose last dependence is d.

Our idea is then to propagate these ranges of d along every path to all other statements, where we now accept the loss of information by using the path descriptions from Kleene's algorithm, in order to keep the computational effort reasonably small.

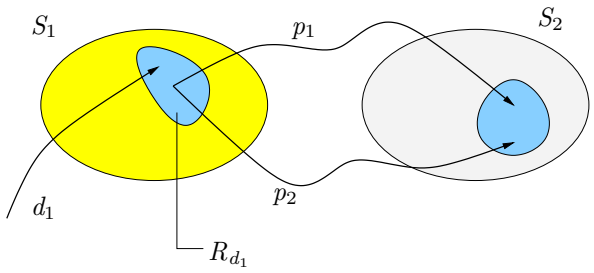


Figure 6.4: An illustration of the splitting algorithm

Our examples have illustrated that this mixture of working with separate dependences on the one hand and combined path descriptions on the other exhibits a good balance between power and execution complexity of our algorithm.

6.5 The proposed splitting algorithm

With the ideas explained in the previous section, we can now formulate a first version of the effective splitting algorithm:

Algorithm 58 (index set splitting).

1. For all dependences d, compute a polytope R_d (as small as possible) containing the range of d, and split the index set $\mathcal{I}(T)$ of the target statement of d into R_d and $\mathcal{I}(T) \setminus R_d$.

2. Compute a description of the set of all paths in the statement dependence graph, using Kleene's algorithm.

3. For every pair of statements (T, S) and for every dependence d with target statement T and every path p from T to S, interpret path description p as a composition of relations which maps points of the index set $\mathcal{I}(T)$ to points of the index set $\mathcal{I}(S)$, and compute the image $p(R_d)$ of this composed relation when applied to the polytope R_d computed in Step 1. This will divide the index set $\mathcal{I}(S)$ into a part which is in the image of R_d under p, and the rest. Usually, this step can be computed with the Omega calculator [PW92]. However, if p contains a cycle whose transitive closure cannot be computed precisely by Omega, then delete the cycle from p before the propagation.

4. For any statement S, combine all splits obtained in this way as in Section 6.4.1.

The algorithm is illustrated in Figure 6.4. In Step 1, the index set of S_1 is split into the image R_{d_1} of d_1 and the rest. Step 2 computes all paths between every pair of statements. Assume that there are two paths from S_1 to S_2, denoted with p_1 and p_2. Step 3 computes

the image of R_{d_1} w.r.t. $p_1|p_2$ within the index set of S_2, where the possible images are all in the dark subset.

Basically, this algorithm computes, for every statement, the approximate ranges of all (transitively) incoming dependences. It is obvious that, for each such range, we might obtain a different possible schedule and, thus, want a separate template.

However, in practice, there are some additional considerations:

- It is easy to see that splitting an index set due to a uniform dependence is useless, since the range of a uniform dependence d is (approximately) the complete index set of the target statement of d. Therefore, we optimize the algorithm by applying Step 1 only to non-uniform dependences. Note that, in Step 3, we still need to consider uniform dependences, as we shall see in Example 60.

- Due to the condensed description of the set of all paths and the overestimate of the reflexive transitive closure by Omega, it often happens that we lose precision and, thus, do not find a desired split: in some cases, we obtain an approximated, i.e., slightly different split and, in some cases, we do not find the split at all (if two approximations yield the same set).

  Note that these overestimates by Omega are due partly to the theoretical impossibility of computing the reflexive transitive closure as a finite union of polyhedra and partly to technical limitations of Omega.

- Sometimes it is useful to unroll a cycle a fixed number of times, in order to achieve the optimal split (cf. Example 62). If, for a cycle c, we can find this fixed number k by the methods described in the original publication [GFL00, appendix], we extend the computation of $p(R_d)$ in Step 3 of our splitting algorithm: if the propagation path p contains c, we rewrite p as $p_1.c^*.p_2$, and we compute $(p_1.c^i.p_2)(R_d)$ for $0 \leq i \leq k$ instead of the approximation $(p_1.c^*.p_2)(R_d)$.

6.6 Examples

Let us reconsider our initial example and modify it in several ways in order to get a feeling for the performance of our method.

Example 59. Let us apply our algorithm to Example 55. The index set is the set $[0, 2*n]$ *and the range* R *of the two dependences is* $[n+1, 2*n]$. *The set of all paths in the statement dependence graph is given by* $(d^{true} \mid d^{anti})^*$. *Propagating* R *along* $(d^{true} \mid d^{anti})^+$ *(*$^+$ *meaning at least once) is not possible since the domain of* $(d^{true} \mid d^{anti})^+$ *does not intersect* R. *Thus, the algorithm terminates after the initial step and splits the index set into* $[0, n]$ *and* $[n+1, 2*n]$, *as expected.*

```
for i := 0 to 2 * n
  for j := 0 to m
    A[i, j] := A[2 * n − i, j+m] + A[i, j−1]
  endfor
endfor
```

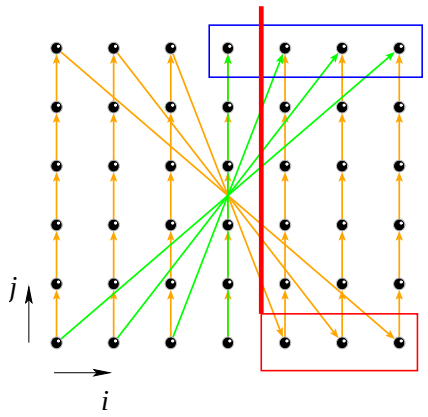


Figure 6.5: Splitting a two-dimensional index set by propagation

Example 60. Extending the program of Example 55 to a two-dimensional example, let us consider the program in Figure 6.5. For the uniform dependence, no split is derived. The non-uniform true dependence has range $[n{+}1, 2{*}n] \times [0, 0]$ *(the lower box in Figure 6.5, where* $n = 3$ *and* $m = 5$*). Propagating along the combined self dependence leads to the desired split into* $[0, n] \times [0, m]$ *and* $[n{+}1, 2{*}n] \times [0, m]$ *(the vertical line in Figure 6.5). The non-uniform anti dependence has the range* $[n, 2 * n] \times [m, m]$ *(the upper box in Figure 6.5), which is not increased by propagation. So, we end up with three subsets:* $[0, n] \times [0, m] \setminus \{(n, m)\}$*, the singleton* $\{(n, m)\}$*, and* $[n{+}1, 2{*}n] \times [0, m]$*.*

Note that, in the previous example, treating the singleton individually does not improve the schedule and, hence, should be avoided in practical implementations, if possible. However, in general, it is impossible to decide locally, i.e., at one given statement, whether a split is useful or not, since global information of the dependence graph is necessary. We would have to run the scheduler in order to detect whether we should split the input of the scheduler! Thus, in our current implementation, we accept possibly useless splits.

Example 61. In order to get a better feeling for the behavior of our algorithm in the case of multiple statements and imperfect loop nests, let us consider the program in Figure 6.6, left. In this example, we find two uniform true dependences and four non-uniform dependences (three true and one anti), i.e., we obtain four initial splits. After propagating each of them along the combined path to each of the three statements, we have to merge the four initial splits with the 12 propagated splits. After simplification, we get the desired splits as indicated in Figure 6.6, right.

A constant number of propagations In the case of a self dependence d, a constant number of propagations may give us an interesting schedule. Let I be the domain of the dependence. Successive propagation yields subsets $I \setminus d(I), d(I) \setminus d^2(I), \ldots, d^r(I) \setminus d^{r+1}(I)$ and, in the interesting case, there exists a constant k such that $d^k = \emptyset$. It can be proved that k is bounded by the cardinality of I, but this bound depends on the size parameters

```
        for i := 0 to 2 * n
S1:       A[i, 0] := A[2 * n − i, m]
          for j := 1 to m
S2:         A[i, j] := A[i, j−1]
          endfor
        endfor
        for i := 0 to 2 * n
          for j := 1 to k
S2:         A[i, j+m] := A[i−1, j+m−1]
          endfor
        endfor
```

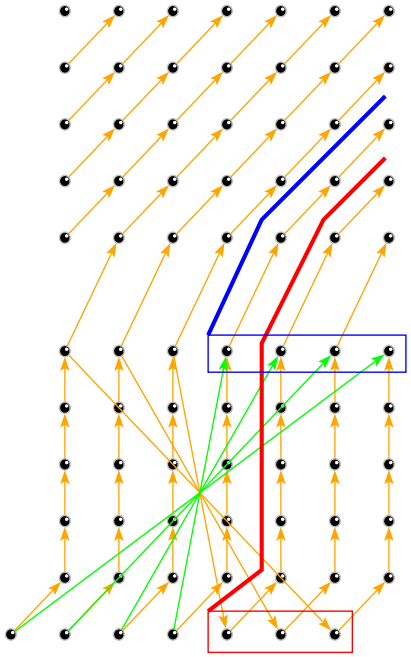


Figure 6.6: Splitting for imperfect loop nests

of the program and cannot be evaluated at compile time. The problem can be solved in the special case that d is defined by a square matrix A in one of the following senses:

$$(x, y) \in d \Rightarrow x = Ay$$

$$\text{or} \quad (x, y) \in d \Rightarrow y = Ax$$

One can prove that, if $A^k = I$, then $d^k(\mathcal{I}(S)) = \emptyset$ [GFL00, appendix]. There are mathematical methods to determine possible candidates for k, and since the dimensionality of the index set is typically a small number, the number of tests is very small. For more detail, we refer to the original publication [GFL00]; instead we demonstrate the effect of a constant number of propagations on a small example.

Example 62. Consider the following program:

```
for i := 1 to n
  for j := 1 to n
    for k := 1 to n
      A[i, j, k] := A[j, k, i]
    endfor
  endfor
endfor
```

Classical scheduling yields $\theta(i, j, k) = \binom{i}{j}$, *or* $O(n)$ *parallelism. There are two dependences:*

$$\begin{aligned} & d_1 = \{\langle i,j,k\rangle \to \langle j,k,i\rangle \,:\, i<j\} \\ \textit{and} \quad & d_2 = \{\langle i,i,k\rangle \to \langle i,k,i\rangle \,:\, i<k\} \end{aligned}$$

For brevity, we omit the bound constraints $1 \le i \le n$ and the like, which stay invariant in the splitting process.

d_1 is generated by the permutation matrix:

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

We can see that A^3 is the unit matrix. The corresponding subsets are:

$$\begin{aligned} & d_1(\mathcal{I}(S)) &&= \{\langle i,j,k\rangle \,:\, k<i\} \\ \textit{and} \quad & {d_1}^2(\mathcal{I}(S)) &&= \{\langle i,j,k\rangle \,:\, k<i, i<j\} \\ \textit{and} \quad & {d_1}^3(\mathcal{I}(S)) &&= \emptyset \end{aligned}$$

The domain of d_2 is disjoint from the domain of d_1, hence we are justified in handling both dependences independently. We leave to the reader to check that ${d_2}^2(\mathcal{I}(S))$ is empty. All in all, the latency is 2 and the degree of parallelism is $O(n^3)$.

6.7 Variants of index set splitting

This section first presents an iterative version of index set splitting (Section 6.7.1), then shows some trade-offs between the quality of the computed splits and the time needed to compute them (Section 6.7.2), and concludes with the modifications we made in our implementation (Section 6.7.3).

6.7.1 Iterative splitting

As described so far, the method treats the program as a whole and is a search for all potentially useful splits for improving the schedule. However, especially for large programs, it is desirable to apply index set splitting only when and where it is necessary. Furthermore, we are often only interested in increasing parallelism by orders of magnitude, but not by constant factors.

Our method can be easily adapted to these situations. The basic idea is that we first schedule the program without splitting and, if the result is not satisfactory, try to improve the solution by index set splitting:

Algorithm 63 (iterative index set splitting).

1. As a first step, compute a schedule following one of the usual methods [Fea92a, Fea92b, DV94, DV96c] without any splitting.

2. Analyze the schedule, with the intention of selecting an interesting candidate for splitting. For this purpose, take the statement S with the highest dimensionality of the schedule. If there are several possible candidates, choose one which is minimal w.r.t. the order imposed by the acyclic condensation of the statement dependence graph.

3. As we have seen in Section 6.3, significantly improving a schedule for S means cutting a cycle going through S. Edges of such cycles belong to the strongly connected component (SCC) of S in the dependence graph. Thus, we must split the statements in the SCC according to the algorithm in Section 6.5:

   (a) For all non-uniform dependences d of the SCC, compute the initial split as in Step 1 of Algorithm 58.

   (b) For every pair of statements (T, R) of the SCC, propagate the initial splits of T to R as in Step 3 of Algorithm 58. If the composite dependence d from R to R is non-uniform and generated by a matrix A of index k (see Example 62), continue propagating the split through $d \quad k-1$ times.

4. Schedule the new dependence graph.

5. When the dimensionality of the schedule is satisfactory or when all SCCs of the dependence graph have been considered, then stop; else go to Step 2.

Remark 64 (difference in the procedure of Algorithms 58 and 63). Note that the original method is completely automatic, whereas the iterative splitting approach just described leaves more freedom (e.g., we can decide that a SCC is not worthwhile being split) and, thus, is better used for interactive loop parallelization. In the original setting, index set splitting is viewed as a preprocessing phase for the loop program to be parallelized – any other parallelization method can then be applied as usual (except for a potentially increased complexity due to the fact that split parts are viewed as individual statements).

In the iterative setting, the user first runs a parallelization algorithm (e.g., computes a schedule), and then decides whether the result is satisfactory or not. If not, the user calls index set splitting, reapplies the parallelization, and decides again. It is unclear how this decision can be fully automated.

Remark 65 (difference in the result of Algorithms 58 and 63). Note that our iterative splitting approach yields fewer splits than the original algorithm: dependences which are not part of a strongly connected component are never considered. On the one hand, this limits the complexity of the target code and, on the other hand, the detected parallelism is still in the same order of magnitude as if we used Algorithm 58.

6.7.2 Trading off quality for time

Our method can be adapted simply for trading off quality of parallelism for analysis time:

- One can search for more splits and, thus, even satisfy a variety of optimality criteria, but at the cost of a doubly exponential search.
- One can turn off the propagation phase in order to save compilation time at the price of a loss of some useful splits (e.g., in Example 60).
- On the other hand, one can try to improve the solution by propagating a split through cycles more than once, as indicated in Example 62.
- Alternatively, one can also first schedule a program without splitting and afterwards try to split only those statements which are responsible for a possibly bad quality of the schedule, as shown in Section 6.7.1.

6.7.3 Implementation notes

We have integrated index set splitting into our prototype parallelizer LooPo (cf. Chapter 4). Our implementation takes as input the results of the existing dependence analysis modules and is very close to the algorithm in Section 6.5, with the following technical modifications:

- In Step 1, we do not actually execute the splits but just store them per statement in a list, in order not to increase the number of statements in this phase.
- In Step 3, we only compute the images; the computation of the rest (i.e., the non-images) does not commence before the end of Step 4 and, again, the derived splits are just stored in lists in order to limit complexity.
- In Step 4, we first merge all stored splits and then compute the rest, i.e., the non-images – which may lead to further splitting, due to technical limitations, if the non-images cannot be described by a single polytope but only by a union of polytopes. Finally, every partition of a split, i.e., every subset of the original index set gets a copy of the body statements and, thus, can be viewed by all subsequent parallelization phases as if it were a separate statement with surrounding loops in the source program. Note that our model-based approach saves us from computing a loop nest which really enumerates all these split subsets; this is advantageous because the construction of such a loop nest would be very costly.

Chapter 7

Spatial distribution: forward communication only placements

Communications The main task of the placement is to reduce the communication overhead, because communications are the most important reason for slowing down a parallel program. As already mentioned in Remark 36, there are two ways to determine the communications: an ownership-driven approach, which we follow in Sections 7.2–7.3 to determine a first version of our suggested placement algorithm and, alternatively, a dependence-driven approach, which we use for the general considerations in Section 7.1, and to which we adapt our placement algorithm in Section 7.4. Section 7.5 briefly discusses its applicability to practical examples.

Existing placement algorithms We have worked with several placement algorithms; in the terminology of Lim and Lam [LL98], they focus on the case that constant parallelism is not sufficient for taking benefit of all processors.

The method of Feautrier takes h-transformations as input; it assumes the owner computes rule, but is also applicable with the computer owns rule. It tries to avoid potential communications by placing the dependent operations on the same (virtual) processor, i.e., it is dependence-driven. The method uses a greedy heuristic in order to determine which dependence relations shall be considered first [Fea94].

The method of Dion and Robert is an ownership-driven approach (cf. Remark 36). It takes direction vectors or dependence polyhedra as input, thereby accepting to lose some precision [DR95].

Forward communication only placements

In Chapter 11, a central problem will be to avoid communication cycles. It is clear that, if all communications have the same orientation in every spatial dimension, i.e., no communication goes backward, a cycle can never exist. More formally, if all communication directions are contained in the cone $(0+, \cdots, 0+)$, no cycle can exist. Note that this is not a necessary but sufficient constraint to avoid cycles (cf. Section 11.2).

Definition 66 (FCO). A placement satisfies the *forward communications only* property (*FCO* property) iff all communication directions are contained in the cone $(0+, \cdots, 0+)$.

Besides the fact that we rely on this property in Chapter 11, which deals with medium- and coarse-grained parallelism, it can also be beneficial in the case of fine-grained parallelism, as pointed out in Section 7.1.

Since we know of no method that guarantees an FCO placement, we derive two versions of a placement algorithm that guarantees this property in Sections 7.2–7.4 – if possible.

7.1 General considerations

The idea of using FCO placements is not new. It has been suggested many times as a certain way of avoiding deadlocks. In our context, FCO placements enable arbitrary time tiling, as indicated in Section 11.2. But note that, even outside that context, the idea of FCO placements may be interesting in some cases since the method can halve the number of communication partners in every dimension, compared to the general case.

Example 67. Consider the following synthetic program (it is adapted from a numeric iteration method for solving systems of linear equations, called successive over-relaxation SOR *[Frö85]):*

```
for k := 2 to m
  for i := 1 to n-1
    A[k,i] := (A[k-2,i] + A[k-1,i] + A[k,i-1] + A[k-1,i+1])/4
  endfor                  (1)      (2)      (3)      (4)
endfor
```

A first solution *A valid schedule is $t(k,i) = 2*k + i - 5$. The communication minimal solution w.r.t. the communication volume is obtained by the placement $p_1(k,i) = i$, which is computed, e.g., by Feautrier's method [Fea94]. The space-time mapped dependence graph is shown in Figure 7.1; the numbers in the legend for the dependences refer to the access numbers in the code.*

As we can see in Figure 7.1, every processor p needs two communications – from different directions – in order to get its input values: this placement does not satisfy FCO. No message coalescing is possible since the data must be sent upwards and downwards.

An FCO solution *In contrast, the placement $p_2(k,i) = k$ satisfies FCO (Figure 7.2). At a first sight, the number of dependences between processors, i.e., communications, is higher than for p_1, since we have three non-local accesses. However, concerning accesses (2) and (3), we need not resend the same value twice from a processor to its upper neighbor, even if the value is needed at two different time steps. Thus, access (2) does not cause a communication.*

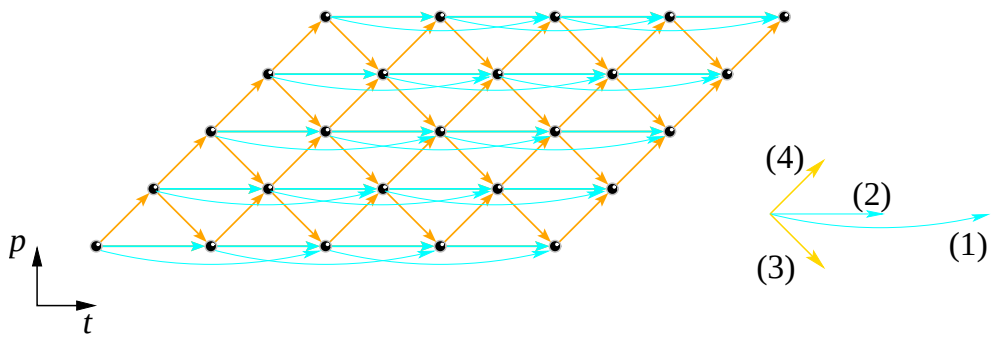


Figure 7.1: Non-FCO placement

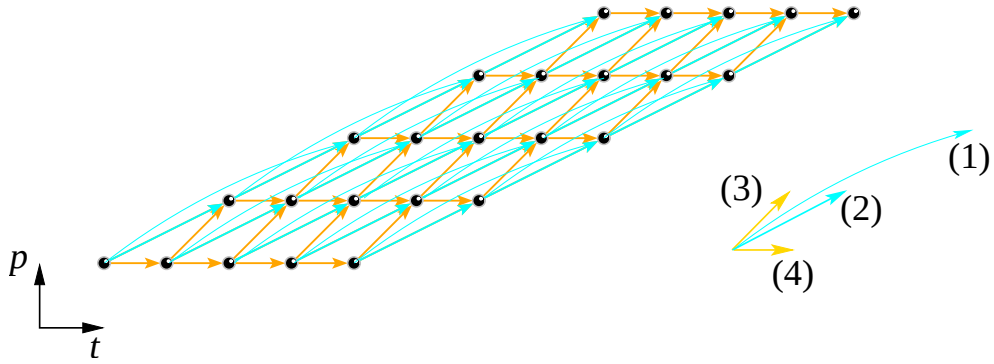


Figure 7.2: A possible FCO placement

Concerning access (1), we first note that the corresponding communication is not between neighbors, but has a distance of two processors. However, after space-time mapping, we usually must coalesce virtual processors in order to match the (high) number of virtual processors with the (typically lower) number of physical ones. If we do so, we coalesce at least two virtual processors and, thus, the distance of the communication is reduced to (at most) one, i.e., to neighbor communication. In addition, the value to be sent is already sent due to accesses (2) and (3). Hence, access (1) does not cause a communication either.

All in all, we end up with about half as many communications than for p_1.

FCO and coalescing processors The latter is a more general effect: after coalescing virtual processors to physical processors, FCO guarantees that there is at most one communication partner in every dimension (and also along the diagonals), if we have uniform dependences with a sufficiently small length. For non-FCO, we might have two communications per dimension (and also along the diagonals). More detail on counting communication partners is given in Section 9.4.

Load imbalance A drawback of FCO placements is that, occasionally, due to skewing the index set until all communications go forward, the load is distributed unevenly among

the processors. This is a problem frequently arising in the model-based parallelization, and code generation must be improved in order to deal with this phenomenon properly. Parts of our ongoing research activities are devoted to this question (cf. also Remark 81) [Sei04].

7.2 FCO in the case of explicit data placements

Let us first generate an FCO placement algorithm for the case that every data element has its fixed owner, given by a data placement, and the computation placement is independent from the data placement. (A different approach is presented in Section 7.4.)

Constraints Let r be the occurrence of the access to array A in statement S. If the array cell $A[a_{A,r}(i,n)]$ is not on the same processor as the computation of operation $\langle i; S\rangle$, a communication must take place. If r is a read access, the communication will be forward if

$$\pi_A(a_{A,r}(i,n),n) \leq \pi_S(i,n). \tag{7.1}$$

On the other hand, if r is a write, the communication will be forward if:

$$\pi_S(i,n) \leq \pi_A(a_{A,r}(i,n),n). \tag{7.2}$$

These inequalities are to be understood component-wise. They are to be verified everywhere in the index set of S, represented by a matrix D_S as in (3.13). The conjunction of these properties over all references in the program defines a forward communication only (FCO) placement. (Note that the definition of the direction is arbitrary: we can always reorder processors independently in each dimension).

Let us now consider one of the FCO conditions, (7.1) for instance. It can be rewritten as:

$$\left(\forall \begin{pmatrix} i \\ n \end{pmatrix} : D_S \begin{pmatrix} i \\ n \end{pmatrix} \geq 0 \Rightarrow \Pi_S \begin{pmatrix} i \\ n \end{pmatrix} - \Pi_A \mathcal{A}_{A,r} \begin{pmatrix} i \\ n \end{pmatrix} \geq 0\right). \tag{7.3}$$

Affinity problem Note that this system looks affine – but it is not. For this purpose, let us check what are the unknowns, what are numeric constants, and what are parameterized constants for the optimization problem. Obviously, the unknowns in this system are the coefficients in the matrices Π_S and Π_A; the matrices $\mathcal{A}_{A,r}$ and D_S contain numeric values that can be determined from the program. However, the iteration vector $\binom{i}{n}$ is a vector of parameters: the entries are not subject to the optimization task (they are input, i.e., not free), but symbolically denoted (we cannot – and would not want to – write all possible iteration vectors explicitly). Thus, if we multiply, e.g., Π_S with $\binom{i}{n}$, we multiply variables with parameters – a contradiction to our definition of affinity (cf. Remark 8).

Solution to the affinity problem In the context of scheduling, Feautrier proposed to solve such a problem by using Farkas' lemma [Sch86] for quantifier elimination. This lemma states how such a system of affine inequalities can be transformed into an equivalent system of affine equalities by adding non-negative variables.

Lemma 68 (Affine form of Farkas' lemma [Fea92a]). *Let $\mathcal{D}$ be a non-empty polyhedron defined by p affine inequalities*

$$a_k\, x + b_k \geq 0, \qquad k = 1, p$$

Then an affine form Ψ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination

$$\Psi(x) = \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k\, x + b_k), \qquad \lambda_k \geq 0$$

A proof sketch is given by Feautrier [Fea92a]; mathematical details can be found in Schrijver's book [Sch86, Corollary 7.1h]. The non-negative variables λ_k will be called *Farkas multipliers*.

Thus, by Farkas' lemma, (7.3) is equivalent to:

$$\textit{read constraint} \qquad \Pi_S - \Pi_A\, \mathcal{A}_{A,r} = \lambda_{A,r} D_S \tag{7.4}$$

where the Farkas multipliers $\lambda_{A,r}$ are non-negative. In this equality, Π_S, Π_A and $\lambda_{A,r}$ are unknowns, while $\mathcal{A}_{A,r}$ and D_S can be deduced from the source program. Obviously, this system is affine (cf. Definition 7).

Analogously, (7.2) is equivalent to:

$$\textit{write constraint} \qquad \Pi_A\, \mathcal{A}_{A,r} - \Pi_S = \lambda_{A,r} D_S. \tag{7.5}$$

Note that (7.4) and (7.5) are systems of equalities, since the number of rows in Π_A and Π_S determines the dimensionality p of the processor grid. Thus, $\lambda_{A,r}$ also must have p rows. For a clearer presentation, we will not stress this aspect in the following.

Generating a combined constraint system Now, let us combine all constraints arising from (7.4) and (7.5) into one big system. For that purpose, we concatenate the matrices Π_A and Π_S in some order and obtain a vector Π, and also we concatenate the matrices $\lambda_{A,r}$ and obtain a vector λ. Finally, we generate coefficient matrices C and D which represent the constraints arising from (7.4) and (7.5) in our new big system. Thus, we obtain that the FCO condition is equivalent to:

$$C\,\Pi = D\,\lambda, \tag{7.6}$$

$$\lambda \geq 0 \tag{7.7}$$

Solutions of the constraint system Every solution of this system represents a placement for each statement and each array, together with a set of corresponding Farkas multipliers.

Let (Π, λ) be such a solution. Let us consider a specific reference r to A. There is a part of λ which corresponds to $\lambda_{A,r}$ in (7.4) or (7.5). If this part is null, then the distinguished reference entails no communication.

Note that the vector $(\Pi, \lambda) = (0, 0)$ is always a (trivial) solution to the system: all statements and data are mapped to processor 0.

Interpreting the solutions as a cone The set of all solutions is a cone $\mathcal{C}$ since it is closed both under addition and under multiplication by a non-negative constant, as one can verify easily.

Thus, for two solutions (Π, λ) and (Π', λ'), the sum $(\Pi + \Pi', \lambda + \lambda')$ is another solution whose residual communications are the union of the residual communications of the two initial solutions. This leads us to consider only extremal solutions, which cannot be obtained as a weighted sum of other solutions.

Lines and rays of the solution cone Let us now consider a line l in cone $\mathcal{C}$, represented by a solution $(\Pi, \lambda) \in \mathcal{C}$. Since l is a line, $(-\Pi, -\lambda)$ is also in $\mathcal{C}$. By (7.7) we obtain $\lambda \geq 0$ and $-\lambda \geq 0$ which implies $\lambda = 0$.

Conversely, if $(\Pi, 0)$ represents a ray, then $(-\Pi, -0)$ is also a solution and the ray is a line.

Hence, with the above considerations, lines correspond to communication-free placements, and rays correspond to FCO placements with, in general, residual communications. Furthermore, an analysis of the part of a ray representing λ allows to identify residual communications (those $\lambda_{A,r}$ which are non-zero after multiplication with D_S). If we assign a weight to each reference (e.g., an estimate of the number of transmitted values), we can associate a weight to each ray and select the one with minimal weight (remember that, in this context, lines will show up as zero-weight solutions).

Dimensionality of the solution However, we still have to consider parallelism. For this aspect, it is important to realize that each Π is a matrix with p rows, where p is the dimensionality of the processor grid. Let Π_S be the part of a solution which corresponds to statement S. The set of active processors for statement S is the image of the index set of S by Π_S. In order to preserve efficiency, we want this set to have the same dimensionality as the processor grid (however, this dimensionality cannot be higher than the dimensionality of the index set of S). Finding the dimensionality of the set of active processors is a simple rank computation; it can be applied to every solution independently and may be viewed as a filter that cancels solutions with a too small dimensionality.

Sketch of an FCO placement algorithm Thus, we propose the following algorithm:

- Construct the matrices C and D from the source program.

- Construct the rays and lines of the cone $\mathcal{C}$ associated to C and D.
- Filter out the rays and lines that do not satisfy the rank condition above.
- Compute the weight of each remaining ray or line.
- Select the ray or line with the smallest weight.

If a line has survived the filtering process, it has zero weight and will be selected, giving a communication free placement. If the selectee is a ray, it will specify an FCO placement with minimal communication volume. Lastly, if there are no survivors, then the problem has no FCO placement of the required dimensionality.

Evaluation of the algorithm We cannot claim that the placement we find in this way is the best one, in the sense of giving the best speedup. However, if the weights we assign to communications are estimates of the communication volumes, then our algorithm is a greedy solution to the problem of finding a minimum-communication FCO placement.

Let us note that the severity of the filtering increases with the dimensionality of the processor grid. Hence, we can always try again with a grid of fewer dimensions. In general, the higher the dimensionality, the higher the volume of residual communications, but also the higher the bandwidth of the communication network. Since the relative importance of these two opposing factors depends on details of the architecture, the best choice can only be found experimentally.

Let us now apply our algorithm to our motivating example and, thereby, also see how the matrices C and D in Equation (7.6) are constructed automatically.

Example 69. The access matrices for the code in Figure 2.1 have been derived in Example 24; the matrix description of the index set is shown in Example 23.

Furthermore, let us assume that we are looking for a one-dimensional placement. For this case, the templates for a computation placement and a data placement are presented in Examples 30 and 33, respectively.

With these definitions, we can compose the FCO constraints for the writes, as presented in Equation (7.5):

$$\begin{aligned} \Pi_{tmp}\,\mathcal{A}_{tmp,T0} - \Pi_T &= \lambda_{tmp,T0}\,D_T \\ \Pi_a\,\mathcal{A}_{a,S0} - \Pi_S &= \lambda_{a,S0}\,D_S \end{aligned}$$

The constraints for the reads are, according to Equation (7.4):

$$\begin{aligned} \Pi_T - \Pi_a\,\mathcal{A}_{a,T1} &= \lambda_{a,T1}\,D_T \\ \Pi_T - \Pi_a\,\mathcal{A}_{a,T2} &= \lambda_{a,T2}\,D_T \\ \Pi_S - \Pi_a\,\mathcal{A}_{a,S1} &= \lambda_{a,S1}\,D_S \\ \Pi_S - \Pi_{tmp}\,\mathcal{A}_{tmp,S2} &= \lambda_{tmp,S2}\,D_S \\ \Pi_S - \Pi_a\,\mathcal{A}_{a,S3} &= \lambda_{a,S3}\,D_S \end{aligned}$$

The combined system *We now combine all these equalities to one big system, as indicated in Equation (7.6). The variables are the entries of all matrices* Π_* *and* $\lambda_{*,**}$*, where a* $*$ *is a wildcard for any existing index:*

- Π_T *contributes 4 variables (cf. Example 30)*
- Π_S *contributes 5 variables (cf. Example 30)*
- Π_{tmp} *contributes 4 variables (cf. Example 33)*
- Π_a *contributes 4 variables (cf. Example 33)*
- *since* D_T *has 5 rows and appears in 3 constraints, we obtain* $5 * 3 = 15$ *Farkas multipliers concerning* T *(i.e.,* $\lambda_{*,T*}$*)*
- *since* D_S *has 7 rows and appears in 4 constraints, we obtain* $7 * 4 = 28$ *Farkas multipliers concerning* S *(i.e.,* $\lambda_{*,S*}$*)*

Thus, the combined system consists of 7 equalities, $15+28 = 43$ *inequalities for the Farkas multipliers, and* $4+5+4+4+15+28 = 60$ *variables.*

Note that each of these equalities is, in fact, a matrix equality, i.e., the equalities must be satisfied for every entry. To make this more obvious, we rewrite the constraint system as follows:

$$\Pi_{tmp}\,\mathcal{A}_{tmp,T0} - \Pi_T - \lambda_{tmp,T0}\,D_T = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.8}$$

$$\Pi_a\,\mathcal{A}_{a,S0} - \Pi_S - \lambda_{a,S0}\,D_S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.9}$$

$$\Pi_T - \Pi_a\,\mathcal{A}_{a,T1} - \lambda_{a,T1}\,D_T = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.10}$$

$$\Pi_T - \Pi_a\,\mathcal{A}_{a,T2} - \lambda_{a,T2}\,D_T = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.11}$$

$$\Pi_S - \Pi_a\,\mathcal{A}_{a,S1} - \lambda_{a,S1}\,D_S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.12}$$

$$\Pi_S - \Pi_{tmp}\,\mathcal{A}_{tmp,S2} - \lambda_{tmp,S2}\,D_S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.13}$$

$$\Pi_S - \Pi_a\,\mathcal{A}_{a,S3} - \lambda_{a,S3}\,D_S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.14}$$

In order to feed these constraints to our mathematical tool, the Polylib [Wil93], we must resolve these matrix equalities to basic equalities by considering every entry separately. In our case, each of the three constraints due to statement T *generates 4 equalities, and each of the four constraints due to statement* S *generates 5 equalities, so that we end up with* $12+20 = 32$ *equalities. This new system is now given to the Polylib in order to determine the lines, rays, and vertices of the described set.*

Implementation note: Rewriting the matrix equalities to basic equalities can be achieved automatically by using transposed matrices: if we transpose, e.g., Equations (7.8)–(7.14), we obtain several rows for each equation, and every row represents one basic equality. This is the input format we need for our mathematical tool.

In more detail: for every placement dimension, we rewrite the constraints in Equation (7.4) and (7.5) into normal form

$$\textit{FCO: read} \qquad \Pi_S^\top - \mathcal{A}^\top \Pi_A^\top - D_S^\top \lambda_{A,r}^\top = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \tag{7.15}$$

and

$$\textit{FCO: write} \qquad -\Pi_S^\top + \mathcal{A}^\top \Pi_A^\top - D_S^\top \lambda_{A,r}^\top = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{7.16}$$

where the length of the zero vector is the length of the iteration vector of statement S in homogeneous representation.

The combined system in normal form In order to combine these systems of constraints for all accesses into one system suitable for Polylib, we also convert Equations (7.6) and (7.7) to normal form:

$$\begin{pmatrix} C & D \end{pmatrix} \begin{pmatrix} \Pi \\ -\lambda \end{pmatrix} = \mathbf{0}, \tag{7.17}$$

$$\lambda \geq \mathbf{0}, \tag{7.18}$$

where $\mathbf{0}$ is a column vector of zeros of appropriate length.

Composing matrices C and D We combine all variables, i.e., the entries of all matrices Π_* and $\lambda_{*,**}$ within one long column vector, as already noted. Thus, if these variables are shifted to the right factor in Equation (7.17), the remainder of the constraints (7.15) and (7.16) for the left factor in Equation (7.17), i.e., for matrices C and D is as follows:

- every occurrence of $\Pi_S^\top$ contributes an identity matrix to C at those columns that match the rows of $\Pi_S^\top$ within $\begin{pmatrix} \Pi \\ -\lambda \end{pmatrix}$;
- every occurrence of $\mathcal{A}^\top \Pi_A^\top$ contributes $\mathcal{A}^\top$ to C at those columns that match the rows of $\Pi_A^\top$ within $\begin{pmatrix} \Pi \\ -\lambda \end{pmatrix}$;
- every occurrence of $D_S^\top \lambda_{A,r}^\top$ contributes $D_S^\top$ to D at those columns that match the rows of $\lambda_{A,r}^\top$ within $\begin{pmatrix} \Pi \\ -\lambda \end{pmatrix}$.

With this translation scheme, we can easily generate C and D by processing each constraint in turn and joining the obtained matrices, i.e., by concatenating them vertically.

Example 70. Let us continue our example. We combine the matrices in the following order: $\Pi_T, \Pi_S, \Pi_{tmp}, \Pi_a, \lambda_{tmp,T0}, \lambda_{a,S0}, \lambda_{a,T1}, \lambda_{a,T2}, \lambda_{a,S1}, \lambda_{tmp,S2}, \lambda_{a,S3}$*. Then, we obtain the following combined matrix:* $\left(\begin{array}{c|c} C & D \end{array}\right) =$

$$\left(\begin{array}{cccc|cccccccc}
-Id_4 & Zero_{4,5} & \mathcal{A}^{\top}_{tmp,T0} & Zero_{4,4} & D^{\top}_T & Zero_{4,7} & Zero_{4,5} & Zero_{4,5} & Zero_{4,7} & Zero_{4,7} & Zero_{4,7} \\
Zero_{5,4} & -Id_5 & Zero_{5,4} & \mathcal{A}^{\top}_{a,S0} & Zero_{5,5} & D^{\top}_S & Zero_{5,5} & Zero_{5,5} & Zero_{5,7} & Zero_{5,7} & Zero_{5,7} \\
\hline
Id_4 & Zero_{4,5} & Zero_{4,4} & -\mathcal{A}^{\top}_{a,T1} & Zero_{4,5} & Zero_{4,7} & D^{\top}_T & Zero_{4,5} & Zero_{4,7} & Zero_{4,7} & Zero_{4,7} \\
\hline
Id_4 & Zero_{4,5} & Zero_{4,4} & -\mathcal{A}^{\top}_{a,T2} & Zero_{4,5} & Zero_{4,7} & Zero_{4,5} & D^{\top}_T & Zero_{4,7} & Zero_{4,7} & Zero_{4,7} \\
\hline
Zero_{5,4} & Id_5 & Zero_{5,4} & -\mathcal{A}^{\top}_{a,S1} & Zero_{5,5} & Zero_{5,7} & Zero_{5,5} & Zero_{5,5} & D^{\top}_S & Zero_{5,7} & Zero_{5,7} \\
\hline
Zero_{5,4} & Id_5 & -\mathcal{A}^{\top}_{tmp,S2} & Zero_{5,4} & Zero_{5,5} & Zero_{5,7} & Zero_{5,5} & Zero_{5,5} & Zero_{5,7} & D^{\top}_S & Zero_{5,7} \\
\hline
Zero_{5,4} & Id_5 & Zero_{5,4} & -\mathcal{A}^{\top}_{a,S3} & Zero_{5,5} & Zero_{5,7} & Zero_{5,5} & Zero_{5,5} & Zero_{5,7} & Zero_{5,7} & D^{\top}_S
\end{array}\right)$$

where all matrices can be copied from Example 69.

We give this system to Polylib and obtain the following dual representation (cf. Equation (3.9)):

- *There are two lines:* $p^n_T = p^n_S = p^n_{tmp} = p^n_a = 1$ *and* $p^1_T = p^1_S = p^1_{tmp} = p^1_a = 1$ *(0 for all other variables). Unfortunately, these two lines have rank 0: the dimensions they span are not dimensions of the index set but constant dimensions, i.e., the lines represent solutions which map all computations to one fixed processor (whose number may be any linear combination of n and 1). Consequently, these solutions do not survive the subsequent filtering process.*

- *There is one vertex: the trivial solution (0 for all variables). Note that this is always the case, since the set of solutions is a cone.*

- *There are four rays:*

  1. $p^i_T = p^k_S = p^{1st}_{tmp} = p^{2nd}_a = \lambda^{1st}_{tmp,S2} = \lambda^{7th}_{tmp,S2} = 1$*, all other variables are 0, i.e.,* $\pi_T(i,j,n) = i, \pi_S(i,j,k,n) = k, \pi_{tmp}(x,y,n) = x, \pi_a(x,y,n) = y$*, and only the access to tmp in statement S is not local.*
  2. $p^i_T = p^k_S = p^{1st}_{tmp} = p^1_{tmp} = p^{2nd}_a = \lambda^{5th}_{tmp,T0} = \lambda^{1st}_{tmp,S2} = 1$*, all other variables are 0, i.e.,* $\pi_T(i,j,n) = i, \pi_S(i,j,k,n) = k, \pi_{tmp}(x,y,n) = x+1, \pi_a(x,y,n) = y$*, and the accesses to tmp in statements S and T are both not local.*
  3. $p^j_T = p^j_S = p^{2nd}_{tmp} = p^{1st}_a = \lambda^{1st}_{a,T2} = \lambda^{5th}_{a,T2} = \lambda^{3rd}_{a,S3} = \lambda^{7th}_{a,S3} = 1$*, all other variables are 0, i.e.,* $\pi_T(i,j,n) = j, \pi_S(i,j,k,n) = j, \pi_{tmp}(x,y,n) = y, \pi_a(x,y,n) = x$*, and the rightmost accesses to a in statements S and T are not local.*
  4. $p^i_T = p^k_S = p^{1st}_{tmp} = p^{2nd}_a = \lambda^{1st}_{tmp,S2} = \lambda^{5th}_{a,T1} = \lambda^{5th}_{a,T2} = 1$*,* $p^1_S = p^1_a = -1$*, all other variables are 0, i.e.,* $\pi_T(i,j,n) = i, \pi_S(i,j,k,n) = k-1, \pi_{tmp}(x,y,n) = x, \pi_a(x,y,n) = y-1$*, and accesses to tmp in S and the two accesses to a in statements T are not local.*

Interpretation *This means that we have no communication-free solution, but four extremal rays which correspond to (infinitely many) solutions that guarantee the FCO property. The minimum number of non-local accesses is 1, which is only achieved by the*

first ray. Thus, we end up with the optimal FCO placement $\pi_T(i,j,n) = i$, $\pi_S(i,j,k,n) = k$, $\pi_{tmp}(x,y,n) = x$, *and* $\pi_a(x,y,n) = y$, *yielding a single communication due to the access to tmp in statement* S.

So far, we have seen an algorithm that generates an FCO placement – if it exists. We have also applied it to an example and given some implementation hints. Thus, the task stated at the beginning of this chapter is solved.

In the remainder of this chapter, we shall first note some ideas for making the approach presented so far more flexible. Then, we develop an alternative approach which can be used in the case of the computer owns rule (cf. Remark 35). We conclude with a few remarks about our experiments.

7.3 On the use of redistribution

The ownership-driven approach has the drawback that an array has only one placement during the course of the entire execution of a program. This is unsatisfactory: many programs can be divided into successive phases which have different access patterns for the same array. Hence, we need the ability to determine a data placement, as presented so far, but also to change this data placement during program execution. Let us discuss this on an example.

```
        for i := 0 to n−1
S :       C[i] := 42
          for j := 0 to n−1
T :         A[i, j] := A[i, j−1] + C[i]
          endfor
        endfor
        for l := 1 to n−1
          for k := 0 to n−1
U :         B[l, k] := A[l, k] + C[l−1]
          endfor
        endfor
```

Figure 7.3: A source program that needs redistribution

Example 71. Consider the source program in Figure 7.3. Following Feautrier [Fea94], we place higher-dimensional arrays before we place lower-dimensional ones, because non-local accesses to higher-dimensional arrays lead to a larger communication volume.

Optimal placement *Hence, a good mapping is the following:* $A[x, y] \mapsto x$ *and* $\langle i, j; T\rangle \mapsto i$ *(this eliminates the dependence cycle inside* T*), and* $B[l, k] \mapsto l$ *and* $\langle l, k; U\rangle \mapsto l$ *(this eliminates the dependence from* T *to* U *due to* A *and enables a local store of* B*). This mapping avoids any communication due to the two-dimensional (hence, most important) accesses to arrays* A *and* B*.*

Furthermore, we map $\langle i; S\rangle \mapsto i$ *and* $C[z] \mapsto z$ *in order to eliminate communications due to accesses of* C *in* S *and* T*. This solution is optimal if we allow one mapping per array and per statement – even if, as in this case, everyone of the* n^2 *accesses to* $C[l-1]$ *in* U *causes a communication.*

Desired placement *A much better solution would be to remap array* C *between its uses in* T *and* U*. If we remap* $C[l-1]$ *to* l *before executing* U*, then* U *causes no communication. The cost for the redistribution is one read/store per element of* C*, i.e., the redistribution causes only linearly many communications.*

How can we modify our placement algorithm in order to find this solution? The first step is to split the first loop. Then we add redistribution points in the source program, i.e., technically, we add artificial statements that read all elements of the array to be redistributed and copy them to a new array (and update the subsequent accesses to the new array). This scheme has the added advantage of limiting the complexity of each elementary placement problem, thus improving the scalability of our approach.

After inserting redistribution points for array C *between the loops on* S *and* T*, and also between the loops on* T *and* U*, and applying our placement algorithm, we obtain:*

- *between* S *and* T*:* $C'[z] \mapsto z$
- *between* T *and* U*:* $C''[z] \mapsto z+1$

This means that we should not redistribute C *between* S *and* T*, but between* T *and* U *– the expected result.*

Potential for future work The central question in this approach is where to insert redistribution points, and for which arrays. One heuristics is to try redistribution along the edges of the acyclic condensation of the statement dependence graph, i.e., the edges that connect the strongly connected components of the graph [Har73]. On the one hand, this allows redistribution between different phases of an algorithm (where redistribution might be most important); on the other hand, it guarantees that the expensive remapping is not executed too often, esp. not executed repeatedly back and forth, since it forbids redistribution inside dependence cycles. Of course, other strategies can be imagined as well.

In addition, there are other possibilities to make placement algorithms more flexible (e.g., to allow replication of arrays or even redundant computations). Ongoing research is focusing on these generalizations [GFG02, FGL03]. In addition, piecewise affine placements can be introduced, e.g., via index set splitting (Chapter 6).

7.4 Another approach: dependence-driven placements

The placement algorithm presented in Section 7.2 computes one computation placement per statement and one data placement per array, i.e., it is an ownership-driven placement algorithm (cf. Remark 37). Let us now convert it to a *dependence-driven placement algorithm*, which is based on the computer owns rule (cf. Remark 35).

Principle As indicated in Section 7, a communication in the setting of the computer owns rule is generated for every space-mapped dependence whose source and destination are on different processors. Hence, the principal idea of this second approach to FCO placements is, that a communication is forward if every space-mapped dependence goes forward.

Deriving the method The construction of a dependence-driven FCO placement progresses along the same lines as above. There is one placement condition per dependence in the program.

Let a dependence d be represented as the relation

$$\textit{dependence} \qquad \{\langle i, n; S\rangle \longrightarrow \langle j, n; T\rangle \ : \ R_d \begin{pmatrix} i \\ j \\ n \end{pmatrix} \geq 0\},$$

where we assume that the dependence is representable as one polyhedron.

For every such dependence, we require the FCO property:

$$\textit{FCO property} \qquad \pi_S(i, n) \leq \pi_T(j, n). \tag{7.19}$$

This can be rewritten as

$$\left(\forall i, j, n \ : \ R_d \begin{pmatrix} i \\ j \\ n \end{pmatrix} \geq 0 \ \Rightarrow \ \Pi_T \begin{pmatrix} j \\ n \end{pmatrix} - \Pi_S \begin{pmatrix} i \\ n \end{pmatrix} \geq 0\right). \tag{7.20}$$

From this point on, the algorithm follows the same lines as above. We eliminate quantifiers with the help of Farkas' lemma, and obtain:

$$\textit{dependence constraint} \qquad \Pi_T - \Pi_S \, H_d = \lambda_d \, R_d, \tag{7.21}$$

where H_d is the matrix representing the h-transformation of dependence d.

Then, we find the rays and lines of the solution cone of this system of constraints, and select the best one.

As before, we demonstrate exemplarily, mainly for implementors of this method, how the necessary matrices are computed and composed, and how the solution of the mathematical tool is interpreted.

Example 72. Let us again consider the program in Figure 2.1. The dependences together with their ranges are computed according to Section 5.2:

$$
\begin{array}{lllll}
d_1 = & \langle i-1,j,i;S\rangle & \longrightarrow & \langle i,j;T\rangle & \text{where } 1\le i\le j-1, j\le n-1\\
d_2 = & \langle i-1,i,i;S\rangle & \longrightarrow & \langle i,j;T\rangle & \text{where } 1\le i\le j-1, j\le n-1\\
d_3 = & \langle i,j;T\rangle & \longrightarrow & \langle i,j,k;S\rangle & \text{where } 0\le i\le j-1, j\le n-1, i{+}1\le k\le n-1\\
d_4 = & \langle i-1,j,k;S\rangle & \longrightarrow & \langle i,j,k;S\rangle & \text{where } 1\le i\le j-1, j\le n-1, i{+}1\le k\le n-1\\
d_5 = & \langle i-1,i,k;S\rangle & \longrightarrow & \langle i,j,k;S\rangle & \text{where } 1\le i\le j-1, j\le n-1, i{+}1\le k\le n-1\\
d_6 = & \langle i-1,j,k;S\rangle & \longrightarrow & \langle i,j,k;S\rangle & \text{where } 1\le i\le j-1, j\le n-1, i{+}1\le k\le n-1
\end{array}
$$

In matrix notation, we have for the range matrices R_d:

$$
R_1 = R_2 = \begin{pmatrix} 0 & -1 & 1 & -1\\ 1 & 0 & 0 & -1\\ -1 & 1 & 0 & -1 \end{pmatrix}
$$

$$
R_3 = R_4 = R_5 = R_6 = \begin{pmatrix} -1 & 1 & 0 & 0 & -1\\ 0 & -1 & 0 & 1 & -1\\ 1 & 0 & 0 & 0 & 0\\ 0 & 0 & -1 & 1 & -1\\ -1 & 0 & 1 & 0 & -1 \end{pmatrix}
$$

The h-transformations are

$$
H_1 = \begin{pmatrix} 1&0&0&-1\\ 0&1&0&0\\ 1&0&0&0\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix} \qquad H_2 = \begin{pmatrix} 1&0&0&-1\\ 1&0&0&0\\ 1&0&0&0\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix}
$$

$$
H_3 = \begin{pmatrix} 1&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1 \end{pmatrix} \qquad H_4 = \begin{pmatrix} 1&0&0&0&-1\\ 0&1&0&0&0\\ 0&0&1&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1 \end{pmatrix}
$$

$$
H_5 = \begin{pmatrix} 1&0&0&0&-1\\ 1&0&0&0&0\\ 0&0&1&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1 \end{pmatrix} \qquad H_6 = \begin{pmatrix} 1&0&0&0&-1\\ 0&1&0&0&0\\ 0&0&1&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1 \end{pmatrix}
$$

Since dependence d_4 and d_6 have the same range and the same h-transformation, we ignore dependence d_6.

Including the necessary trivial inequality $1\ge 0$, we have $2*4+3*6=26$ inequalities, i.e., 26 Farkas multipliers. The template for the computation placement is again (cf. Example 30)

$$
\Pi_T = \begin{pmatrix} p_T^i & p_T^j & p_T^n & p_T^1 \end{pmatrix} \quad \text{and} \quad \Pi_S = \begin{pmatrix} p_S^i & p_S^j & p_S^k & p_S^n & p_S^1 \end{pmatrix}
$$

As in Example 69, these templates contribute $4 + 5 = 9$ *further variables, but now, in contrast to Example 69, there are no variables due to a data placement. In total, we have a system with* $26 + 9 = 35$ *variables, 5 matrix equalities which become* $2 * 4 + 3 * 5 = 23$ *basic equalities, and 26 inequalities:*

$$\left(\begin{array}{c|c} C & D \end{array}\right) = \left(\begin{array}{cc|ccccc} Id_4 & -H_1^\top & -R_1^\top & Zero_{4,4} & Zero_{4,6} & Zero_{4,6} & Zero_{4,6} \\ \hline Id_4 & -H_2^\top & Zero_{4,4} & -R_2^\top & Zero_{4,6} & Zero_{4,6} & Zero_{4,6} \\ \hline -H_3^\top & Id_5 & Zero_{5,4} & Zero_{5,4} & -R_3^\top & Zero_{5,6} & Zero_{5,6} \\ \hline Zero_{5,4} & Id_5 - H_4^\top & Zero_{5,4} & Zero_{5,4} & Zero_{5,6} & -R_4^\top & Zero_{5,6} \\ \hline Zero_{5,4} & Id_5 - H_5^\top & Zero_{5,4} & Zero_{5,4} & Zero_{5,6} & Zero_{5,6} & -R_5^\top \end{array}\right)$$

When giving this system to Polylib, we obtain the following dual representation:

- *There are two lines:* $p_T^n = p_S^n = 1$ *or* $p_T^1 = p_S^1 = 1$*, all other variables are 0, which, again, represent the trivial placements to any linear combination of* n *or* 1 *without spanning a parallel dimension.*
- *There is again one vertex: the trivial solution (0 for all variables).*
- *There are five rays:*
  1. $\pi_T(i, j, n) = i$ *and* $\pi_S(i, j, k, n) = k$ *and* $\lambda_3(a, b, c, d, e, f) = a + f$*, all other variables are 0, i.e., this solution avoids communications for all dependences except* d_3*;*
  2. $\pi_T(i, j, n) = j$ *and* $\pi_S(i, j, k, n) = j$ *and* $\lambda_2(a, b, c, d) = a + d$ *and* $\lambda_5(a, b, c, d, e, f) = a + d$*, all other variables are 0, i.e., this solution avoids communications for all dependences except* d_2 *and* d_5*;*
  3. $\pi_T(i, j, n) = i$ *and* $\pi_S(i, j, k, n) = k - 1$*, which avoids communications for dependences* d_4 *and* d_5*;*
  4. $\pi_T(i, j, n) = i$ *and* $\pi_S(i, j, k, n) = i + 1$*, which avoids communications for dependences* d_1 *and* d_2*;*
  5. $\pi_T(i, j, n) = i$ *and* $\pi_S(i, j, k, n) = i$*, which avoids only the communications for dependences* d_3*.*

Evaluating the solutions *Note that all five rays guarantee the FCO constraint, and they all span a parallel dimension, i.e., they are valid solutions – in contrast to the two lines. However, when considering the weight, the first solution is the best: only one dependence leads to communication, and, by chance, the residual communication starts from* T *which is only two-dimensional.*

In order to be able to compare it with the ownership-driven approach, we explicitly denote the corresponding data placement due to the computer owns rule: $\pi_{tmp}(x, y, n) = x$*, and* $\pi_a(x, y, n) = y$*. (Note that we only have a fixed data placement since, in this example,*

the elements of tmp are never overwritten, and the overwriting of a for different instances of the loop index i always takes place by the same processor, i.e., k.)

With this information, we see that, in this example, the solutions of the ownership-driven approach (Example 70) and the dependence-driven approach are, by chance, equivalent. Both approaches find out that the optimal solution leads to a single communication due to the read of tmp in statement S, whose value is set by statement T.

Technical remarks: If the domain (range) of a dependence is overestimated in order to be describable as a union of polyhedra (e.g., for affine but non-uniform dependences), we obtain, in general, overly restrictive constraints for FCO, and therefore fewer solutions.

Another point is that the set of dependences to be taken into account depends on the architecture and the communication scheme. For instance, on distributed memory machines and two-way communications (e.g., send/receive), we need not create communications due to anti or output dependences. Therefore, we only require the FCO constraint for true dependences in this setting (cf. Section 13.4).

7.5 Experiments

Our placement algorithm has been implemented as an extension to the LooPo parallelizer and tested on about 10 kernels, some real and some artificial. We found FCO placements for all examples, and even some communication free placements. The largest kernels are "burg" (a signal processing kernel with 22 lines of code, which is based on Burg's algorithm [Bur75]) and "LCZOS" (a Lanczos iteration with 60 lines, which is a numerical method for computing Eigenvalues [Saa93]). The algorithm has reduced 44 potential communications to 13 in the first case and 64 to 2 in the second case.

Chapter 8

Basic necessity: granularity control

The methods described so far aim at the extraction of maximal parallelism. However, the resulting performance is poor due to the fine grain of the found parallelism. The goal of this chapter is to provide methods for improving performance.

8.1 Overview

Tiling is a well-known technique for sequential or parallel program optimization which focuses on the efficient execution of nested loops. The first papers on tiling date back to the mid-Eighties [MF86, RAP87, Wol89c]. The key idea of tiling is to coalesce operations, i.e., to partition the index set. Hereby, the application domain determines which operations belong together and, therefore, should be coalesced.

Components of a tile

For technical reasons, the sets of operations to be coalesced are congruent, non-intersecting polyhedra, so-called *tiles*, the union of which covers all operations. In many cases, an additional restriction is that the tiles must be *parallelepipeds*, i.e., a parallelogram of arbitrary dimensionality.

Shape, form, and size of a tile With the restriction to parallelepipeds, a tile is traditionally determined by three components:

- the *shape* is determined by d families of parallel hyperplanes [HCF99]: the bounding hyperplanes of the parallelepiped are fixed except for parallel translation; alternatively, we can determine the shape by the directions of its spanning vectors;
- the *form* is determined by the ratio of the lengths of the parallelepiped in the d dimensions;
- the *size* is determined by the scaling factor, which is the same for all d dimensions.

Width of a tile The *width of a tile* in a dimension is the extent of the tile in that dimension. We use this term more frequently than form and size in this thesis since, in our approach, we occasionally obtain fixed widths in some dimensions, i.e., we cannot decide a form in the traditional sense and then scale it to the desired size (cf. Chapters 9–11). Scaling in our setting is only allowed in the dimensions without a fixed width.

Example 73. Consider the code and dependence graph in Figure 8.1 (we only depicted the flow dependences, not the output dependences). It denotes the kernel of one-dimensional SOR, as, e.g., used to solve partial differential equations by finite difference methods, taken from the Web [Sal].

```
for j := 1 to m
  for i := 2 to n−1
    A[i] := (A[i−1] + A[i+1])/2.0
  endfor
endfor
```

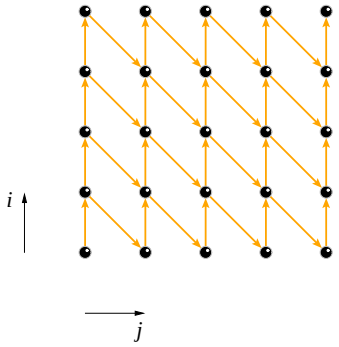


Figure 8.1: One-dimensional SOR – the source

Tile shape *Without going into further detail (this will be the topic of Chapter 9), we decide that the hyperplanes bounding the tile shall be parallel to the two dependence vectors. This decision determines the vertical and the diagonal hyperplanes in Figure 8.2, left, and, thus, the tile shape. As a representative of the two families of parallel hyperplanes, we choose the ones containing the origin of the coordinate system. Then, our two families of bounding hyperplanes are given by $j = 0$ and $i = -j$. The directions of the spanning vectors in our case are equivalent with the direction of the dependence vectors.*

Tile form and size *Let us assume, we want both dimensions to be of equal length. This determines the tile form. In Figure 8.2, right, we have depicted tiles with this form, and with a size of 4 operations per tile. The width is 2 in every dimension.*

Unbounded tiles

Sometimes, we only want to tile a single dimension, leaving all other dimensions unchanged. In this case, the tiles are unbounded in all but one dimensions, i.e., the tiles are polyhedra, not polytopes, and the dimension that is subject to tiling is simply partitioned into pieces of fixed width.

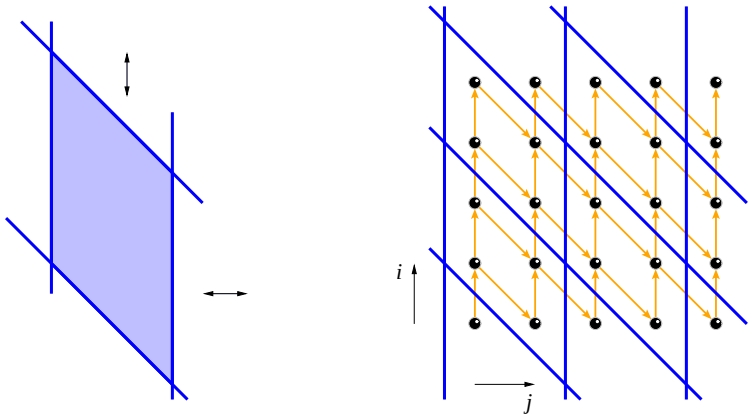


Figure 8.2: One-dimensional SOR – tiled dependence graph

On the one hand, this technique can be considered to be more restricted than tiling because it requires that the hyperplanes are orthogonal to the coordinate axes; on the other hand, it can be considered more general since classical tiling uses bounded tiles.

In this thesis, we consider this simple partitioning technique as a restricted form of tiling, because we do not enforce tiles to be bounded.

General restriction The only restriction we impose on the tiles is quite usual in the literature: we require tiles to be (potentially unbounded) parallelepipeds.

Organization of the rest of this chapter

The two main application areas of tiling, cache optimization and granularity control of parallelism, are compared in more detail in Section 8.2.

Section 8.3 then focuses on the traditional way of tiling in the context of parallelization, which is the application domain addressed in this thesis. It gives a brief historical summary and describes the state of the art.

Section 8.4 presents our tiling approach: it describes the combination of space-time mapping techniques followed by tiling methods. This section is restricted to traditional tiling methods, i.e., these techniques do *not* take advantage of the fact that the loops are sequential or parallel.

Finally, Section 8.5 gives an interpretation of tiling space dimensions, i.e., parallel loops, and time dimensions, i.e., sequential loops. Chapters 9–11 elaborate on these aspects in more detail.

8.2 Tiling for cache or for parallelism

There are two main domains using tiling: cache optimization and loop parallelization.

Tiling for cache locality For cache optimization, a tile should contain all operations which work on data that are close together in memory and, therefore, can and should reside in the cache simultaneously. Therefore, the shape of the tiles is determined by the access pattern of different array cells (for spatial locality) and the reuse pattern of the same array cells (for temporal locality); the size is mainly determined by the cache size [Wol89c, XH98].

In our view, tiling for cache optimization is a *subsequent* step to tiling for parallelism: one should determine first which operation is placed on which virtual processor, then which virtual processor is mapped on which physical processor, and finally, when the distribution of the operations – and the data – among the processors is known, which ordering of the operations to be executed on a single processor is cache-optimal. The reason is that the communication cost between different processors is typically orders of magnitude higher than the cost due to a cache miss.

We shall not focus on tiling for cache optimization any further.

Tiling for parallelism For loop parallelization, a tile should coalesce sufficiently many operations to make a communication phase worth-wile. Here, the shape of the tiles is determined by the data flow between the operations; the size is determined by the computation-per-communication ratio of the hardware on which the parallel program is to be executed, as well as by the number of physical processors (cf. Section 8.3).

Comparison At first sight, both tasks seem to be very similar, esp. since the reuse pattern in the case of cache optimization and the data flow in the case of loop parallelization both specify special kinds of dependences of the source program. However, there are also big differences which prevent an easy transfer of methods between the two application domains:

- In the framework of tiling for cache optimization, the data exchange modeled usually occurs between two partners (memory and cache); occasionally a hierarchy of cache levels is modeled. In the setting of loop parallelization, we have typically general affine communication patterns, and the number of partners is often not known before run time.
- For cache optimization, the limited capacity of the receiver (the cache) is *the* critical factor; for parallel programs, the receiver's capacity is not explicitly modeled, i.e., the methods assume unbounded capacity.
- Data that are stored in cache should be used early – they might not be in cache later. On the other hand, sent data are available for the receiver as long as desired.
- On typical hardware platforms, the programmer has only indirect control of caches – the actual cache management is performed directly by the hardware. (Note that this may be changing: on Intel's Itanium processors assembly programs can control the

cache level at which data should be stored.) In the setting of loop parallelization, the compiler generates all necessary messages – this is one of the most important tasks during code generation.

- Tiling for loop parallelization has not only the goal of reducing the amount of transferred data (as in the case of cache optimization), but also the conflicting goal of leaving enough parallelism.

Tiling for parallelism in hardware design The first papers on tiling have still another focus: they have been used in systolic array design, i.e., in the development of VLSI processor arrays. There, the goal is to obtain systems with a fixed number of VLSI processors, thereby reducing the number of communications. I.e., the subject is nearly identical with tiling for program parallelization, even though the VLSI community uses the term partitioning instead of tiling [MF86, TT93]. Also the principal procedure is quite similar.

8.3 Traditional tiling for loop parallelization: state of the art

Historical remarks In 1988, Irigoin and Triolet [IT88] introduced tiling as a loop parallelization method. The shape of the tile is first chosen such that deadlocks are avoided. For this purpose, they presented a sufficient criterion for the validity of tiling (or supernode partitioning, as they called it). The parallel program is then constructed by applying Lamport's hyperplane method [Lam74], which is a very simple space-time mapping technique for programs with uniform dependences only. Lastly, the size of the tiles is adjusted for minimal run time.

In the Nineties, much research was done to find good (or even optimal) sizes of rectangular or rhomboidal tiles [AR97, ARY98, OSKO95], and then to adapt the shape of the tiles to the dependences [SD90, BDRR94, Xue97a, Xue97b, HCF99, XH98, HS00], resulting in parallelepipeds as tiles.

Traditional way of tiling Tiling nowadays typically still proceeds the same way:

1. the three parameters shape, form, and size for a suitable (in some cases provably optimal) tile are computed;
2. the tiles are explicitly or implicitly mapped to space and time;
3. a new higher-dimensional loop nest is generated that enumerates the tiles and the points inside the tiles.

Example 74. Let us continue Example 73, in which we have determined the shape, from, and size of the tiles.

Inter-tile dependences *Before we can compute a space-time mapping of the tiles, we first need the dependences between the tiles. Figure 8.3, left, depicts all non-empty tiles which appear in Figure 8.2, together with all inter-tile dependences.*

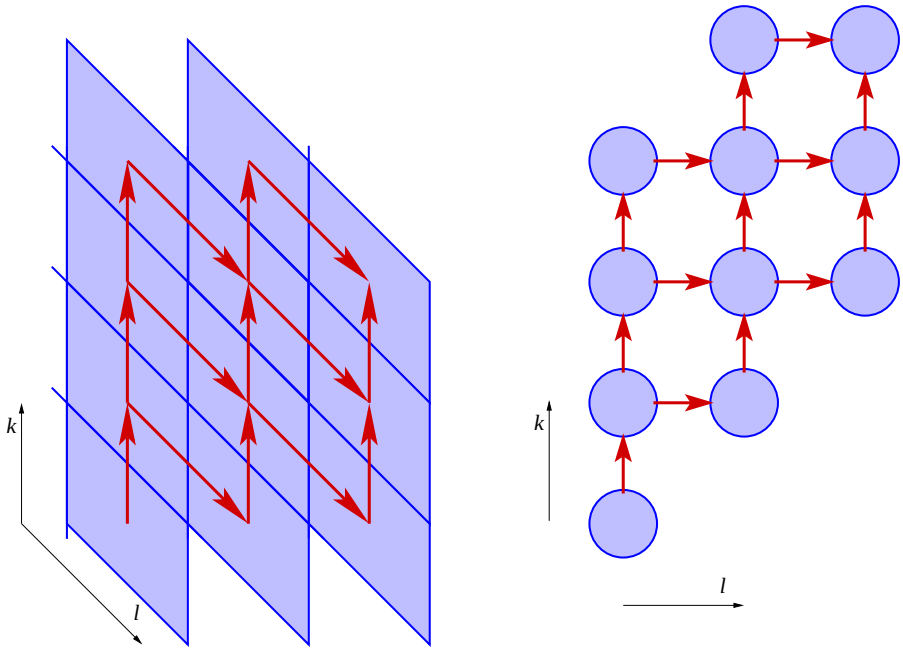


Figure 8.3: One-dimensional SOR – dependence graph of the tiles

Tile coordinates *As next step, we abstract from the fact that tiles haven an internal structure and consider them as a kind of super-computations, instead. The coordinate axes of these super-computations are determined by the vectors spanning the tile. They are also depicted in Figure 8.3, left, labeled k and l. (Aside: this is not an obligatory, but the simplest choice [AI91].)*

Since in this coordinate system the axes are not orthogonal to each other, we skew the l axes upwards and obtain the tile dependence graph in a new orthogonal coordinate system, depicted in Figure 8.3, right. There, each tile is represented as a big bullet, indicating that we have completed the abstraction from tiles with internal structure to super-computations. The distance vectors of the inter-tile dependences w.r.t. this new coordinate system are given by the distance vectors $(1, 0)$ *and* $(0, 1)$.

Parallelization *For the tile dependence graph in Figure 8.3, right, we now compute a space-time mapping, e.g.,* $\theta(k, l) = k+l$, *and* $\pi(k, l) = k$, *and, thereof, a parallel loop nest enumerating all tiles. Of course, further loops are generated that enumerate the interior of the tiles.*

```
for i := 1 to m
   for j := 1 to n
      A[i, j] := A[i−1, n−j]
   endfor
endfor
```

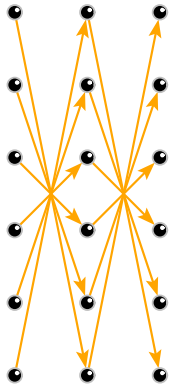


Figure 8.4: Full permutability may destroy all parallelism – the source

Full permutability Many of the methods mentioned have an initial phase which converts the source loop nest into a fully permutable loop nest, i.e., they first compute a skewing which makes all dependence directions non-negative in every component [Wol89c, BDRR94]. This ensures the correctness for the subsequent optimizing tiling phase. I.e., the task of finding optimal tiles can be solved independently of correctness issues. However, this approach to generating coarser grained parallelism may lead to an undesired loss of parallelism in the presence of non-uniform but affine dependences.

Example 75. Consider the program and dependence graph in Figure 8.4. If we convert the program to a fully permutable loop nest, we must skew the j dimension into the i dimension with a factor of n. The resulting dependence graph, given in Figure 8.5, shows that this initialization step of tiling destroys all parallelism.

On the other hand, we may consider the i dimension as schedule and the j dimension as placement. Since the placement dimensions are always fully permutable – in our case there is only one, which is trivially permutable – we can tile the space dimensions arbitrarily. E.g., we distribute the n iterations of the j loop across the physical processors and obtain a load-balanced parallel program.

8.3.1 Typical limitations of traditional tiling

Dependence representation

The most precise dependence description in most of the work cited is with dependence polyhedra. This level of abstraction results in a loss of parallelism compared to using, e.g., Feautrier's scheduling algorithm [Fea92a, Fea92b] – which works on a precise dependence description – followed by tiling.

Example 76. Consider the program and operation dependence graph in Figure 8.6. If we use the full precision of the dependence description, we obtain a schedule that gives us

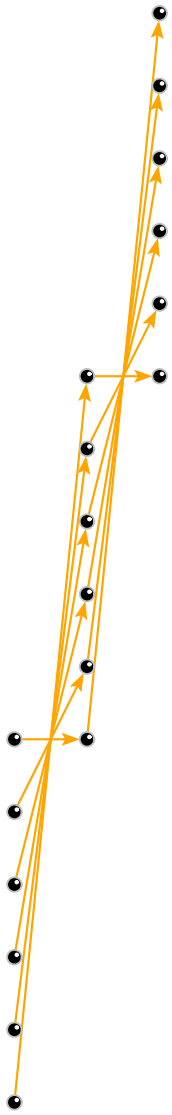


Figure 8.5: Full permutability destroys all parallelism – the target

```
      for i := 0 to n−1
        for j := 0 to n−1
S :       A[i, j] := A[i, j−1] + B[i−1, 0]
        endfor
        for k := 0 to n−1
T :       B[i, k] := B[i, k−1] + A[i, 0]
        endfor
      endfor
```

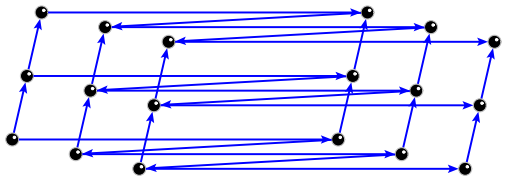


Figure 8.6: Dependence approximation by dependence polyhedra is harmful

```
      for i := 0 to n
        for j := 0 to n
S :       A[i, j] := ...B[i−1, j]...
T :       B[i, j] := ...A[i, j−1]...
        endfor
      endfor
```

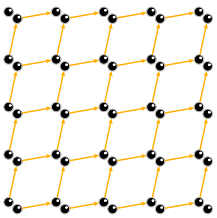


Figure 8.7: Per-statement view is crucial

linear execution time: Feautrier's method results in the one-dimensional schedule $\theta_S(i,j) = 2 * i + j$ *and* $\theta_T(i,k) = 2 * i + k + 1$ *[Fea92a].*

However, if we approximate the dependences by dependence polyhedra, we lose the information that the dependences between the two statements are at the same "level" in Figure 8.6, i.e., the values of j *and* k *are always identical. As a consequence, we lose all parallelism. Hence, the method of Darte and Vivien, which is based on dependence polyhedra, returns a two-dimensional, i.e., sequential, schedule:* $\theta_S(i,j) = (2 * i, j)$*, and* $\theta_T(i,k) = (2 * i + 1, k)$ *[DV97].*

Of course, no tiling can reintroduce parallelism in an obligatorily sequential program, i.e., if tiling is based on dependence polyhedra, it will also lead to a fully sequential target program.

Atomic loop body

Furthermore, many tiling methods work on a per-body basis, i.e., consider the statements in the body of a loop as one big atomic statement. This may again cost orders of magnitude of parallelism (and also performance).

Example 77. Consider the code in Figure 8.7. The program is perfectly nested and has two uniform dependences with distance vectors $(1,0)$ *and* $(0,1)$*. The according operation*

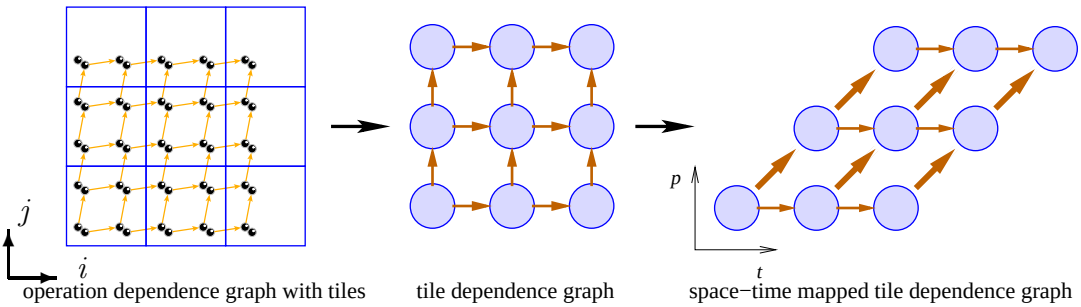


Figure 8.8: Traditional way of tiling

dependence graph is also given in Figure 8.7. Every pair of bullets represents the two body statements at some iteration vector (i, j). (A very similar example can also be found in Lim and Lam [LL98].)

Traditional solution *For that dependence graph, state-of-the-art tiling techniques determine the rectangle to be a valid tile shape. This solution is even optimal (with respect to communication minimization), if the dependence graph is a uniform dependence grid (as we assume so far).*

Tiling techniques then construct the tile dependence graph by converting the inter-operation dependences to an inter-tile level (Figure 8.8, middle), and finally lay out the tile dependence graph in space p and time t (Figure 8.8, right).

With this solution, every processor initiates one communication per executed tile, indicated by the bold arrows in Figure 8.8, right. Hence, the total number of communications per processor is linear in the number of tiles per processor.

Our suggestion *In contrast, if we consider the statements individually, the uniform dependence grid dissolves into independent dependence chains parallel to the diagonal (Figure 8.7, right). So, if we apply first the statement-based space-time mapping method, we can eliminate all communications by placing the operations of the first and the second statement on processor $\pi_S(i, j, n) = j - i + n + 1$ and $\pi_T(i, j, n) = j - i + n$, respectively (Figure 8.9). Of course, a subsequent trivial tiling, i.e., a partitioning of the p dimension, does not introduce any new communications. Hence, we obtain a parallel program with no communication at all.*

Perfect nesting

An immediate consequence of the inability of treating different statements differently is the restriction of many tiling methods to perfect loop nests. These methods can only be used for tiling arbitrarily nested loops if some preprocessing – e.g., the proposed space-time

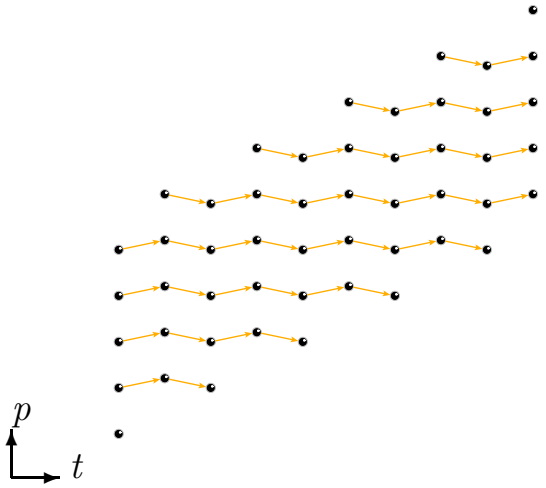


Figure 8.9: After space-time mapping

mapping – has been applied. (Note that this limitation does not exist for the phase which computes a fully permutable loop nest.)

Example 78. Both examples of Chapter 2 are imperfectly nested programs and we are not aware of a tiling method that computes the optimal tiles and the corresponding target program, e.g., for the LU decomposition algorithm in Figure 2.2 directly and automatically.

To overcome this limitation has been one of the main motivations for our approach.

Special approaches devoted to imperfectly nested loops There are specialized methods for the treatment of imperfectly nested loops [SL99, SL00, AMP00a, AMP00b]. Unfortunately, these methods have been developed specifically for cache optimization, and tiling methods for cache optimization cannot be used directly in tiling for coarse-grained parallelism, even if both aim at improving locality, as we have seen in Section 8.2.

Apart from the different purpose of these specialized methods, the approaches are also quite different from ours. Song and Li [SL99] use a program model which is more restricted than ours: it only allows one enclosing loop whose body contains several loops. In addition, this outermost loop is not subject to tiling.

Newer work of Song and Li [SL00] is dedicated to DSP applications and is based on the idea of unroll-and-jam. In this technique, several iterations of an outer loop are unrolled so as to increase the stride of this outer loop. Therefore, the body statements must be copied correspondingly often – also inside the inner loops. Still, the work focuses on the interaction of one surrounding loop and its next inner loop.

Another idea is to embed the index sets of the different statements in one new, common index set. Teich and Thiele, who extended tiling in the context of VLSI design to allow

more irregularities, noted long ago that this is always possible [TT93]. However, this idea of one common index set leads directly to the problem demonstrated in Example 77.

Subsequently, Ahmed, Mateev and Pingali [AMP00a, AMP00b] proposed a similar approach: it works on a space whose dimensionality is as high as the number of loops in the program (7 for the LU algorithm). This high-dimensional domain is the subject of tiling. In addition, the embedding of the original index sets into the product space offers a very wide range of choices, from which one is chosen heuristically.

In our approach, in contrast, the dimensionality is minimal: the maximum nesting depth of the source program (3 for the LU algorithm). The embedding is the space-time mapping, which yields provably optimal parallelism (for a restricted application domain). The reduced dimensionality is crucial, since the time complexity of space-time mapping methods is usually exponential in the dimensionality.

A non-specialized approach In addition to the technical differences between these specialized methods and our approach, there is a fundamental difference: as we shall see in Section 8.4, specialized methods for imperfectly nested loops are not necessary, i.e., we are going to present a unified approach for whatever nesting of loops.

8.3.2 Tiling for communication cost reduction

One of our main goals is to reduce communication cost. Let us try to classify existing tiling approaches for parallelism with respect to this goal.

Ignoring communication A first approach which is, among others, intended for distributed systems simply ignores communication cost and minimizes only the idle time instead [HCF97, HCF99, DDRR98]. On the one hand, this estimate for the execution cost can be far from reality. On the other hand, our task is much simpler because space tiling does not affect the idle time, and for tiling time we have proved that only a very limited form of rectangular tiles is valid (cf. Corollary 103). Note that this is not a restriction, but a feature of the preceding scheduling phase: all operations are already "aligned" to logical time dimensions. Hence, we need neither consider non-rectangular tile shapes nor restrict ourselves to special shapes of index sets (such as, e.g., [DDRR98]), in order to find time tiles which minimize the idle time – provided the schedule is minimal, i.e., it has minimal latency.

Communication volume A second approach minimizes the communication *volume* [Xue97a], not taking into account the number of startups. But, in state-of-the-art parallel architectures, the communication startup cost is much higher than the cost of transferring an additional value, so this should be taken into account. In addition, as stated in Chapter 3, the preceding space mapping already focuses on reducing the total communication volume.

Synchronizations A third approach reduces the number of global *barrier synchronizations* necessary, i.e., the number of communication phases [LF97, LL98, ARY98]. This is a more abstract, and hence, less precise cost function than the number of startups, since it does not take into account the number of communications and, thus, the number of communicating processors per communication phase. Some of the research of this approach [LF97, ARY98] is additionally restricted to rectangular partitioning, e.g., due to the limitations of BLOCK distribution in HPF. Furthermore, some techniques [ARY98] do not coalesce messages, but issue one communication per variable.

In contrast, we shall present in Chapter 9 a method that computes the optimal tiles w.r.t. the number of startups and guarantees that there is at most one communication between any pair of processors at any time step.

Total execution cost There is also a fourth approach [ABRY01], which is based on a cost model (BSP [SHM97]) and which minimizes the execution time. This is a very hard task and is therefore limited to two-dimensional index sets and uniform dependences with additional constraints on the tile and index set borders. Similarly, we reduce execution time w.r.t. a cost model (cf. Section 11.3), but we cannot guarantee optimality since, first, we separate space and time dimensions (cf. Section 8.5) and, second, space-time mapping algorithms themselves do not guarantee the optimality of the overall execution time. This is the price which we pay for extending the applicability of tiling to arbitrarily nested loops with affine dependences. Our experiments (cf. Section 11.3) show that our analytical minimum is very close to the real minimum.

8.4 Using traditional tiling after a space-time mapping

In this section, we demonstrate how traditional tiling techniques can be integrated into the framework of space-time mapping.

We have already mentioned the traditional way in which tiling is applied to the sequential source program in order to increase granularity *before* the (explicit or implicit) parallelization, e.g., [WL91, ARY98]. In contrast, we suggest to apply tiling *after* the parallelization.

Note that, in the latter case, the initial idea and the technical basis of tiling is the same as in traditional ways of tiling, namely to coalesce iterations, but the purpose of tiling is different.

- In the traditional setting, tiling yields a description of the portions of computations that *may* (e.g., due to an initial phase converting a loop nest into a fully permutable form) and *should* (e.g., according to some cost model) be distributed among the processors; in that sense, tiling is said to *generate* parallelism.
- In our setting, the previously computed schedule guarantees correctness, and the previously computed placement specifies which operations are to be executed on

different processors (under the assumption that there is no bound on the number of processors). I.e., parallelism already exists at this stage, and the task of tiling is to *limit* parallelism to a practically useful amount by applying tiling techniques.

8.4.1 Benefits of tiling after space-time mapping

Let us now examine how already existing tiling techniques may benefit from a preceding space-time mapping.

Unique coordinates. If we tile after the space-time mapping, all operations of the parallelized loop nest are expressed in one unified coordinate system, consisting of space and time dimensions. Hence, we need no special methods for tiling imperfectly nested loops [SL99, AMP00b], but may reuse existing tiling techniques that have proved useful (cf. Example 79). (Aside: note that we apply these techniques before target code generation, since the latter usually represents the target index sets in the unified coordinate system by imperfectly nested loops in order to obtain an efficient enumeration (cf. Chapter 13).)

Identical tiles. As an immediate consequence of the unified coordinate system, we have the option to restrict ourselves to identical tiles for the entire space-time coordinate system.

Tiling power. An alternative option is to take advantage of the direct access to the logical execution time (the indices of the loops in time) – which only exists after a space-time mapping. This permits a time-dependent tiling. E.g., the tile size for the dimensions in space may depend on logical time, enabling some kind of static load balancing. Ongoing work explores this idea [Sei04].

Non-uniform dependences. The space dimensions are always fully permutable (and are extracted by more sophisticated techniques than just traditional loop skewing). Consequently, in our approach, any tiling of the space dimensions is legal – also for non-uniform dependences (cf. Example 75).

Number of processors. The space-time mapping reveals the number of virtual processors required for the parallel execution. Thus, when computing the tiles after space-time mapping, we can take this number into account when determining which virtual processors must be mapped to the same physical processor.

Flexibility of parallelization. If we coalesce operations of the source program, we may give up automatically detectable parallelism unnecessarily, e.g., parallelism which is exposed by iteration graph partitioning [Ban93] (except if we apply some specialized methods [D'H92]), or parallelism exposed only after index set splitting [GFL00]. Our approach can overcome this weakness of traditional techniques.

Another already mentioned limitation of many current tiling techniques is that they view the loop body as indivisible. We have seen in Example 77 that, if we apply a

```
for t1 := 0 to 3 * n - 4
  parfor p1 := 1 to n
    parfor p2 := 0 to n
      if (0 = t1 and 0 = p2) then
        L[p1, 1] := A[p1, 1]
      endif
      if (...) then
        U[1, p1] := A[1, p1]/A[1, 1]
      endif
      :
      :
    endfor
  endfor
endfor
```

Figure 8.10: Target structure of LU decomposition

statement-based parallelization method before tiling, we may end up with a program which requires orders of magnitude fewer communications than if we applied the same tiling method without or before another statement-based parallelization method.

8.4.2 Imperfectly nested loops in our approach

Let us demonstrate the idea that we need no special method for tiling imperfectly nested loops on an example.

Example 79. Consider again the LU decomposition algorithm. If we apply a space-time mapping to the code in Figure 2.2, we obtain, e.g., from the code generator of LooPo (Chapter 4), an (intermediate) target program, which is a perfect loop nest whose body statements are guarded by individual conditions (Figure 8.10).

Since this program is perfectly nested, we can apply any existing tiling method (provided that it can deal with the existing affine dependences). In this example, all dependences (except for the first two initialization steps) are contained in the dependence cone $(1+, 0+, 0+)$*, which allows for, e.g., rectangular tiling.*

Advantages First, in contrast to more brute-force methods [AMP00b, AMP00a], the nesting depth of the target program is only the maximal nesting depth of the imperfectly nested source program (instead of the number of loops in the source program; cf. Section 8.3.1). A second, even more important feature is that the merge of the different index sets is based on the dependence structure (instead of the program text; cf. Example 77). These two aspects are central advantages of our method.

Limitation On the other hand, note that our method does not distinguish the different source index sets (which have, in general, different dimensionalities), once they are embedded into a common target coordinate system. This is probably its most serious limitation. Hence, in extreme cases, some statements with small source index sets might enforce a globally unsuitable tile shape. In order to avoid this, we could apply index set splitting to the target program, and use different tilings for the different parts of the index set. However, since we have not met such a situation in practice so far, we leave this aspect for future work.

Summary Technically, the suggested combination of tiling after space-time mapping just replaces the preprocessing step of the usual tiling procedure: instead of a unimodular transformation (loop skewing) which converts the source loop nest to a fully permutable one (if possible), we use the more general affine space-time transformation (including all related sophisticated techniques, e.g., index set splitting, described in Chapter 6) – but, in contrast to the traditional setting, with the goal of extracting parallelism.

This new combination improves, at the same time, the applicability (Example 79) and/or the quality (Example 77) of existing tiling techniques.

From a more abstract point of view, the reason for these improvements is the fact that we apply more flexible techniques first, which means that we maintain flexibility and, thus, exploitable information as long as possible. In our case, this flexibility concerns the individual treatment of different statements.

8.5 A refined view: tiling space and/or tiling time

So far, we have treated the mapping to space and time in unison. Let us now consider the space and time loops separately. We shall see that tiling space loops reduces the communication *volume* and the number of communication *partners* per communication phase, and tiling time loops reduces the number of communication *phases*.

Space tiling Tiling space (processor) dimensions means aggregating neighboring virtual processors in blocks and mapping them to a common physical processor (this is also often called partitioning). Since all dependences are carried by time loops, we have no restrictions on the tiles for the space dimensions.

However, there are inter-processor dependences (i.e., communications) between any two successive time steps. In this context, our task is to find tiles which lead to a minimal number of communication partners. Chapter 9 focuses on this topic.

Note that, at this point, we ignore the communication volume. The reason is that the preceding space mapping already aims at a global minimum of the amount of data that needs to be communicated. However, it does not take into account startup costs.

A startup occurs every time a processor sends a message and, in the case of point-to-point communications, for every partner. Thus, as already mentioned in Section 8.3.2, it is a more precise measure of the communication cost (which we aim to reduce) than the

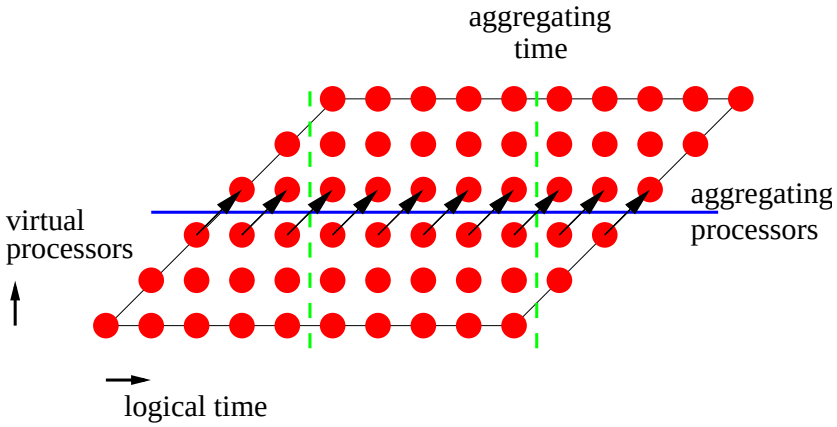


Figure 8.11: Principal idea of tiling time

number of global barrier synchronizations: it also considers the number of communication partners, i.e., processors, and the number of communications between each pair of processors (which can be reduced to at most 1; cf. Section 13.4.1). Since startup costs have been neglected during the space mapping, we must – and may – focus on this aspect during space tiling. The communication volume will become part of the discussion again in Chapter 11, since tiling time can, in general, change the communication volume, and this aspect is not being considered by the scheduler.

Time tiling Tiling time dimensions seems less intuitive, at first glance. The reason why it is useful is that, even if the granularity is increased by tiling parallel dimensions, we still have to issue communications between any two (logical) time steps (see the arrows in Figure 8.11)!

We can avoid this by way of *message vectorization*: we tile time dimensions, i.e., we aggregate time steps, as indicated by the dashed lines in Figure 8.11, and we allow communications only between two different aggregated time steps. For that purpose, we collect the data to be transferred locally on every processor during the execution of a tile, and generate the actual communication only at the tile borders. This satisfies the key restriction of Wolfe [Wol89a] that tiles run to completion without preemption. In contrast to logical time, we call the aggregated time *global time*.

Chapter 11 presents constraints and technical details for tiling time dimensions and suggests a cost model together with an algorithm which determines the optimal time tiles.

Individual tiling of space and time Tiling space and time dimensions separately might restrict the search space in comparison to tiling space and time dimensions in one step – however, this is not the case in our approach. Placement algorithms typically aim at reducing the communication cost [Fea94, DR95, Fea00], and we go by the premise that they do so successfully. Hence, we require that the tiling must honor the allocation, i.e.,

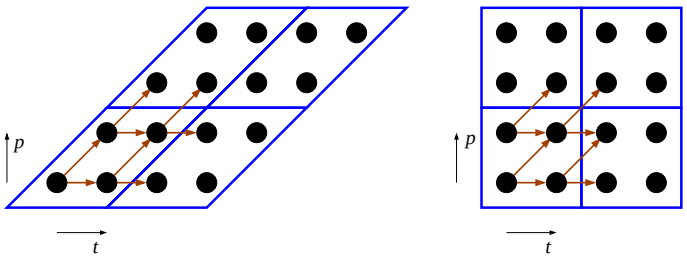


Figure 8.12: Change of communication startups when skewing the tiles

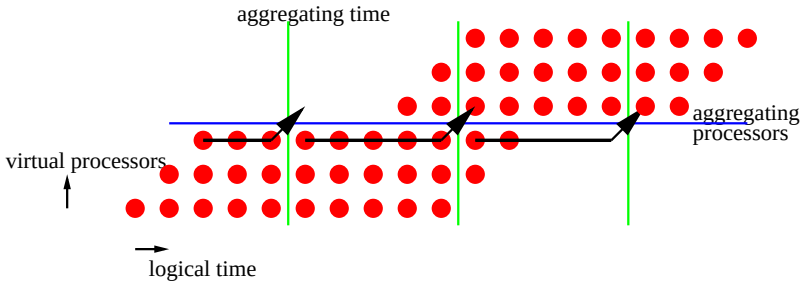


Figure 8.13: Target space after partitioning time

if the allocation maps two operations to the same (virtual) processor, then the tiling must keep them on the same processor. With this constraint, we have the following property.

Lemma 80. *For every legal parallelepiped tiling τ which respects a given allocation π and which causes c communication startups with a total volume v, there is a tiling τ' in which space and time dimensions are orthogonal to each other and which also causes c communication startups with volume v.*

Proof sketch. Let t be the number of time dimensions. Since τ respects π, we have t spanning vectors $\{v_1, \cdots, v_t\}$ of τ which are only in time. We create a new tiling τ' by skewing the remaining spanning vectors of τ (we do not skew the index set) so that they are orthogonal to the subspace spanned by $\{v_1, \cdots, v_t\}$. Note that this skewing does not change the number of communication partners or directions – nor the volume – but the number of communication startups (see Figure 8.12). However, due to the correctness criterion for τ, all dependences are inside the pointed cone spanned by the spanning vectors of τ; hence, we can postpone the receiver (without generating a cycle) until all data for it have been computed, and again initiate only a single communication for every pair of communication partners (see Figure 8.13).

Our suggested approach Based on the above ideas, we propose the following procedure.

1. We apply space-time mapping to the source program.

2. We tile the space dimensions. This minimizes the number of communication partners and maps the operations to the physically available processors. It cannot modify the number of time steps at which communication takes place. Note that the size is chosen such that we have as many tiles as we have physical processors (this aspect is discussed in more detail in Remark 81).

3. When we know the space tiles, we tile the time dimensions. This reduces the frequency of the communication startups with the communication partners just computed. Note that this second step does not change the communication partners, but it adapts the granularity of the parallelism to the given parallel architecture (for which we need the size of the space tiles).

In general, this separation cannot lead to a globally minimal number of communication startups because the two phases interact: first, a space tiling may disallow a desired time tiling and, second, the size of the space tiles influences the optimal size of the time tiles.

However, if a given space tiling does not prevent time tiling (as is frequently the case), we obtain the minimal number of communication partners by our separation – for a given number of processors to be used by the parallel program (cf. Chapter 9).

Chapter 9

Obvious task: space tiling

9.1 The task of tile optimization

Our goal is now to derive an algorithm that determines a tiling of a space-time mapped loop program, taking into account the number of physically available processors. We aim at minimizing the number of communication partners for the case of uniform dependences in the space dimensions. Note that, as already mentioned in Section 8.5, we may ignore the communication volume in this chapter. It will again become part of our discussion in Chapter 11.

Minimal number of startups If the computed space tiles do not prevent a tiling of time, we search for the minimal number of communication startups for a given number of physical processors in the following sense: if another tiling caused fewer startups, it would be due to fewer communication partners per communication phase (which is not possible if our method constructs tiles with the minimal number of partners), or due to fewer communication phases (which we could mimic by using coarser time tiles – down to one communication per processor, which is minimal for a not embarrassingly parallel program on a parallel computer with a given number of processors).

Since, after the tiling of time dimensions, we coalesce all messages for a given communication partner, our cost function is equivalent to minimizing the *number* of communications, ignoring the message *size*. This so-called HKT model was first used by Hiranandani et al. [HKT94]. It may appear over-simplified, but has been shown to be quite accurate [OSKO95, PSCB94, ARY98]. As explained in Section 8.5, this model is sufficient for computing space tiles after a space-time mapping since the space mapping already aimed at reducing the communication volume.

9.2 Technical overview

In order to obtain the optimal tiling, we determine first the shape of the tile (Sections 9.3 and 9.4). Then, we compute the form and the size of the tiles. In our case, we also

constrain the absolute lengths of the spanning vectors before determining the final size of the tiles (Section 9.5). Hence, when scaling the tiles to the desired size, we must exclude some dimensions. This size is easy to determine at this point, since every tile is executed by one physical processor, and the number of processors – and so the number of tiles – is a (parametric) input for our method.

Remark 81 (cyclic or block distribution). There are two ways of mapping tiles to a fixed number of physical processors: cyclic distribution and block distribution. Cyclic distributions are best suited for programs with dynamic control flow since they allow load balancing. However, they cause more communications than a block distribution. The reason is that communications are coalesced and vectorized per tile, and smaller tiles mean more tiles, i.e., more communications. Block-cyclic distributions are a compromise between the two extreme distributions.

We choose the block distribution since, first, our goal is to minimize the number of communications and, second, we focus on programs with a relatively static control flow. Programs with a very dynamic control flow should not be tackled with static parallelization or tiling methods but with appropriate methods at run time, e.g., the inspector-executor scheme. For load balancing of static control programs, we propose to explore the idea of time-dependent tile sizes instead of using cyclic tiling – in order not to lose the minimality of the number of communication startups. However, as already noted, this is still ongoing research [Sei04].

Our decision for block distribution leads to a one-to-one correspondence of the tiles in the space dimensions and the physical processors.

Note that the suggested procedure requires the number of needed virtual processors as input, in order to produce the target code for the physical processors. Fortunately, this information is provided by the space-mapping.

Structure of the technical presentation Technically, we start with *short* dependences, i.e., dependences whose lengths do not exceed the width of the tile in every dimension. The extensions for *long* and for non-uniform dependences are discussed in Chapter 10.

Also, we ignore the borders of the index set – this aspect is added in Section 9.5. I.e., up to Section 9.5 we assume that the index set contains more than one tile (indeed an unbounded number of tiles) in every dimension.

9.3 At most one uniform dependence per source loop

Exactly one dependence for every loop in the loop nest

Let us first consider the simplest case of n linearly independent uniform dependences, where n is the dimensionality of the index set.

Remark 82 (space-mapped dependence vectors). Dependences expressed in the source coordinates are transformed by the space-time mapping to dependences in the target (i.e.,

spatial and temporal) coordinates. Since only the spatial dimensions determine the communication partners, we use, until the end of Chapter 10, a simplified terminology: when speaking of a dependence vector d, we mean d projected onto its spatial components.

Lemma 83 (optimal space tiles). *Let D be a set of n linearly independent uniform dependence vectors $d_1, \cdots, d_n$ in an n-dimensional index set. Then, a tile whose one-dimensional faces are parallel to $d_1, \cdots, d_n$ incurs the minimal number of communication partners.*

Proof sketch. A one-dimensional face of a tile is the intersection of n–1 hyperplanes forming the tile. Hence, since every dependence d_i is parallel to one one-dimensional face, we have: d_i is parallel to all but one hyperplanes forming the tile. Thus, d_i crosses only one hyperplane. Since we assume short dependences compared to the tile size, d_i causes at most a single communication – covering all existing instances of d_i inside the tile. This local minimum (only one partner per d_i) is also the global minimum, since all d_i are linearly independent.

Consequently, the optimal tile shape is given directly by the n linearly independent uniform dependence vectors D.

Less dependences than loops

Let us briefly discuss the – very rare – case that we have m dependences, with $m < n$, where n is again the dimensionality of the index set. In this situation, we suggest not to apply a regular tiling of the n dimensions, but to simply partition the subspace of the $n-m$ dimensions that are not spanned by the dependence vectors. This results in a communication-free parallel program.

9.4 Arbitrarily many uniform dependences

Let us now consider the more typical case of m uniform dependences (for $m > n$). Here, we select a basis $D' = \{d_1, \cdots, d_n\}$ of the n-dimensional space and use these vectors as tile spanning vectors and as the basis for the tile coordinate system. In our approach, we suggest to take the basis vectors from the set of all dependences D; we come back to this point in Section 9.4.1. We express the remaining dependences $D'' = \{d_{n+1}, \cdots, d_m\}$ as linear combinations $L = \{l_{n+1}, \cdots, l_m\}$ of D'. How many communication partners arise from a linear combination?

Relative neighbor coordinates Since we assume short dependences, we can only reach neighbor tiles. Consequently, we may abstract from the precise linear combinations in L and use the vector $\bar{l}_j = (s_1, \cdots, s_n)$ as abstract representation of $l_j \in L$, where s_i is the sign of the i-th component of l_j.

Every such vector describes precisely one dependent neighbor tile. We call this representation the *relative neighbor coordinates* w.r.t. a given tile.

Counting communication partners

Let us now enumerate communication partners caused by a dependence $d_j \in D''$. In our discussion, we argue from the sender point of view, i.e., the communication partner is the receiver, but the situation is symmetric.

For clarity of presentation, we assume for a moment that also an intra-tile dependence leads to a communication; the receiver has relative neighbor coordinates $(0, \cdots, 0)$.

Relative neighbor coordinate 0 If, at some coordinate k, $k \leq n$, the entry $\bar{l}_j[k] = 0$, then d_k does not appear in the linear combination for d_j. Hence, d_j causes no communication along dimension d_k or, in other words, the relative neighbor coordinate of the receiver is 0 in dimension k.

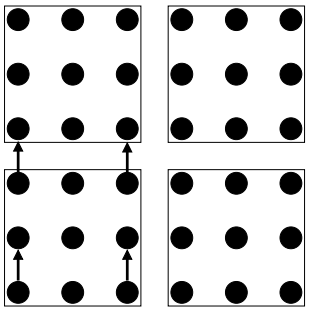


Figure 9.1: Receiver for relative neighbor coordinate $(0, 1)$

Example 84. Consider the horizontal dimension in Figure 9.1. The corresponding entry in the dependence vector $d_j = (0, 1)$ is 0. The figure makes obvious that, wherever the source of d_j may be inside the tile, the destination has always the same tile coordinate in the horizontal dimension as the source. In other words, the relative neighbor coordinate of the receiver is 0 in the horizontal dimension.

Relative neighbor coordinate 1 If, at some coordinate k, $k \leq n$, the entry $\bar{l}_j[k] = 1$, then d_k does appear in the linear combination for d_j with positive sign, i.e., d_j causes a communication which goes forward in dimension k, and we have a receiver with relative neighbor coordinate $+1$ in dimension k. Note that, in addition, also the tile with relative neighbor coordinate 0 in dimension k is a receiver: for a d_j whose source is far from the upper tile border in direction d_k, the destination is not beyond this border (we assume short dependences), i.e., the receiver has the same tile coordinate in dimension k.

Example 85. Consider now the vertical dimension in Figure 9.1. The corresponding entry in the dependence vector is 1. The figure shows that the destination tile are

- *the upper neighbor, i.e., it has relative neighbor coordinates $(0, 1)$, if the source of the dependence is close to the upper tile border, and*

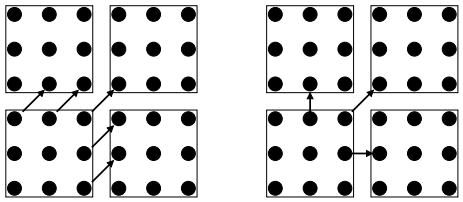


Figure 9.2: Dependences and communication partners

- *the tile itself, i.e., it has relative neighbor coordinates* $(0, 0)$*, if the source of the dependence is far from the upper tile border.*

Relative neighbor coordinate −1 The case that, for some coordinate k, $k \leq n$, the entry $\bar{l}_j[k] = -1$ is symmetric with the case that $\bar{l}_j[k] = 1$.

Algorithmic enumeration of the communication partners Let us now summarize these ideas by presenting a more formal scheme that allows to easily enumerate the number of communication partners (receivers) caused by a dependence $d_j \in D''$, which is represented by $\bar{l}_j$. To be a communication partner of a fixed tile, the relative neighbor coordinates of a tile (the receiver) must be

- 0 for those dimensions k for which $\bar{l}_j[k] = 0$, and
- 0 or x for those dimensions k for which $\bar{l}_j[k] = x$ $(x = \pm 1)$.

Example 86. We summarize the receivers for Figure 9.1 as follows:

- $\bar{l}_j[1] = 0$*, i.e., the receiver has relative neighbor coordinate 0 in the first dimension;*
- $\bar{l}_j[2] = 1$*, i.e., the receiver has relative neighbor coordinate 0 or* $+1$ *in the second dimension.*

Thus, the possible receivers are $(0, 0)$ *and* $(0, 1)$*, as expected.*

Example 87. Let us consider the dependence $\bar{l}_j = (1, 1)$ *in Figure 9.2, left. The first coordinate is 1. Thus, the first dimension of the neighbor coordinate must be 0 or 1. For the second coordinate, we have the same constraint. Hence, the communication partners are* $(0, 0)$, $(0, 1)$, $(1, 1)$, $(1, 0)$.

No self communication At this place, we are ready to consider also the special case that intra-tile dependences do not lead to communications. For this purpose, we simply remove the entry with relative neighbor coordinates $(0, \cdots, 0)$ from the list of possible communication partners.

Example 88. Hence, the possible receivers in Example 87 are $(0, 1)$, $(1, 1)$, *and* $(1, 0)$, *as indicated in Figure 9.2, right.*

An upper bound for the number of communication partners Finally, let us compute the maximal number of communication partners. In the worst case, $\overline{l}_j$ has no zero entries, i.e., we get $2^n - 1$ possible communication partners (all neighbors, and not the tile itself). The maximal number of communication partners over *all* dependences is $3^n - 1$, because, in every dimension, there might be a dependence with entry $+1$, and another dependence with entry -1, which, together, lead to -1, 0, and $+1$ as possible relative neighbor coordinates for the communication partner.

An algorithm to compute the tile shape

The above ideas lead to the following algorithm; it computes the tile shape in the case of m uniform dependences (for $m > n$).

Algorithm 89 (tile shape for uniform dependences).

1. For every possible partitioning p_k of D into D'_k and D''_k as above do:
   (a) For every vector $d_j \in D''_k$, compute the set of relative neighbor coordinates $C_{k,j}$ of the possible communication partners.
   (b) Compute the union U_k of all $C_{k,j}$, with $d_j \in D''_k$, and the n unit vectors of length n (representing the communication caused by the dependences in D'_k).
   (c) Set u_k to the cardinality of U_k.
2. Choose a partition p_k whose u_k is minimal.
3. The searched tile shape is spanned by the vectors in D'_k.

The step which dominates the time complexity of this algorithm is the computation of the possible communication partners of cost $O(2^n)$, which is executed $\binom{m}{n}$ times. Since n is a very small integer, this is acceptable.

Example 90. Let us consider the code in Figure 9.3.

After space-time mapping, we obtain a perfect loop nest with one dimension in time and two in space. From the originally 9 dependences only 4 dependence vectors must be considered for tiling (the others are made local by the placement and, hence, cause no communication). Their projections to the space dimensions are $d_1 = (1, 1)$, $d_2 = (1, -1)$, $d_3 = (-1, 0)$, *and* $d_4 = (0, 1)$ *(Figure 9.4, middle).*

Now we apply the above algorithm in order to select the two basis vectors for D'. *As examples for all pairs out of the 4 dependences, we present the result after Step 1(b) for two selected pairs in Figure 9.4. If we take* d_1 *and* d_3 *as basis (right), we end up with 6 communication partners; if we take* d_2 *and* d_3 *(left), we have only 5, which is minimal over all pairs. Hence,* d_2 *and* d_3 *are the tile's spanning vectors.*

```
for j := 2 to n
  for i := j to n
    X[j, i] := X[j−1, i] + Y[j+2, i] + Y[j+1, i]
  endfor
  for i2 := j+1 to n
    for k := 1 to j−1
      S[j, i2] := S[j, i2] + ...
    endfor
    Y[j, i2] := S[j, i2] + Y[j+1, i2−1] + X[j, i2]
  endfor
endfor
```

Figure 9.3: Source program for Example 90

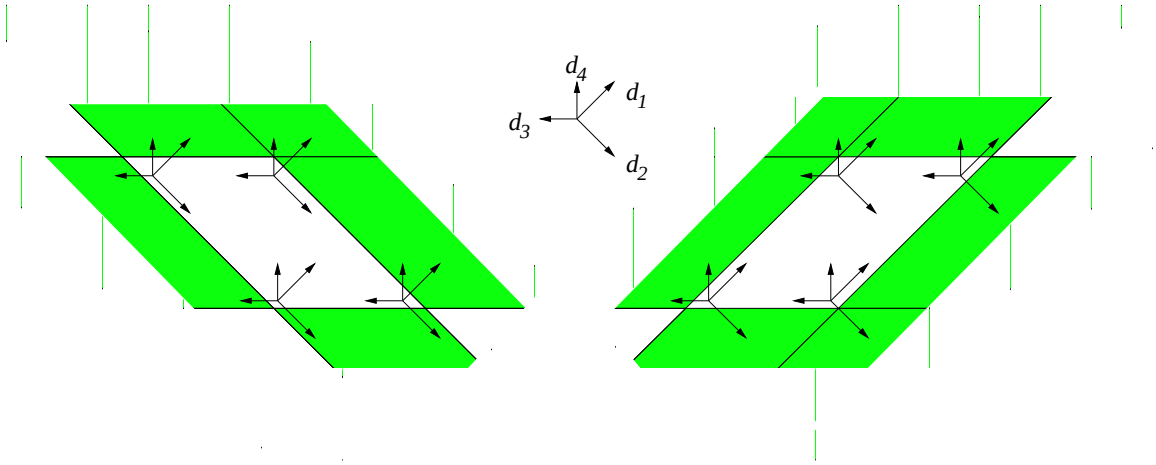


Figure 9.4: Communication partners for d_2 and d_3 (left) or d_1 and d_3 (right) in D'

9.4.1 Optimality considerations

Let us now check the optimality of this approach. The main limitation is that we restrict our search for the tile spanning vectors to the set of dependence vectors. Indeed, it turns out that this is not necessarily optimal, as the following example shows.

Example 91. Let us consider three perfectly nested loops with the following uniform dependences: $d_1 = (1, 1, 1)$, $d_2 = (1, 1, 0)$, $d_3 = (1, -1, 1)$, $d_4 = (0, -1, 1)$.

An optimal choice according to our presented algorithm is to take $D' = \{d_1, d_2, d_3\}$. *Thus,* d_4 *is represented as* $d_4 = d_1/2 - 2 * d_2 + d_3/2$, *which leads to communications with those tiles whose relative neighbor coordinates are* $(+1, -1, +1)$, $(+1, -1, 0)$, $(+1, 0, +1)$, $(0, -1, +1)$, $(+1, 0, 0)$, $(0, -1, 0)$, $(0, 0, +1)$. *From the three dependences in* D' *we obtain one additional communication:* $(0, 1, 0)$, *due to* d_2; *the communications due to* d_1 *and*

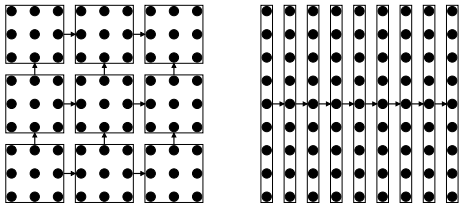


Figure 9.5: Square and thin but long tiles

d_2, i.e., $(1,0,0)$ and $(0,0,1)$, are already caused by d_4. Hence, in total, we have eight communications.

However, if we take $D' = \{d_1, d_2, b\}$ as a base, where $b = (1,0,1)$, we end up with fewer communications: $d_3 = -d_1 + b$, and $d_4 = -d_2 + b$, which leads to communication directions $(-1,0,1)$, $(-1,0,0)$, and $(0,0,1)$ for d_3, and $(0,-1,1)$, $(0,0,1)$, and $(0,-1,0)$ for d_4. Eliminating the doubly listed vector $(0,0,1)$ and adding the unit vectors due to d_1 and d_2, we obtain seven communications.

For a two-dimensional placement, we could not construct a similar example, neither could we prove the following conjecture up to now.

Conjecture 92 (Optimality for two-dimensional placements). *Algorithm 89 returns a tile shape that leads to the minimal number of communication partners for arbitrarily many uniform dependences.*

For three- and higher-dimensional placements, our algorithm is a greedy heuristics for finding optimal tile shapes in the sense that it first maximizes the number of dependences that only cause a single communication.

9.5 Tile form for uniform dependences

If we do not consider effects at the border of the index set, the form of the tiles has no influence on the number of communication partners in the case of uniform dependences. Thus, the only possibility of avoiding communications due to a different tile form is at the border.

Square vs. thin but long tiles In the presence of fine-grained parallelism and/or a small number of processors, tiles can become quite large. In practice, we can frequently collapse a complete dimension of the index set into a single tile and still have enough parallelism (as, e.g., in LU decomposition). It turns out that this way of using thin but long tiles reduces the number of communication startups, compared to nearly square tiles. For an illustration, see Figure 9.5 with $\sqrt{P} * (\sqrt{P}-1) * 2$ communications in the square and $P-1$ in the non-square case, where P is the number of tiles and also the number of processors.

Note that, if the tile extends to all of a single dimension i, this can eliminate *multiple* communications: all those communications whose relative neighbor coordinates have a non-zero entry in dimension i.

In order to determine which tile dimensions should be extended to all of the index set and which should not, we compute first the communication partners as already described. Then we count, for every dimension, how many communication vectors have a non-zero entry in that dimension. Startup costs are minimized if we extend the dimension with the highest count. (We select one arbitrarily if multiple dimensions have the same count.)

Chapter 10

Advanced task: heuristic extensions

In this section, we go beyond uniform dependences. Since optimality is hard to reach, we focus on heuristics which are simple to implement.

First, we look at long dependences. The interesting new phenomenon is that the width of the tile in some dimensions may influence the number of communications – not only due to boundary effects. Therefore, we derive an algorithm for choosing this tile width.

10.1 Long dependences

Formally, a dependence with a constant direction and a distance which is a symbolic parameter is not uniform. However, such dependences occur in real programs and should be treated more precisely than general affine dependences.

Definition 93 (long dependence). A dependence is called *long*, if its distance vector is constant but contains parameters, or if it is uniform but longer than a desired tile width.

In both cases, we assume that a long dependence always crosses at least one tile border. Let us now compute appropriate tiles for the case of long dependences.

First, we might want to consider the direction of a long dependence d as a candidate for D' (the set of tile spanning vectors). Since, by definition of long, d is longer than the width of a tile, we need to cut the length of d to 1, obtaining a vector d', which we then can put into D' (if we put d into D', d is not long anymore).

10.1.1 Communication partners

Let us now compute the number *partners* of communication partners for a long dependence d: $\mathit{partners} = \prod_{i=1}^{n} f_i$, where

- $f_i = 1$ if d maps the hyperplane, which limits the tile in dimensions i, to a parallel hyperplane, which itself limits another tile in dimension i, i.e., starting from a border of a tile, d reaches the equivalent border of another tile, and

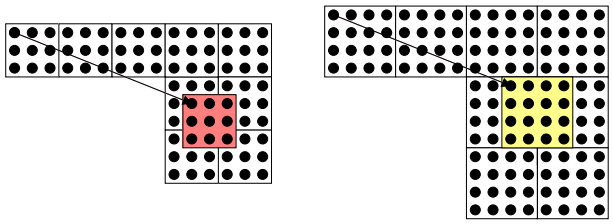


Figure 10.1: Long uniform dependences

- $f_i = 2$ otherwise.

Example 94. Let us count the number of communication partners for the leftmost tile τ in Figure 10.1.

- *In the left part, the tile width in both dimensions is 3, and the depicted dependence vector $d = (10, 4)$ (interpreted as an affine translation function, which is defined by the offset d) maps both the vertical and the horizontal boundary to the interior of another tile; thus, partners $= 2 * 2 = 4$, given by those tiles which intersect with the image of τ under d. (In the general n-dimensional case we have 2^n partners).*

- *In the right part, where the tile width in both dimensions is 4, the horizontal boundary is mapped to another horizontal boundary; thus partners is reduced to $2 * 1 = 2$.*

- *If we take a tile width of 2 in both dimensions (or 2 in the horizontal and 4 in the vertical dimension), both boundaries are mapped to boundaries again, resulting in partners $= 1 * 1$, i.e., a single communication.*

The effect of long dependences The important difference to the case of Sections 9.3 and 9.4 is that the widths of a tile in the various dimensions have an impact on the number of startups, as seen in the example.

Fortunately, due to this effect, the number of startups can only be reduced, i.e., in our heuristics, we may first treat long uniform dependences as candidates for the tile shape directions as above, and reduce the number of startups later when we decide the form of the tile (cf. Section 10.1.2).

Remark 95 (estimates for the number of startups). We can specify lower and upper bounds for the number of startups without considering the form: a long uniform dependence vector d causes one or two startups if (the direction vector of) d is a spanning vector of the tile, depending on the width of the tile in the direction of d, and between one and 2^n if not. This way, we can sometimes reduce the candidate set before minimizing the number of startups with the help of the form.

Overlapping communication partners A more difficult problem is that we have to check whether any of those communication partners overlap with the communication partners computed in Section 9.3 or with the partners caused by other long uniform dependences. The only precise solution method is to count the number of communication partners using Ehrhart polynomials (cf. Section 3.3.4). For simplicity, our heuristics assumes that the communication partners of long dependences do not overlap, and we leave a precise formalization for future work.

10.1.2 Appropriate tiles

Let us now derive an algorithm which computes constraints on the absolute widths of the tiles in order to reduce communication. As indicated above, we can halve the number of communication partners for a long dependence d along every tile dimension i in which the dependence expands, if the i-coordinate of d (in the basis of the tile spanning vectors) is an integer multiple of the tile width in dimension i. A suitable width is computed as follows.

Every dependence d can be expressed as $d = \sum_{i=1}^{n} \lambda_i * d_i$, where $d_i \in D'$, i.e., d_i is one of the tile spanning dependence vectors. Now, we try to factorize λ_i into $\lambda_i = s_i * a_i$, for every $i = 1, \cdots, n$ with $a_i \in \mathbb{Z}$, $d_i * s_i \in \mathbb{Z}$, and $s_i \geq 1$. The reason is as follows: s_i (the value we are interested in) is intended to be the scaling factor of the tile in the direction of d_i, i.e., $d_i * s_i$ will be the – integral – tile width in direction d_i. s_i must not be smaller than 1, because shrinking the spanning vectors would convert some uniform dependences to long dependences. Finally, from Section 10.1.1 we know that any integral multiple a_i of the tile width $d_i * s_i$ halves the number of communication partners due to dependence d along dimension i w.r.t. the general case.

Example 96. Let us consider the horizontal dimension in Figure 10.1, where we assume a tile spanning vector of $d_1 = (1, 0)$ for the horizontal dimension, and a long dependence of $d = 10 * d_1 + 4 * d_2$. We have the following possible factorizations of 10: $(s_1, a_1) \in \{(10, 1)$, (5,2), (2,5), $(1, 10)\}$. If we choose, e.g., $s_1 = 2$, we obtain a horizontal tile width of 2, which halves the number of communications in this direction, as mentioned in Example 94.

In order to find the optimum over all dependences, we enumerate, for every dimension i, the multi-sets S_i of all valid pairs (s_i, a_i) for all dependences. These multi-sets are finite (usually small), since $a_i \in \mathbb{Z}$ and $|a_i| \leq \lambda_i$ (since $s_i \geq 1$), and λ_i is fixed for every dependence. Then we select an $\hat{s_i}$ which occurs most frequently as first component in S_i. The resulting tile vertex $d_i * \hat{s_i}$ maximizes the number of dependences which satisfy the constraint $a_i \in \mathbb{Z}$, i.e., minimizes the number of communication startups (per time step) globally.

Example 97. Suppose we had a second long dependence $d' = (15, 8)$. We factorize 15, and see that $s_1 = 5$ appears both for d' and d, i.e., a horizontal tile width of 5 causes a quarter of the dependences (that are due to the long dependences), compared to the general case.

Note that this reduction can be achieved independently for the various dimensions. However, we need to keep (at least) one dimension which can be extended until the tile has the desired size (volume). In order to determine which dimensions we should keep, we can use a counting scheme for the various directions, which is similar to the one described previously for multiple dependences (Section 9.4).

10.2 Affine dependences

For the general affine case, we need to compute the precise number of communication partners in order to minimize the number of startups. Again, this can be solved precisely using Ehrhart polynomials: since, in our approach for communication generation, we already compute the sets of all processors which send/receive values, we just have to count the number of points in these sets, which are described as parameterized polytopes (cf. Sections 11.4.1 and 13.4).

However, since Ehrhart polynomials are difficult to handle, our implementation takes the direction vectors as an approximation and uses them as candidates for D' instead – if the number of uniform candidates is less than the number of space dimensions; otherwise, we determine the tile shape based on the uniform dependence vectors.

Example 98. Let us return to our LU decomposition algorithm in Figure 2.2. Let us apply Feautrier's placement method [Fea94] (as implemented in LooPo [GL97]), setting the desired dimensionality of the placement to 2. In this case, we obtain a placement which makes 10 of 19 dependences local, i.e., there are 9 dependences left which cause communications. All of them are non-uniform, some are not even forward communications. Therefore, we peel off the first two time steps, in which the first and second statement compute some initial values which are broadcast immediately.

The core of the program then has five remaining dependences. Their direction vectors are $(2, 1+, 0)$, $(1, 1+, 0+)$, $(2, 1+, 1+)$, $(1, 1+, 0)$, *and* $(1, 0, 1+)$, *where the first dimension is time and the other two are space dimensions.*

This leads to the following candidates for spanning vectors of the space tiles: $(1, 0)$, $(1, 1)$, *and* $(0, 1)$. *The communication-minimal choice thereof is* $(1, 0)$ *and* $(0, 1)$, *(using, e.g.,* $(1, 0)$ *and* $(1, 1)$ *as a spanning vector leads to an additional communication partner due to the dependence in direction* $(0, 1)$*). Therefore, we should use rectangular tiling in the space dimensions.*

Chapter 11

Surprising task: tiling time dimensions

Application scenario If the source program contains several dimensions in space, tiling these space dimensions usually leads to parallelism whose granularity is coarse enough to be efficient. However, this is not the case if the parallelization phase can only find a single space dimension.

Example 99. Consider the program fragment

```
for k := 0 to m
  for i := 1 to n−1
    A[k, i] := (A[k, i−1] + 2 ∗ A[k−1, i])/3
  endfor
endfor
```

After space-time mapping and tiling (partitioning) the one-dimensional processor space, we obtain a space-time mapped index set as in Figure 8.11 (without the dashed lines). The black arrows represent communications.

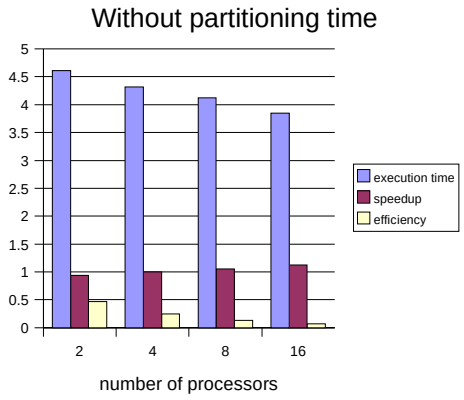


Figure 11.1: Execution times, speedup and efficiency after tiling space dimensions only

We executed the according target program on an SCI-connected network of 32 nodes with 512 MByte of main memory and two Intel Pentium-III, 1000 MHz processors per node (however, we used only one processor per node for our experiments). We took gcc-2.96 -O2 for the compilation and SCAMPI as communication library.

The execution times, speedups and efficiency for $(n, m) = (3 * 2^{17}, 2^7)$ *are given in Figure 11.1. The speedups for 2, 4, 8, and 16 processors are 0.94, 1.0, 1.05, and 1.13, which gives poor efficiency values of 0.47, 0.25, 0.13, and 0.07, respectively.*

Principal procedure For a situation as just described, we suggest to add another tiling phase, which increases the granularity of the parallelism further by coalescing virtual time steps.

As announced in Section 8.5, we are going to postpone and collect all send operations within a time partition and to execute the communications only at the end of the partition.

Of course, this requires a possible postponement in the scheduling of dependent tiles, compared with the original parallel program, as we have already seen in Figure 8.13. Note that the task of scheduling tiles is a typical subtask of existing tiling methods. However, in our case, i.e., under the constraints of Section 11.2, this schedule is program-independent: one must simply skew every space dimension (the physical processors) into time with factor $+1$.

Effect The general effect of tiling time dimensions is as follows. By increasing the tiles, we increase the computation volume of a tile and also the communication volume per message. This makes the parallelism coarser and reduces the number of communication startups. However, we pay a two-fold price for this coarsening due to postponing the start of the execution of the first tile for all but the first processor: a larger number of logical time steps and a worse processor utilization.

Example 100. Before going into technical detail, let us verify this idea along Example 99.

If we apply this idea to the space-time mapped index set of Figure 8.11, we obtain the index set in Figure 8.13, which shows the reduced number of communications and also the increased latency for the upper processor. The efficiency for the same problem size as above and for different values of the width of the time partitions is depicted in Figure 11.2. The presence of a maximum in the efficiency curve clearly points to a trade-off between fewer communications and shorter latency.

Remark 101 (implementing the postponement). In asynchronous systems, our typical target architecture, this postponement happens automatically by using blocking receive commands. In synchronous systems, we have to rewrite the loop nest accordingly, so that every tile starts executing after all predecessors have finished.

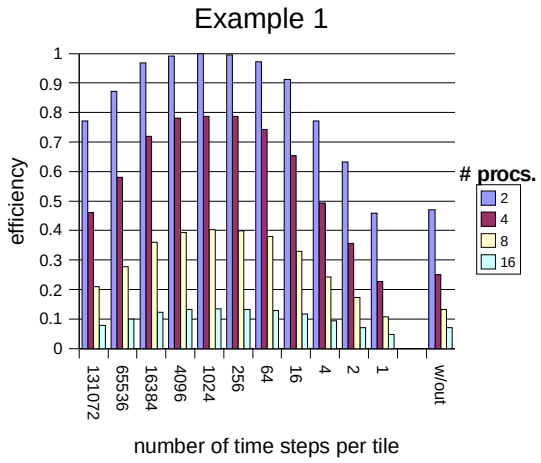


Figure 11.2: Efficiency after partitioning time

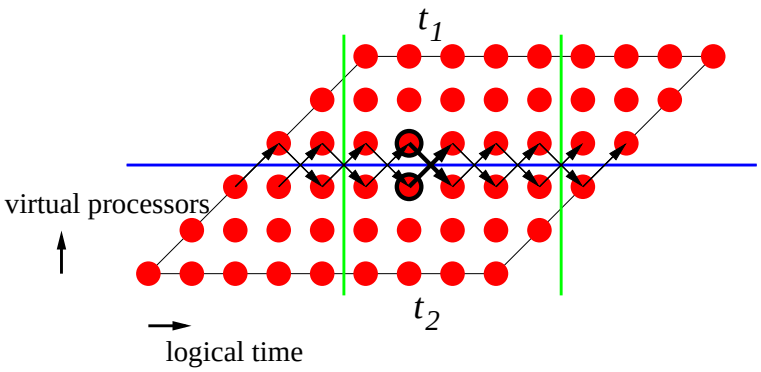


Figure 11.3: Tiling time is impossible

Danger The described postponement of tiles incurs the danger of deadlocks: suppose that some operation in tile t_1 generates data for a later operation in t_2 while an operation in t_2 generates data for t_1 (Figure 11.3).

It is clear that no deadlock can occur if the time is not tiled, i.e., the width of the time tiles is 1 (since, for a cycle, we would need at least two operations with different schedules in each tile).

A deadlock is also impossible if all communications roughly go into the same direction (e.g., from t_1 to t_2 but not the reverse). This is the property we want to achieve; a formal definition of this property and a placement algorithm that generates this property are presented in Chapter 7.

Organization of this chapter In the next sections, we focus on the following questions:

1. Which tile shapes can be used for tiling time dimensions (Section 11.1)?

2. When can we apply this idea of tiling time dimensions (Section 11.2)?
3. If tiling time is possible: how can we find the optimal tiles (Section 11.3)?
4. How can we estimate the execution time for a tiled program (Section 11.4)?

11.1 How to tile nested time loops

Let us first check how a nest of time loops can be tiled at all.

Lemma 102 (shape restriction for time tiles). *If, in some time dimension k, at least two successive iterations I and I' are aggregated to one tile $\mathcal{T}$, then all time dimensions k' with $k' > k$, i.e., nested inside the loop at level k, must be contained in $\mathcal{T}$.*

Formally:

$$(I = (i_1, \cdots, i_k, 0, \cdots, 0) \in \mathcal{T} \ \wedge \ I' = (i_1, \cdots, i_k + 1, 0, \cdots, 0) \in \mathcal{T}) \quad \Rightarrow$$

$$I'' = (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n) \in \mathcal{T}$$

for every I'' in the index set, where n is the maximal nesting depth.

Proof sketch. Multi-dimensional time can be linearized to one-dimensional time (if we allow non-affine expressions). Since, by definition, tiles enumerate successive points, $(I \in \mathcal{T}) \wedge (I' \in \mathcal{T})$ implies that all points between I and I' must belong to $\mathcal{T}$.

Note that this lemma does not enforce that the width of the tile is 1 for every time dimension. Usually, there exists one time dimension which is tiled with a width greater than 1; more details on how to compute the optimal width are given in Section 11.3. A more precise statement is given by the following corollary.

Corollary 103 (tiling time is partitioning). *There is at most one time dimension with a tile width greater than 1 and less than the maximal extent of the loop.*

Proof sketch. By contradiction: if there were two such loops, the inner loop l would contradict Lemma 102, if the width of the tile in the dimension of l were less than the maximum extent of l.

Consequently, we cannot – and need not – apply any sophisticated tiling method for the time dimensions. Once we have chosen a time dimension to be tiled with a width greater than 1, we just have to check that all inner time loops can be *collapsed* into one common tile, i.e., we check whether we can consider every inner loop as enumerating one dimension of the tile. An algorithm for tiling sequential dimensions is presented in Section 11.3.

Remark 104 (not a limitation but a fact). Note that the lemmata just presented do not *assume* rectangular tiling or simple partitioning of only one dimension – they establish it (for the time dimensions)!

11.2 Constraints

Before we present the algorithm for tiling time, we first need to elaborate on the interference of tiling space and time dimensions. We have already defined that a tiling has forward communications only (FCO), if all communication directions are contained in the cone $(0+, \cdots, 0+)$ (cf. Chapter 7).

11.2.1 The importance of FCO for tiling time

FCO enables tiling in time In the desirable and frequent case of FCO, the time dimension for tiling as well as the according tile width can be chosen freely, since the situation of Figure 11.3 cannot appear (a more formal reasoning is presented elsewhere [GFG02]). Thus, the FCO property is also the reason that we do not need a general (re-)scheduling phase after tiling time, but just a skewing of every space dimension (that carries communications) into time with a factor of $+1$, as indicated in the description of the principal procedure before. On the other hand, if we do not have FCO, two physical processors might cause a dependence cycle between two successive time steps, which forbids to coalesce these two steps.

FCO and time tiling dimensionwise Let us refine the idea of FCO by considering all dependences δ that are *not* carried by the outer time loops up to level $k-1$. We may tile one of the inner time dimensions $k, k+1, \cdots$ arbitrarily, if all communications due to the dependences δ satisfy FCO.

11.2.2 The creation of FCO

FCO by loop peeling Sometimes the FCO constraint is violated only at the bounds of a loop, but is satisfied by the inner iterations. In this case, we may peel off boundary iterations and compute a tiling for the remaining loop iterations.

Example 105. If we apply the scheduling and placement algorithm by Feautrier [Fea94] to the LU decomposition algorithm in Figure 2.2, we obtain a solution in which only four dependences cause a communication. Unfortunately, a single communication due to a dependence from statement S_2 to S_8 is directed backward, since S_2 and, hence, $U[1, j1]$ are mapped to the processor with number $j1$, and S_8 and, hence, the read access to $U[k, i2]$ are mapped to j, where $j < i2$ due to the loop bounds and $i2 = j1$ due to the existence of the dependence.

Fortunately, statement S_2 is scheduled at time 0, i.e., we can peel off the first iteration of the time loop, together with the according communications (which go in both directions), and consider the remaining target index set for time tiling, since there, we only have forward communications.

```
for i := 0 to n
  for j := 0 to n
    A[i+1, 4*j-n] := A[i, 2*j]
  endfor
endfor
```

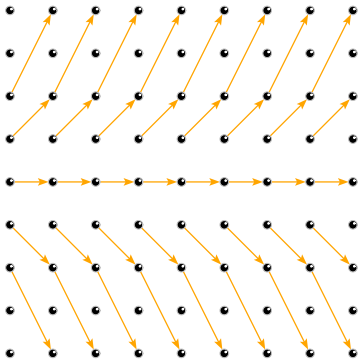


Figure 11.4: FCO is sufficient but not necessary for tiling time

FCO by construction There exists a placement algorithm that guarantees the FCO constraint, if possible (cf. Chapter 7).

11.2.3 Delimitation

FCO vs. full permutability Note that the FCO limitation is similar to full permutability in the traditional tiling framework but with three differences.

1. In our case, we require this constraint only for *space* dimensions. For time dimensions, the scheduler guarantees that all direction vectors are lexicographically strictly positive. On the one hand, the strict order is an even more restrictive constraint but, on the other hand, the lexicographic order allows negative vector entries if they are preceded by a positive entry.

2. We require the FCO constraint only if space tiling does not yield coarse enough parallelism, i.e., if we need to apply time tiling.

3. We ignore the FCO requirement, if we can assure otherwise that no communication cycle exists (cf. Example 106 below).

FCO is sufficient but not necessary Note that FCO is sufficient but not necessary for arbitrary time tiling. E.g., if we have *only* backward communications in some (or all) space dimensions, we can also tile time: we simply reverse the enumeration order of the respective processor dimension; problems can only arise if the sign of the direction vector in at least one space component changes. (Aside: if we have only backward communications and, for whatever reason, we do not want to reverse the enumeration order, thereby expressing the space mapping as an FCO placement, then we must skew those space dimensions into time by a factor of -1 – instead of $+1$ as noted before.)

Example 106 (FCO not necessary). Consider the program and dependence graph in Figure 11.4. A valid schedule is $\theta(i,j) = i$, and a suitable placement is $\pi(i,j) = j$. In this

example, even the sign of the direction vector changes, but still no dependence cycle can arise: within any chain of communications, the communication direction does not change.

Tiling time without FCO *Consequently, any time tiling is allowed in this case. This means that we can end up with a rectangular tiling of the iteration dependence graph in Figure 11.4, which partitions the processor dimension j into as many parts as we have physical processors, and the time is partitioned in a suitable width – according to the algorithm in Section 11.3 and the cost model in Section 11.4.*

FCO leads to load imbalance *In contrast, if we converted the program to a fully permutable loop nest first, or, in other words, if we required an FCO placement, we would obtain a poor load balance, as depicted in Figure 11.5.*

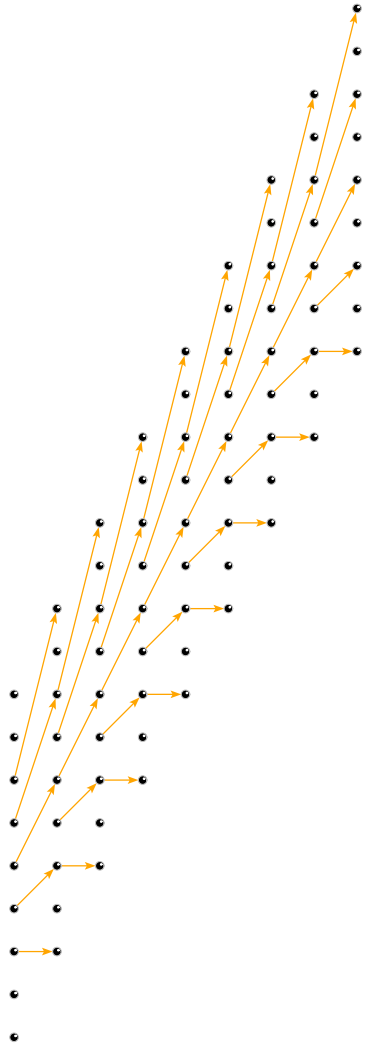


Figure 11.5: Full permutability or FCO may cause load imbalances

11.3 An algorithm to compute the optimal tile width

The above ideas lead to an algorithm that iterates through the time dimensions inside-out. For every time dimension, we check whether we coalesce iterations of this dimension, i.e., whether we select a tile width larger than 1.

Two constraints Before we do so, we have to answer two questions:

- **Correctness** Do the dependences allow to coalesce iterations?
- **Performance** Is it more efficient to coalesce operations, i.e., to increase the granularity of the target program, thus, reducing startup cost at the price of increasing the total number of time steps? And if coalescing is advantageous, how many iterations should be coalesced?

The second question can only be answered with a cost model that estimates the execution time of the target program. We discuss such a cost model in detail in Section 11.4. Our algorithm to compute the optimal tile width is as follows.

Algorithm 107 (optimal time tile width).

1. Let t be the innermost time loop.
2. Check whether the dependences allow to *collapse* dimension t, or at least to coalesce some iterations into one tile. More formally: consider only those dependences which are not carried by one of the outer loops $1, \cdots, t-1$. For these dependences, check whether the communications cause no cycle, e.g., because the direction is forward. Let $\overline{t}$ be the maximal number of time steps which may be coalesced (the full extent of dimension t in the most preferable case).
3. If $\overline{t} > 1$, compute the tile width $\underline{t}$ that has the minimal estimated execution time w.r.t. the cost model. The simplest way to find this minimum is by using mathematical derivation of the cost function. (Section 11.4.1 shows that this is not possible for our cost function and presents an alternative method to find the minimum.)
4. Set the optimal tile width for dimension t to $t_w = \min(\underline{t}, \overline{t})$.
5. If the optimal tile width for dimension t equals the full extent of dimension t, then let t be the next outer time loop (if exists) and go to Step 2.
6. The optimal tiling of the time dimensions is as follows: in one time tile, coalesce t_w many iterations of loop t and, for these indices, all iterations of all loops nested inside loop t.

Note that the cost model for Step 3 must be more precise than just the number of startups. Otherwise, even in the case of FCO, we may reach the absolute global minimum of startups (one communication per processor), at the price of losing all parallelism.

11.4 An appropriate cost model

In this thesis, we apply the standard linear cost model, in which a computation costs *cc* units of time, each communication startup costs *sc* and each transferred data element costs *vc* units of time. Note that we assume that startup and transfer do not overlap.

Thus, the cost of the execution of the parallel target program is:

$$\begin{aligned} C \;=\; & \#startups * sc + \\ & \#transferred\ data\ elements * vc + \\ & \#sequential\ computations * cc \end{aligned}$$

where # denotes the number of elements in a set.

We shall verify by some experiments in Section 11.5 that this cost model is sufficiently accurate,

- independently of the ratio of the three parts of the cost function w.r.t. each other, and
- whether the dependences are uniform or non-uniform but affine.

Input for the cost function The machine-specific parameters *sc*, *vc*, and *cc* are treated as constants. Given a source program, the difficult part is to count the number of startups, transferred data elements and computations. We solve this task in two steps:

1. we describe the sets of startups, transferred data elements and computations of a program as polytopes (Section 11.4.1), and
2. we apply the existing counting techniques in our context (Section 11.4.2).

11.4.1 Polyhedral descriptions

First, we describe the sets of startups, transferred data elements and computations of the parallelized program as polytopes.

Sequential computations

The sets of all computations are already described (or approximated) as polytopes in our model: if we remove the spatial dimensions and consider the coordinates in those dimensions as parameters instead, we obtain a description of the sequential computations for every processor.

Transferred data elements

The sets of transferred data elements are, roughly speaking, the sets of dependence instances that cross processor borders in the target code.

A polytope representing all information of a dependence Hence, for every dependence relation, we construct a polytope which contains dimensions for both the source and the destination statement of the dependence; its constraints are given by the inequalities of the statements' index sets and the equalities of the dependence relation. Every point in that polytope corresponds to one dependence instance.

Example 108. Let us consider dependence d_3 in Example 72:

$$d_3 = \quad \langle i, j; T\rangle \longrightarrow \langle i, j, k; S\rangle \qquad \text{where } 0 \leq i \leq j{-}1, j \leq n{-}1, i{+}1 \leq k \leq n{-}1.$$

Since the index sets of T and S are two- and three-dimensional, respectively, we obtain a five-dimensional polytope whose bounds are determined as follows:

- *the index set of T bounds two dimensions;*
- *the index set of S bounds the other three dimensions;*
- *the equalities hidden in the h-transformation bound the polyhedron such that two dimensions have extent 1: the source operation $\langle i', j'; T\rangle$ is $\langle i, j; T\rangle$, i.e., $i' = i$ and $j' = j$;*
- *the range $0 \leq i \leq j{-}1, j \leq n{-}1, i{+}1 \leq k \leq n{-}1$ restricts the polytope further.*

Thus, d_3 itself can be seen as the description of the polytope.

The polytope after space-time mapping Then, we apply the schedule and placement functions and obtain a *communication polytope*.

Example 109. From Example 31, we take the schedules $2 * i$ and $2 * i + 1$, and from Example 72, we take the optimal one-dimensional placements i and k for statements T and S, respectively. Thus, for Example 108, operation $\langle i, j, k; S\rangle$ is mapped to $(t, p) = (2 * i + 1, k)$, and operation $\langle i, j; T\rangle$ is mapped to $(t, p) = (2 * i, i)$, where t represents the temporal coordinate, and p represents the spatial coordinate. Hence, we can denote the communication polytope similarly to d_3 in Example 108:

$$c_3 = \quad \langle 2 * i, i; T\rangle \longrightarrow \langle 2 * i + 1, k; S\rangle \quad \text{where } 0 \leq i \leq j{-}1, j \leq n{-}1, i{+}1 \leq k \leq n{-}1.$$

Equivalently, we can denote c_3 in target indices:

$$c_3 = \quad \langle t, \tfrac{t}{2}; T\rangle \longrightarrow \langle t{+}1, p; S\rangle \quad \text{where } 0 \leq \frac{t}{2} \leq j{-}1 \leq n{-}2, \frac{t}{2}{+}1 \leq p \leq n{-}1, t\%2 = 0.$$

Note that this communication polytope is embedded in five dimensions: the time and space dimension of source and destination and, additionally, dimension j which does neither appear in space nor in time but spans one dimension of the set of destinations.

The example makes obvious that communication polytope cannot be embedded in fewer dimensions than the polytope derived in the previous paragraph. However, it can be embedded in a higher-dimensional space.

Example 110. If we take a two-dimensional instead of the one-dimensional placement, we obtain a communication polytope of the form

$$c_3' = \quad \langle t', p_1', p_2'; T\rangle \longrightarrow \langle t, p_1, p_2; S\rangle \qquad \text{where } \ldots$$

I.e., this communication polytope is embeded in a six-dimensional space.

The communication polytope after space tiling After space tiling the communication polytope, we obtain again a polytope, every point of which corresponds to the communication behavior of one operation due to one dependence relation: a communication is necessary iff the physical processor coordinates of source and destination operation differ. Since negated equalities cannot be expressed in the framework of polytopes, we use pairs of polytopes: one polytope contains all dependence instances, i.e., potential communications, and the other contains those instances that do *not* cause a communication, i.e., it has the additional constraint that the physical processor indices of source and destination operations are equal.

Example 111. Tiling the space dimensions converts every processor coordinate into two coordinates: one for the tile number, i.e., the physical processor number, and one for the local offset inside the tile. Hence, the five dimensions of the polytope in Example 109 turn into seven: the j-dimension as before, and six dimensions according to the following template:

$$c_3^s = \quad \langle t', p'_{physical}, p'_{offset}; T\rangle \longrightarrow \langle t, p_{physical}, p_{offset}; S\rangle \qquad \text{where } \ldots$$

For simplicity, let us tile (i.e., partition) the only space dimension into pieces of 64 iterations. Then, we have

$$\begin{aligned} p - p_{offset} &= 64 * p_{physical} \\ 0 &\leq p_{offset} \leq 63 \end{aligned}$$

Introducing this into the communication polytope of Example 109, we obtain:

$$c_3^s = \quad \langle t, p'_{physical}, p'_{offset}; T\rangle \longrightarrow \langle t+1, p_{physical}, p_{offset}; S\rangle$$

where

$$0 \leq \frac{t}{2} \leq j-1, j \leq n-1, \frac{t}{2}+1 \leq p \leq n-1, t\%2 = 0,$$

$$64 * p'_{physical} = \frac{t}{2} - p'_{offset}, 0 \leq p'_{offset} \leq 63,$$

$$64 * p_{physical} = p - p_{offset}, 0 \leq p_{offset} \leq 63.$$

For arbitrary tilings, the bounds can be computed similarly with standard methods [AI91]. This polytope represents the set of all potential communications.

For the polytope representing the non-communications, we add the constraint that the coordinates of the physical processors must be equal in both dimensions for sender and receiver, i.e., $p_{physical} = p'_{physical}$.

The communication polytope after time tiling The motivation for introducing tiling time has been to increase the granularity of communications by adding dimensions that split logical time into a global component and a local offset. Since the actual communications only take place at the borders of global time steps, each send or receive subsumes all other source or destination operations of the dependence that have the same global time but any of the possible local time offsets. The formalization is like in the case of tiled space dimensions, just without the special treatment for avoiding self-communications.

Remark 112 (repeated sends due to imprecise modeling). Note that, if we generate communications as just described, we might send the same data element multiple times: e.g., the use of a value just computed at different statements in the program is represented by different dependences, and every dependence is treated separately.

A proper solution would be to use the union of the set of data to be transferred, instead of the operations in dependence. However, the union itself is not trivial to handle since a union causes non-convex sets, and, furthermore, using the set of data causes an additional restriction: the access matrix that describes the array indices must be unimodular, since we can only count the number of integer points, i.e., stride 1, inside a polytope.

Since this representation may be quite complex, the execution time of the target program is increased, and in some cases this additional cost is higher than the communication cost saved due to the reduced communication volume, which is the benefit of the precise description. A more precise analysis would be necessary to evaluate this trade-off in practice.

Startups

A startup occurs once for every pair of physical processors that exchange data, and between any two aggregated time steps. Hence, we may start again with the communication polytope and project it to the physical space and aggregated time dimensions, and then count the number of points inside. Note that the number of startups depends on the time tile width, the variable we are looking for. Thus, the time tile width must be expressed as a parameter and, consequently, we obtain a parameterized cost value, i.e., a cost function. In order to find the minimum of this cost function, we try to use derivation.

Fundamental problem However, there is a mathematical problem: tiling with parametric widths – in space and/or time – leads to non-affine terms in the descriptions of

the tiled index sets and, thus, of the communication polytope. As a consequence, the method for counting integer points using Ehrhart polynomials is not directly applicable, i.e., we have no closed cost function that we could derive; we can just compute the cost for individual values of the tile width.

Solution Fortunately, we can avoid parametric time tile widths. For this purpose, we look at the three terms in our cost function in more detail. We know that, if all other parameters are constant,

- the number of startups drops approximately by a factor of $1/B$ for time tile width B, whereas
- the number of sequential time steps additionally executed increases with increasing B. The precise number of these steps additionally executed equals $B * (\#P - 1)$, where $\#P$ is the number of processors in the longest communication chain, which is, in the case of FCO, at most the sum of the lengths of the dimensions of the multi-dimensional physical processor grid;
- the amount of transferred data is independent of B.

Due to this simple setting of one increasing and one decreasing term in the sum, the cost function has one minimum for varying values of the time tile width B, i.e., it is *unimodal*.

Consequences for our cost model For a unimodal function f, there exists an iterative method to find its minimum: we select two positions x_1 and x_2 in the domain of f and compute $f(x_1)$ and $f(x_2)$. If $f(x_1) \leq f(x_2)$, the minimum is located at some x_o with $x_o \leq x_2$, whereas if $f(x_1) > f(x_2)$, the minimum is located at some x_o with $x_o > x_1$. Hence, we continue our search for the minimum in the accordingly restricted domain.

One possible choice for selecting x_1, x_2 is to split the domain into pieces of $1 - \frac{\sqrt{5}-1}{2}$ and $\frac{\sqrt{5}-1}{2}$ of the original domain. In this case, only a single new function value needs to be computed at every iteration step – the second function value needed can be reused from the previous iteration [Kie53].

We can apply this method for computing the width causing the minimal cost in Step 3.

11.4.2 Using Ehrhart polynomials in our context

At this point, we can express all necessary sets as polytopes and compute the cardinality and, thus, the cost of a program with any desired tiling.

Example 113. Let us illustrate the performance of the computation of an Ehrhart polynomial counting the number of communication startups generated by one of the dependences of our LU decomposition algorithm in Figure 2.2. We consider the simple case of no time tiling, i.e., the time tile width equals 1. We do not need any knowledge about that arbitrarily chosen dependence, except for the size of the polytope describing the startups: it is

represented by a system of 18 inequalities with 6 variables (the time and the 2-dimensional physical processor coordinate both for sender and receiver) and 2 parameters (the matrix size and the constant 1).

If we use a square tiling of the space dimensions with a fixed width of 2, the Ehrhart polynomial is computed in about 3 seconds on a 1.4 GHz PC (Athlon XP 1600+) with 512 MByte RAM. It is interesting to see that, already in this case, the implementation in the Polylib switches to the (slow) arbitrary precision mode due to the internal growth of the coefficient values.

If we use a square tiling of the space dimensions of width 8 instead, the computation of the Ehrhart polynomial takes already about 3.5 hours.

The dominating cost factor in this experiment is the second phase, in which the coefficients of the Ehrhart polynomials are computed by counting the number of integer points in several non-parametric polytopes (cf. Section 3.3.4) – the parameter domains for the different shapes and the respective parameterized vertices are computed in far less than a second.

Approximations In order to save execution time, we could try to approximate the number of points inside the polytope. The main idea is that the terms with the highest degree approximate the real volume. These terms have no periodic coefficients and their dimensionality and degree are given by the number of parameters and the dimensionality of the polytope, respectively. There are (at least) two approaches to compute these terms with the highest degree:

1. Similarly to the original method of computing the coefficients of the Ehrhart polynomials (cf. Section 3.3.4), we can try several parameter settings and count the number of integral points in the non-parameterized polytopes. Since the term with the highest degree is only an approximation, we obtain no strict equalities from which we could compute the coefficients. We have to look for appropriate statistic or numeric methods instead.

2. An alternative solution is to compute, as before, the parameter domains and the respective (parameterized) vertices of the polytope. Now, we can decompose the polytope into its simplices and compute the real volume of the simplices using the determinant of the spanning vectors.

The execution time of the first approach increases with larger parameter values, whereas the second approach becomes more expensive as the number of vertices (simplices) grows. For a practical comparison, both methods should be implemented.

Independently of the concrete method, we must check how such an approximation of the volume influences the choice of the tile width and, thus, the execution time of the parallel program. For that purpose, we present in Section 11.5 some experiments which explore the relationship between different time tile widths and the execution time.

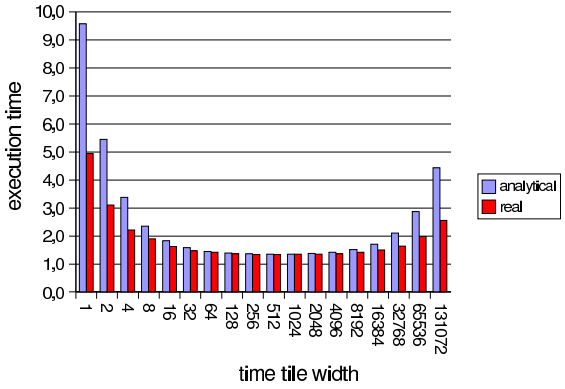


Figure 11.6: Analytical and experimental results for uniform dependences (Example 115)

11.5 Experimental validation

We ran our experiments on the SCI-connected network of PCs, described in Example 99. For this configuration, we found as a rough approximation $sc = 100$ and $vc = cc = 1$ (we only need relative costs). Fortunately, it turns out that these parameters need not be computed very precisely; even if the real value of a parameter in the model differs by a factor of almost an order of magnitude in this example, the analytical result is sufficiently precise. The main reason for this behavior is that the optimal execution time usually has a very flat minimum, i.e., around the minimum, the execution time stays nearly constant for different values of the time tile width (see Figure 11.6). Note that the analytical execution time in Figure 11.6 is computed in logical units of time, which we scaled so as to fit into the histogram with the real execution times (cf. Example 115). Furthermore, this means that the number of computations and communications may be approximated in a similarly coarse manner.

Remark 114 (a more dynamic approach). If a program is difficult to analyze, so that the approximation of the number of computations and communications remains seriously imprecise, we suggest to execute the first iterations of the parallel target program with the time tile width computed at compile time. During this execution, measure the real amount of work for computations and communications, and then adapt the time tile width according to the new run-time values. However, in our practical examples, this has never been necessary.

Examples Let us now apply time tiling to some examples which have different ratios of the most important cost parameters and, thus, evaluate the stability of our cost model:

- Example 115 has a trade-off between the number of communication startups vs. the computation time;

- Example 116 with two-dimensional placement has a trade-off between the volume-dependent communication time vs. the computation time;
- in Example 116 with one-dimensional placement, the computation cost dominates both.

Example 115 (Example 99 revisited). We consider first a rectangular index set with uniform distance vectors $(1,0)$ *and* $(0,1)$*, as already used in Example 99. The communication graph after space-time mapping is given in Figure 8.11 (without the dashed lines). Let us apply our algorithm for tiling time:*

1. *This example has only one dimension in time, so we set* t *to this dimension.*
2. *Since the program satisfies the FCO property, collapsing* t *is permitted.*
3. *Therefore, we can now compute the optimal tile width. Let* m *be the number of virtual processors, and* n *the number of logical time steps per processor. The program has* $n * m$ *computations, and every physical processor sends at every time block one package with* w *pieces of data to its upper neighbor, where* w *is the time tile width (cf. Figures 8.11 and 8.13).* NP *denotes the number of physical processors. The terms in our cost function are composed as follows:*

computation amount per block	$\frac{n*m}{NP*(n/w)}$
number of blocks on the critical path	$n/w + (NP-1)$
computation cost	$\frac{n*m}{NP*(n/w)} * (n/w + (NP-1)) * cc$
startup cost	$(n/w + (NP-1)) * sc$
volume-dependent transfer cost	$n * vc$

Hence, the estimated execution time is

$$\begin{aligned} C(w) &= \frac{n*m}{NP*(n/w)} * (n/w + (NP-1)) * cc + \\ & \quad (n/w + (NP-1)) * sc + n * vc \end{aligned}$$

We eliminate the double fractions and obtain

$$\begin{aligned} C(w) &= \frac{n*m*cc}{NP} + \frac{(NP-1)*m*w*cc}{NP} + \\ & \quad sc * n/w + (NP-1) * sc + n * vc. \end{aligned}$$

In order to find the minimum, we derive the cost function:

$$\frac{dC}{dw} = \frac{(NP-1)*m*cc}{NP} + sc * n * \frac{-1}{w^2}$$

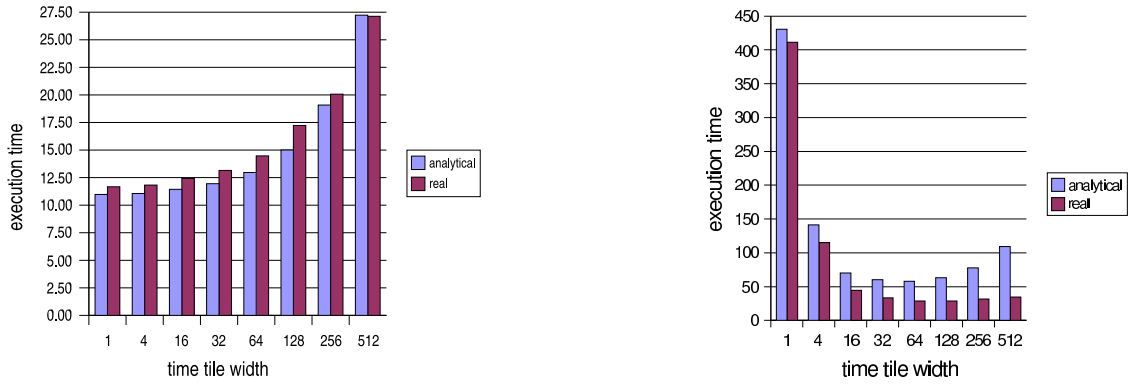


Figure 11.7: Analytical and experimental results for one-dimensional (left) and two-dimensional (right) placement for LU decomposition on 4 processors

For the minimum, $\frac{dC}{dw}$ must be 0:

$$\frac{dC}{dw} = 0 \quad \Leftrightarrow \quad \frac{(NP-1) * m * cc}{NP} = \frac{sc * n}{w^2}$$
$$\Leftrightarrow \quad w^2 = \frac{sc * n * NP}{(NP-1) * m * cc}$$

Thus, for sufficiently large numbers of processors, $\frac{NP}{NP-1} \approx 1$, and we obtain the minimum at

$$w_0 \approx \sqrt{\frac{n * sc}{m * cc}}.$$

For our experiment ($n = 3 * 2^{17}, m = 2^7$), this minimum is at $w_0 \approx 554$.

The scaled theoretical and the real execution times are presented in Figure 11.6.

In this example, the startup and computation costs are of the same order of magnitude (whereas the volume-dependent communication cost is much smaller), so we can observe the trade-off between reducing the number of startups vs. reducing the number of computations on the critical path.

Note that the optimal tile width w_0 is between 0 and 1, for the case of $m * cc \geq n * sc$. This analytical result means that tiling time is useless, i.e., we must use a time tile width of 1. This matches our intuition: in this situation, we have sufficiently coarse-grained parallelism by just mapping the (large number of) m virtual processors to the physical processors.

Example 116. We also apply time tiling to our LU decomposition algorithm.

Two-dimensional placement *We derive first a space-time mapping defining a two-dimensional placement. This placement guarantees forward communications (except for the first*

two iterations). Thus, the only time dimension can be collapsed. Our cost model shows that the volume cost dominates the startup cost by orders of magnitude: by code inspection, we were able to determine that

- *the average number of startups per time block is about $\mathcal{NP}/2$, and*
- *the average transferred volume per broadcast is about $1.5 * n^2/\sqrt{\mathcal{NP}}$.*

For simplicity and efficiency reasons, we decided to prefer broadcasts to point-to-point communications in this example. One consequence is that, in contrast to Example 115, the total volume cost decreases for larger time tiles, since our target program distributes a constant number of data elements per broadcast, and the number of broadcasts decreases for increasing tile width. The formula for the computation cost is similar to the one in Example 115, except that we have n^3 instead of $n * m$ computations in the body. Using a matrix width of $n = 1024$, and a number of processors of $\mathcal{NP} = 4$ or 16, the volume cost is of the same order of magnitude as the computation cost. Hence, we again have a minimum for the execution time.

By evaluating the cost function at several time tile widths (a practical alternative to derivation, cf. last paragraph of Section 11.4.1), we obtain a minimum of about 64 in both cases, which happens to be the correct global optimum for $\mathcal{NP} = 4$ (see Figure 11.7, right). On 16 processors, the measured minimal execution time is 22.9 seconds for an optimal tile width of 128 (in contrast to 24.5 seconds for a tile width of 64). The sequential execution time is 29.1 seconds.

One-dimensional placement *For a one-dimensional placement of LU decomposition, our model predicts that the amount of computation (cubic as in the two-dimensional case, of course) dominates the amount of communication by one to two orders of magnitude: with the help of Ehrhart polynomials, we count that about $\mathcal{NP} * n^2/3$ data elements must be transferred, and the number of startups without tiling time is approximately $\mathcal{NP} * n$. Hence, the optimal theoretical and actual time tile width is 1, i.e., no time tiling should be done – the computations in the loop body take enough CPU time (in our case, the body contained an additional loop for summing up the values of SUM and SUMM, respectively).*

The measured execution times for 4 and 16 processors and $n = 1024$ are as follows:

	$\mathcal{NP} = 4$	$\mathcal{NP} = 16$
execution time for time tile width 1:	*11.7 sec.*	*5.9 sec.*
optimal time tile width:	*1*	*2*
execution time for optimal time tile width:	*11.7 sec.*	*5.5 sec.*

Comparison *Note that, for LU decomposition, a two-dimensional instead of a one-dimensional placement leads to longer execution times, whether we use the optimal time tiling, or we do not use time tiling at all (compare Figure 11.7, left and right, and note the different scale). Without tiling time, the execution time differs by a factor of about 30, which is reduced to a factor of about 2 with optimal time tiling. The reason for the bad performance of the two-dimensional case is that a set of operations with many dependences*

between them must be distributed across the processors, just in order to span a second spatial dimension.

Concerning time tiling, we have seen that,

- *on the one hand, the execution time is in the same order of magnitude when we apply time tiling, whether we start from the preferable one-dimensional placement, or from the unfavorable two-dimensional placement;*
- *on the other hand, the result is still not the same, i.e., tiling cannot fully compensate for the wrong choice of the placement's dimensionality.*

The fact that a one-dimensional placement without time tiling performs better than a two-dimensional placement with time tiling will be explored in more detail in Chapter 12.

Conclusion from the experiments The examples show that our cost model is sufficiently accurate for uniform dependences (Example 115) as well as for affine dependences (Example 116), and that it is stable regardless of the ratio of the most important cost parameters.

Chapter 12

Interplay: low-dimensional placement or tiling

The LU example in the previous chapter shows that a reduced number of parallel dimensions makes tiling useless. In that example, we have potentially two parallel dimensions, and we obtain the shortest execution time when we lay out only one of them in space and execute the other sequentially. The effect is, that

- the one dimension in space generates enough parallelism and,
- the second, sequentially executed dimension leads to a sufficiently coarse-grained parallel program.

This gives rise to a new question: is this a general property, i.e., is tiling necessary for programs with at least two possibly parallel loops? This chapter answers this question in more detail.

12.1 Low-dimensional placements as an alternative to tiling

Most existing placement procedures have an input parameter specifying how many dimensions of the index set are to be laid out in space [Fea94, DR95]. An obvious value for this parameter is the dimensionality of the index set reduced by the dimensionality of the schedule.

On the one hand, this choice leads to the maximal amount of exploitable parallelism but, on the other hand, it causes too many small communications (with distributed memory) or synchronizations (with shared memory) in order to be usable in practice. Therefore, we suggested to apply tiling techniques in order to increase the parallelism's granularity.

Tiling cannot compensate for high-dimensional placements However, as just seen, tiling often cannot find as efficient coarse-grained parallelism as we obtain if we

just ask the placement computation algorithm for fewer dimensions of parallelism (even despite the fact that the bandwidth of a network typically decreases with reduced dimensionality). The reason is that the placement spreads the statements individually in space, whereas tiling only coalesces iterations without being able to treat the different statements of one loop step individually.

```
      for i := 0 to n
        for j := i to n
S1:       A[i,j] := A[i-1,j]
        endfor
        for k := i to n
S2:       B[k,i] := B[k-1,i] + A[k,i]
        endfor
      endfor
```

Figure 12.1: Source program for two-dim. tiling vs. one-dim. placement

Example 117. For a clearer presentation, we extract the core of the LU algorithm and simplify it until we get the program in Figure 12.1. Note that we assume a third surrounding, mandatorily sequential loop that carries all dependences, which is already projected away.

The program has two loop-carried uniform dependences and one loop-independent dependence which only exists on the diagonal (see Figure 12.2):

$$\begin{aligned} d_1 &= \langle i-1, j; S_1\rangle \longrightarrow \langle i, j; S_1\rangle \\ d_2 &= \langle i, k-1; S_2\rangle \longrightarrow \langle i, k; S_2\rangle \\ d_3 &= \langle i, i; S_1\rangle \longrightarrow \langle i, i; S_2\rangle \end{aligned}$$

d_1 and d_2 result in a square, d_3 only in a linear number of communicated data. Hence, d_1 and d_2 should be cut first, i.e., source and destination of the dependence should be put on the same processor. This results in an – at most – one-dimensional placement: j and i for S_1 and S_2, respectively. This also cuts d_3, and we obtain a communication-free parallel program.

On the other hand, we cannot cut d_1 nor d_2 if we enforce a two-dimensional placement. We can only eliminate d_3 and obtain (i, j) and (i, k) as simplest two-dimensional placements for statements S_1 and S_2, respectively.

Since this placement is the identity, we can tile the program before or after the placement phase and always obtain rectangular tiles as the communication-optimal solution. There is no chance to eliminate all communications – by whatever tiling (see the right part of Figure 12.2, in which any hyperplane bounding a tile intersects with a dependence arrow).

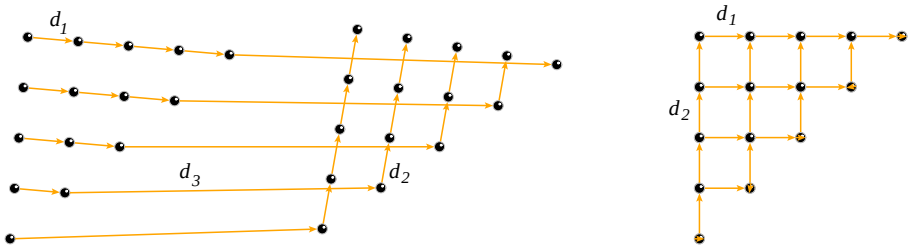


Figure 12.2: Operation and iteration dependence graph for Figure 12.1

12.2 Hoping for an algorithm

The idea

In order to avoid the effect just described, we could try to start with a one-dimensional placement and check whether the parallelism generated is sufficient for the given target architecture. If so, we use this placement, otherwise we compute a two-dimensional placement, check again, and so on. This way, we *reduce the granularity* step by step, until we obtain the minimal placement dimensionality which we need in order to exploit all possible parallelism of the given architecture.

For this placement, we then compute the optimal space tiling which adapts the number of virtual and physical processors and, at the same time, reduces the number of communication partners, as described in Chapters 9 and 10. If necessary, we can finally *increase the granularity* by tiling time, as described in Chapter 11.

Advantages

Minimal communication The idea behind this approach is that it takes the minimal number of space dimensions (that gives us enough parallelism for the parallel architecture). This is optimal since every additional desired space dimension introduces additional constraints for the placement algorithm which avoid that some dimensions collapse. Hence, every additional space dimension typically causes many more – certainly never fewer – communications.

Sufficient parallelism Beyond that, typically a single dimension in space induces more parallelism than needed since, for many of today's parallel computers, the total number of processors is less than the number of iterations of one parallel loop.

Easy adaptation At run-time, we often do not have exclusive access to a parallel computer, but we only use a partition of it. The size of this partition may change for different runs. Thus, in principle, we could have to adapt our parallel program to these changes. Fortunately, in the described setting, this is not necessary: since the parallelism generated by one dimension in space is sufficient for the whole parallel computer, it is

	number of startups	comm. vol./send	total communication cost
π_1^1	$\mathcal{NP}-1$	n	$(\mathcal{NP}-1)*(sc+n*vc)$
π_2^1	$\mathcal{NP}-1$	n	$(\mathcal{NP}-1)*(sc+n*vc)$
π_2^2	$2*(\sqrt{\mathcal{NP}}-1)*\sqrt{\mathcal{NP}}$	$\frac{n}{\sqrt{\mathcal{NP}}}$	$2*(\sqrt{\mathcal{NP}}-1)*\sqrt{\mathcal{NP}}*(sc+\frac{n}{\sqrt{\mathcal{NP}}}*vc)$

Table 12.1: Communication cost for different placements

sufficient for any partition. So, independently of the size of the partition of the parallel architecture we use at run time, we need only a single version of the parallel program, which has one dimension in space; the width of the partitions of this spatial dimension is a parameter whose value is set at run time in order to adapt the program to the number of physical processors.

Extended applicability There is also a technical benefit. Tiling time dimensions is not always legal. A *sufficient* condition for time tiling is that all communications satisfy the FCO property. In contrast, directly computing a low-dimensional placement is always possible.

12.3 Low-dimensional placements are not always an alternative

Unfortunately, the situation is more complex than just described.

Example 118. Consider the program in Figure 12.3 which has three uniform dependences. A valid schedule is $\theta(i,j,k)=i$, a possible one-dimensional placement is $\pi_1(i,j,k)=j$, and a possible two-dimensional placement is $\pi_2(i,j,k)=(j,k)$.

After space tiling we obtain

- $\pi_1^1(i,j,k)=\frac{j}{\mathcal{NP}}$ *for the one-dimensional placement,*
- $\pi_2^1(i,j,k)=(\frac{j}{\mathcal{NP}},k)$ *for the two-dimensional placement with one-dimensional space tiles, and*
- $\pi_2^2(i,j,k)=(\frac{j}{\sqrt{\mathcal{NP}}},\frac{k}{\sqrt{\mathcal{NP}}})$ *for the two-dimensional placement with two-dimensional space tiles.*

For simplicity, we assume that the fractions have no remainder. Furthermore, we assume that the elements of array A are stored where they are computed (i.e., computer owns), and that time tiling is not necessary. Let us now compute the cost of a single time step for these placements; the results are summarized in Table 12.1.

π_1^1 *The number of startups is $\mathcal{NP}-1$ (every processor sends to its right neighbor), and the communication volume per send is n (all elements in the k dimension). Hence, the*

total communication cost is

$$C_1 = (\mathcal{NP}-1) * (sc + n * vc).$$

π_2^1 *The number of startups and the amount of transferred data is equivalent with the case where we start with a one-dimensional placement; the logical difference is that k is first interpreted as a space dimension which causes communications at the virtual processor level, but this difference is made obsolete by the one-dimensional space-tiling that maps all k virtual processors into the same tile. Hence, the cost for this case is C_1.*

π_2^2 *The number of startups is $(\sqrt{\mathcal{NP}}-1) * \sqrt{\mathcal{NP}}$, both for the horizontal and the vertical space dimension. The amount of transferred data per send is $\frac{n}{\sqrt{\mathcal{NP}}}$. Hence, the total communication cost is*

$$C_2 = 2 * (\sqrt{\mathcal{NP}}-1) * \sqrt{\mathcal{NP}} * (sc + \frac{n}{\sqrt{\mathcal{NP}}} * vc).$$

Now, let us check when the one-dimensional layout is better:

$$\begin{aligned}
& C_1 < C_2 \\
\Leftrightarrow \quad & (\mathcal{NP}-1) * (sc + n * vc) \ < \ 2 * (\sqrt{\mathcal{NP}}-1) * \sqrt{\mathcal{NP}} * (sc + \frac{n}{\sqrt{\mathcal{NP}}} * vc) \\
\Leftrightarrow \quad & (\mathcal{NP}-1) * sc + (\mathcal{NP}-1) * n * vc \ < \\
& 2 * \mathcal{NP} * sc - 2 * \sqrt{\mathcal{NP}} * sc + 2 * (\sqrt{\mathcal{NP}}-1) * \sqrt{\mathcal{NP}} * \frac{n}{\sqrt{\mathcal{NP}}} * vc \\
\Leftrightarrow \quad & (\mathcal{NP}-1) * n * vc \ < \\
& (\mathcal{NP}+1) * sc - 2 * \sqrt{\mathcal{NP}} * sc + 2 * (\sqrt{\mathcal{NP}}-1) * n * vc \\
\Leftrightarrow \quad & (\sqrt{\mathcal{NP}}-1) * (\sqrt{\mathcal{NP}}+1) * n * vc \ < \\
& (\mathcal{NP} - 2 * \sqrt{\mathcal{NP}} + 1) * sc + 2 * (\sqrt{\mathcal{NP}}-1) * n * vc \\
\Leftrightarrow \quad & (\sqrt{\mathcal{NP}}-1) * n * vc * ((\sqrt{\mathcal{NP}}+1) - 2) \ < \ (\sqrt{\mathcal{NP}}-1)^2 * sc \\
\Leftrightarrow \quad & (\sqrt{\mathcal{NP}}-1) * n * vc * (\sqrt{\mathcal{NP}}-1) \ < \ (\sqrt{\mathcal{NP}}-1)^2 * sc \\
\overset{\mathcal{NP} > 1}{\Leftrightarrow} \quad & n * vc \ < \ sc \\
\Leftrightarrow \quad & n \ < \ \frac{sc}{vc}
\end{aligned}$$

This means that, even in this simple case, the optimal placement dimensionality depends on the problem size and the machine-specific ratio between communication startup cost and volume dependent communication cost.

```
for i := 1 to n
  for j := 1 to n
    for k := 1 to n
      A[i, j, k] := A[i−1, j, k] + A[i−1, j, k−1] + A[i−1, j−1, k]
    endfor
  endfor
endfor
```

Figure 12.3: Uniform dependence grid

Our solution: several placements The consequence of this observation is that we must compute placement functions with different dimensionalities, and then choose the cost minimal version.

Fortunately, the computation of a placement is one of the least computation-intensive parts of the suggested parallelization method, i.e., the execution time of the parallelizer does not increase too much by computing two or three placements with different dimensionalities.

Main limitation of tiling in general The big drawback for practical application of tiling is the limitation to linear problems already mentioned, which is a harsh restriction in this context since the number of physical processors and the problem size are structure parameters which appear frequently as coefficients for the variables in the description of the tiled index set. Our ongoing research is devoted to relaxing this constraint (cf. Remark 8) [Grö03, GGL04].

Chapter 13

Towards target code: tuning the abstract results

In this last technical section, we point out some aspects which must be considered in order to convert the target model into efficient target code. In Sections 13.1 and 13.2, we suggest two kinds of postprocessing of the space-time mapping; in Section 13.3, we deal with the central task of code generation, namely the scanning of sets of polyhedra. Section 13.4 introduces the communication code, which is necessary since we aim at distributed memory architectures, and 13.5 briefly compares candidates for the target programming language.

13.1 Consistency of schedules and placements

First, we face the problem that our space-time matrix might not be of full rank. This case appears in practice, since in most implementations of the polytope model the schedule and the placement are computed independently.

Nevertheless, this issue of relaxing the invertibility requirement has not been addressed in the literature until recently, probably because the invertibility seems to be indispensable for deriving the source indices from the target indices in the loop bodies as well as for computing the target polytope (cf. Equation (3.21)). Among the various techniques for scanning polyhedra in Section 13.3, only Cloog [Bas02, Bas03] is able to handle directly space-time matrices that are not invertible. In addition, the way how Cloog deals with the problem is hard-coded and does not take into account the types of the dimensions (e.g., spatial or temporal) – which is natural since Cloog is a generic tool for generating loop nests from sets of polyhedra.

In contrast, let us now look at the initially suggested solution to the problem of singular space-time matrices [GLW98], and explain its interpretation as a modification of the space-time mapping. This will also show us that the solution is not unique in general.

Example 119. Let us return to our motivating example in Figure 2.1. We take the schedule from Example 31, which is

$$\theta_T(i,j,n) = 2{*}i, \quad \textit{and} \quad \theta_S(i,j,k,n) = 2{*}i{+}1.$$

In contrast to that example, we now use a one-dimensional placement. Example 72 shows that the best one-dimensional FCO placement, based on the dependence-driven approach, is

$$\pi_T(i,j,n) = i, \quad \text{and} \quad \pi_S(i,j,k,n) = k.$$

Thus, obviously, the schedule and placement for statement T are linearly dependent. The space-time matrices are:

$$\mathcal{T}_T = \left(\begin{array}{cc|cc} 2 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right), \text{ and } \mathcal{T}_S = \left(\begin{array}{ccc|cc} 2 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right). \tag{13.1}$$

Because of the singularity (more precisely: the non-injectivity) of $\mathcal{T}_T$, some processors will have to perform several operations at the same time step.

Let us now see how to deal with this situation.

13.1.1 Arbitrary space-time matrices

Our initial solution consists of the following steps:

Algorithm 120 (generating an invertible space-time matrix).

1. Eliminate linearly dependent rows in the space-time matrix $\mathcal{T}_S$ of statement S.
2. Extend the matrix to a square invertible matrix, i.e., to a basis.
3. Generate code w.r.t. the resulting, modified matrix.
4. Insert code for the rows eliminated in Step 1.

Let us discuss these steps in more detail in the following paragraphs. For simplicity, we assume that the rows of the space-time matrix are ordered from top to bottom according to the desired outside-in order of the target loops. (Note that this is just a convention, not a restriction.)

Eliminating linearly dependent rows (Step 1)

In a first step, we eliminate, iteratively from top to bottom, all linearly dependent rows and store them together with their row number for further processing later on (Step 4). Technically, we use Echelon reduction [Ban93] to determine whether row μ is linearly dependent on rows $1, \cdots, \mu-1$. The result of this elimination process is a matrix $\mathcal{T}'_S$ which has full row rank but is not necessarily square: the number of rows is less than or equal to the number of columns.

Extending to a square matrix with full rank (Step 2)

Matrix $\mathcal{T}'_S$ can have fewer rows, i.e., target dimensions, than columns, i.e., source dimensions. To obtain an invertible square matrix, we extend $\mathcal{T}'_S$ to a basis by simply adding linearly independent unit vectors at the bottom of $\mathcal{T}'_S$, each of which spawns one missing dimension. The result is an invertible square *transformation matrix* $\widetilde{\mathcal{T}_S}$.

Remark 121 (homogeneous extension is irrelevant). Note that, for the task of obtaining a basis (Steps 1 and 2), we are only interested in the non-homogeneous part of the space-time matrix. A row or column that belongs to the homogeneous extension, i.e., that represents a symbolic parameter (including the constant 1), does not really span a dimension – the index in that dimension is constant.

Generating code (Step 3)

The transformation matrix $\widetilde{\mathcal{T}_S}$ can be used to derive the target program parts from the target polytopes by standard methods (including the extensions for non-unimodular transformations), as described in Sections 3.3.8 and 13.3.

Interpretation Before continuing with our description of the code generation method, let us briefly reflect on the changes from $\mathcal{T}'_S$ to $\widetilde{\mathcal{T}_S}$. How can we interpret the artificial rows which we have added and which correspond to target dimensions but are given neither by the schedule nor by the placement?

Since these dimensions are not laid out in time, they obviously will not carry a dependence (otherwise, the schedule would be incorrect). Therefore, we could lay them out in space.

On the other hand, the placement algorithm did not distribute iterations along these dimensions. Therefore, in order to preserve the effect of the placement, we decide to lay out these artificial dimensions in time and put the respective loops inside the loops enumerating the schedule (indeed we make them the innermost loops as described above). This means that we refine the time given by the scheduler with additional dimensions. This changes neither the global schedule (which is respected by the outermost, i.e., dominant loops on time), nor the placement (since it is not modified).

The remaining question is: what happens with the rows which have been eliminated? The next section will rectify the fact that we have ignored these dimensions.

Reinserting eliminated dimensions (Step 4)

First, note that a target dimension given by row μ, which is linearly dependent on rows $1, \cdots, \mu-1$ of $\mathcal{T}_S$, collapses to a singleton, the value r_μ of which can be computed from the coordinates $r_1, \cdots, r_{\mu-1}$ in dimensions $1, \cdots, \mu-1$. This leads directly to our solution: we compute r_μ from $r_1, \cdots, r_{\mu-1}$ and insert a loop enumerating only the singleton r_μ. The nesting level at which this loop is inserted is given by the previously stored number of the eliminated row (Step 1). The type of the loop is sequential if the eliminated row

was produced by the scheduler, and parallel if it is a part of the placement. Of course, a "parallel" loop with only one iteration does not really specify parallelism, but we keep this labelling convention since the index of a parallel loop (even for singletons) carries the information on which processor the body operation(s) shall be executed. (Aside: if the target language provides us with an explicit possibility to denote this processor, we use it instead of the degenerated loop.)

Note that this reinserting scheme ensures that the outermost target loops enumerate precisely all coordinates according to the given space-time mapping, i.e., the loop nest implements $\mathcal{T}_S$ correctly, whereas the innermost, artificially added target loops spawn the missing dimensions, which is necessary to be able to enumerate all transformed source index coordinates.

Application to an example

Example 122. Continuing Example 119, the elimination step (Step 1) reduces $\mathcal{T}_T$ to

$$\mathcal{T}'_T = \left(\begin{array}{cc|cc} 2 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right),$$

since the placement in the second row is linearly dependent with the first row: the placement index p equals $t/2$, where t is the schedule index. (Aside: since statement T is only executed at even time steps, p is always integer.)

The extension step (Step 2) then generates the invertible transformation matrix

$$\widetilde{\mathcal{T}}_T = \left(\begin{array}{cc|cc} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right).$$

This invertible matrix is used to generate target code (Step 3). Following the method described in Section 3.3.8, the target loop structure for statement T with transformation matrix $\widetilde{\mathcal{T}}_T$ is:

T' :
```
for t := 0 to 2 * (n-1) step 2
  for x := t/2 + 1 to n-1
    tmp[t/2, x] := a[x, t/2]/a[t/2, t/2]
  endfor
endfor
```

where t is the index of the schedule with $t = 2 * i$, and x is the loop index of the dimension added in Step 2 with $x = j$.

Finally, the eliminated processor dimension must be reinserted as second loop (Step 4). Since $p = t/2$, we obtain:

T' :

```
for t := 0 to 2 * (n-1) step 2
  parfor p := t/2 to t/2
    for x := t/2 + 1 to n-1
      tmp[p, x] := a[x, p]/a[t/2, p]
    endfor
  endfor
endfor
```

As already mentioned, the processor loop does not span a dimension, but it specifies on which processor the body computations take place.

Remark 123 (compatibility by construction). Note that these steps can be avoided if one guarantees that placement and schedule are computed in sequence, and the latter of the two phases, whichever it is, ensures the return of a solution that is compatible, i.e., linearly independent, with the result of the first phase. This requires to adapt the methods for the latter of the two phases, but it allows to put more emphasis on the first phase: since the results are compatible, none of the rows need to be eliminated, i.e., the result of the first phase (which is computed without any constraints) enters the transformation matrix without any change, whereas the result of the latter phase (considered less important) must satisfy the constraint of linear independence. One approach in this direction has been suggested by Darte, Diderich, Gengler, and Vivien [DDGV00]. They consider the placement more important and, thus, present an adapted scheduling method.

13.1.2 Insufficient dimensionality of the space-time matrix

A problem, similar to the incompatibility scenario, occurs if the sum of the dimensionalities of schedule and placement of a statement is smaller than the nesting depth of that statement. We apply again Algorithm 120 and, thus, obtain an invertible transformation matrix.

Example 124. Consider again Example 119, but now with a focus on statement S. The rows of the space-time matrix $\mathcal{T}_S$ are linearly independent, but the number of rows is one less than the number of columns.

Following Algorithm 120, Step 1 yields

$$\mathcal{T}'_S = \mathcal{T}_S = \left(\begin{array}{ccc|cc} 2 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right).$$

Step 2 inserts an additional third row in order to span the dimension of j:

$$\widetilde{\mathcal{T}}_S = \left(\begin{array}{ccc|cc} 2 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

We have the following equalities between source and target variables: $p = k$, $y = j$, and $t = 2 * i + 1$, i.e., $i = (t-1)/2$.

Step 3 generates the following code:

S' :
```
for t := 1 to 2 * (n-1) + 1 step 2
  parfor p := (t-1)/2 + 1 to n
    for y := (t-1)/2 + 1 to n-1
      a[y, p] := a[y, p] - tmp[(t-1)/2, y] * a[(t-1)/2, p]
    endfor
  endfor
endfor
```

Remark 125 (more intelligent extension to a basis). Note that the extension to a basis in Step 2 is not restricted to unit vectors – this is just the simplest valid extension. In principle, any vector that is linearly independent with the already existing rows of the matrix is valid. Thus, future work might elaborate on whether some other vector improves the performance of the resulting program, e.g., by increasing cache efficiency.

13.2 Multi-dimensional schedule

In the (typical) case of statementwise schedules, the target code often contains calls to modulo functions in guards and array index expressions. The consequence is an extremely weak performance, frequently even a slowdown of the source program.

This section shows a solution to this problem: we start with the introduction of a typical example and present several steps towards a target program without any guards (Section 13.2.1). Then, we derive a basic procedure from the example (Section 129) and finally present several extensions to this basic procedure (Section 13.2.3).

Example 126. Consider the target loops for statements T' and S' in Examples 122 and 124, respectively. In order to obtain a single target program with one common time loop, we must enumerate all integral time steps between 0 and $2 * n - 1$, and a guard in the body must select statement T for execution if the index t of the time loop is even, and statement S if t is odd. The structure of the target program is:

```
        for t := 0 to 2*n-1
          if t%2 = 0 then
            ...
T' :          tmp[t/2, x] := a[x, t/2]/a[t/2, t/2]
            ...
          else
            ...
S' :          a[y, p] := a[y, p] - tmp[(t-1)/2, y] * a[(t-1)/2, p]
            ...
          endif
        endfor
```

where % represents the modulo function.

Reasons against modulo In this section, we derive methods to avoid modulo functions for the following reasons:

- The computation of modulo is usually very time-consuming (more than a floating point multiplication or an addition), which means that, for simple body statements, the computation of the modulo in the guard is more costly than the computation of the body statement itself.
- Postprocessing the target code (e.g., if an HPF compiler must generate the communications, cf. Section 13.5) is much more complicated in the presence of modulo functions (e.g., the HPF compiler Adaptor [Bra98] cannot handle modulo properly).

13.2.1 Principal idea

Replacing modulo by additional dimensions As we have seen in Section 3.3.5, we can express modulo with two-dimensional linear functions. Thus, a naïve approach would be to postprocess the target program accordingly. In contrast, we suggest to avoid the generation of modulo functions in the target code by adding dimensions to the schedule and placement functions. This makes code generation easier and the generated code faster.

Example 127. The guard in the program skeleton of Example 126 is a consequence of the schedule, which is

$$\theta_T(i,j,n) = 2{*}i, \quad \text{and} \quad \theta_S(i,j,k,n) = 2{*}i{+}1.$$

The coefficient of i in both schedules is 2. Thus, we can use instead the equivalent two-dimensional schedule

$$\theta_T(i,j,n) = (i,0), \quad \text{and} \quad \theta_S(i,j,k,n) = (i,1),$$

in which the second dimension enumerates the different values of the modulo function – without an explicit call to modulo.

The resulting code structure is:

```
for t1 := 0 to n−1
  for t2 := 0 to 1
    t := 2 ∗ t1 + t2
    if t2 = 0 then
      ...
T′ :    tmp[t/2, x] := a[x, t/2]/a[t/2, t/2]
      ...
    else
      ...
S′ :    a[y, p] := a[y, p] − tmp[(t−1)/2, y] ∗ a[(t−1)/2, p]
      ...
    endif
  endfor
endfor
```

Removing the guard by loop unrolling At this stage, we can (and, for efficiency reasons, should) unroll the loop that enumerates the modulo values, i.e., the inner time loop. Essentially, we undo the scheduler's distribution of operations over different loop iterations. On the one hand, the elimination of the corresponding control overhead results in much simpler and more efficient target code. On the other hand, this procedure makes the target code generation method slightly more complicated since we need to take care of synchronization and/or communication also within a loop body.

Example 128. The resulting code structure for Example 127 is:

```
for t1 := 0 to n−1
  t := 2 ∗ t1
  ...
T′ :  tmp[t/2, x] := a[x, t/2]/a[t/2, t/2]
  ...
  ;
  t := 2 ∗ t1 + 1
  ...
S′ :  a[y, p] := a[y, p] − tmp[(t−1)/2, y] ∗ a[(t−1)/2, p]
  ...
endfor,
```

where the semicolon indicates the necessary synchronization (in the case of shared memory) or communication (in the case of distributed memory) that is, in general, due to the increment of the logical time step.

Further code simplifications Frequently, the outer time loop (the only one left after unrolling the inner time loop) has an index that is identical with the source loop index. This can simplify the target code further.

Example 129. The code of Example 128 can finally be simplified by exploiting the facts that $t/2 = t_1$ in statement T', and $(t-1)/2 = t_1$ in statement S'. Since, for both statements, $t_1 = i$, we can eliminate all occurrences of t and obtain the final structure of the target program:

$$
\begin{array}{ll}
 & \mathsf{for}\ t_1 := 0\ \mathsf{to}\ n-1 \\
 & \quad \cdots \\
T': & \quad\quad tmp[t_1, x] := a[x, t_1]/a[t_1, t_1] \\
 & \quad \cdots \\
 & \quad ; \\
 & \quad \cdots \\
S': & \quad a[y, p] := a[y, p] - tmp[t_1, y] * a[t_1, p] \\
 & \quad \cdots \\
 & \mathsf{endfor}
\end{array}
$$

Note that, for the dimension under consideration, we have removed any control overhead, and everything related with the interleaving schedule of T and S or the modulo function. The resulting loop is identical with the source loop, i.e., concerning computations due to this loop, the target code is as efficient as the source code.

Comparison with text-based methods

It is interesting to see that the use of multi-dimensional schedules can bridge the gap between the flexible and, in principle, more powerful model-based parallelization methods and the simple but, if applicable, sometimes very efficient text-based methods.

Example 130. In order to observe the combination of increased flexibility and the use of a multi-dimensional schedule, let us return to the program of Example 77:

$$
\begin{array}{ll}
 & \mathsf{for}\ i := 0\ \mathsf{to}\ n \\
 & \quad \mathsf{for}\ j := 0\ \mathsf{to}\ n \\
S: & \quad\quad A[i, j] := ...B[i-1, j]... \\
T: & \quad\quad B[i, j] := ...A[i, j-1]... \\
 & \quad \mathsf{endfor} \\
 & \mathsf{endfor}.
\end{array}
$$

Since both loops carry a dependence, simple text-based methods cannot find parallelism. More elaborate text-based methods like loop skewing do find parallelism, but the solution requires linearly many communications (cf. traditional solution of Example 77).

In contrast, the model-based approach finds a placement that avoids communications (cf. our solution of Example 77 and Figure 8.9):

$$\pi_1(i, j, n) = j - i + n + 1, \quad \textit{and} \quad \pi_2(i, j, n) = j - i + n.$$

The schedule computed by LooPo is

$$\theta_1(i,j,n) = 2 * j + 1, \quad \textit{and} \quad \theta_2(i,j,n) = 2 * j.$$

Following the ideas just presented, we rewrite the schedule to

$$\theta'_1(i,j) = (j,1), \quad \textit{and} \quad \theta'_2(i,j) = (j,0).$$

The corresponding target program is again free of guards:

```
      for t1 := 0 to n
        parfor p := t1 to t1 + n
S' :      A[t1−p+1, t1] := ...B[t1−p, t1]...
        endfor
        parfor p := t1 + 1 to t1 + n + 1
T' :      B[t1−p, t1] := ...A[t1−p, t1−1]...
        endfor
      endfor
```

13.2.2 Technique

Let us now generalize the above example to a more general procedure. The resulting basic method is simple to implement, but its applicability is limited severely. Possible extensions are presented in Section 13.2.3.

Increasing the dimensionality of the schedule The essential idea of the method is to split every schedule dimension into two components, as we have seen in Example 127. A schedule dimension can be split if it contains only a single source loop index for every statement, i.e., schedule dimension i of every statement S is of the form $c_i * v_i + o_S$, where

- v_i is the same source loop index for all statements,
- $c_i \geq 1$ is the same coefficient of v_i for all statements,
- o_S is an additive constant offset that depends on the statement but on no index,
- the difference between the maximal and the minimal offset is less than the coefficient, i.e., $\max_S o_S - \min_S o_S < c_i$, and
- the index set for all statements is the same.

With these constraints, we simply replace schedule dimension i with

$$(c_i * v_i + \mathit{mino}, o_S),$$

where *mino* is the minimal offset $\min_S o_S$.

At target code generation, we unroll the loop on o_S and simplify the resulting program with standard techniques (e.g., constant propagation, simple forms of symbolic evaluation,...). The loop for the first dimension, enumerating $c_i * v_i + \mathit{mino}$, has a constant stride of c_i, and, in contrast to the original code, it enumerates the same time steps for all statements. Thus, it causes no modulo functions or conditionals in the target code.

Example 131. Let us review Example 126 at this technical level. It is easy to see that variables v_1, c_1, o_T, o_S, and mino of the algorithm are $i, 2, 0, 1$, and 0 in the example, respectively. The first dimension of the resulting schedule is $2 * i$; the second dimension is unrolled, which leads to the program of Example 128.

Eliminating fractions However, with the method just described, the array indices usually still contain (remainderless) fractions ($t/2$ for i in the above example). This can be avoided by modifying the first of the new schedule dimensions again. For correctness reasons, especially for the extensions below, this schedule dimension starts at the same value as before, but we set the stride to 1 (and reduce the upper loop bound correspondingly). This is valid since

- the time steps for the dependent statements of the computation cycle are laid out in the second, new time dimension (the dimension for o_S), and
- $c_i \geq 1$, i.e., we simply reduce the number of scheduled time steps in this dimension and
- there exist no other scheduled time steps before the last scheduled time step of the statements in the cycle.

Hence, the new schedule is

$$(\mathit{minsch}_i + v_i - \mathit{minv}_i, o_S)$$

where $\mathit{minv}_i = \min v_i$ and $\mathit{minsch}_i = (c_i * \mathit{minv}_i + \mathit{mino})$.

Example 132. Continuing Example 131, we find that $\mathit{minv}_1 = \min i = 0$, which is scheduled at $\mathit{minsch}_1 = 2*0+0 = 0$. Hence, the first dimension of the new schedule is i, as presented in Example 129.

No matter whether we apply this second change of the schedule, we call the two new schedule dimensions which replace dimension i the *cycle schedule* θ_c and the *offset schedule* θ_o, in this order.

Note that some scheduling algorithms return, for most inputs, by default multi-dimensional schedules, e.g., the scheduler by Darte and Vivien [DV94]. However, the time steps for iterations which are inside a strongly connected component of the dependence graph are still enumerated by a single loop (if possible), i.e., represented by a single dimension. Hence, also these methods can benefit from our approach.

13.2.3 Extensions

The basic idea can be generalized in various ways.

Non-intersecting time intervals

Sometimes, the above method is not directly applicable because there are a few statements whose schedules have different variables or coefficients; typical candidates leading to this scenario are initialization statements. However, if

- there are two time steps $t_<$ and $t_>$ and we can partition the set of statements into three groups G_1, G_2, G_3, such that all statements in G_1, G_2, G_3 are scheduled before $t_<$, between $t_<$ and $t_>$ (including the borders), and after $t_>$, respectively, and

- the statements in G_2 satisfy the above requirements,

then we can apply the above method for the statements in G_2 without changing the schedules for the other statements. The main reason that this is correct is that $t_<$ is equal to the minimum of the cycle schedule θ_c, and that time steps with different dimensionalities are ordered lexicographically.

More offset values

Sometimes statement instances are launched round-robin with a period of c_i, as just described, but there exist trailing computations, i.e., some postprocessing of the results of each round. In this case, we cannot find a partitioning as suggested before, since the schedule for these postprocessing steps overlaps with the schedule of other statements of group G_3. The same can happen at the beginning of the computation cycle, e.g., with preprocessing via the initialization statements which overlaps with the computations of the cycle.

In this situation, we peel off border iterations of the time loop in dimension i for these statements in the computation cycle, i.e., we partition the index set of these statements which satisfy the constraints in Section 13.2.2 in three parts P_1, P_2, P_3, such that the minimum and maximum of the scheduled time steps in P_2 can be used as $t_<$ and $t_>$. The price is that we must copy the code of the statements with a partitioned index set to every part; these partitions, each with its copy of the body statement, are then treated as if they were individual statements with a separate (usually very small) index set.

To simplify code generation, P_2 should containt only complete cycles, i.e., the number of instances in P_2 should be identical for every statement of the cycle. Therefore, we move some of the first iterations of P_2 to P_1 (or some of the last iterations of P_2 to P_3), until we reach this goal.

After this preparation, we can modify the schedule as in the original case of Section 13.2.2.

Shifted index sets

In some cases, the index sets of the statements in the cycle are not precisely equivalent, but they are shifted with a small offset. In this situation, we partition the index sets into the (typically big) common part and the border pieces which are different for different statements. Again, we consider the parts of the index sets, together with their copied body statement, as if they were individual statements. Then, we apply the method from above.

Further extensions

Combining several modulos In principle, we can extend the principal idea also to the case of different modulos for different statements.

```
        for i := 1 to n
I_1 :     C[i,0] := ···
I_2 :     E[i,0] := ···
          for j := 1 to n
S_1 :       A[i,j] := C[i,j-1]
S_2 :       B[i,j] := A[i,j]
S_3 :       C[i,j] := B[i,j]
S_4 :       D[i,j] := E[i,j-1]
S_5 :       E[i,j] := D[i,j]
          endfor
        endfor
```

Figure 13.1: Different periods for different statements

Example 133. Consider the program in Figure 13.1. An obvious placement for every statement is the projection to i; optimal one-dimensional schedules are 0, 0, $3*j-2$, $3*j-1$, $3*j$, $2*j-1$, and $2*j$ for statements $I_1, \cdots, S_5$, respectively. Again, we would like to omit the arising modulo tests.

It is easy to see that we must use the least common multiple of the modulos as the period, i.e., as the length of the loop to be unrolled. Note, however, that unrolling the loop causes a lot of code duplication. In addition, the array indices become slightly more complex than in the case of Example 129. Furthermore, the longer periods usually incur guards in order to control proper termination. Consequently, one replaces the control overhead of the modulo with the control overhead of guards in this case. However, if done carefully, one can at least split the loops, e.g., peel off border iterations and, hence, move the control statements out of the loop. The target program for Example 133 with the minimal number of control computations is presented in Figure 13.2, where the semi-colons

illustrate the different time steps within the loop body – which enforce synchronization or communication in the general case.

This is a simple example; the general situation might result in quite complex target programs. A code generation scheme for the general case is probably not theoretically difficult but still a tedious task.

Solving the problem at the root A better strategy might be to divide the schedule dimension into parts that match simpler cases, if possible. Typically, these separate simpler partitions result from scheduling different strongly connected components of the dependence graph, and the acyclic condensation of the dependence graph is the graphical representation of the desired partitioning. Hence, this method should better be incorporated into scheduling algorithms that work on the dependence graph [DV94] and that have all necessary information at their disposal.

However, this procedure only works if there are no independent dependence cycles such that the ranges of their schedules may overlap mutually. A counter-example is Example 133: in that situation, the above disadvantages cannot be fully avoided when using multi-dimensional schedules.

An alternative approach An alternative code generation scheme that eliminates the problem of code duplication entirely is given by Collard [Col94]. His approach is not to use multi-dimensional schedules but to simplify the guards in the target program. His method needs only a single integer comparison – but at every (logical) time step. Unfortunately, a sufficiently precise cost model, that allows to decide which of the alternatives to use in a given situation, is not yet available.

At this point, we have completed the postprocessing of the space-time mapping and we are ready to generate the target code.

13.3 Code generation: scanning unions of polytopes

The final step in the parallelization procedure is to enumerate the target index sets of all statements. I.e., we want to generate a loop nest that scans a set of polyhedra $\mathcal{P}_1, \cdots, \mathcal{P}_k$, where k is the number of statements in the target program.

In the next paragraphs, we list several approaches to this task, not all of which turned out to be usable in practice, and we decide which one is most beneficial.

A first approach to avoid guards Since our experiments with the naïve code generation method of Section 3.3.8 were unsatisfactory, we have developed a method that works without any guard inside loops. The idea is to partition the polyhedra $\mathcal{P}_1, \cdots, \mathcal{P}_k$ into different overlapping regions such that, for every region $\mathcal{R}$ and every polyhedron $\mathcal{P}_i$, $1 \leq i \leq k$, the intersection $\mathcal{R} \cap \mathcal{P}_i$ is either the empty set or the complete region $\mathcal{R}$. Thus,

```
        parfor i := 1 to n
I1 :      C[i,0] := ···
I2 :      E[i,0] := ···;
          for j := 1 to ⌊2*n/6⌋
S1 :        A[i,2*j-1] := C[i,2*j-2]
S4 :        D[i,3*j-2] := E[i,3*j-3];
S2 :        B[i,2*j-1] := A[i,2*j-1]
S5 :        E[i,3*j-2] := D[i,3*j-2];
S3 :        C[i,2*j-1] := B[i,2*j-1]
S4 :        D[i,3*j-1] := E[i,3*j-2];
S1 :        A[i,2*j] := C[i,2*j-1]
S5 :        E[i,3*j-1] := D[i,3*j-1];
S2 :        B[i,2*j] := A[i,2*j]
S4 :        D[i,3*j] := E[i,3*j-1];
S3 :        C[i,2*j] := B[i,2*j]
S5 :        E[i,3*j] := D[i,3*j];
          endfor
          if n%3 = 1 then
            j := ⌊2*n/6⌋ + 1
S1 :        A[i,2*j-1] := C[i,2*j-2]
S4 :        D[i,3*j-2] := E[i,3*j-3];
S2 :        B[i,2*j-1] := A[i,2*j-1]
S5 :        E[i,3*j-2] := D[i,3*j-2];
S3 :        C[i,2*j-1] := B[i,2*j-1];
            if n > 1 then
S1 :          A[i,2*j] := C[i,2*j-1];
S2 :          B[i,2*j] := A[i,2*j];
S3 :          C[i,2*j] := B[i,2*j];
            endif
          elsif n%3 = 2 then
            j := ⌊2*n/6⌋ + 1
S1 :        A[i,2*j-1] := C[i,2*j-2]
S4 :        D[i,3*j-2] := E[i,3*j-3];
S2 :        B[i,2*j-1] := A[i,2*j-1]
S5 :        E[i,3*j-2] := D[i,3*j-2];
S3 :        C[i,2*j-1] := B[i,2*j-1]
S4 :        D[i,3*j-1] := E[i,3*j-2];
S1 :        A[i,2*j] := C[i,2*j-1]
S5 :        E[i,3*j-1] := D[i,3*j-1];
S2 :        B[i,2*j] := A[i,2*j];
S3 :        C[i,2*j] := B[i,2*j];
          endif
          for j := 2*(⌈2*n/6⌉ + 1) - 1 to n
S1 :        A[i,j] := C[i,j-1];
S2 :        B[i,j] := A[i,j];
S3 :        C[i,j] := B[i,j];
          endfor
        endfor
```

Figure 13.2: Target code for Example 133

for all iteration vectors within one region, the same statements must be executed. These different regions are enumerated by different loop nests.

If there are no structure parameters (symbolic constants), this method is optimal in the sense that it leads to the minimal number of guards to be executed: in this case, target programs do not contain guards at all. Otherwise, we derive a separate loop nest for every possible scenario that can arise due to different values of the structure parameters. One big case distinction selects at run time which loop nest is appropriate for the actual parameter setting. This method is also theoretically optimal since the only guard generated must be executed once, before entering any loop nest.

However, it turns out that this method is unusable in practice, since it leads to a dramatic code explosion. The reason is that the case distinction has as many branches as there are orderings, i.e., permutations of possible values of the structure parameters: for every ordering, e.g., of two parameters n and m, we generate a separate program, e.g., one program for $n \leq m$ and one for $n > m$. Hence, the number of branches is at least the factorial of the number of structure parameters. Even worse, we also have to consider the ordering of the parameters with respect to constant loop bounds.

In our experiments, a tiny example program eventually leads to 8 loop bounds which must be compared. This crashed our PC with 128 MB of main memory (which it used completely), because there are $8! = 40320$ possible permutations, for each of which a separate version of the target program must be computed and added to the case distinction.

The Omega approach One of the most popular software tools for the code generation task is the Omega library by Pugh [PW92, KMP+96], which is based on Presburger arithmetic. However, since establishing the connection to LooPo became too elaborate, we decided not to investigate it in detail. A comparison of Omega with the method presented next is given by Quilleré [QRW00].

The approaches by Quilleré and Bastoul As an alternative, we experimented with the scanning method by Quilleré et. al. [QRW00]. This method does not evaluate all possible combinations eagerly but only generates the necessary distinctions, exploiting the fact that usually many branches of the case distinction consist of the same target loop nest. Furthermore, Quilleré's method proceeds hierarchically through the dimensions, and it allows to stop the elimination of the guards at a desired level. In that case, all guards within the innermost, hence, most important levels are removed, and the outer loops still contain guards in order to limit the code explosion.

For our tests, we have used a stand-alone implementation by Quilleré based on the Polylib [Wil93] and also a newer implementation by Bastoul, the tool *Cloog* already mentioned [Bas02, Bas03]. In the practical application, both implementations yielded good results.

In a comparison of the two implementations, Cloog turned out to be more flexible as it allows to specify the source index spaces and the space-time mapping, and it returns the target program in various syntactic forms. Furthermore, Cloog's execution time scales

very well with increasing input programs [Bas03].

Remark 134 (non-unimodularity and linearly bounded lattices for free). Note that allowing the source index sets and the space-time mapping as input is more than just a convenience which saves a few matrix multiplications arising from Equation (3.22); it takes the treatment of non-unimodular space-time mappings completely off the user's shoulders. The same is true for the complications arising from the fact that we are not really working on rational polytopes but on linearly bounded lattices (cf. Remark 22).

Due to the resulting enormous facilitation and based on the encouraging results of our experiments, we decided to apply Cloog as the code generation module for our prototype parallelizer LooPo.

13.4 Communication code generation

In Chapter 7, we have proposed an ownership-driven and a dependence-driven placement algorithm, and for both versions, we have described, in Remark 36, how these approaches lead, in principle, to communications in a parallel target program.

The purpose of this section is to demonstrate in more detail how the necessary communication statements can be generated automatically. We restrict ourselves to the dependence-driven approach – the adaptation to the ownership-driven approach is straight forward: we simply assume a dependence from the producer to the owner, and from the owner to every consumer.

13.4.1 Principal idea

In principle, the generation of communication code is again a scanning of polyhedra. As described in Section 11.4.1, the set of all communications due to one dependence relation can be represented as a polyhedron. Thus, we introduce a send and, for two-sided communication primitives, also a receive statement for every dependence that causes communication, and apply the fully developed code generation techniques mentioned in Section 13.3 in order to merge the computation and communication statements.

The set of send and receive operations The index set of the send is given by projecting the tiled communication polytope (cf. the communication polytope after space and time tiling in Section 11.4.1) to the global time and physical processor coordinate of the dependence's source statement. Analogously, the index set of a receive statement is given by projecting the space-tiled communication polytope to the time and physical processor coordinate of the dependence's destination statement.

The set of data to be sent If we project the tiled communication polytope to all coordinates of the source statement, we obtain the set of all iterations that compute data to be sent. In this set, we consider the global time t and physical processor coordinates $p_{physical}$ as a fixed parameter vector $(t, p_{physical})$; the remaining dimensions then enumerate the iteration vectors that compute data to be transferred by instance $(t, p_{physical})$ of the send statement.

Example 135. In Example 111, we have derived a space-tiled communication polytope c_3^s. The projection to the source coordinates, denoted as loop nest instead of a system of inequalities in order to improve readability, can be computed using LooPo; the result is as follows:

$$\begin{array}{l}\mathsf{for}\ t := 0\ \mathsf{to}\ 2*n-4\\ \quad \mathsf{for}\ p'_{physical} := \lceil \frac{t-126}{128} \rceil\ \mathsf{to}\ \lfloor \frac{t}{128} \rfloor\\ \quad\quad \mathsf{for}\ p'_{offset} := \lceil \frac{t}{2} \rceil\ \mathsf{to}\ \lfloor \frac{t}{2} \rfloor\\ \quad\quad\quad \mathsf{for}\ j := \lceil \frac{t}{2} \rceil + 1\ \mathsf{to}\ n-1\end{array}$$

Interpretation Note that, for even values of t, the loop on $p'_{physical}$ enumerates only one instance; also, the loop on p'_{offset} enumerates only at most one instance; it also filters away odd values of t. This matches our understanding of the communication polytope c_3 in Example 109: at every time step t, there is only one processor (with virtual processor number $t/2$, i.e., physical processor number $\lfloor t/2/64 \rfloor$) that must send data.

We also see that, along dimension j (which is mapped neither in time nor in space due to the linear dependence of schedule and placement, cf. dimension x in Example 122), several data are computed that must be transferred by the one send at time t.

Generating the complete send statement Let us now assume that we want to send the data using a broadcast, i.e., a send operation in which one processor sends one set of values to all other processors. In this scenario, we need no details for the receiver, and we can generate the communication code with the information we have already at hand. Statement T in Example 109 computes the values of array tmp (Figure 2.1); thus the details for the broadcast at time loop index t are as follows:

- *sender: $p'_{physical} = \lfloor t/128 \rfloor$*
- *first data element to be sent: $tmp[t/2, t/2+1]$*
- *last data element to be sent: $tmp[t/2, n-1]$*
- *numbers of data items: $n - 1 - t/2$*

If, fortunately, the data elements are stored contiguously in memory – as is the case here, e.g., for the programming language C – we need not even copy the values to a communication buffer.

Message vectorization automatically included It is important on today's parallel computers with distributed memory, e.g., clusters of PCs, to coalesce the messages between two physical processors at a given time [Wol95]. This idea is called message vectorization (cf. Section 8.5).

As we have just seen, our principal procedure automatically generates vectorized messages: it enumerates all iteration vectors that compute data which must be transferred by one send at a fixed space-time coordinate.

Example 136. Consider again Example 135. All values, computed at the different iterations of the j loop, are sent in a single broadcast.

Similarly, our approach vectorizes messages of various virtual processors that are mapped to one physical processor: all data to be transferred due to the enumeration of virtual processors are packed into a single send statement, just as the different data due to the iterations of the j dimension.

Point-to-point communications need further information from the tiled communication polytope In the case of point-to-point communications, where a processor sends a set of values to precisely one other processor, we also need information about the identity of the receiver. This necessary information is obtained analogously by projecting the tiled communication polytope to the global time and physical processor coordinates of the destination statement.

Similarly, if the receiver needs the local offsets of the sender, e.g., in order to know the ordering of the data in the communication buffer, we project the tiled communication polytope to the global time and physical processor coordinate of the destination statement and to the local offsets of the source statement.

Central observation In all cases, we obtain projections of a polytope, i.e., polytopes again, which can be merged by the methods described in Section 13.3 or represented as loop nests directly.

Remark 137 (integers need additional consideration). Note that this argument is, in principle, only valid in the rational or real numbers. In the integer numbers, the projection of a polytope is not necessarily a polytope, but a linearly bounded lattice (cf. Remark 22). As previously in Section 13.3, we delegate the resulting complications to the loop generation method, i.e., in our case, to Cloog.

Remark 138 (irregular communications). If we have to deal with dynamic changes in the communication behavior that cannot be treated by the methods suggested here, we can apply an ad-hoc method: we send the value together with an identification of the variable that contains this value. This procedure is universally applicable, at the price of increasing

- the communication volume by, roughly, a factor of 2 (where we assume that the identification of the variable is about as long as the value to be sent), and

- the control overhead necessary to write and read the communication buffer.

We shall not focus on this possibility further in this thesis.

13.4.2 Disadvantages of the principal communication scheme

In this section, we point out some drawbacks of the principal idea just presented.

No data reuse on the receiver side If, in the source program, some value is needed at different operations, this is coded by different dependences. Consequently, there may exist several space-mapped dependences, i.e., communications, which all have the same source and different destinations. Note that the space components of the destinations might also be identical. In this case, it does not make sense to generate several communication statements, but there should be only one communication, and the receiver should remember and reuse the received value. This idea can be generalized and leads to the following two observations.

Much implicit network buffering As described so far, we generate the send and the receive separately. This means that, if a value is computed many time steps before its use on a different processor, then the communication system buffers the value sent but not yet received. This can dramatically increase the amount of buffer memory – and the time for message administration – needed by the communication system.

Too many messages Furthermore, at the time when the value is needed and the receive takes place, it might happen that there is another receive with the same communication partner but with a different time step at which it is sent. Therefore, between any pair of processors, we can have as many point-to-point communications as there are *pairs* of time steps. Thus, the number of point-to-point communications grows with $O(N^2)$ where N represents the number of global time steps.

Message vectorization across multiple dependences Finally, we have seen how messages due to a single dependence are vectorized automatically. However, so far, we have no way to do this for different dependences.

13.4.3 A more practical communication scheme

Let us present a variant of the principal communication scheme that is simple to implement and that overcomes all drawbacks mentioned in Section 13.4.2 simultaneously.

The essential idea is to manage the communication buffers manually.

Sender side The change at the sender side is very simple to implement. We introduce, on every processor, one communication buffer for each communication partner and, instead of generating a send statement for a data item x to processor p, we generate only a statement that copies x to the communication buffer for p. The actual sends are launched at the end of each global time step.

Receiver side The main modification when compared to the principal scheme is that we do not postpone the receive until the according destination statement of the dependence is executed, but we receive all data immediately after they are sent, i.e., at the beginning of the next global time step.

For this purpose, we take the code for filling the send buffer as described before and reuse it to unpack the receive buffer – only some indices must be shifted due to a new global time step and a different processor number, and the direction of the copy statement from main memory to the communication buffer must be reversed. This procedure guarantees that the enumeration ordering of the data elements in the communication buffer is the same at the sender and the receiver side, even if, from a theoretical point of view, there is some degree of freedom in this ordering.

Correctness consideration This procedure is trivially correct if we do not apply time tiling, because our communication scheme models precisely the remote assignment or the assignment to shared memory, which is the initial model of the assignment after a space mapping. The issue of sufficient synchronization will be dealt with in more detail in Section 13.4.4.

For time-tiled programs, we need some additional consideration, since we change the schedule in the rescheduling (skewing) phase. The reason that it works despite of this change is the FCO constraint. Let us elaborate.

Assume some read R on some physical processor $p^R_{physical}$. Before the rescheduling (skewing) takes place, the schedule guarantees the correct order of all other accesses on the same memory cell M. Let W be the last write operation to M before R; W is executed on processor $p^W_{physical}$. Now, let us consider the – potentially – conflicting cases.

- If there are several other writes to M before W, they must be executed on processors p' with $p' \leq p^W_{physical}$, due to the FCO constraint for output dependences. Hence, the skewing postpones W most among all writers that are scheduled before R. Consequently, the value for M that has been sent the latest, and that, thus, will be read at R, is the value written by W. Note that, if the conflicting write takes place in the same global time step as W, the local time offset orders the execution and, again, W will be executed last.

- Due to the FCO constraint for the true dependence, $p^W_{physical} \leq p^R_{physical}$, and, thus, R is skewed at least as far as W, which means that the value arrives at R before it is needed. Again, the case of the same global time step is covered by the local offset, which is determined by the original schedule.

- The last case to check is that the value sent by W cannot be overwritten before R uses it. Again, due to the FCO constraint for anti dependences, potential overwriters are placed on processors p'' with $p^R_{physical} \leq p''$ and, thus, skewed (postponed) at least as far as R. Consequently, they write to their send buffers after R is executed and, so, they cannot overwrite M beforehand – even if the receive and unpack operations take place immediately after the send.

Thus, it is guaranteed that every read operation reads the correct value, i.e., that the communication scheme presented in this section is correct.

Remark 139 (transfer of only one value per memory cell). Similarly, every sent package must only carry a single value of a memory cell M. Assume one communication buffer contains two values for M. If the first of the two is a necessary value for the receiver, then there must exist a true dependence from the sender of the first value to the receiver, and an anti dependence from the receiver to the sender of the second value. However, since both dependences meet the FCO constraint after the space-mapping, sender and receiver must be the same processor – and no communication would have been necessary at all. Otherwise, if the receiver did not need the first value, then it would not have had to be sent either. Note that the same argument holds also for different global time steps, which means that different values of a memory cell M are never sent by the same processor.

Remark 140 (approximation of the number of communications). Note that our communication scheme guarantees that there is at most one communication at every global time step between any pair of processors, which is the most powerful message vectorization scheme possible.

13.4.4 Dependence types and their influence on communications

In this section, we explore the difference between true dependences and anti or output dependences.

Effects of communications

Data transfer At first glance, the only effect of a communication is to get some data transferred between processors. The necessity is obvious for true dependences: the receiver needs the transmitted value for its further computations. If the data is not yet available, it must wait until the data arrives. In distributed systems without a common clock tick, this is most easily achieved by using a blocking receive command for a point-to-point communication.

Note that, besides transferring data, this blocking receive also has a second effect, which we consider next.

Synchronization A blocking receive introduces some amount of synchronization, also in an asynchronous parallel program. However, this is not a barrier over all physical processors as in synchronous programs, but only a synchronization of those processors that must exchange data – and only at those time steps at which there are data to be exchanged. Hence, with different execution speeds of the processors, e.g., also due to bad load balance, this may reduce the total latency.

True vs. anti or output dependences

In contrast to true dependences, anti and output dependences do not incur a data transfer; their purpose is only to enforce a synchronization, i.e., to avoid that a memory cell is overwritten too early. In other words, true dependences always cause a communication, but anti and output dependences might not. We shall elaborate on different scenarios and check whether we can avoid to generate any kind of communication or synchronization statement due to anti and/or output dependences.

Anti or output dependences in synchronous systems

We can ignore anti and output dependences at the code generation phase if we use a synchronous target architecture. Here, it is not important whether the synchronization is due to the hardware or due to explicit synchronization commands (e.g., barrier statements) or collective communication operations (e.g., broadcast statements) that we generate. Also the memory model of the target architecture, shared or distributed, is irrelevant. In any of these cases, we need not generate code due to anti or output dependences.

Anti or output dependences in asynchronous systems with distributed memory

In order to discover whether we need communication or synchronization statements due to anti or output dependences in asynchronous systems with distributed memory, let us consider two cases:

- If the dependent operations are on the same processor, there is trivially no communication (and the schedule guarantees correctness – which, reversely, means that we need to consider these dependence types in the analysis phase and the scheduler).
- If the dependent operations are on different processors, and if we generate no communication/synchronization for anti or output dependences, we obtain multiple copies of one variable on different processors (due to true dependences) with differently updated values (due to anti or output dependences). This can only cause problems when some operation wants to read one of the updated values. In this case, however, it is clear that there is a true dependence from the producer to the consumer of the new value – and, thus, a communication of this value.

Consequently, for distributed memory systems, we need not consider anti and output dependences for communication code generation.

Comparison with single assignment form Another approach that eliminates anti and output dependences in all parallelization phases is the conversion of the source program into *single assignment form* [Fea91]. This form contains no reassignments, which eliminates automatically all anti and output dependences. The price for a single assignment form is frequently a horrible increase of memory needed by the program (due to the prohibition of reassignments). This is typically solved by applying a memory reduction phase after the parallelization – often at the price of generating expensive computations like expressions containing modulo functions [LF98, Coh99]. (Aside: also, single assignment form is not desirable for dynamic control programs [BCC98].)

The approach we suggest is less memory consuming than converting the program into single assignment form, since we do allow reassignments. In the worst case, every processor gets its own (possibly out-of-date) copy of every variable – which is an increase in memory usage by a constant factor w.r.t. the problem size. (This worst case has never occurred in our case-studies.) In contrast, single assignment conversion often leads to a linear or polynomial increase of memory usage (e.g., a scalar variable that is repeatedly reassigned inside a loop nest of depth d is converted to a d-dimensional array).

On the other hand, single assignment form allows to find more parallelism, since it eliminates all anti and output dependences before a schedule is computed; omitting these dependences only in the communication generation scheme just reduces the amount of communications, not the extracted parallelism.

Shared memory systems

In shared memory systems, all dependence types must be treated equivalently since there cannot exist different values for one variable at a time: every dependence leads to a synchronization. The only exception is the case that the synchronization is given from outside, e.g., from the hardware, as described above.

Summary: anti and output dependences are rarely needed

Anti and output dependences lead to communication or synchronization code only if the execution scheme is asynchronous and the memory system is shared.

Consequently, in this setting, we may ignore anti and output dependences for the computation of a placement. To be more precise: we may ignore them in the part of the placement algorithm that evaluates the quality of different placement candidates, since this evaluation is based on the number of communications caused (cf. Section 7.2). However, it is more difficult to answer the question whether anti and output dependences may be ignored in the part of the placement algorithm that is responsible for the FCO constraint. Finding an answer to this question is the topic of the next section.

13.4.5 The relevance of anti and output dependences for FCO

Anti and output dependences are relevant The following example illustrates that the FCO constraint must also be satisfied by anti dependences (even if they do not lead to communication code).

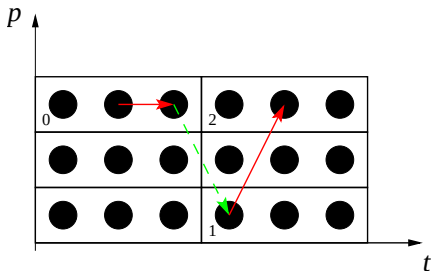


Figure 13.3: FCO for anti dependences – before rescheduling

Example 141. Consider the operations after space-time mapping and tiling, but before the necessary rescheduling (skewing), depicted in Figure 13.3. For simplicity, we assume that virtual and physical processors are identical, i.e., the tile width in the space dimension is 1; furthermore, the desired time tiling aggregates 3 logical time steps, as indicated by the vertical borders of the tiles. The numbers in Figure 13.3 just identify the tiles.

Let us assume that the dependences in Figure 13.3 all exist due to accesses to the same single memory cell M. The true dependences, represented as solid arrows, satisfy the FCO constraint, but the anti dependence, represented as a dashed arrow, points backward in the processor dimension. Obviously, all dependences are forward in time, i.e., the schedule is valid: the horizontal arrow in tile 0 represents a dependence due to a definition of M, immediately followed by a use. Later on, in tile 1, M is reassigned, and the new value is eventually used in tile 2.

The situation after rescheduling (skewing) is depicted in Figure 13.4. Again, we can find the horizontal arrow inside tile 0 but, now, it is enclosed between the definition of M in tile 1 and the use in tile 2. This means that, if the producer in tile 1 sends the value of M as soon as it is computed, indicated by the dotted arrow, the intended user in tile 2 can never see this value since M is always overwritten in tile 0 before the actual use. The reason is that the dashed arrow points – illegally – backward in time after the skewing.

In other words, this example illustrates that our idea to replace a general rescheduling by a simple skewing with a factor of 1, as introduced in Chapter 11, is only correct if our placement guarantees the FCO constraint. (The correctness for FCO placements has already been established in Section 13.4.3.)

A way out for non-FCO If one does not want to be restricted to placements that must satisfy the FCO constraint also for anti and output dependences, there is another solution. However, this changes the communication scheme again. The essential requirement is to

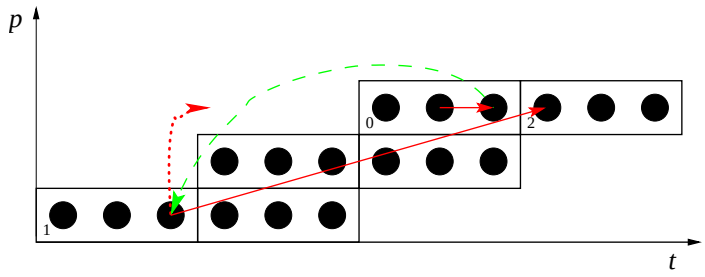


Figure 13.4: FCO for anti dependences – after rescheduling

avoid that the rescheduling (skewing) shuffles the order in which the communication buffers are received and unpacked. A simple solution is to postpone the receiving, more precisely, the unpacking of the communication buffer until the receiving processor r has reached the global time step that – *before* rescheduling – directly followed the global time step of the send.

This is correct since, logically, the data has been sent to be used at the next global time step. However, after rescheduling, the time difference between the sending and the receiving, measured in global time steps, is no longer necessarily 1, but Δ, where Δ is the distance of the sending and the receiving processor (the Manhattan distance in multi-dimensional space). This means that every processor r needs a ring of $\mid r - p \mid$ communication buffers for every communication partner p. At each global time step, we take the next buffer in the ring.

Note that this scheme requires additionally that the elements to be transferred are copied to and from communication buffer at each individual logical time step, because now there might be several (different) values of a single variable that must be transferred, and it must be assured that the receiver has access to each actual value of this variable at each logical time step. In other words, receiving the communication buffer takes place once at every global time step, whereas unpacking the relevant data takes place at every logical time step.

Example 142. Let us return to Example 141, where the value computed in tile 1 is unpacked on the third processor after *the execution of tile 0, namely in tile 1. Hence, the reads in tiles 0 and 1 both read the correct value. In other words, this scheme compensates for the change of the schedule due to the skewing by postponing the communication by the same amount.*

Again, as in Section 13.4.4 (and also in Section 5.3), we succeeded to replace the effect of anti and output dependences by increasing memory and, again, memory usage increases by a constant size, since the number of physical processors and the buffer size per global time step are finite.

13.4.6 Further research directions

Transferring garbage In the frequently arising case that the set of data to be transferred by one send operation is not contiguous in memory, we must copy the data to a communication buffer. An interesting alternative might be to send a superset of the data to be transferred, if this superset is contiguous in memory and its size is of the same order of magnitude as the size of the set of data to be transferred necessarily, e.g., an array of which only the entries at even array indices must be transferred. Note, however, that this procedure might cause consistency problems: the data sent additionally might contain garbage values that overwrite important values of the receiver, e.g., if the receiver computes the array entries at the odd indices.

In this situation, it cannot be avoided that the receiver must copy the non-garbage values from the communication buffer to the original array, i.e., we only can save the copying from the original array to the communication buffer at the sender side – at the price of increasing the transfer volume. Our cost model is not precise enough to evaluate which version performs better since the transfer of one byte is assumed to have a fixed constant cost. Instead we would need a cost model that uses separate cost functions for moving bytes over the network and for moving bytes locally between memory buffers. Thus, we leave this aspect for future work.

Transfers to additional processors Similarly, we do not investigate in detail when it would be favorable to use point-to-point communications and when to use broadcasts, even if some of the receivers of a broadcast might not need the sent values. In Example 135, we have used broadcasts, and in our practical experiments we have found that this is slightly faster than using point-to-point communications – even though broadcasts enforce a stronger synchronization scheme than point-to-point communications. Again, a more precise cost model would be necessary to predict the performance of the two variants, and we also leave this aspect for future work.

13.5 Target language

Since the purpose of parallelism is often to increase performance, the right choice of the target language is an important issue. In the LooPo project, we mainly tried C/MPI, HPF, and Java – each for slightly different purposes.

HPF

At a first glance, High-Performance-Fortran is a very good backend for parallelizing compilers. The idea is to compute an *abstract* parallel program based on our mathematical model: a program that consists of annotations specifying the desired parallelism, but that does not contain explicit code for communication. In HPF, the main annotations are templates for the distribution of data, together with some explicit assertions that our desired parallel loops can be executed independently (and on which processor).

Since HPF compilers can generate code for distributed memory architectures, we hoped to obtain efficient communication code via the HPF compiler for free. This proved true for very simple programs and space-time mappings. However, in most of our examples, the array indices generated by methods based on the polytope model are much too complex for HPF compilers to generate efficient communications. Additional annotations can occasionally improve the situation, but this procedure seems to be tedious, probably not fully automatable, and, practically most important, very compiler-dependent. This means that, for every new version of the HPF compiler, our annotation generator would have to be adapted.

Furthermore, for the development of new methods, it can be difficult to establish the cause of some phenomena: is it the new method, or the HPF compiler, or its underlying communication library? Faber discusses the problems of HPF as target language for a parallelizer in more detail [FGL01]. He tries to deal with the known difficulties by adapting the methods of the polytope model, whereas the approach described in this thesis avoids the problems mentioned before by avoiding HPF as target language – at the price of losing one level of abstraction and, thus, being forced to generate the communication code explicitly (cf. Section 13.4).

C/MPI

One possible target language that requires explicit communication code is C/MPI.

The communication library MPI offers a large variety of communication methods. In Section 13.4, we have just mentioned a few: should we use broadcast or point-to-point communications, and should we use blocking or non-blocking versions and, also, do we need an extra communication buffer? Many more options exist in MPI, and further research (especially a more detailed cost model) is needed to determine which option is the best for a given transfer requirement in a program.

In addition, the generation of the communication code itself is a very tedious and error-prone task: the low abstraction level of C allows for efficient target programs but makes the generation of correct code more difficult and, furthermore, the necessity of writing code for both sender and receiver in MPI is an additional source of errors.

However, the goal of this thesis is to present methods that can be used to build a parallelizing compiler, i.e., we need not generate the communication code by hand. And we have seen in Section 13.4 that the generation methods for the communication code can mainly be constructed by methods that are deeply rooted in the polyhedron model. Based on this mathematical framework, it should be possible to prove the correctness of the communication code generator, if desired.

Dialects of Java

During the design of Java, concurrency and distributed systems have already been taken into account. Thus, there might be hope that Java offers an abstract interface for parallel programming. However, if we require real parallelism, it turns out that we need

specialized dialects of Java focusing on parallelism. We have made case-studies with the following languages: Titanium [YSP+98], two different languages which are both called High Performance Java [BG97, ZCF+88], and JavaParty [PZ97].

The best results are obtained with JavaParty, where multiple Java virtual machines, distributed on several processors, act from a programmer's point of view like a single virtual machine with shared memory. The only new keyword in JavaParty is the modifier remote, which declares that an object may be located on a different processor. This way, the user is relieved of the technical details of RMI in Java: the compiler converts the modifier remote into RMI.

This means that, in principle, it is a lot easier to write parallel programs than in C/MPI; however, the performance is much worse. In order to obtain a performance which is at least of the same order of magnitude as the one of C/MPI, a lot of low-level tuning is necessary [Wol00]. Since this tuning is essentially an encoding of all the details of a C/MPI program in Java, we have decided to stick to C/MPI for our target code.

Chapter 14

Excursion: functional specifications as input

In the previous chapters, we have considered imperative programs with nested loops as the input to our methods. In this chapter, we illustrate that we can take functional specifications instead, provided that the dependences are affine.

For our presentation, we use Haskell, a pure, strongly typed functional language with lazy evaluation [Jon03, Has]. Our running example is again LU decomposition.

Example 143. Consider the following mutual recursive definition of LU decomposition [Ger78]:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad j \leq i, \quad i = 1, 2, \ldots, n \tag{14.1}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}}, \quad j > i, \quad j = 2, \ldots, n \tag{14.2}$$

The Haskell implementation used for this example is taken from Ellmenreich et al. [ELG99] and shown in Figure 14.1.

Transfer of the basic terms

Before applying our methods we have to determine what the statements, operations, arrays, and array cells in this context are. We assume that the functional language supports arrays, so that array definitions and accesses to array cells can be identified directly in the program text. Since, in a purely functional program, every array cell is assigned a value at most once, we can view the definition of an array cell as an operation and, thus, the defining equality as a statement. The various instances of the definition of the array cells are generated by array comprehensions in Haskell. In order to stay within the polyhedron model, we require, in analogy to the imperative setting, that the guards of the array comprehensions enumerate affinely bounded index sets. Hence, the index sets of the defining equalities, i.e., the statements, are polyhedra.

```
lu_decomp :: Array (Int, Int) Float → (Array (Int, Int) Float, Array (Int, Int) Float)
lu_decomp a = (l, u)
where
l :: Array (Int, Int) Float
l = array ((1, 1), (n, n))
[((i, j), a!(i, j) − sum [l!(i, k) ∗ u!(k, j) | k ← [1..(j−1)]] )
          | i ← [1..n], j ← [1..n], j ≤ i]

u :: Array (Int, Int) Float
u = array ((1, 2), (n, n))
[((i, j), (a!(i, j) − sum[l!(i, k) ∗ u!(k, j) | k ← [1..(i−1)]])/l!(i, i))
          | j ← [2..n], i ← [1..n], i < j]

(_, (n, _)) = bounds a
```

Figure 14.1: Haskell code for LU decomposition

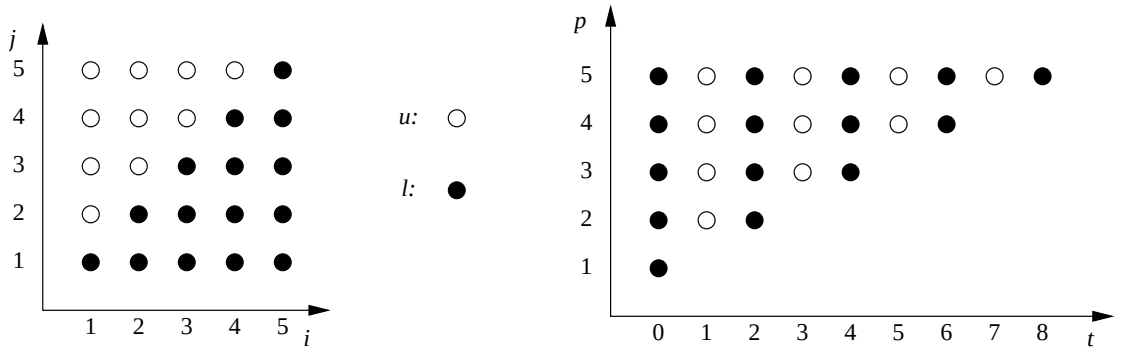
Dependence analysis is now even simpler than in the imperative setting, since we have only flow dependences: for every defining equality (statement) and for every read access on the right-hand side of the equality, we add a dependence relation from the operations that define the read values to the according operations of the current statement.

This analysis provides us with all the input required by our methods, and we compute the parallel target program as in the imperative setting.

Example 144. For the LU decomposition of Figure 14.1, we have two defining equalities, one for l and one for u. The definition of $l(i, j)$ uses the values $l(i, k)$ and $u(k, j)$, which are computed at the operations $\langle i, k; l\rangle$ and $\langle k, j; u\rangle$, respectively, for values of i, j, k as given in the program. The list of all dependences, together with the description of the dependences' index sets, is given in Figure 14.2.

	source	dest.	restricted index set
d_1	$\langle i, k; l\rangle$	$\langle i, j; l\rangle$	$(i, j, k) \in$
d_2	$\langle k, j; u\rangle$	$\langle i, j; l\rangle$	$\{1, \ldots, n\} \times \{1, \ldots, i\} \times \{1, \ldots, j-1\}$
d_3	$\langle i, k; l\rangle$	$\langle i, j; u\rangle$	$(i, j, k) \in$
d_4	$\langle k, j; u\rangle$	$\langle i, j; u\rangle$	$\{1, \ldots, j-1\} \times \{2, \ldots, n\} \times \{1, \ldots, i-1\}$
d_5	$\langle i, i; l\rangle$	$\langle i, j; u\rangle$	

Figure 14.2: Dependences in the LU example



(a) Index space of l and u

(b) Target space of l and u

Figure 14.3: Before and after the space-time mapping

Modification of the implementation

In order to allow functional specifications as input for LooPo, we just added another input interface, by which we explicitly supply the statements and their index sets, and also the dependences generated by the functional specification. Thus, Figure 14.2 illustrates the essential part of the input for LooPo using this functional interface.

Let us now complete our example.

Example 145. First, the source index set for $n = 5$ are depicted in Figure 14.3(a). Note that we do not show the dependences to make the presentation clearer.

Using Feautrier's scheduling method [Fea92a] as implemented in LooPo [Wie95] we obtain:

$$\theta_l(i,j) = 2*(j-1) \tag{14.3}$$
$$\theta_u(i,j) = 2*(i-1)+1 \tag{14.4}$$

This schedule honors the fact that the definitions of l and u are mutually recursive, so that their overall computation is interleaved.

A suitable placement, computed by Feautrier's placement method [Fea94], is

$$\pi_l(i,j) = i \tag{14.5}$$
$$\pi_u(i,j) = j \tag{14.6}$$

The resulting target index sets are depicted in Figure 14.3(b).

The target code (after using the techniques of Sections 13.2 and 13.3, but before tiling) is presented in Figure 14.4. Note that the body statements contain free loop variables k which come from the computation of the sum (cf. Figure 14.1); this summation must be expanded manually to a third loop, since our functional input specifies a two-dimensional computation (assuming that summation can be done as part of the body statement) instead

of a three-dimensional computation (as is typical for the imperative programming style). Thus, the third dimension does not become part of the parallelization process, resulting in a non-linear execution time.

Alternatively, we could have started with a functional specification containing explicit assignments to the cells of a summation array and so would have obtained an imperative program with linear execution time.

```
        for t := 1 to n−1
          parfor p := t to n
l :         l[p, t] := a[p, t] − sum(k, 1, t−1, l[p, k] ∗ u[k, t]);
u :         u[t, p] := (a[t, p] − sum(k, 1, t−1, l[t, k] ∗ u[k, p]))/l[t, t]
          endfor;
        endfor;
l′ :    l[n, n] := a[n, n] − sum(k, 1, n−1, l[n, k] ∗ u[k, n])
```

Figure 14.4: Target code for the functional specification

All in all, we have illustrated that the methods of the polyhedron model are suited for a (mutually recursive) functional specification – in which no execution order is defined – and convert it to an imperative parallel program.

The first comprehensive tool for program transformation based on the polytope model that also starts from a functional specification as input is Alpha [LMQ91]. The core of the transformation is very similar in Alpha and LooPo; the main difference is that Alpha targets at hardware description whereas LooPo targets at software – a program for a parallel computer.

Chapter 15

Conclusions

Multi-processor computers are becoming more and more popular in today's computing systems, but software that exploits all the computation power is often not available to the desired extent. This thesis is a step towards meeting this challenge.

In order to do so, it presents techniques for parallelizing compilers. Its focus is on fully automatic parallelization of nested loops. This approach has the huge advantage that the derived parallel programs are correct by construction, i.e., provably correct w.r.t. the source program or specification. Consequently, the programmer can focus on writing correct programs without thinking of parallelism – which is already a sufficiently difficult task, as many incorrect programs demonstrate (from badly formatted print-outs of dates due to the millennium bug, up to the crash of Ariane 5 in 1996 due to a counter overflow). The complexity of writing correct parallel programs is even an order of magnitude more difficult, because one must usually consider different speeds of processors, i.e., orderings of operations which are mapped to different processors. This can – and, therefore, should – be left to a compiler.

Another important aspect of today's software development is the time to market, i.e., the time from the idea for the software to the finished product. This is often determined by the time needed for programming and the time for testing and debugging. With the ideas just mentioned, it should be clear that both of these times can be reduced significantly by using automatic parallelization methods.

At the same time, the maintenance of the simpler (since non-parallel) program or specification obviously becomes easier. In addition, the issue of portability to other hardware platforms disappears completely – machine-specific details should be delegated to the compiler for that architecture, but not integrated in every application individually.

Of course, a rigorous commitment to fully automatic methods has its price:

- limited knowledge at compile time,
- limited freedom due to the representation of the input,
- limited applicability.

Let us now discuss these issues in more detail, together with possible counter-measures.

Limited knowledge at compile time

If we apply parallelization methods at compile time only, we have incomplete knowledge about the program to be executed. This can result in a loss of performance due to some missed optimizations. Typical applications are, e.g., computations on sparse data structures.

Solutions Special techniques for this field have been presented in the literature [GGL93, BW95]. In addition, general-purpose parallelization techniques for dynamic programs have been developed, e.g., inspector-executor schemes, various kinds of dynamic scheduling methods, etc. [FTYZ90, RP99]. The common idea is that the selection and/or the instantiation of the target code is shifted to run time. The consequent continuation of this idea is to shift even parts of the compilation itself to run time. This approach of just-in-time compilation has become a standard technique with the pervasiveness of Java during the last decade [Mic01, Kra98].

The methods presented here can, in principle, also be moved to run time. Obviously, this solves the problem of missing information. The only important issue in this case is that the run-time compiler must execute fast enough.

Fortunately, there are various different methods for the parallelization phases, which take different sides in the trade-off between the execution time of the parallelization phase and the quality of its result. In LooPo, we have integrated several techniques for each phase, typically with quite a number of options for every implemented method, in order to provide a testbed for comparisons (Chapter 4).

In addition, the hot-spots in programs tend to be small code portions, so that it should be possible to apply (at least the faster versions of) the presented parallelization methods in a run-time compiler. However, this idea has not been explored in detail so far; first steps in this direction are currently considered by some researchers, but no concrete results or publications are available yet.

Limited freedom due to the representation of the input

In some cases, the input to our methods might be over-constrained, e.g., due to sequentializations that are introduced by the notation of the source algorithm as an imperative loop program, but that are not necessary for the source algorithm to work. Typically, the sequential, imperative programmer is not aware of this fact, but even if he *is* aware, he has no easy way to give this information to the automatic parallelization system.

Solutions Essentially, there are two options for tackling this problem.

One is to allow user annotations in the source program. This approach is taken, e.g., by HPF, where the user annotates loops and arrays with independence and distribution suggestions. In principle, this approach is fully compatible with our methods; e.g., an adaptation of the dependence analysis to explicitly parallel programs is presented by Col-

lard et al. [CG97]. We did not stress this aspect any further since it contradicts the general idea that the user should usually not worry about parallelism.

The second solution is that the user (programmer) should write abstract specifications instead of sequential code. This avoids, at least partly, the introduction of irrelevant dependences. E.g., one could take benefit from a higher-order function called *map*, which applies a function to all elements of a list. Its usage introduces no dependences whereas an imperative, sequential loop could do so. We have seen that, in principle, our methods can accept functional specifications as input as well. However, up to now, there is no detailed analysis of which functional concepts can be transferred to our model, and how the non-transferable concepts can be handled and integrated in the parallelization process.

Limited applicability

Typically, the most serious drawback of fully automatic methods is their limited applicability. The search space for automatic solutions must be restricted in order to obtain decidable search problems and efficient methods and tools.

Solutions One goal of this thesis has been to reduce the number of limitations. A very important step in this direction is the development of a dependence analysis method which can deal with unstructured, non-recursive programs (cf. Section 5.2 and the corresponding original publication [CG99]).

Also some work has been done on recursive programs or data structures, which, unfortunately, cause severe extensions of our parallelization methods, and therefore have neither been integrated in LooPo nor in this thesis [CCG96b, CCG96a, Coh99].

Another extension of the applicability is the treatment of linearly dependent schedule and placement dimensions in Section 13.1.

Furthermore, Chapter 6 extends the applicability of index set splitting techniques [Wol95] by giving them a mathematical foundation.

Finally, all presented tiling methods originate from the question of how to tile arbitrarily nested loops with non-uniform dependences efficiently. Practical solutions to this important question have been presented in detail in Chapters 8–12.

Remaining limitation for general-purpose applications A central remaining limitation is that the only data structure allowed by our methods is the array. This limitation is acceptable for scientific computing and for programming languages which traditionally rely on arrays (e.g., Fortran); programs from other application domains, written in C, or even in object-oriented programming languages like Java are often not a suitable input to the presented parallelization method.

Note that there is no principal problem, since arrays, vectors, lists, and similar data structures are still frequently used in many programs. However, the interplay with the other central concepts of these languages (as mentioned before for functional languages) is not studied in sufficient depth.

This limitation must be tackled before the presented parallelization techniques can have a chance to be included in future general-purpose compilers.

Remaining limitation for scientific computing In the field of scientific computing, which requires much computation power and, thus, parallelism, the biggest remaining limitation of our methods is that the array indices must be affine expressions in indices of surrounding loops and structure parameters. If this constraint is not satisfied, the methods of the model can still be applied but, due to the approximations that are necessary in that case, the amount of extractable parallelism is not always sufficient.

Finally, the presented methods must be adapted to an interprocedural application, which requires, at least, a substantial amount of technical work.

All in all, this thesis proposes several methods that increase the applicability and/or quality of model-based parallelization. We hope that the presented methods help to make this way of generating provably correct and efficient parallel programs more prominent.

Bibliography

[ABRY01] Rumen Andonov, Stefan Balev, Sanjay Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 153–162. ACM Press, July 2001. Extended version available as technical report: IRISA, nr. 1392, dec. 2001.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'91)*, pages 39–50. ACM Press, 1991.

[AK87] John R. Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AMP00a] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Conference proceedings of the 2000 International Conference on Supercomputing*, pages 141–152. ACM SIGARCH, May 2000.

[AMP00b] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests (revised). Technical Report TR 2000–1782, Cornell University, Computer Science Department, Ithaca, NY 14853, 2000.

[AR97] Rumen Andonov and Sanjay Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *J. Parallel and Distributed Computing*, 45:159–165, 1997.

[ARY98] Rumen Andonov, Sanjay Rajopadhye, and Nicola Yanev. Optimal orthogonal tiling. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98: Parallel Processing*, LNCS 1470, pages 480–490. Springer-Verlag, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban79] Utpal Banerjee. *Speedup of Ordinary Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report 79-989.

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

[Ban94] Utpal Banerjee. *Loop Parallelization*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1994.

[Ban97] Utpal Banerjee. *Dependence Analysis*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1997.

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

[Bas03] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, October 2003.

[BCC98] Denis Barthou, Albert Cohen, and Jean-François Collard. Maximal static expansion. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 98–106, January 1998.

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

[BDRR94] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *INTEGRATION*, 17:33–51, 1994.

[Ber66] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15(5):757–763, October 1966.

[BG97] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, June 1997.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *Proc. Int. Conf. on Supercomputing (ICS'88)*, pages 407–417. ACM SIGARCH, ACM Press, June 1988.

[Bra98] Thomas Brandes. *ADAPTOR Programmer's Guide, Version 6.0*, June 1998. Available via anonymous ftp from ftp.gmd.de as gmd/adaptor/docs/pguide.ps.

[BS89] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik*. Harri Deutsch, 1989.

[Bur75] John Parker Burg. *Maximum Entropy Spectral Analysis*. PhD thesis, Stanford University, 1975.

[BW95] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *J. Parallel and Distributed Computing*, 31:14–24, 1995.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *Proc. 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'93)*, pages 92–102. ACM Press, July 1995.

[CCG96a] Albert Cohen, Jean-François Collard, and Martin Griebl. Array data-flow analysis for imperative recursive programs. Technical Report 96-035, PRiSM, December 1996.

[CCG96b] Albert Cohen, Jean-François Collard, and Martin Griebl. Data flow analysis of recursive structures. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, Konferenzen des Forschungszentrums Jülich 21, pages 181–192. Forschungszentrum Jülich, 1996.

[CF99] Larry Carter and Jeanne Ferrante, editors. *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99*, LNCS 1863. Springer-Verlag, 1999.

[CG97] Jean-François Collard and Martin Griebl. Array dataflow analysis for explicitly parallel programs. *Parallel Processing Letters*, 7(2):117–131, 1997.

[CG99] Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Carter and Ferrante [CF99], pages 286–302.

[Cha92] Zbigniew Chamski. Scanning polyhedra with DO loop sequences. In Blagovest C. Sendov and I. Dimov, editors, *Proc. Workshop on Parallel Architectures (WPA'92)*. Elsevier (North-Holland), 1992.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. Tenth ACM Int. Conf. on Supercomputing*. ACM Press, May 1996.

[Coh99] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, PRiSM, Université de Versailles, 1999.

[Col94] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, April 1994.

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

[Dar94] Alain Darte. Mapping uniform loop nests onto distributed memory architectures. In G.R. Joubert, D. Trystram, F.J. Peters, and D.J. Evans, editors, *Parallel Computing: trends and applications*. Elsevier Science B.V., 1994.

[DDGV00] Alain Darte, Claude Diderich, Marc Gengler, and Frédéric Vivien. Scheduling the computations of a loop nest with respect to a given mapping. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000: Parallel Processing*, LNCS 1900, pages 405–414. Springer-Verlag, 2000.

[DDRR98] Frédéric Desprez, Jack Dongarra, Fabrice Rastello, and Yves Robert. Determining the idle time of a tiling: New results. *Journal of Information Science and Engineering*, 14:167–190, 1998. Also available as technical report: INRIA, Nr. 3272, Oct. 1997.

[D'H92] E.H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. on Parallel and Distributed Systems*, 3(4):465–476, 1992.

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

[DR95] Michèle Dion and Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing & Networking (HPCN'95)*, LNCS 919, pages 184–189. Springer-Verlag, 1995.

[DRV01] Alain Darte, Yves Robert, and Frédéric Vivien. Chapter 5. Loop Parallelization Algorithms. In Santosh Pande and Dharma P. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems Languages, Compilation Techniques, and Run Time Systems*, LNCS 1808, pages 141–171. Springer-Verlag, 2001.

[DSV03] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'03)*. ACM Press, 2003. to appear.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, September 1994.

[DV96a] Alain Darte and Frédéric Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96: Parallel Processing, Vol. I*, LNCS 1123, pages 379–388. Springer-Verlag, 1996.

[DV96b] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Proc. Int. Conf. on*

Parallel Architectures and Compilation Techniques (PACT'96), pages 281–291. IEEE Computer Society Press, October 1996.

[DV96c] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Technical Report 96-06, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, April 1996.

[DV97] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6):447–496, December 1997.

[ELG99] Nils Ellmenreich, Christian Lengauer, and Martin Griebl. Applicability of the polytope model to functional programs. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing (LCPC'99)*, LNCS 1863, pages 219–235. Springer-Verlag, 1999.

[FB94] Robert W. Floyd and Richard Beigel. *The Language of Machines – An Introduction to Computability and Formal Languages*, chapter 4.4. Computer Science Press, 1994.

[Fea88] Paul Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[Fea00] Paul Feautrier. Automatic distribution of data and computation. Technical Report 2000/3, Laboratoire PRiSM, Université de Versailles, URL: http://www.prism.uvsq.fr/rapports/2000/abstract_2000_3.html, March 2000. English translation of TSI vol. 15 pp 529–557, 1996.

[FGL01] Peter Faber, Martin Griebl, and Christian Lengauer. Issues of the automatic generation of HPF loop programs. In Samuel P. Midkiff, José E. Moreira, Manish Gupta, Siddharta Chatterjee, Jeanne Ferrante, Jsn Prins, William Pugh, and Chau-Wen Tseng, editors, *13th Workshop on Languages and Compilers for Parallel Computing (LCPC 2000)*, LNCS 2017, pages 359–362. Springer-Verlag, 2001.

[FGL03] Peter Faber, Martin Griebl, and Christian Lengauer. Replicated placements in the polyhedron model. In Harald Kosch, Lázló Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003: Parallel Processing*, LNCS 2790, pages 303–308. Springer-Verlag, 2003.

[Fos95] Ian Foster. *Design and Building Parallel Programs*. Addison-Wesley, 1995.

[Frö85] Carl-Erik Fröberg. *Numerical Mathematics – Theory and Computer Applications*. Benjamin/Cummings, 1985.

[FTYZ90] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990.

[FvDF+94] James D. Foley, "Andries van" Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley, 1994.

[Gei97] Max Geigl. Parallelization of loop nests with general bounds in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, March 1997. http://www.fmi.uni-passau.de/loopo/doc/geigl-d.ps.gz.

[Ger78] Curtis F. Gerald. *Applied Numerical Analysis*. Addison-Wesley, 2nd edition, 1978.

[GFG02] Martin Griebl, Paul Feautrier, and Armin Größlinger. Forward communication only placements and their use for parallel program construction. In *Languages and Compilers for Parallel Computing, 15th International Workshop, LCPC'02*, LNCS, 2002. To Appear.

[GFL00] Martin Griebl, Paul A. Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Programming*, 28(6):607–631, 2000.

[GGL93] Alan George, John R. Gilbert, and Joseph W.H. Liu. *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[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.

[GGL04] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In *Twelveth Int. Workshop on Compilers for Parallel Computers (CPC 2004)*, 2004.

[GL97] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo—Announcement. In David Sehr, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 603–604. Springer-Verlag, 1997. More details at http://www.infosun.fmi.uni-passau.de/cl/loopo.

[GLW98] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111. IEEE Computer Society Press, 1998.

[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.

[Grö03] Armin Größlinger. Extending the polyhedron model to inequality systems with non-linear parameters using quantifier elimination. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, September 2003.

[Har73] Frank Harary. *Graph Theory*. Addison-Wesley, 1973.

[Has] Web document. URL: http://www.haskell.org.

[HCF97] Karin Högstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, New York, January 1997. ACM Press. Also available as UCSD Tech Report CS96-489.

[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.

[HKT94] S. Hiranandani, K. Kennedy, and C-W. Tseng. Evaluating compiler optimizations for Fortran D. *J. Parallel and Distributed Computing*, 21:27–45, 1994.

[HS00] Edin Hodžić and Weijia Shang. On time optimal supernode shape. In *Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 367–379, Boca Raton,FL, 2000. CRC Press.

[IT88] François Irigoin and Remi Triolet. Supernode partitioning. In *Proc. 15th Ann. ACM Symp. on Principles of Programming Languages (POPL'88)*, pages 319–329, San Diego, CA, USA, January 1988. ACM Press.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[Kie53] J. Kiefer. Sequential minimax search for a maximum. *Proceedings of the American Mathematical Society*, 4(3):502–506, June 1953.

[KL80] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 8.3. Addison-Wesley, 1980. Previously published as: Systolic arrays for VLSI, in *SIAM Sparse Matrix Proceedings*, 1978, 245–282.

[KLS+94] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Stelle Jr, and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Department of Computer Science, Univ. of Maryland, College Park, April 1996.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[KPR94] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Department of Computer Science, Univ. of Maryland, 1994.

[Kra98] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 205–212. IEEE Computer Society Press, 1998.

[Kun88] Sun-Yuan Kung. *VLSI Processor Arrays*. Prentice-Hall Int., 1988.

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

[Leh] Lehrstuhl für Programmierung, Universität Passau. The polyhedral loop parallelizer: LooPo. http://www.fmi.uni-passau.de/loopo/.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LF97] Hyuk-Jae Lee and Jose A.B. Fortes. Communication-minimal partitioning and data alignment for affine nested loops. *The Computer Journal*, 40(6):302–310, 1997.

[LF98] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, May 1998.

[LL98] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[LMQ91] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Processing*, 3:173–182, 1991.

[LP94] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *Int. J. Parallel Programming*, 22(2):183–205, April 1994.

[Max46] E.A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogeneous Coordinates*. Cambridge University Press, 1946.

[MF86] Dan I. Moldovan and José A. B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Trans. on Computers*, C-35(1):1–12, January 1986.

[Mic01] SUN Microsystems. The Java HotSpot Virtual Machine, 2001. Technical White Paper, available via http://java.sun.com/products/hotspot/docs/whitepaper.

[MJ95] Zaher Mahjoub and Mohamed Jemni. Restructuring and parallelizing a static conditional loop. *Parallel Computing*, 21(2):339–347, February 1995.

[MJ96] Zaher Mahjoub and Mohamed Jemni. On the parallelization of single dynamic conditional loops. *Simulation Practice and Theory*, 4:141–154, 1996.

[Mol83] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, January 1983.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1988.

[OSKO95] Hiroshi Ohta, Yasihiko Saito, Masahiro Kainaga, and Hiroyuki Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *Proc. 1995 Int. Conf. on Supercomputing*. ACM Press, July 1995.

[PSCB94] D. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomuters. In *International Conference on Parallel Processing*. IEEE, August 1994.

[PW92] William Pugh and Dave Wonnacott. Eliminating false data dependences using the Omega test. *ACM SIGPLAN Notices*, 27(7):140–151, July 1992. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'92)*.

[PW94] William Pugh and Dave Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[QR00] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Programming*, 28(5):469–498, 2000.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In Françoise F. Soulié, Yves Robert, and Maurice Tchuenté, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.

[Qv89] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *J. VLSI Signal Processing*, 1(2):95–113, October 1989.

[Ram92] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing '92*, pages 214–223. IEEE Computer Society Press, 1992.

[Ram95] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, October 1995.

[RAP87] Daniel A. Reed, Loyce M. Adams, and Merrell L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers*, C-36(7):845–858, July 1987.

[RK88] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, March 1988.

[RP99] Lawrence Rauchwerger and David A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *TPDS*, 10(2):160–180, feb 1999.

[Saa93] Y. Saad. *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, 1993.

[Sal] Abdusamad A. Salih. http://www.sali.freeservers.com/engineering/cfd/cfdcodes/fd4.html.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.

[SD90] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report CS-90-108, University of Tennessee, Computer Science, May 1990.

[Sei04] Georg Seidel. Methods for adaptive tiling in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004. to appear.

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[SL99] Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In Carter and Ferrante [CF99], pages 185–200.

[SL00] Yonghong Song and Yuan Lin. Unroll-and-jam for imperfectly-nested loops in dsp applications. In *Proceedings of the ACM International Conference on Compilers, Architectures, Systhesis for Embedded Systems*. ACM Press, November 2000. Also available as http://www.cs.purdue.edu/homes/songyh/research/unroll.ps.

[TT93] Jürgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *INTEGRATION*, 14(3):297–332, 1993.

[TVSA01] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A unified framework for schedule and storage optimization. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5, pages 232–242. ACMPress, 2001.

[Wet95] Sabine Wetzel. Automatic code generation in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, November 1995. http://www.fmi.uni-passau.de/loopo/doc/wetzel-d.ps.gz.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, May 1995. http://www.fmi.uni-passau.de/loopo/doc/wieninger-d.ps.gz.

[Wil93] Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, December 1993.

[WL91] Michael Wolf and Monica Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Wol89a] Michael Wolfe. More iteration space tiling. In *Supercomputing '89*, pages 655–664. ACM Press, 1989.

[Wol89b] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[Wol89c] Michel Wolfe. Iteration space tiling for memory hierarchies. In Gary Rodrigue, editor, *Proc. of the 3rd conference on Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1989.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[Wol00] Andreas Wolf. Parallele Schleifenprogramme in Java. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, April 2000.

[Wüs97] Alexander Wüst. Grafische darstellung des indexraums von anweisungen in schleifenprogrammen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, March 1997. http://www.fmi.uni-passau.de/loopo/doc/wuest-d.ps.gz.

[XH98] Jingling Xue and Chua-Huang Huang. Reuse-driven tiling for improving data locality. *Int. J. Parallel Programming*, 26(6):671–696, December 1998.

[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.

[Xue97b] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[YAI95] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations: Extended version. *International Journal of Parallel Programming*, 23(4):359–388, August 1995.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998. Special Issue: Java for High-performance Network Computing.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley (ACM Press), 1990.

[ZCF+88] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xinying Li, and Yuhong Wen. A high level spmd programming model: HPspmd and its Java language binding. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1988.

Index