UNIVERSITÄT PASSAU
Fakultät für Informatik

Automatic Code Generation for Distributed Memory Architectures in the Polytope Model

Diplomarbeit

Autor:
Michael Claßen

Aufgabensteller:
Priv. Doz. Dr. Martin Griebl
Lehrstuhl für Programmierung
Universität Passau

Passau, 30. September 2005

Abstract

The increasing availability of parallel systems leads to an increasing demand for automatic parallelization. The polytope model can be used for analyzing input programs and performing transformations on the obtained model, in order to increase the amount of parallelism in the resulting target program. In contrast to shared memory systems, a compiler for distributed memory architectures has to generate additional code for the necessary communication between processors. The goal of this thesis is to provide a method for automatic generation of efficient, parallel target code for distributed memory architectures, that uses the polytope model for the parallelization process and the generation of efficient communication code.

Acknowledgements

I would like to thank the following people for supporting me. First of all, I thank Dr. Martin Griebl for the many discussions about this topic, his valueable remarks and suggestions, and for proofreading this thesis. I thank Prof. Christian Lengauer for his comments on a draft of this thesis. I am also grateful for all the feedback in discussions with the other members of the **LooPo** team, especially Armin Größlinger, Peter Faber, Thomas Wondrak, Georg Seidel, Tilman Rabl and my brother Philipp. Thanks to Tobias Langhammer for helping me with some LaTeX-typesetting issues. Finally, I would like to thank Cédric Bastoul for his great support with **CLooG**.

Contents

Chapter 1

Introduction

1.1 Motivation

Recently, parallel computing has gained increased significance. From *hyper-threading* or *dual-core* technologies in modern CPUs to clusters of affordable desktop computers or high performance computing clusters, there are lots of different areas in which hardware manufacturers try to increase computing power by using more parallelism.

Despite of a multitude of developments on the hardware sector, there remains an obvious need of improving parallel software technology. Most software systems today provide a mechanism called *explicit parallelism* that enables the user of those systems to add parallelism manually by using language extensions or parallel programming libraries like *OpenMP* [Ope, Qui04] or *MPI* [Mes94, Qui04].

Unfortunately, these methods can be very complex to understand and lead to correctness and maintainability problems (e.g. lack of suitable debugging techniques).

Alternatively, a mechanism called *implicit parallelism* could be applied. Here, a special *parallelizing compiler* analyzes the source code to find as much parallelism as possible without a need for the user to change the sequential semantic of his code. This allows the user to concentrate on the sequential algorithm and increases the development speed and maintainability of parallel projects. Correctness can also be addressed more easily, because if the parallelizing compiler has been verified once, it can be used to generate correct parallel code from any (sequentially) correct program.

The field of parallelizing compilers has been subject to a lot of research. Automatic methods have been developed to analyze the given source program and use a mathematical model to get a better representation to work with. For this work, the *polytope model* [KMW67, Lam74, Len93] is used, which is introduced in Section 1.2.

Various methods have been developed to find certain affine functions (schedule and allocation), which can be applied to a polytope model representation of the input program in order to obtain as much parallelism as possible in the transformed program. Other transformations can be used that increase

the dimensionality of the target program in order to tune the granularity of the parallelism to get *coarser* parallel programs. These so called *tiling* techniques will also be dealt with in the course of this thesis in the context of communication between processors.

Finally, existing methods [QRW00, Bas03] can be used to generate executable code from the transformed representation of the program. Augmented by special parallel constructs, this code can be run on so-called *shared memory architectures*, where all processes use a common, *shared* memory in order to exchange data between them.

On the other hand, code generation for *distributed memory architectures* turns out to be quite challenging, because unlike in *shared memory architectures*, the generated code has also to deal with communication between processors that work on data that is not stored in their local memory.

It was the goal of this project to develop a tool for automatic code generation for distributed memory architectures in the Polytope Model.

1.2 The polytope model

As mentioned above, it is often very useful to provide an abstraction in the form of a mathematical model in which all analysis and transformation can be performed much more easily (model approach), instead of working directly on the source code of the input program.

The polytope model concentrates on parallelizing `for` loops, because most programs spend most of their execution time on iterating through loops. Also, for-loops can be very elegantly represented by a mathematical structure called *polytope*. In the course of this section, some commonly used terms concerning the polytope model will be defined.

Operation In the polytope model, for-loops are used as the only control structure. For every statement S in a loop-body, the index vector I of all surrounding loop-indices defines the iterations of S. These iterations of statements are called *operations* and each operation can be identified by its statement S and iteration vector I.

Affine transformation, affine space An affine transformation is a linear transformation followed by a translation. Similarly, an affine space is a vector space whose origin is translated by a constant vector.

Polytope A *polytope* is the mathematical structure on which the polytope model is based. It is used to represent the operations in the input program. Mathematically, it can be defined as a bounded *polyhedron*. Polyhedra are subsets of the n-dimensional space of real numbers $\mathbb{R}^n$ that are intersections of a finite number of *half-spaces*. A half-space can be defined as one of the two parts of an affine space that result when dividing that space by a hyperplane. For this work, we mostly use systems of linear inequalities to represent a polytope. The inequalities can be derived by using the bounding expressions of for-loops in the input program.

Domain, index space If we define a polytope as an intersection of half-spaces, then a polytope is necessarily *convex*. To get a more general subset of the n-dimensional space, we can use unions of (convex) polytopes, which in the context of this work will be called *domains*. Sometimes the term *index space* is also used to describe the set of of all operations for all statements.

Dependence Using domains (or unions of polytopes), we can model the set of operations in the input program, but without a given execution order. From the sequential order of operations in the input program, we can derive a partial order which prescribes that some operations are required to be executed before other operations. This partial order is defined by the *dependences* between operations in the input program. Each dependence is caused by two accesses to the same memory cell M. We call the first memory access to M the *source* of the dependence, whereas the second access is called the *destination*. If two or more operations are not comparable in the partial order, it means that these operations may be executed in arbitrary execution order, because no operation depends on the other.

Dependence types Both source and destination of a dependence involve some kind of memory access, which can consist either of writing or of reading of data. This leads to four types of dependences:

	source	
destination	reads	writes
reads	input	true
writes	anti	output

In this thesis, input dependences will be disregarded, because the execution order of two operations without a writing access should be arbitrary on most modern hardware, i.e. the hardware supports multiple concurrent reads.

We can use domains and dependences to model all operations and their execution order in the input program. This model is called the *polytope model*.

Space-time mapping, target space Having a representation of our input program in the polytope model, we can use affine linear functions to transform the original index space (and the corresponding dependences) in such a way, that the resulting transformed index space allows operations to be executed in parallel. For that purpose, we define an affine linear transformation function that we call *space-time mapping*. Every operation will be assigned to a processor within an m-dimensional field of processors (its *space* coordinate) and a certain execution time given as a n-dimensional *time* coordinate. The index space containing all transformed operations of all statements in the target program is called *target space* or *target index space* (*TISPC*).

Schedule, allocation, placement We call a *schedule* an affine function that assigns each operation from the input program to a certain time coordinate

in the transformed target program while preserving the partial order derived from the sequential execution order of the input program.

We call an *allocation* an affine function that assigns each operation from the input program to a certain processor coordinate from the field of processors in the target program. Another name for that function is *placement*, because it *places* each operation on a virtual processor.

The search for a *good* schedule and allocation is an optimization problem that has been a subject for research over the years. There are a number of well developed techniques [Lam74, Ban92, Fea92, DV94, GFG02], that try to optimize different metrices like execution time, amount of parallelism and amount of necessary communication between processors.

It is clear that, in order to use the polytope model for practical reasons, one has to transform the representation in the model back to actual executable target code. This involves generating code that originates from the statements in the input program. Furthermore, code has to be generated that enables the program to be executed on parallel architectures. There are some approaches for shared memory architectures, where no communication between processors is necessary, but for distributed memory architectures, the required additional code can become quite complex and there are still several problems to solve.

The following chapters will show an implementation of an automatic code generation tool that is also able to deal with distributed memory target architectures. This implementation is part of the **LooPo** project, in which a prototype implementation of parallelization methods has been developed.

Chapter 2

Basic approach

This chapter briefly introduces the basic approach to the design of an automatic code generation tool for the polytope model. A more detailed overview of this subject is given by Griebl [Gri04].

2.1 Input information

This code generation tool can be understood as a back-end for a parallelizing compiler that takes a representation of a sequential input program and generates a parallel target program.

Therefore, the first step consists of running a scanner and parser that parses the source code and constructs an abstract syntax tree of the input program. The polytope model is used to represent the index space of all statements as polyhedra. After analyzing the dependence structure, an affine space-time-mapping is applied to this polyhedra, in order to increase the amount of parallelism in the target program.

As we will see in Section 3.3 and later in Section 3.4.3, we have to require this *space-time mapping* function to have certain other properties in order to be usable for generating code for distributed memory architectures.

2.2 Scanning polytopes

As the result of the parallelization process, a representation of the input program in the polytope model is available. This model representation has been transformed using a space-time mapping function.

We now have to find a way to generate code consisting of for-loops that enumerate the points belonging to each corresponding polytope in our model representation. At the same time, we have to maintain the order of operations as given by the schedule function.

This method of enumerating polytopes in the correct order is called *scanning*. There are a number of different techniques that have been developed to deal with it.

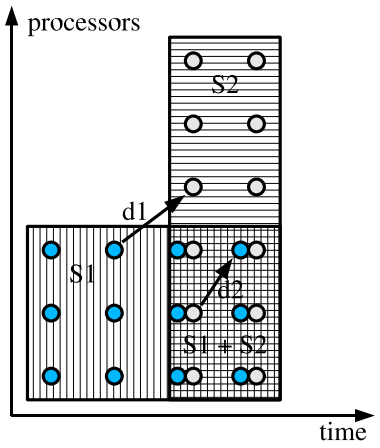

Figure 2.1: scanning polytopes

2.2.1 Basic method

The basic method described here can be implemented very easily, but has certain drawbacks, because of poor run time efficiency of the generated code.

Single polytopes

In the simplest case that we have only one single polytope to be enumerated, we can use *Fourier-Motzkin elimination* [DE73], a mathematical method for projecting polytopes (given as inequality systems) onto certain dimensions by successively eliminating variables. Using this projection method, we can derive loop bounds for our outer loop dimensions that do not contain indices of inner loops in their expressions. In this way, we can construct all loop bounds for a single polytope and generate the corresponding code.

Unions of polytopes

Unfortunately, if we have multiple statements with possibly overlapping polytopes that have to be scanned, we cannot use this simple approach separately on each polytope. The problem that arises is that our scheduling algorithm has defined a certain execution order between the points of the union of all polytopes that makes it necessary to merge loops of statements with overlapping polytopes.

Consider the example illustrated in Figure 2.1. Here, two statements ($S1$ and $S2$) with their corresponding polytopes are displayed. Note that both index spaces are overlapping and there are two dependences, $d1$ that stretches from $S1$ to $S2$ and the second dependence $d2$ going the other way round.

If we generate the resulting loop nests separately for each statement, using Fourier-Motzkin elimination, we obtain the following pseudo code:

```
for t=1..4
  for p=1..3
    S1
  end
end
for t=3..4
  for p=1..6
    S2
  end
end
```

In this case, $d2$ is violated because its destination coordinate (belonging to $S1$) is enumerated *before* its source. Of course, enumerating the index space of $S2$ first would lead to a violation of a dependence, in that case $d1$.

Thus, in order to deal with unions of polytopes, we have to extend our approach to treating all polytopes simultaneously. Therefore, we use a superset of the union of all polytopes that itself must be again a polytope. We can use a rectangular or convex hull algorithm to construct this superset of all polytopes. Now we can use Fourier-Motzkin elimination again to construct the necessary loop bounds for our loop nest.

Finally, we have to take care that each statement is only executed at the specified points within the superset of all polytopes corresponding to its own polytope. We can use `if` statements for the statements within the loop-body to assure the correct points to be executed for each statement and its corresponding polytope.

Advantages and drawbacks

As already mentioned, this method is quite basic and therefore easy to understand and implement. The generated code is also quite small in size compared to other methods, which can be seen as an advantage in some cases, because the code is more likely to fit into the instruction cache. Also, the small code size looks nicer and is more suitable for small explanatory examples than the result of the more complex methods.

However, the drawbacks of this method make it not suitable for our purposes of generating efficient target code. The main disadvantages result from constructing the convex (or rectangular) hull of all polytopes. This can lead to two problems:

Enumeration of empty subsets Because we constructed a convex superset of all polytopes, there may be "holes" within the resulting convex polytope, which leads to unnecessary loop iterations.

Control overhead Another problem results from the `if` guards that have to be introduced to specify the execution of multiple statements within the convex superset. For each iteration, there are a number of guards to be

evaluated, no matter whether the statements have to be executed or not. This can lead to huge control overhead and, thus, to poor efficiency of the target program.

Extensions to this method eliminate the control overhead from guards in the innermost loop dimensions completely, but the size of the resulting code can increase dramatically which makes it unusable in practice, as Wetzel [Wet95] illustrates.

For this method, the applicable space-time-mapping functions are restricted to matrices that holde the following property.

Definition. *A square matrix of full rank is* unimodular *if its determinant is* ± 1 *and all its entries are integer values.*

Non-unimodular transformation functions can lead to holes in the target polytopes that have to be treated carefully in the code generation algorithm. Therefore, further extensions are necessary, in order to deal with such non-unimodular functions.

It is possible to use additional guards within the loop body to exclude these statements, which represent holes in the polytope from execution. But this only increases the problems mentioned before and leads to a further decrease in efficiency.

Fortunately, there is another code generation technique for the polytope model that allows more flexibility in choosing a relation between control overhead and code size. For the project covered by this thesis, an implementation of that method by Cédric Bastoul [Bas03] is used, which also can be used for non-unimodular transformation functions.

2.2.2 Methods by Quilleré and Bastoul

We have seen some of the problems arising from our basic code generation methods like control overhead and unnecessary enumeration of empty subsets. The algorithm described by Quilleré [QRW00], implemented with some additional extensions and improvements by Cédric Bastoul [Bas03] in the ***CLooG*** *(chunky loop generator)* [Bas], was introduced to deliver a combination of efficient target code without control overhead and with acceptable code size.

CLooG can deal with (possibly overlapping) unions of polytopes and also introduces so called *scatter functions* in order to specify an execution order on all statements.

Quilleré's algorithm

The algorithm used for code generation in **CLooG** is described in detail by Bastoul [Bas03]. The basic idea of Quilleré [QRW00] is a recursive algorithm, which starts with the outermost loop dimensions of all polytopes, separates them into disjoint parts and then accumulates conditions for guards while descending deeper into the inner dimensions. Thus, it is possible to generate efficient code for the innermost loop dimensions without control overhead,

because most guards can be placed in the outer dimensions that are executed less frequently. The main advantage is that common expressions in the guards can be used for multiple statements, because the information about guards of all dimensions of all statements can be exploited when returning from the recursive descent.

CLooG

In addition to an implementation of the basic Quilleré algorithm, **CLooG** is also able to generate efficient target code for polytopes that result from *non-unimodular* space-time mapping functions. To that effect, it again tries to generate guards in the outermost dimensions rather than within the loop body. Thus, the innermost loops can be executed without control overhead caused by checking guard conditions. Of course this again leads to an increase in code size, but **CLooG** supports different options for fine-tuning the resulting target program in either control optimization or code size. Further optimizations like *loop unrolling* are also implemented. This results in target code with minimal control overhead.

Another special feature of **CLooG** is the use of so called *scatter functions*. Normally, we have a description of all statements in the target program as polytopes, which define the corresponding index spaces for each statement. In order to also specify the execution order given by the schedule function, we can use a certain affine function (the *scatter function*) for each statement, that specifies a number of additional dimensions (*scatter-dimensions*) and defines equalities between those scatter-dimensions and the dimensions of our polytope description.

Section 4.1.3 will describe, how scatter functions are used in our implementation.

2.3 Synchronous and asynchronous parallelism

In principle, we can choose between two basic schemes for loop nests in parallel programs, which differ in their enumeration order for time and space dimensions:

Synchronous target program In a *synchronous* target program, all polytopes are projected (sorted) in such way, that the outermost dimensions of the resulting loop nest contain the time dimensions, with the space dimensions of our program following in the inner part of our loop nest.

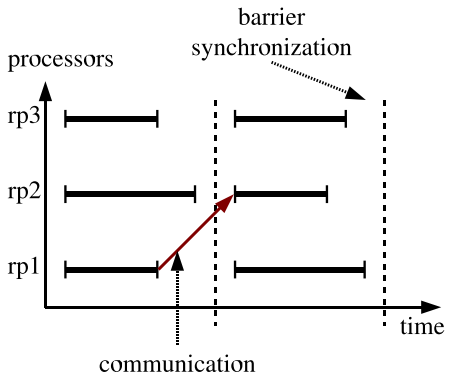

Figure 2.2: synchronous parallel target program

```
for t1=...
  ...
  for tn=...
    parfor p1=...
      ...
      parfor pm=...
        // loop-body
      end
    end
  end
end
```

In this scheme, the execution of the target program is divided into time steps that are indexed by a tuple $(t_1, \ldots, t_n)$. At each time step, the logical processor specified by the tuple $(p_1, \ldots, p_m)$ executes its loop body concurrently with all other processors (as specified here by the `parfor` construct).

At the end of each time step, a synchronization occurs between all processors. Note that it is possible in this scheme that one processor has to wait for other processors, even if there is no dependence between the other processors and the waiting processor, as illustrated for `rp3` in Figure 2.2. Similarly, all processors have to wait for the slowest processor to finish its computation.

Asynchronous target program In an *asynchronous* target program, we project our polytopes in a different order, resulting in a loop nest that contains space dimensions in the outermost loops, with time dimensions further within the loop nest.

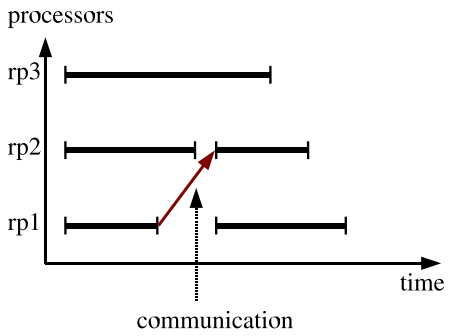

Figure 2.3: asynchronous parallel target program

```
parfor p1=...
  ...
  parfor pn=...
    for t1=...
      ...
      for tm=...
        // loop-body
      end
    end
  end
end
```

Here, the tuple $(p_1, \ldots, p_n)$ specifies a logical processor coordinate. This processor then executes inner time loops (indexed by the tuple $(t_1, \ldots, t_m)$ in parallel with all other processors (as defined by the `parfor` construct), without an explicit synchronization taking place at regular time intervals.

However, if one processor has to wait for data computed by another processor, an implicit synchronization between those processors takes place, as illustrated in Figure 2.3. This synchronization is performed by the communication statement.

Note that both schemes can be derived from the same polytope representation, using a different order of the space and time dimensions.

For this work, an early design decision had been made to use *synchronous* parallelism as target structure for the generated programs (as described in the next chapter), because of its very regular communication scheme for distributed memory systems, which can be implemented very efficiently by collective operations. These collective operations use specialized implementations for the underlying hardware architecture to provide optimal run time efficiency. On the other hand, *asynchronous* programs can theoretically result in higher speedups than *synchronous* ones when used for input programs that contain very few data dependences. In this case, each processor can execute

in parallel, most of the time without need for synchronization between processors and, thus, no required waiting time. However, in general, programs may contain enough data dependences to nullify the speedup margin of asynchronous programs compared to synchronous programs.

It should be noted that it is also possible to use combined forms of synchronous and asynchronous programs, for example:

```
for t1=1..n
  parfor p1=1..m
    for t2=1..i
      parfor p2=1..n
        ...
      end
    end
  end
end
```

It this case, alternating time and space dimensions are used, thus constructing a kind of hybrid synchronous/asynchronous target program. For now, this structure is not used in our implementation.

Remark. *Execution schemes like this could be useful to implement a hierarchy of parallelism in a parallel program, as we will discuss later in Section 5.1.2.*

2.4 Generating parallel target code

So far, we have used the polytope model to analyze and transform our input program in order to allow it to be executed in parallel on a given target architecture with the goal of a speedup in execution time compared to the original program. We also have transformed our model representation of the target program back into executable code using **CLooG** as an implementation of Quilleré's algorithm.

However, we still have to add additional code that enables our program to be executed not only sequentially, but in parallel, because our `parfor` construct is only an abstract notation for parallelism.

For the generation of this parallel code, we have to consider the type of the intended parallel target architecture.

2.4.1 Code for shared memory architectures

If we decide to generate code that should be executed on a shared memory parallel architecture, we have to to annotate our sequential code with parallel keywords (or compiler directives).

An often applied method is to use the *single-program-multiple-data* (*SPMD*) approach for the parallel target program. This means that all processors execute the same program that is parameterized by a unique number for each processor, which makes it possible to access different data on each processor.

SPMD is an extension of *SIMD* (*single-instruction-multiple-data*), a term Flynn [Fly66] uses in his taxonomy of parallel programs. It is also often used for programs running on shared memory architectures. In SIMD a single instruction can be used to execute operations on different data for each parallel processors, whereas the program still looks sequential.

If we want to use SPMD semantics to introduce parallelism to our generated sequential program, we have to pick these dimensions of our loop nest, that enumerate the processor coordinates resulting from our placement function. These loops now can be annotated using some special parallel keyword or compiler directive to indicate that each iteration of this processor loop can be executed in parallel on different processors. An example of such a compiler directive is the `#omp parallel for` directive that is used in the *OpenMP* library [Ope, Qui04] for the language *C*.

If *OpenMP* is used in a program written in *C*, in order to specify a `for` loop to be executed in parallel on all available processors, the corresponding for-loop is annotated with a `#pragma omp for` compiler directive. The resulting code for a simple one-dimensional for-loop looks as follows:

```
#pragma omp for
for (i=0; i<N; i++)
{
  c[i] = a[i] + b[i];
}
```

If the dependence structure of the target program contains dependences between different processors, we also have to introduce some form of *synchronization* construct in order to suspend the computation on a processor that is the target of a data dependence until the source processor has completed the computation of the required data.

In the case of our OpenMP example, this synchronization happens implicitly at the end of each iteration of the time loop.

2.4.2 Code for distributed memory architectures

If the intended target system is based on a distributed memory architecture, there are additional requirements for our generated code. The necessary communication between processors introduces additional problems that have to be dealt with, such as choosing adequate communication operations depending on the combination of language or additional libraries that are used and the specifics of the used network architecture. This also implies problems like buffering data during the communication or coalescing operations in order to achieve some kind of message-vectorization. These aspects of code generation that are specific to distributed memory architectures will be dealt with in more detail in the next chapter.

Chapter 3

Code generation for distributed memory architectures

This chapter deals with the aspects of automatic code generation that are specific to distributed memory target architectures.

3.1 Communication code

For our task of automatic code generation, we have to introduce additional statements that carry out the necessary sending and receiving of communicated data. Therefore, we have to consider all dependences that can lead to communication of data and choose an appropriate communication structure. The following communication scheme is also described in more detail by Griebl [Gri04].

3.1.1 Polytope representation of communication statements

Basically, we have to generate two different kinds of communication statements, one for each side of a communication: statements for *sending* data to the corresponding processor that requires it and statements for *receiving* data from the sender and updating the local array entries respectively. We can use a representation of dependences in the polytope model for finding corresponding index spaces for send and receive statements. Therefore, we use a polytope for each dependence, which is composed of the following information:

- Inequalities describing the original index spaces of the corresponding source and destination statement, before the application of the space-time transformation.
- Equalities between source and destination statements' index spaces indicating the common data access that causes the dependence.
- Further inequalities that restrict the dependence to its domain, in the case that the dependence relation only exists for a subset of the index spaces (e.g. in the case of piecewise schedules).

In order to obtain a description for the required communication between processors for later code generation, we apply the space-time transformation to the dependence polytope mentioned above, thus obtaining a *communication polytope* that describes relations between source and destination coordinates of transformed dependences in the target program.

We can use this communication polytope to create code for the sender or receiver side of a communication corresponding to a certain dependence relation. This is achieved by projecting the dimensions for sender and receiver coordinates in a different order for each side:

- For the sender side, we have to specify all destination processors for all destination time steps for a given source location in space and time of each dependence, leading to a projection order of outer dimensions for the source of the dependence and inner dimensions for the destination of the dependence.
- For the receiver side, we first enumerate all receiver locations and then enumerate nested loops for all sender locations where data has to be received from, therefore inverting the projection order of source and destination dimensions of the dependence.

Point-to-point communication operations can be used where the statements at the sender side of each dependence use operations to transmit the data to their corresponding communication partners where the data is received by explicit receive operations.

Remark. *For distributed memory architectures, using synchronous target programs, we can ignore anti and output dependences for the generation of communication code because they only indicate the need for synchronization between statements, without data transfer being necessary.*

In the course of this chapter, different extensions to this simple approach to communication code generation will be introduced.

3.1.2 Drawbacks of simple point-to-point communication

As we have seen before, we can easily derive send and receive statements from our communication polytopes. This simple point-to-point communication scheme can lead to practical problems:

Overhead from communication startups As described earlier, we generate a send statement and a corresponding receive-statement for each space-time coordinate for the source and destination of each communication polytope respectively. This means that if we have N time steps and P logical processors, we can have $O(N \cdot P)$ possible destination coordinates for each source of a dependence and each logical source processor, which all lead to separate communications. Altogether, this can result in $O(N^2 \cdot P^2)$ communications and corresponding communication startups.

Implicit network buffering Because the execution of each send statement and its corresponding receive statement can be separated by many time steps, the message passing system has to perform implicit message buffering, which can lead to excessive memory usage for buffering and an overhead of managing the increasing buffer sizes.

As a result, this simple communication scheme is obviously not well suited for distributed-memory architectures, where the cost of a communication startup tends to be the bottleneck of run time efficiency.

However, it is possible to reduce the number of startups. For this purpose, the *tiling* technique, which will be described in the next section, can be used to aggregate computation operations, thereby obtaining a coarser parallelism in the target program. We will see that communication only takes place at the end of those aggregated blocks of operations, thus reducing the startup cost. Collective communication operations can be used for the actual transmission of the aggregated data to achieve more efficiency. Certain restrictions to the space-time mapping have to be applied for tiling to work, as described later in the section about *FCO placements*.

It will also be necessary for this modified communication scheme to manually manage buffering at both sides of the involved communication partners, which will be covered in more detail in Section 3.4.2. The resulting communication structure is described in Section 3.4

3.2 Tiling

In this section, the *tiling* optimization method will be described that can be used to transform the target program in order to reduce overhead resulting from communication.

3.2.1 Tiling as an optimization technique

Tiling is an optimization technique that aims at increasing efficiency in the execution of nested loops by coalescing operations. It can be defined as partitioning the index space of a given statement into equal polyhedra, called *tiles*. As the mathematical definition of a partition implies, these tiles have to be non-intersecting and the tiled index space has to be covered completely by the union of all tiles.

The parameters that can be adjusted to define a tile are its *shape* (the direction of its spanning vectors), its *form* (which defines the size ratio in all dimensions) and its *size* (a scaling factor in all dimensions). In this implementation, an algorithm is used that restricts the shape of tiles to *parallelepipeds*, i.e. multidimensional parallelograms. Griebl [Gri04] describes in more detail the algorithm for finding optimal parameters for shape, form and size.

For the index spaces of example input programs, we only use one-dimensional processor dimensions in this thesis for the sake of simplicity, which leads to rectangular tile shapes in the corresponding figures.

However, the implementation covered by this thesis allows an arbitrary number of spatial dimensions to be tiled, which leads to tile shapes that represent general parallelepipeds, which are not necessarily rectangular.

Tiling can be used on sequential programs in order to optimize for cache efficiency, but the focus of this work is its use for optimizing the granularity of parallelism in parallel programs.

3.2.2 Tiling for parallelism

If we use our methods to generate a parallelized version of our input program, we are likely to find the speedup in execution time not satisfactory, especially on distributed memory target architectures. One of the main reasons for this problem usually is the amount of *fine grained* parallelism in the generated target program, that leads to a communication overhead that is too costly compared to the small amount of computations executed on each virtual space-time-step. Fortunately, we can use the *tiling* technique to coalesce operations, thus reducing the communication overhead and gaining *coarser* parallelism in our program.

For the project covered by this thesis, tiling is used as an additional optimization technique that is applied on the index spaces of the transformed, parallelized program, i.e. *after* the space-time mapping. The reasons for this application order are explained in detail by Griebl [Gri04], e.g. more flexibility in the parallelization methods and the possibility to use identical tiles for all transformed statements. It also allows to extract information about communication in the transformed program for the optimization algorithm of the tile-shape.

In the course of this section, two aspects of tiling will be discussed, that can be distinguished depending on whether operations are aggregated in space or time.

3.2.3 Space tiling

Generally, the application domain in which tiling is applied implies a certain way of aggregating operations. For our application domain, the code generation for distributed memory architectures, the aggregation algorithm aims at minimizing the cost of communication between processors.

Consider the example illustrated on the left side of Figure 3.1, where a simple index space is displayed. For simplicity reasons, assume that both dimensions (P_1 and P_2) are spatial dimensions.

In this case, we can use a very simple tiling to aggregate several virtual processors into larger tiles, the so-called processor tiles, as displayed on the right side of Figure 3.1. As tile shape for this example, a two-dimensional parallelogram is used.

It is now possible to execute all virtual processor coordinates belonging to the same processor tile locally on the same real processor and, thus, reduce the number of communications to the number of real processors (for programs

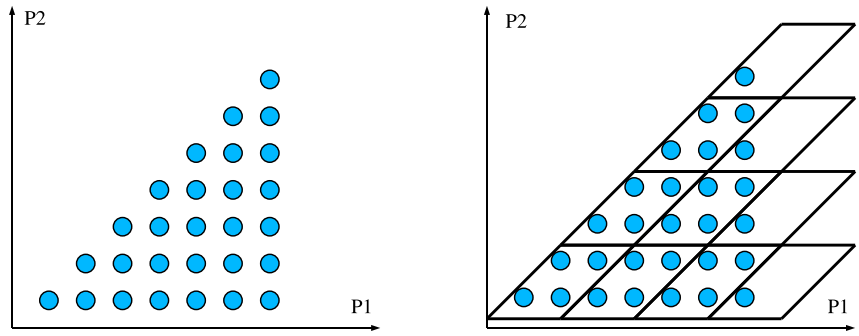

Figure 3.1: space tiling

that contain dependences between processors). This use of tiling for aggregating virtual processors into larger processor tiles is called *space tiling*.

The communication polytope described in Section 3.1.1 is extended by additional dimensions for the processor tile dimensions. Thereby, each logical processor dimension is replaced by two dimensions: one for the processor tile coordinate and the other for the processor coordinates within that tile coordinate.

Using this modified communication polytope for tiling, it is possible to derive inequality systems that specify all virtual processor coordinates belonging to a certain processor tile. These additional processor tiles can also be enumerated using **CLooG**, followed by loops for virtual processor dimensions. Our parallelization method guarantees that the processor dimensions never carry dependences and thus no care needs to be taken of a possible change in execution order when tiling processor dimensions.

3.2.4 Time tiling

In the last section, a simple point-to-point communication structure was introduced, which had some drawbacks concerning high startup cost. In order to achieve a more efficient communication scheme, tiling can be used to aggregate several logical time steps into larger chunks. This so-called *time tiling* technique results in a reduced number of *global time steps*, each global time step consisting of several logical time steps being executed within these blocks.

Instead of communicating data at each logical time step, the data is aggregated and communication of this aggregated data only takes place at the end of each global time step. Thus, the ratio of startup costs compared to computation time is much smaller, which means that the overhead for communication startup is reduced.

The aggregation of data within these global time steps requires additional buffer management at the sending and receiving side of communication. This mechanism will be explained further in Section 3.4.2.

Because time dimensions usually carry dependences, care has to be taken in order to restore the correct execution order when using time tiling.

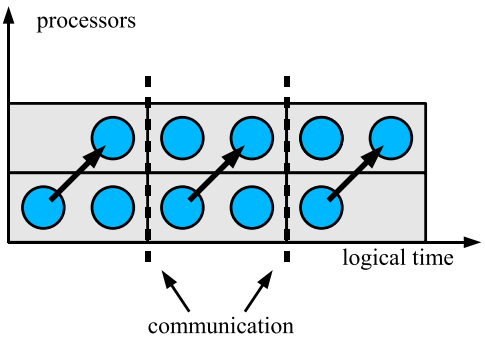

Figure 3.2: time tiling: dependences causing execution order problem

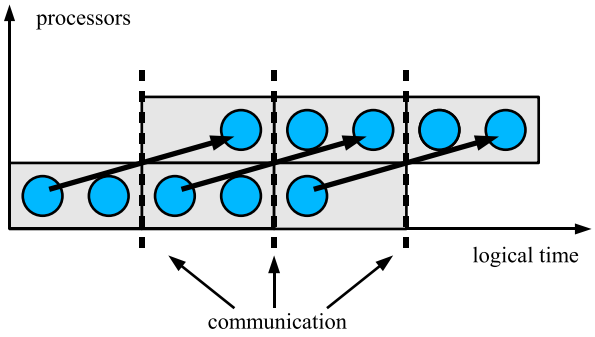

Figure 3.3: time tiling: skewing

Consider Figure 3.2, where the arrows indicate dependences that lead to a correctness problem. Without time tiling, the dependences lead to a communication between two logical time steps that is perfectly legal. However, if time tiling is used, the communication is delayed to the end of the global time step, which means that the destination processor will receive the data too late and uses undefined memory content when computing its local data. In this case, where only two processors are used, it is necessary to delay the destination time tile by one global time step, as shown in Figure 3.3. For the general case, the following definition is used:

Definition. *For vectors* $(p_1, \ldots, p_n)$ *and* $(q_1, \ldots, q_n)$*, the* 1-norm distance *or* Manhattan distance *is defined as:* $\sum_{i=1}^{n} |p_i - q_i|$.

It is necessary to delay each tile by its *manhattan distance* from the 0-processor coordinate, i.e. by the sum of all processor dimensions in its space-time coordinate.

This leads to a transformation of the index space that represents a *skewing* of the polytope representing the tile coordinate. We also implicitly assumed that all communications are directed towards processors with a component-wise higher coordinate than the corresponding source processors of the dependence. In the next section, we will see that this assumption is indeed very important.

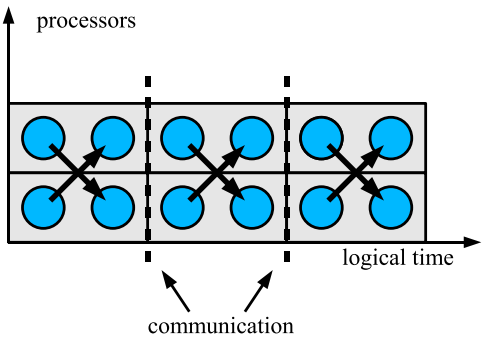

Figure 3.4: time tiling: non-fco dependence structure before skewing

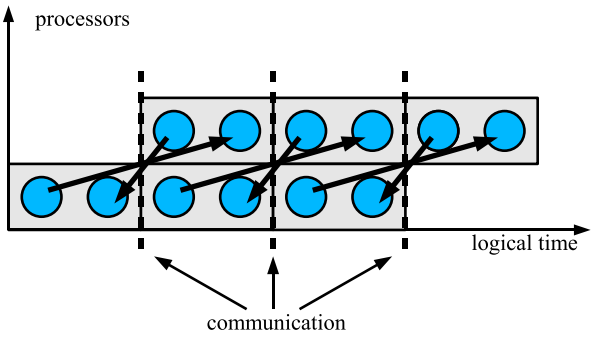

Figure 3.5: time tiling: deadlock occurs after skewing

3.3 FCO placements

In the last section, the tiling technique was presented. We have seen that its application causes no problems when tiling space dimensions, because only time dimensions carry dependences that could be violated by tiling. However, in the case of tiling time dimensions, the tiled index space has to be skewed, delaying tiles on receiving processors.

Also we implicitly assumed a certain dependence structure: all dependences are directed towards processors with higher processor coordinates. To understand the reason behind this assumption, consider the dependence structure in Figure 3.4. Here, without time tiling, the dependence structure is perfectly legal, as all dependences are directed forward in time. We can see the block structure introduced by time tiling, but skewing has not been performed yet. Figure 3.5 shows the same index space *after* skewing is applied. It is now obvious that some of the dependences are directed backwards in time, thus being illegal.

It can be easily seen that this scenario can always be avoided by using a placement that satisfies the property of *forward communications only* (*FCO property*), as described by Griebl, Feautrier and Größlinger [GFG02].

Definition. *A placement satisfies the* forward communications only *property*

*(*FCO *property), if and only if all communication vectors are component-wise positive in their spatial components, i.e.:*
$\forall d \in D : destination(d) - source(d) \geq \vec{0}$*, with* D *being the set of all* true dependences*, projected on their spatial dimensions.*

An existing implementation of an FCO allocator [GFG02] within the **LooPo** framework can be used to generate a valid placement for the use within the code generation tool, when time tiling is used.

In Section 3.4.3, the FCO property will be extended from true dependences to anti and output dependences.

3.4 Block structure for communication

With the time tiling technique described in the last section, it is possible to reduce the overhead from communication startup by aggregating blocks of several logical time steps into larger global time steps.

In this section, the buffer management used during this communication scheme is further explained. Some restrictions to the space-time mapping are introduced that arise from the use of this communication scheme. Finally, some alternatives to the approach used in this work are also discussed.

3.4.1 Buffer layout

For the block communication scheme introduced above, send and receive operations have been substituted by buffer management operations that gather data in buffers at the sender side and unpack transmitted data from buffers at the receiver side. The layout of buffers varies from on side to the other. It is strongly related to the use of collective operations for communication (i.e. the `MPI_Alltoall` operation). For this description of buffer layout, assume that space tiling has been applied and each processor tile is mapped directly onto one real processor.

Buffer layout for sending processor Figure 3.6 shows the buffer layout for one processor at the sender side. Here, each processor (i.e. each processor tile) uses separate buffers for all receiving processors to store data in the corresponding buffer for later communication. This makes it easily possible to manage each buffer separately. Increasing the size of each buffer may be necessary because in this scheme, the size of each buffer is not computed at compile time. An alternative approach, which uses a single buffer, is described in Section 3.4.5.

All data is inserted at the end of each buffer, thus keeping the stored elements ordered for later unpacking.

Finally, at the end of a global time step, all buffers are combined into one single buffer that is used for transmission by the collective operation. This communication operation is executed on each processor and transmits each part of the buffer to the corresponding destination processor.

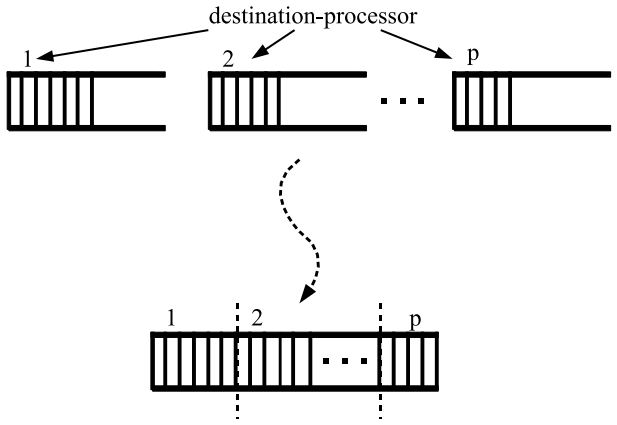

Figure 3.6: buffer layout for one sending processor

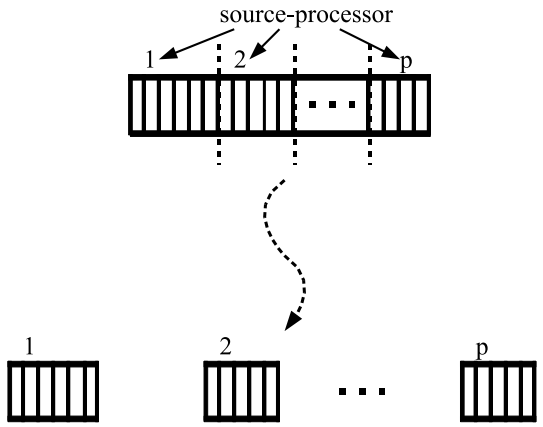

Figure 3.7: buffer layout for one receiving processor

Buffer layout for receiving processor Figure 3.7 shows the buffer layout for one processor at the receiver side. After the collective communication operation takes place, the single buffer used for receiving the data consists of logically separated parts for each processor (i.e. each processor tile) at the sender side.

When unpacking the received data, separate indices are used to keep track of the position of the next element within each of these logical parts. Thus, each of these parts can be as easily accessed as if separate buffers were used for each source processor.

Note that although each processor at the receiver side can access the buffers for each source processor separately, the unpacking of data from these buffers still has to be performed in the exact same order as the insertion of the data at the corresponding source processor.

3.4.2 Buffer management and communication statements

If the communication scheme described above is used for code generation, three types of statements in addition to the transformed computation statements can be distinguished from the input program.

Write-buffer statements At the source processor of a dependence, that leads to communication, locally computed data is stored by the write-buffer statements in the communication buffer of the corresponding target processor.

Unpack-buffer statements At the destination processor, these unpack-buffer statements are responsible for unpacking the received data from the communication buffer and update the corresponding memory cell in the local arrays with it.

Communication statement One communication statement is executed at the end of each global time step to transmit the data aggregated in the communication buffer from all source processors to all target processors (using a collective operation like `MPI_Alltoall`).

These statements are described separately by polytopes, which are used by **CLooG** in order to generate the corresponding loop nests in the target program. The loop nests of all statement types are merged together, in order to realize the required order in which the write-buffer, unpack-buffer and communication statements are executed at each logical time coordinate, relative to the compute statements in the target program. For this purpose, an additional inner schedule dimension is used to order those statements within one logical time step.

To get a better understanding of the buffer management statements, consider Figure 3.8. Here, an index space for an example program is shown, containing only two dependences, $d1$ and $d2$, which lead to communication of the memory cells M and N respectively. Assume that there are enough processor resources, so that each processor tile can be viewed as a real processor. As mentioned above, each of these real processor tiles uses separate buffers for each communication partner, including itself. This is required by the collective operation used for communication, although no actual communication from a processor to itself can occur.

In this example, memory cells M and N are computed in the first tile, with a dependence indicating the later use of N and M in the last global time step. Note that the order of the read accesses in the third global time step is inverted, compared to the order in which the corresponding write accesses occurred. In the first time step, directly following the computation statements, the new values of M and N are written to the communication buffer by the write-buffer statements. After the communication at the end of the first global time step, the unpacking of the received data takes place in the second time step at the corresponding destination processor $rp2$, with all logical coordinates being enumerated exactly as in the corresponding source tile (at the first time step), so that

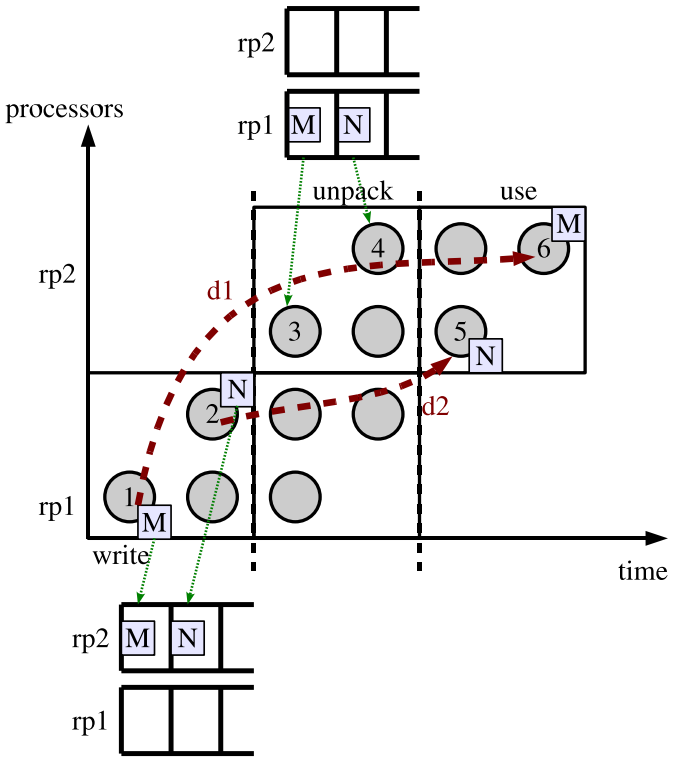

Figure 3.8: buffer-management

memory cells M and N are updated with new values from the receive-buffer in the correct order. Finally, in the last time step, these updated values of M and N are used during local computation statements.

The corresponding loop nests for both types of buffer management statements can be generated from the communication polytope described in Section 3.1.1 as follows.

Write-buffer statements

For each space-time mapped dependence d, the inequalities describing the relation between source and target tile coordinates of d are used to specify which global tiles are communicating with each other. For the write-buffer statements, this results in enumerating loop nests for the source tile coordinates of each dependence (to specify the sender) and enumerating the destination processor number (to specify all receivers of the computed data).

Note that, by leaving out the global time step of the corresponding receiver tiles, we automatically get message vectorization over all following time steps, if the sender side has to send different data to the same destination processor over multiple time steps. For the enumeration of all data that has to be aggregated in the buffer for communication within a certain tile, additional dimensions for logical time and logical processor loops are inserted.

In the example of Figure 3.8, the resulting loop nest for enumeration of

all write-buffer statements looks as follows (here, only one processor is enumerated at the source and destination side respectively and all coordinates are counted starting from 1):

```
for globalTime=1..1
  parfor srcProc=1..1
    for logicalTime=1..2
      for logicalProc=logicalTime..logicalTime
        for destProc=2..2
          /*
            for processor srcProc:
            write data at access(logicalTime,logicalProc)
            to buffer for destProc
           */
        end
      end
    end
  end
end
```

Here, at the global time coordinate 1, for processor 1, only two logical time coordinates $(1, 1)$ and $(2, 2)$ are enumerated, representing the logical time coordinates at which the data elements M and N are written into the buffer, respectively.

For the actual access of memory cells M and N, access functions can be used to compute the exact array index from logical space-time coordinates for the resulting memory access.

The actual generation of the corresponding inequality systems will be treated in more detail in Chapter 4.

Unpack-buffer statements

It is essential for both types of buffer management statements (the write-buffer and read-buffer statements) to use exactly the same order in which the elements are stored in the communication buffer that is used during the collective send operation. This is achieved by using separate buffers for each corresponding processor (as described in the section about buffer layout) and the same order for accessing the elements within each of that buffers for both types of buffer management statements.

For this purpose, the same polytope description for each unpack-buffer statement is based on the inequality system that was used for the corresponding write-buffer statement of the same dependence. Also, the same projection order for this polytope is used, except for the real processor dimensions for source and destination of the dependence, which are interchanged.

This way, for each processor at the *destination* of a dependence, there are loops enumerated for all source-processors from which data is received.

For the global time dimension, an affine function is applied to delay the unpack-buffer statements at the receiver side by one global time step, until the data from the last global time step has been communicated.

As illustrated by Griebl [Gri04], no dependences are violated by this immediate unpacking of the buffered data at the next global time step.

Although a different projection order for processor dimensions of source and destination of a dependence is used, the loop body of each unpack-buffer statement still maintains the correct order in which elements are unpacked. The reason is that the innermost dimensions for source-processors restrict the enumerated logical space-time coordinates to the relevant space-time coordinates for exactly one pair of source and destination of a dependence.

In our example (illustrated in Figure 3.8), the values of memory cells M and N are written to the buffer at the logical space-time coordinates indicated by number 1 and 2 respectively. As described above, the unpacking is performed at the next global time step (although both elements are only used at the third global time steps) by processor $rp2$. Also, both elements are unpacked in the exact same order (first M, then N) at logical space-time coordinates indicated by number 3 and 4.

The resulting loop nest looks as follows:

```
// next global time step:
for globalTime=2..2
  // destination processor tile:
  parfor destProc=2..2
    // same as at sender side:
    for logicalTime=1..2
      for logicalProc=logicalTime..logicalTime
        // now enumerate source processor of communication:
        for srcProc=1..1
          /*
            for processor destProc:
            - unpack data from buffer
            - copy to memory at:
              access(logicalTime,logicalProc)
           */
        end
      end
    end
  end
end
```

Here, at the global time coordinate 2, for receiving processor 2, the same logical time coordinates $(1, 1)$ and $(2, 2)$ are enumerated in the same order as at the sender side of the corresponding dependence, representing the logical time coordinates, at which the data elements M and N are unpacked from the buffer and written to local memory cells.

Communication statement

To get a polytope representation for all operations of the communication statement, the computation statements can be used. Therefore, we use the transformed representation of all computation statements as polytopes in the target space. After the application of time tiling, all polytopes are skewed to avoid communication cycles (cf. Section 3.2.4). These polytopes now can be projected (using Fourier-Motzkin elimination) to the global time dimension, in order to provide a representation of all global time steps that include operations for computation statements. The code generation tool **CLooG**, used in this project for generating loops from polytope descriptions, can be instructed to use these polytope representations of the projected polytopes of all computation statements as the domain description of the single communication statement. The corresponding implementation details are described in Section 3.4.2.

Although only one communication takes place in our example of Figure 3.8, the current implementation generates one communication statement at the end of each global time step, that aggregates at least one computation operation, even if no actual data is to be transmitted. In this case, all separate buffers for corresponding source or target processors are empty, so the combination into one communication buffer is not costly.

The implementation of the code generation could be extended to avoid communication statements at global time steps where no communication is needed, but at the cost of far more complex domain descriptions for each communication statement. Because the current implementation already shows problems because of long run time of the involved **CLooG** code generator for creating loop nests, this extension is not implemented.

3.4.3 Placements with strict FCO restriction

In the communication scheme described above, the communicated data is written to local array cells immediately at the next global time step, instead of delaying the reception of the data until the actual destination time step of the corresponding dependence.

As described by Griebl [Gri04, GFG02], no dependences are violated, as long as tiling is only used for space dimensions, as all dependences must be carried by time dimensions. However, the skewing of the index space required for time tiling can potentially lead to correctness issues, if no further restrictions are made to our placement function.

Consider Figure 3.9. Here, we see an index space with time tiling applied to aggregate three logical time steps into one global time step, but before skewing is applied. Processor tiling can be ignored here (assume a processor tile size of 1). Three tiles are indicated by numbers for explanatory reasons. We also see two true dependences and one anti dependence (indicated by a dotted line). For this example, all dependences can be assumed to be resulting from accesses to the same memory cell M. Thus, M is first overwritten in tile 0, then again in tile 2 with a value computed in tile 1.

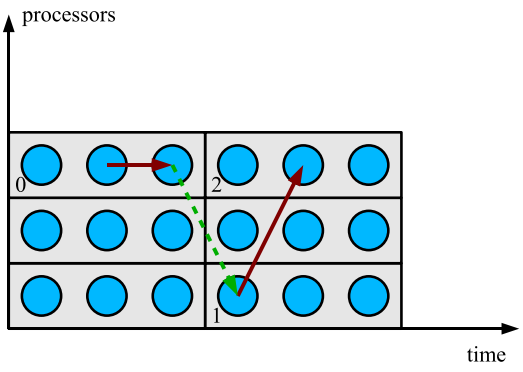

Figure 3.9: before skewing

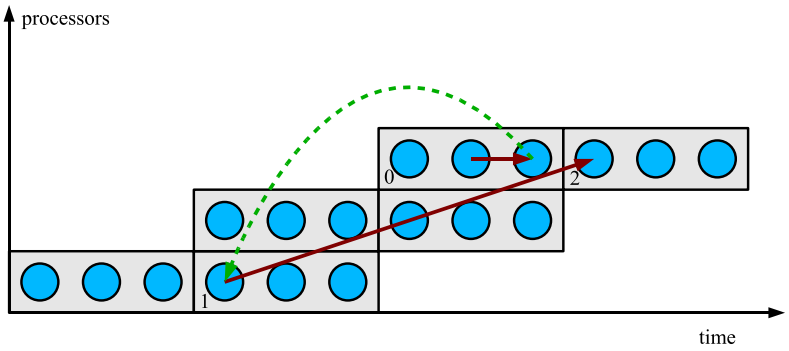

Figure 3.10: after skewing

Note that no skewing has yet been performed. So far, all dependences are directed forward in time, with the true dependences even satisfying the FCO property by also being directed forward in the processor dimension.

Figure 3.10 shows the same example, this time after skewing has been applied. All true dependences are still directed forward in time (and in processor dimensions), but we see the anti dependence now pointing backward in time. This indicates a violation of the correct execution order, in the case that the immediate unpacking of the communication data is used here.

In tile 1, the computed value is written to the communication buffer and communicated at the end of its global time step, immediately followed by unpacking the received data in tile 0. The problem now becomes obvious, because in tile 0 the received data is overwritten by a locally computed value, which is then placed in the communication buffer. Thus, tile 2 receives the value from tile 0 instead of the correct value from tile 1.

This example shows that our restriction to placements that satisfy the FCO property (for true dependences only!) is not sufficient for guaranteeing correctness in our communication scheme, when time tiling is used. In this case, the FCO property must be extended to include also anti and output dependences, thus ensuring that *all* dependences are directed forward in time after a skewing transformation (needed for time tiling).

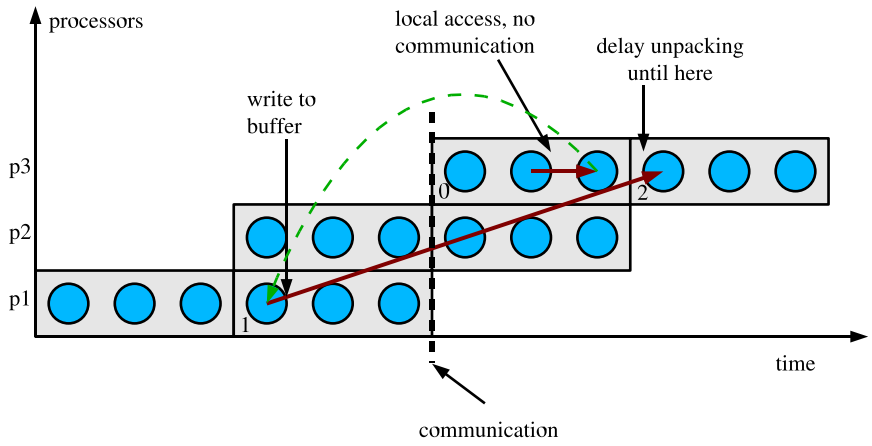

Figure 3.11: alternative buffer management for non-strict FCO placements

Definition. *A placement satisfies the* strict *FCO property, if and only if all dependence direction vectors are component-wise positive in their spatial components, i.e.:* $\forall d \in D : destination(d) - source(d) \geq \vec{0}$*, with* D *being the set of* all dependences *in the transformed program, projected on their spatial dimensions.*

In the case of the communication structure described in Section 3.4, this strict FCO property guarantees that no read access is allowed to a memory cell M between the source and destination time of a dependence that has M as destination, because the direction vector of the resulting anti dependence would not be component-wise positive.

This allows the communication scheme to communicate all aggregated data in the next global time step and immediately write the received data to local memory cell, because each local memory cell may only be accessed for reading *after* the destination of the dependence. Thus, the amount of startup overhead is reduced, because for each global source time coordinate, data is aggregated for all global destination time coordinates.

3.4.4 Coping with non-strict-FCO placements

As mentioned earlier in this section, the buffer management described in Section 3.4.2 is only guaranteed to be correct for placements that satisfy the *strict* FCO property. However, in some cases, space-time transformations that comply with this demand lead to load balance problems.

For an alternative buffer management scheme, consider Figure 3.11. Here, we can see the same example as displayed in Figure 3.10 in Section 3.4.3. All dependences result from accesses to the same memory cell M.

It is possible to use an alternative buffer management scheme for such non-strict FCO placements, where the receiving processor delays the unpacking of data by the *manhattan distance* (1*-norm distance*) of both communication partner processors.

In order to implement the delayed unpacking in this modified communication structure, a circular buffer can be used during the communication.

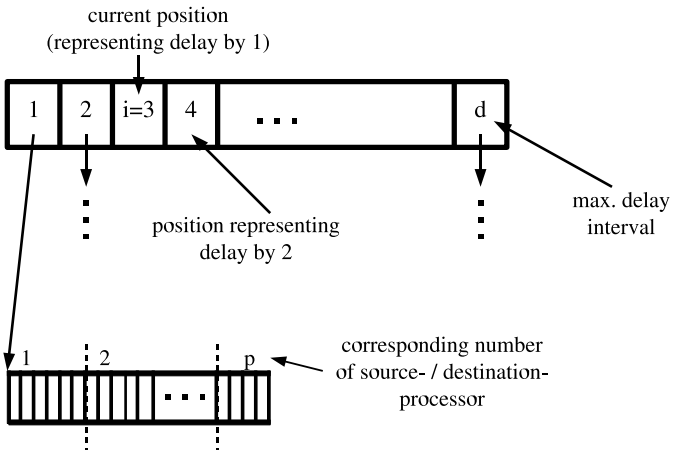

Figure 3.12: buffer layout using circular buffers

After communicated by the collective operation, the data is not immediately unpacked and written into local memory cells. Instead, it is copied from the communication buffer to a circular buffer, in order to delay the actual unpacking of data by the required time interval.

Each entry of this circular buffer represents a corresponding global time interval by which the reception of the corresponding data is delayed. Each processor keeps a counter that indicates the current position in the circular buffer that represents the minimum delay (delay by 1 global time step). The resulting location in the circular buffer for a receiving processor R for data received from sending processor S could be accessed as follows:

```
circularBuffer[( i + distance(S,P) - 1) % d]
```

Here, i is the local index kept by S indicating the current position within the circular buffer and d is the maximum possible delay, i.e. the size of the circular buffer. Because dependences with Manhattan distance 0 do not lead to communication and, thus, are ignored for code generation, the minimum delay interval is 1 global time step. Because the copying from the communication buffer to the circular buffer at the receiving processor takes place at the following global time step after the source time step of each dependence, the location has to be adjusted by subtracting one global time step. At each global time step, i is incremented.

Note that the maximum communication distance d between any pair of processors has to be known to specify the size of the circular buffer.

For each of these entries, the same buffer layout, as described in Section 3.4.1 can be used. The resulting buffer layout for the new communication buffer at each processor is illustrated in Figure 3.12. Here, the situation is illustrated for $i = 3$, which means that the entry 3 holds a pointer to the communication buffer for the shortest delay interval.

For the unpacking of data, each receiving processor has to follow the pointer in the circular buffer at the current position (indicated by $i = 3$ in

the above example) to the corresponding buffer, where the data is stored for each processor at the sender side of the corresponding dependence. Thus, the data stored in the buffer that is pointed to by the current entry in the circular buffer is guaranteed to be delayed sufficiently and can be unpacked.

3.4.5 Using a single communication buffer

As mentioned in Section 3.4.1, the use of multiple buffers for gathering the data for each destination processor involves a certain overhead for copying the buffers into one buffer required for the `MPI_Alltoall` send operation.

In order to avoid this overhead, it is possible to restrict buffer management to the use of only one buffer for gathering the data for all communication partners of a processor. The same buffer can then also be used for communication. However, to keep the gathered data ordered for later unpacking, it is still necessary to divide this single buffer into multiple logical blocks for each destination processor. As the number of communicated data for every destination processor can vary for each source processor and also for each time step, it is required to precompute the size of these blocks at compile time, in order to be able to manage the insertion of the gathered data at the correct position in the corresponding buffer.

For this precomputation of the amount of communicated data for each pair of source and destination processor at each global time step, it is possible to count the points within the corresponding communication polytopes. However, these polytopes may contain parameters that are only known at run time, so a mathematical method has to be used that makes it possible to count points within *parameterized polytopes*. For this purpose, so called *Ehrhart Polynomials* [Cla96] can be used to find a function of the parameters that represents the number of integer solutions to a system of parameterized linear constraints from the corresponding polytope.

For the implementation covered by this thesis, this single-buffer approach has been discarded, because of the overhead caused by the resulting complex functions used in indexing the buffer. Instead, the above mentioned buffer management is applied, using separate buffers for each communication partner of a given processor tile.

3.5 Mapping on real processors

In the communication schemes described in Section 3.4, each logical processor (or processor tile in case that space tiling is used) was assumed to represent exactly one real processor, on which the corresponding operations are executed. However, in order for this scheme to work, the number of available processors would either be sufficiently high for any number of processor tiles or space tiling would have to be applied to produce the exact number of processor tiles for available real processors. As the number of real processors is obviously restricted, only the second solution is practicable. However, for this approach to work, space tiling has to be extended to the case that parameters can be used

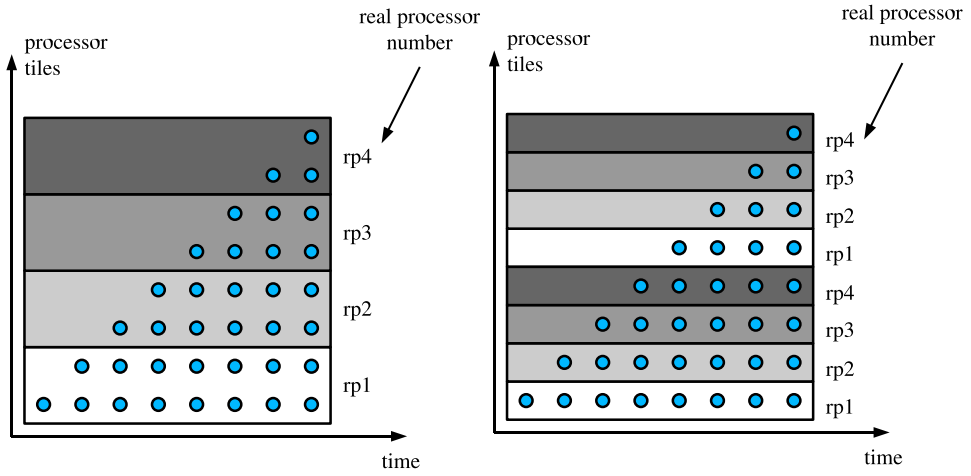

Figure 3.13: mapping to real processors: distribution strategies

for specifying the tile size of processor tiles (cf. Section 5.1), thus specifying the resulting number of processor tiles. This approach was not used in our implementation, because of the already high complexity of the loop generation algorithm.

3.5.1 Distribution strategies

Alternatively, space tiling can still be used to coalesce operations in order to increase the granularity of parallelism, but for the actual mapping to real processors an additional mapping mechanism is introduced that decides for each real processor, which processor tiles are to be executed on it.

Thus, the result is a mapping scheme split into two stages:

1. logical processor coordinate $\longrightarrow$ processor tile coordinate
2. processor tile coordinate $\longrightarrow$ real processor number

As mentioned above, the first stage is realized by the space tiling technique. For the second stage, different mapping strategies can be chosen:

Block distribution As illustrated in the left part of Figure 3.13, in a block distribution strategy adjacent processor tiles are mapped to the same physical processor. This may reduce communication between processors, if communication takes place within those blocks of processor tiles, however, space tiling already has been applied for this purpose, so the additional amount of reduced communication is expected to be negligible. Moreover, block distribution can lead to problems with load balancing, i.e. in the example of Figure 3.13, real processor number 4 has only 3 operations to execute, whereas the first real processor executes 15 operations.

Cyclic distribution Alternatively, processor tiles can be mapped to real processors using a cyclic distribution, as illustrated in the right part of Figure 3.13. Here, each processor tile is assigned to a real processor number that is increased in turn, using modulo arithmetic to limit the range of the counter to the available number of real processors. Cyclic distribution may lead to more communication of data compared to a block distribution strategy, because for adjacent processor tiles communication is necessary. However, cyclic distribution results often times in a better load balance than block distribution, as can be observed in Figure 3.13.

Both mechanisms allow a mapping to an arbitrary number of available real processors that has to be known only as late as run time, without the need to specify the exact number of processors already at compile time, as is necessary when space tiling is used exclusively for distribution of logical processors on real processors.

Because space tiling already provides a good instrument for reducing communication between processors for local communications, and because of the load balancing advantage, it is often sensible to implement cyclic distribution of processor tiles.

In some cases, however, especially when space tiling is not applied or cyclic distribution is not desired because of a dependence structure that leads to local communication, block distribution is better suited as a mapping strategy.

In this implementation, both strategies are implemented, allowing the user to choose one distribution strategy at compile time.

Remark. *The combination of space tiling with cyclic distribution is known as a* block-cyclic *tiling. By experimenting with different values for the size of processor tiles, it is possible to fine-tune the distribution structure for different target architectures and input programs.*

3.5.2 Generating a processor mapping

For the actual code generation, there are different ways to implement the mapping of logical processors (or processor tiles) onto real processors.

Using the rectangular hull

One approach is to use symbolic expressions to compute the corresponding real processor number directly from a given multi-dimensional processor tile coordinate, using a rectangular hull algorithm for determining the size in each processor dimension.

Therefore, in a first step, the possibly multidimensional logical processor (processor tile) coordinates have to be transformed into a one-dimensional coordinate.

Let s_i be the size of processor dimension P_i for $i \in 1..n$, where n is the total number of processor dimensions, then the one-dimensional coordinate P_{one} can be computed from the entries $(P_1, \ldots, P_n)$ of the n-dimensional logical processor coordinate P using a lexicographic order as follows:

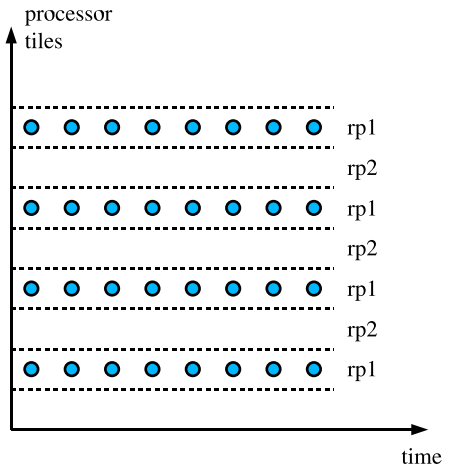

Figure 3.14: load balance problem with rectangular hull

$$P_{one} = (((P_n \cdot s_{n-1}) + P_{n-1}) \cdot s_{n-2} + P_{n-2}) \ldots \cdot s_1 + P_1$$

This one-dimensional number is then mapped to a real processor coordinate, using a modulo-expression in the case that a cyclic distribution strategy is applied (P_{size} being the number of available real processors):

$$P_{real} = P_{one} \bmod P_{size}$$

In order to obtain the size of each processor dimension, a *rectangular hull* of all polytopes of all computation statements can be used, after projecting these polytope representations onto the processor dimensions. From the resulting rectangular polytope, the lower and upper bounds in each dimension can be derived easily and, thus, the size of each dimension can be computed symbolically.

However, this processor mapping algorithm can lead to poor load balance, because holes between polytopes are implicitly enumerated when using a rectangular hull algorithm. Consider the example illustrated in Figure 3.14, assuming two available real processors ($rp1$ and $rp2$). Here, the index space of all computation statements is not contiguous, because every other logical processor coordinate is not executing any computations. However, the rectangular hull ignores these holes in the index space, which leads to a mapping, where all operations are mapped to $rp1$. Similar problems with load balance can also occur, when a block distribution strategy is used. In order to avoid these problems, the enumeration of occurring processor tile coordinates is performed dynamically at run time.

Using a dynamically computed processor map

If it is necessary to avoid holes in the index space when enumerating all processor tile coordinates, an alternative to the above described symbolical approach

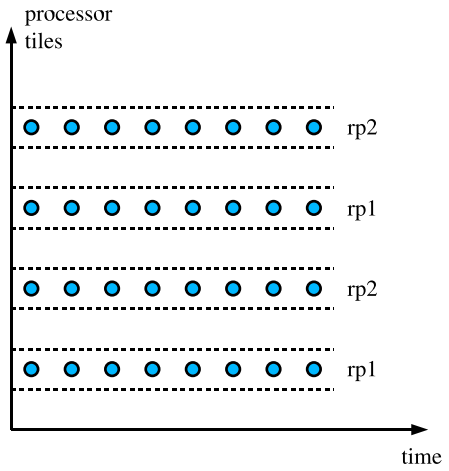

Figure 3.15: exact processor mapping using a precomputed processor map

has to be found. The implementation treated by this thesis uses **CLooG** again, to enumerate all processor tile coordinates at run time. The spoken constructs a map at run time, when the exact number of real processors is available. This map contains entries for each multidimensional processor tile coordinate that specify a corresponding real processor coordinate. During execution of the target program, this map can be used to lookup corresponding real processor coordinates.

For constructing the input description used by **CLooG**, the polytope representations of all computation statements are used. Therefore, each polytope is projected onto its processor tile dimensions. The union of these polytopes then is used as domain description in the input file, from which **CLooG** generates the resulting loop nests. Finally, code is inserted into the generated loop nest, that creates an entry in the processor map for each iteration of the processor tile loop. Here, the user can choose between cyclic and block distribution for the mapping of processor tile coordinates to real processor numbers.

As mentioned in Section 2.2.2, **CLooG** is able to generate code for unions of polytopes, while also handling non-unimodular transformations on these polytopes. Thus, **CLooG** is perfectly suitable for enumerating the required processor tiles and avoiding enumeration of holes in the index space. In the example illustrated in Figure 3.15, the resulting mapping is displayed.

Compared to the mechanism of using a rectangular hull described above, this mapping distributes a real processors onto a certain processor tiles, if and only if there is at least one computation statement operation executed on that processor tile in the target program. However, as this mechanism requires the construction of the map at run time, along with the lookup operations that are required additionally (also at run time), a certain amount of run time overhead is introduced.

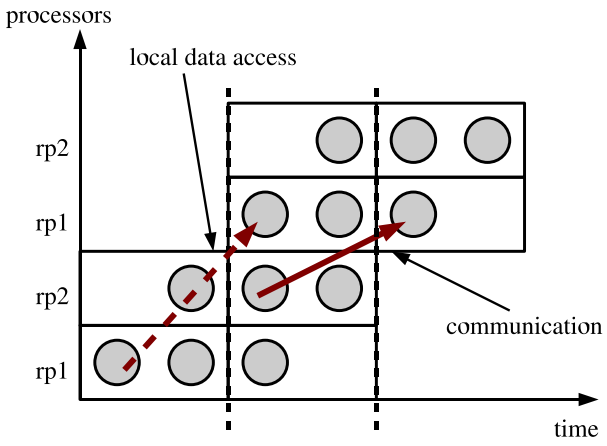

Figure 3.16: modified buffer-management

3.5.3 Modifications for buffer management statements

With the availability of a processor map for dealing with the distribution of processor tiles (or logical processors) onto the available real processors, it is now necessary to revisit the above described buffer management.

As described in Section 3.4.2, two types of statements are generated for managing the aggregation of data in the communication buffer at the sender side and unpacking the corresponding data at the receiver side respectively:

- write-buffer statements
- unpack-buffer statements

Using the processor-mapping mechanism described above, it is now possible for each buffer management statement to determine the exact real processor for both the corresponding processor tile used as communication partner and the processor tile used for the statement itself, thus allowing to distinguish between local and non-local communication at run time. This is done by performing a lookup in the processor-map, using the n-dimensional processor tile coordinate for both involved buffer management statements.

The information about the executing real processors allows the statements to restrict operations on the buffer to the cases of non-local communication, where an actual transmission of data is necessary. This way, no locally available data is transmitted.

For a better understanding, consider Figure 3.16, where the left dependence is between two processor tile coordinates, that are mapped to the same real processor. This dependence does not lead to communication, because the involved memory cell can be accessed locally. However, the destination of the second dependence has a different real processor number assigned to it than the source, thus leading to communication of the value from the corresponding memory cell.

3.5.4 Using processor mapping in SPMD programs

Without processor mapping, the target program generated until this point implicitly assumes that the loops corresponding to processor tile dimensions will be executed, with each iteration of these loops running on a separate real processor. In this thesis, these parallel dimensions have been indicated by using a `parfor` construct instead of the sequential `for` construct.

Now that the processor mapping mentioned above allows the logical mapping from processor tiles to real processors, it is possible to specify the executing real processor (by inserting an `if` statement), instead of the `parfor` construct, in order to create target code that could be executed in parallel on a distributed memory system.

As the next section will deal in more detail with the actual combination of target language and communication library that is used in this project, the next example uses pseudo code to describe the principle:

```
const myRank = initRank()
[...]
for globalTime=..
  for srcProc1=1..n
    for srcProc2=1..m
      if (lookup(srcProc1,srcProc2) == myRank)
        for logicalTime=..
          for logicalProc=..
            for destProc1=..
              for destProc2=...
                if(lookup(destProc1,destProc2) != myRank)
                  writeBuffer()
                [..]
                end if
              end
            end
          end
        end
        [..]
      end if
    end
  end
  communicateData()
end
```

This simplified extract from a parallel target program only shows a small part of a write-buffer statement and indicates a communication statement, that is executed at the end of each global time step. Here, the `parfor` construct is substituted by a sequential `for` loop. Moreover, there is an additional `if` statement inserted that performs a lookup of the two-dimensional processor tile coordinate.

The resulting real processor number is then compared with a constant that is initialized at the begin of program execution with the real processor number of the current executing process. Together with the SPMD principle, that implies the execution of the same program code for all processes, each processor initializes this constant with its specific number. The write-buffer statement is only executed on the current processor, if the corresponding processor tile coordinate is found.

Note also that another lookup of the destination processor tile coordinate is performed for assuring the case of non-local communication.

At the end of the global time step, the communication is performed, here indicated in the pseudo code by the function call `communicateData()`. Because a collective operation is used for the communication of data, this statement is not guarded by the `if` condition as in the case of the write-buffer statement. This allows the code to be executed without restriction to any real processor number and, thus, the collective operation is executed in parallel on all available processors, as required.

Because **CLooG** was not designed primarily for use with SPMD programs, the insertion of the guarding `if` statement mentioned above requires an additional post-processing of the generated loop nest. For this purpose, a simple scanner/parser is used that recognizes the parallel processor loops and inserts the `if` statement at the corresponding location in the target code.

3.6 Target language

In the previous sections, the principles of code generation for distributed memory architectures have been described, but so far only pseudo code has been used in the target program examples. This section introduces combinations of target languages and communication libraries that could possibly be used for code generation. The design choice to use the *MPI* communication library for the target language *C* will also be explained.

3.6.1 Languages using message passing semantics

The approach the is used for the required communication of data between processors in distributed memory architectures is called *message passing*. In this field, several programming languages, libraries or language extensions exist, each providing different abstraction levels and different concepts. For this project, the main options as target languages were:

HPF Based on *Fortran 90*, *High Performance Fortran* (*HPF*) extends the language with parallel constructs in order to allow parallel computing of arrays. The user can specify the distribution of data on processors (using several specialized compiler directives), using SIMD semantics for computing separate parts of this data in parallel on different processors. The compiler uses the annotations to compute the required communication between processors.

The main advantage of *HPF* from a user's point of view is the high abstraction level, because the basic sequential program code can be reused. However, the programmer has to specify, how the computed data is distributed onto the available processors. This is done by inserting additional compiler directives, whereas the difficult task of generating code for message passing is left to the compiler. However, in the context of automatic code generation, this abstraction level is problematic, because for more complex data distributions, available *HPF* compilers often fail to generate efficient communication code. These problems are dealt with by Faber [Fab97, FGL01], who adapts *HPF* to be used as a target language for code generation in the polytope model.

C+MPI The *message passing interface* (*MPI*) [Mes94, Qui04] is available as a library for several programming languages, but is mostly used in combination with *C* or *Fortran* programs. This is an advantage over *HPF*, because using *C* as base language, *C+MPI* allows to generate target code for most available platforms. Compared to *HPF*, the abstraction level of *C+MPI* is considerably lower, because the user has to deal explicitly with all communication between processors, although the concept of *collective operations* provides abstractions for certain specialized communication structures. On the other side, this lower abstraction level provides a more powerful method to deal with complex communications resulting from irregular dependency structures.

Java Although it is possible to use *Java* for parallel computing, its *remote method invocation* (*RMI*) approach is designed primarily for use in large distributed software systems (e.g. distributed data base systems), providing a high abstraction level compared to both *HPF* and *C+MPI*. As a target language in the context of high performance computing, however, it lacks the necessary amount of control over how the processors communicate with each other. There are attempts to extend the language for the use in high performance computing, resulting in several *Java* dialects [PL01]. Nevertheless, these languages (e.g. *JavaParty* [PZ97] or *High Performance Java* [YSP+98]) still require complex adjustments to the generated code in order to achieve a comparable efficiency as, e.g. *C+MPI*.

The experience with both Faber's results [Fab97, FGL01] and experiments with *Java Party*, together with some positive experiments with *C+MPI*, lead us to the decision to use *C+MPI* as target language for this implementation of automatic code generation in the polytope model.

3.6.2 C+MPI as target language

As target language for this implementation of automatic code generation, *C+MPI* provides a very powerful means for expressing different kinds of communication between processors. From the enormous number of available operations (129 in the MPI 1.1 standard!) there are two groups of communication operations that are of interest for the code generation in our project:

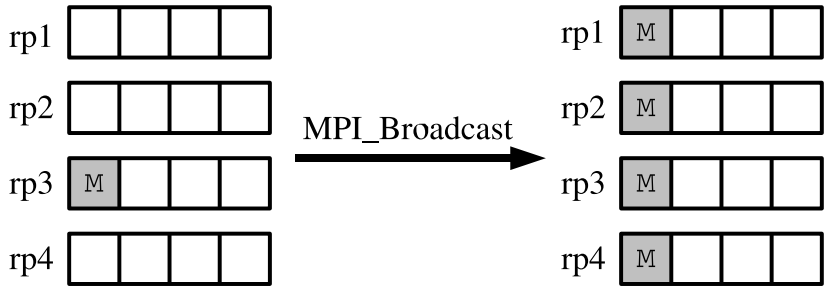

Figure 3.17: collective operations: processor $rp2$ performs broadcast

Point-to-point operations The basic functions for sending and receiving messages between single processors in *MPI* are the *MPI_Send* and *MPI_Recv* operations.

These point-to-point operations can be used in different forms, e.g. blocking or non-blocking, buffered or not buffered. Other functions allow the probing of messages for their availability or type. There are also extensions in the *MPI 2.0* standard [MPI96] to access remote memory directly using one-sided communication operations.

Collective operations In order to provide a higher abstraction level, collective operations are available for certain communication structures. These operations range from the often used broadcast of single data elements onto all other processors (*MPI_Broadcast*, described in Figure 3.17) to the most general collective operation *MPI_Alltoall*, illustrated in Figure 3.18. Here, each processor keeps a separate buffer for each communication partner at the receiver side and likewise uses separate buffers for each communication partner to receive data, when the actual exchange of data by all processors takes place. As the methods used for parallelizing in the polytope model often create quite complex communication structures, requiring many processors to communicate with each other, *MPI_Alltoall* provides a very convenient tool for implementation. Moreover, the *MPI* library can in theory use more specialized constructs for implementing the *MPI_Alltoall* operation, as it would be the case if a sequence of point-to-point communication operations were used instead. For the hardware used for testing this implementation, benchmarks have indeed shown a performance advantage of collective operations compared to point-to-point operations, when used for equal purposes in synchronous parallel target programs, as illustrated by Ellmenreich [Ell04]. There exist further generalizations of the basic all-to-all operation, differing in the kind of data that is transmitted. For sending single data elements only (e.g. single integer values), the basic *MPI_Alltoall* can be used, whereas the more general *MPI_Alltoallv* extends the transmission to arrays (*v*ectors) of (same-typed) data elements. The size of each of these arrays can be specified by an argument to the function call in the form of an array of integers.

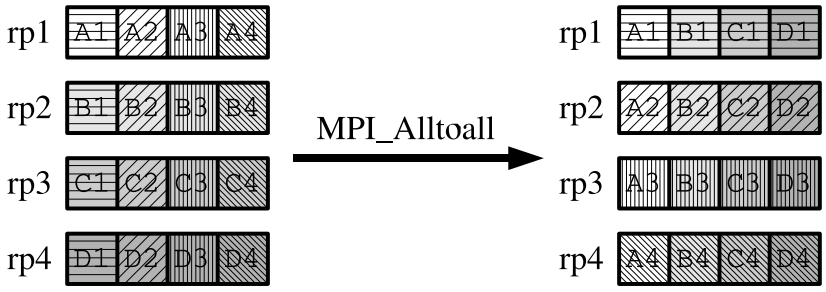

Figure 3.18: collective operations: all-to-all

In the implementation covered by this thesis, the array variant *MPI_Alltoallv* has been used for transmitting the separate communication buffers described in section 3.4.1. Because the receiving processors have to know the exact buffer-size of each buffer (for each communication partner) in which the transmitted values are stored, an additional basic *MPI_Alltoall* operation is used prior to the actual communication of the buffers, in order to propagate the information about buffer-sizes to each involved processor.

3.7 Soaking and draining

Using the methods described in the earlier sections of this chapter, the resulting target program consists of different types of statements that perform the actual computation of values and also the communication of updated data to destination processors, where the data is used during further computations.

However, in order to start computation, each processor implicitly requires the input data to be available in its local memory. Similarly, all data computed has to be gathered at the end of computation. Here, all processors that are owners of a most recent updated memory cell have to send the value to a single processor that holds all computed values.

This section briefly discusses the problem of generating the required code for performing this initial distribution of input data across processors (the so-called *soaking* of the processor array), as well as the *draining* of the final values after completed computation from the processor array.

3.7.1 Soaking

Different approaches can be used to implement the initial distribution of input data across all processors, that require this data for computation.

Additional soaking statement

In order to generate the necessary communication of data for *soaking*, additional statements can be inserted in the original input program. These soaking statements consist of assignments of all data cells to themselves. Assume the

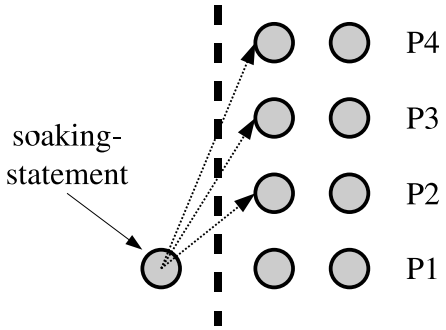

Figure 3.19: communication for additional soaking statement

following loop nest, belonging to an input program that uses only a single two-dimensional array A:

```
for i=1..n
  for j=1..m
       ... = A(i-1,j-1)  // computation operations
    A(i,j) = ...
  end
end
```

This program is augmented by an additional soaking statement S that consists of an assignment of all array cells in A onto themselves:

```
for i=1..n
  for j=1..m
    A(i,j) = A(i,j) // additional soaking statement S!
  end
end
for i=1..n
  for j=1..m
    ... = A(i-1,j-1)  // computation operations
    A(i,j) = ...
  end
end
```

This additional statement S is now placed explicitly onto the first processor P_1 (or onto any other processor that holds the initial data for A), thereby overriding the placement generated by the allocator and leading to dependences from P_1 to any other processor that uses data from A. Likewise, S is scheduled explicitly to be executed at the global time step right before the first time step for which any other computation is scheduled. Figure 3.19 illustrates the resulting communication scheme in a simplified index space. Note that no computation overhead results from this inserted statement, because it

is only used for generating the necessary communications and no actual assignment operations have to be executed during the computation in the generated target program. Also, using this dependence-driven approach, data is only computed when it is actually used in the later computations on the corresponding processor. Currently, this soaking statement is inserted manually before the start of the usual parallelization process, which is then completed automatically. Extending the implementation to insert the soaking statement fully automatically is possible, but for our project, we used the following alternative.

Distributed reading of input data

As an alternative to the insertion of an additional soaking statement, it is also possible to replicate all arrays on all real processors. This can be achieved by copying explicitly the required files on each target processor or, more elegantly, using distributed file systems (e.g. the *network file system*, *NFS*). Every processor then proceeds to read the required input data from its local file, before the start of any computation.

Using this approach, no modifications to the original input file have to be made, but the overall amount of communicated data can be considerably higher.

3.7.2 Draining

At the end of computation, all final values are typically scattered throughout the processor array, which complicates the task of collecting these values from their last location (the so called *draining*) and gather it at one destination processor. However, it is possible to use an approach that is very similar to the one used for soaking.

Here, an additional statement is again inserted, containing assignments from each array cell to itself. Whereas the soaking statement is inserted before the computation statements, the draining statement D is appended at the *end* of the original input program:

```
for i=1..n
  for j=1..m
    ... = A(i-1,j-1)  // computation operations
    A(i,j) = ...
  end
end
for i=1..n
  for j=1..m
    A(i,j) = A(i,j) // additional draining statement D!
  end
end
```

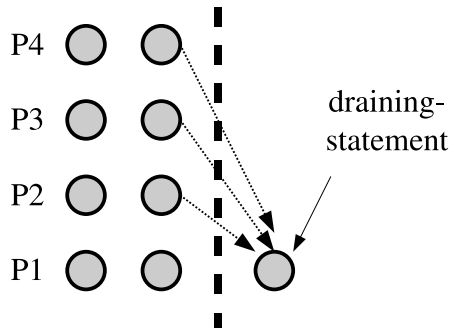

Figure 3.20: communication for additional draining statement

Again, a modified placement is used to place D at the designated destination processor (normally the first processor $P1$ again). Also, a modified schedule is used, in order for D to be executed at the end of the target program. The resulting communication scheme is displayed in simplified form in Figure 3.20. As it is the case for the soaking statement, this draining statement produces no computation overhead, because the assignment code can be omitted for code generation for the target program.

Chapter 4

Implementation

This chapter will describe in detail how the **CLooG** loop generator tool is used for generating the target code. It illustrates the layout of **CLooG**'s input file format and how the code for the different statements in the loop bodies of the target code is constructed.

4.1 Generating the CLooG input data

As mentioned in Section 3.4.2, different types of statements are used in the target code, beside the statements resulting from the input program, which perform the actual computation task. Instead of using single send and receive statements for point-to-point communication, this implementation uses buffers to aggregate the data in global time steps and communicates the content of these buffers at the end of each global time step using a single collective operation (in this case, a combination of `MPI_Alltoall` and `MPI_Alltoallv` is used). For this purpose, two different kinds of statements are used, which perform the aggregation of data at the sender side (write-buffer statements) and the unpacking of received data at the corresponding destination processor (unpack-buffer statements). Including the communication statement and the original computation statements, there are four different types of statements to be generated for the target program. This section describes how **CLooG** is used for generating the corresponding loop nests.

Therefore, a simple input program (illustrated in Figure 4.1) is used as an example for the generation of target code.

```
for i=1,m-1
  for j=1,i
    A(i,j)=A(i-1,j)+A(i,j-1)
  end
end
```

Figure 4.1: Example input program

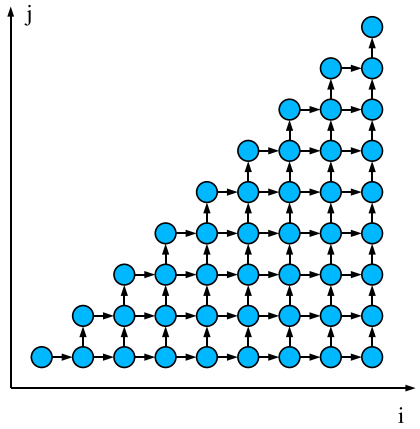

Figure 4.2: Original index space for the example input program

This code example contains a perfectly nested, two-dimensional loop nest, with a corresponding index space as displayed in Figure 4.2 (assuming the value 10 for parameter m). The corresponding dependences are also included in Figure 4.2, as obtained by the dependence analysis method.

4.1.1 CLooG input file format

The required input information for **CLooG** can be passed via ASCII data files (or directly by reading from standard input stream), using a defined structure for the input file to describe the loop nests for the statements in the target program.

This input file is structured into three parts:

Context information In the first part of the input file, the context for the target code can be specified, using linear inequality systems to restrict all occurring parameters to their corresponding domains (in our case, all parameters are always restricted to be positive). Also, the parameter names can be given optionally user-defined names and the target language can be set to either *C* or *Fortran* (although at the moment, only *C* is used for the implementation covered by this thesis).

Statement domain description The next part of the input file contains a description of the polytopes for all statements in the target programs in the form of linear inequality systems, that may also include equations. In order to describe non-convex index spaces, it is possible to use multiple polytopes for each statement. **CLooG** uses this description to enumerate all integral points in the union of all polytopes for each statement in lexicographical order but, so far, no ordering between the different statements is assumed. Also, the names of iterator variables for the corresponding loop nests can be specified optionally here.

Scatter functions The last section of the input file contains a description of so called *scatter functions*. These functions allow to specify additional

dimensions for the target code in order to specify further the execution order on all statements. To this end, equations in the dimensions from the domain description of each statement and the additional *scatter dimensions* can be defined by a linear system of equations, whereas the number of scatter dimensions has to be equal for all target statements.

Using this description, **CLooG** generates a loop nest, which consists of S scatter dimensions in the outermost loop dimensions, using the lexicographic order on all statements for enumerating the corresponding integral points, whereas the following D innermost dimensions result from the separate domain description mentioned above for each statement. Thus, the resulting loop nest consists of a total number of $S + D$ dimensions.

Furthermore, the additional scatter dimensions can optionally be named by the user.

In the course of this section, the domain descriptions and corresponding scatter functions will be described for all types of statements in the target program respectively.

4.1.2 Domain descriptions

The domain description in **CLooG**'s input file specifies the index space for each statement in the target program. Although a lexicographical order is implicitly assumed for enumerating each domain, scatter functions (see Section 4.1.3) can be used to rearrange the order in which each dimension is enumerated in the generated corresponding loop nest. Thus, the inequality systems describing the domain for each statement can in theory be represented in arbitrary order of their variables, if desired. The following domain descriptions are based on the communication scheme described in Section 3.4.

Each domain description can be expressed in **CLooG** in the form of a number of matrices, where each matrix M describes the inequality system

$$M \cdot \begin{pmatrix} \vec{v} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0},$$

in which M is multiplied by the index vector consisting of surrounding loop indices $\vec{v}$, structure parameters $\vec{p}$ and the constant 1. All entries in M have to be integral numbers. Thus, each row in M represents an inequality. By extending M by the columns for parameters and a column for the constant 1, affine linear inequalities can be expressed.

For each statement type, the corresponding domain description is composed of different inequality systems in the course of this section. These inequality systems are first illustrated seperatetly and later combined in the final matrix that defines the corresponding domain.

Computation statement description

For the domain description of each computation statement in the transformed target program, the matrix representation of the inequality system belonging to the corresponding original index space (ISPC) of the statement (M_{ispc}) is used as the basic inequality system:

$$M_{ispc} \cdot \begin{pmatrix} \vec{v}_{ispc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \tag{4.1}$$

In the case of our input program of Figure 4.1, which contains only one statement, the corresponding original index space is described by the following inequality system:

$$\begin{pmatrix} 0 & 1 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ m \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.2}$$

In order to transform this inequality system into the corresponding description for the inequality system in the transformed index space (TISPC), the space-time transformation (trafo) function is included in the form of a matrix M_{trafo}, that contains equations in coordinates in the original and the transformed index space (TISPC). The TISPC coordinates are given as a function of the ISPC coordinates:

$$M_{trafo} \cdot \begin{pmatrix} \vec{v}_{ispc} \\ \vec{p} \\ 1 \end{pmatrix} = \begin{pmatrix} \vec{v}_{tispc} \\ \vec{p} \\ 1 \end{pmatrix}. \tag{4.3}$$

In the case of our input program of Figure 4.1, the corresponding space-time mapping obtained by the applied parallelization methods leads to following equation system:

$$\begin{pmatrix} 1 & 1 & 0 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ m \\ 1 \end{pmatrix} = \begin{pmatrix} t \\ p \\ m \\ 1 \end{pmatrix}. \tag{4.4}$$

Here, the dimensions of the target index space are named t and p denoting time and processor dimensions respectively. This space-time mapping leads to a skewed target index space, as displayed in Figure 4.3.

Thus, the inequalities for enumerating coordinates in the transformed index space is implicitly defined by the combination of inequalities from the original index space (Inequality 4.1) and the equations from the space-time transformation function (Equation 4.3).

For some statements, the space-time mapping is only valid for a certain subset of the original index space. For this purpose, additional inequalities

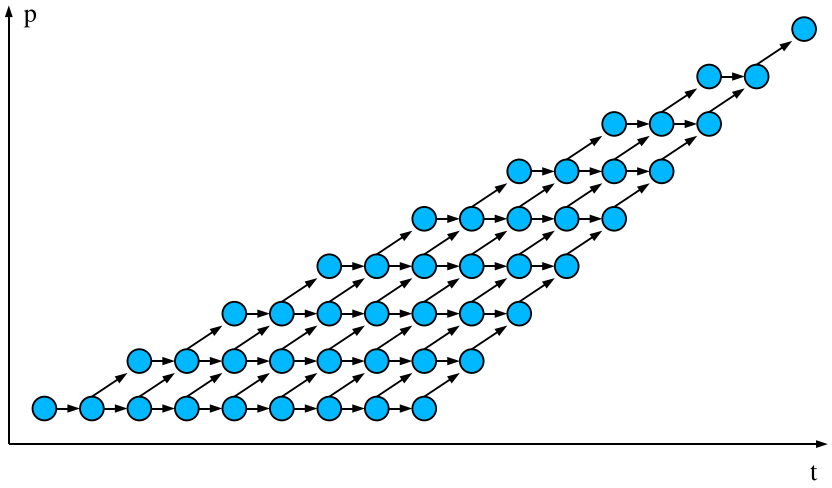

Figure 4.3: Target index space after space-time mapping

(trafo constraints, trc) are included in the form of the matrix M_{trc} to restrict the ISPC inequality system to the valid coordinates:

$$M_{trc} \cdot \begin{pmatrix} \vec{v}_{ispc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.5}$$

In the case of our example input program, the only restriction that is defined by the inequality system corresponding to matrix M_{trc}, is for parameter m:

$$m \geq 2. \tag{4.6}$$

For the implementation of the tiling technique, the corresponding inequalities for describing the relation between space-time coordinates in the transformed target space and their corresponding tile coordinates are included as the tiling matrix M_{tile}:

$$M_{tile} \cdot \begin{pmatrix} \vec{v}_{tile} \\ \vec{v}_{tispc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.7}$$

In our example, we use tiling for both dimensions of the transformed index space, thus, implementing time tiling as well as space tiling. We choose a rectangular tile shape and a tile size of 3 in both dimensions.

The resulting inequality system between tile coordinates and absolute co-

ordinates in the transformed index space looks as follows:

$$\left(\begin{array}{cc|cc|c|c} 0 & -3 & 0 & 1 & 0 & 0 \\ 0 & 3 & 0 & -1 & 0 & 2 \\ -3 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & -1 & 0 & 0 & 2 \end{array}\right) \cdot \left(\begin{array}{c} timeTile \\ spaceTile \\ \hline t \\ \hline p \\ \hline m \\ \hline 1 \end{array}\right) \geq \vec{0}. \tag{4.8}$$

These separate matrices are combined to one matrix that describes the domain of one computation statement in the target index space. Therefore, each matrix M is partitioned into three sub-matrices (V, P, C), containing entries for the corresponding description of variables, parameters and constants, respectively. The resulting matrix M_{Comp} for one computation statement is constructed as follows:

$$M_{Comp} := \left(\begin{array}{c|cc|c|c} V_{ispc} & \multicolumn{2}{c|}{0} & P_{ispc} & C_{ispc} \\ \hline V_{trafo} & 0 & -I & P_{trafo} & C_{trafo} \\ \hline -V_{trafo} & 0 & I & -P_{trafo} & -C_{trafo} \\ \hline 0 & \multicolumn{2}{c|}{V_{tile}} & P_{tile} & C_{tile} \\ \hline V_{trc} & \multicolumn{2}{c|}{0} & P_{trc} & C_{trc} \end{array}\right),$$

Here, the first line can be obtained from matrix M_{ispc}, defined in Inequality 4.1. The second and third line are used to represent Equation 4.3. Here, in the second line, the entries 0 and $-I$ in the second and third column represent a zero matrix and a negated unit matrix of the same dimension as the total number of tiled dimensions and the target index space, respectively. This inequality is negated for the third line, in order to define Equation 4.3.

In the fourth line, the tiling inequalities from Inequality 4.7 are used, where V_{tile} consists of dimensions for tile coordinates and transformed index space coordinates, between which the inequality is defined. The last line represents the inequalities from Inequality 4.5.

The resulting inequality system defined, by matrix M_{Comp}, looks as follows:

$$M_{Comp} \cdot \left(\begin{array}{c} \vec{v}_{ispc} \\ \vec{v}_{tile} \\ \vec{v}_{tispc} \\ \vec{p} \\ 1 \end{array}\right) \geq \vec{0}. \tag{4.9}$$

The dimensionality of the index space defined by Inequality 4.9 is $dim_{ispc} + dim_{tispc} + dim_{tile}$, where dim_{ispc} and dim_{tispc} are the dimensions of the corresponding index spaces and dim_{tile} is the total number of tiled dimensions.

In the case that tiling is not applied, the corresponding rows and columns of matrix M_{Comp} that are defined by matrix M_{tile} are omitted, leading to an index space of dimensionality $dim_{ispc} + dim_{tispc}$.

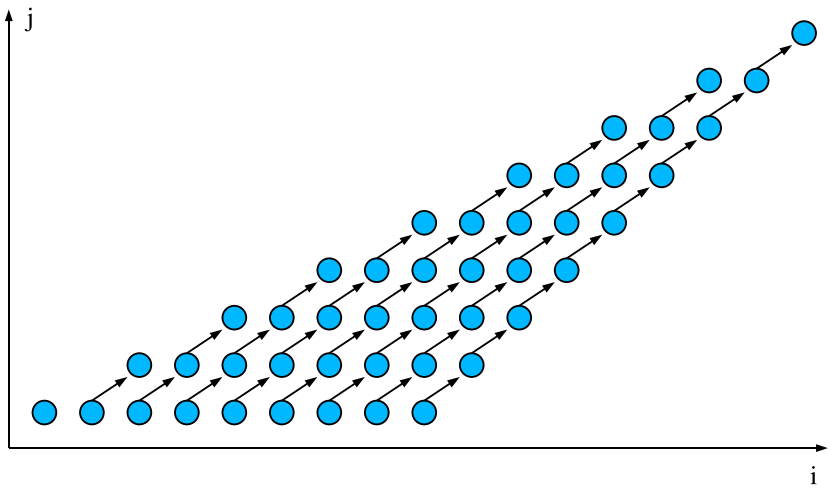

Figure 4.4: Communication-producing dependences

Buffer management statement description

As described in Section 3.4.2, two kinds of buffer management statements are generated for the target program. At the sender side, the *write-buffer* statement manages the aggregation of data in a buffer whereas, at the receiver side, the communicated data is read from another buffer and written to local memory by the *unpack-buffer* statement.

For the domain description of write-buffer and unpack-buffer statements, we can use the same inequality system for both statements. The reason is that the same relation between source and destination coordinates of a dependence is described. A distinction between buffer management at the sender and receiver side is only made in the scatter functions (see Section 4.1.3), were two different projection orders are applied that invert the enumeration of dimensions belonging to the source and destination of a dependence.

Each domain description is based on the information corresponding to one transformed dependence in the target program. However, as mentioned in Section 3.1, anti, input and output dependences are not used during this code generation step, because only true dependences can lead to communication between processors. Furthermore, it is possible that, for a true dependence, the corresponding source and destination are located on the same processor tile. In that case, no communication is required, because both processor tiles are always mapped to the same real processor and these dependences can be omitted for code generation as well. This leaves only one dependence in our input program from Figure 4.1 for code generation, as illustrated in Figure 4.4.

As briefly introduced in Section 3.4.2, the dependence polytope from Section 3.1.1 is used as a base for the domain description inequality system. For this purpose, polytope representations of the source ($M_{SrcISPC}$) and corre-

sponding destination index spaces ($M_{DestISPC}$) are used:

$$M_{SrcISPC} \cdot \begin{pmatrix} \vec{v}_{ispc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \tag{4.10}$$

and

$$M_{DestISPC} \cdot \begin{pmatrix} \vec{v}_{ispc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.11}$$

For our input program, the resulting inequality for the source index space of the relevant dependence is:

$$\begin{pmatrix} -1 & 0 & 1 & -1 \\ 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & -1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ m \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.12}$$

The corresponding index space for all coordinates that are destinations of the relevant dependence is defined by the following inequalities:

$$\begin{pmatrix} -1 & 0 & 1 & -1 \\ 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -2 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ m \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.13}$$

The inequalities for source and destination ISPC of a dependence are combined into one inequality system, along with an equation that defines the corresponding source coordinate for a given destination coordinate (the so-called *h-transformation* (hT) form of the dependence):

$$\left(\begin{array}{c|c} M_{SrcISPC} & 0 \\ \hline 0 & M_{DestISPC} \\ \hline -I & M_{hT} \\ \hline I & -M_{hT} \end{array}\right) \cdot \begin{pmatrix} \vec{v}_{src} \\ \vec{p} \\ 1 \\ \vec{v}_{dest} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.14}$$

In the case of our input program, M_{hT} defines following equation (given in our inequality system as two inverse inequalities) between variables at the source of a dependence ($\vec{v}_{dest}$) and corresponding destination variables ($\vec{v}_{dest}$):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \vec{v}_{dest} \\ m \\ 1 \end{pmatrix} = \begin{pmatrix} \vec{v}_{src} \\ m \\ 1 \end{pmatrix}. \tag{4.15}$$

This polytope is combined with separate M_{trafo} space-time transformation matrices from the corresponding source ($M_{SrcTrafo}$) and destination ($M_{DestTrafo}$) statements (cf. Equation 4.3), in order to obtain a representation of the dependence in the transformed index space, the *communication polytope*

described in Section 3.1.1. Also, additional restrictions to the space-time transformation are included (as described by Inequality 4.5).

If the user chooses to use tiling, the corresponding inequalities from Inequality system 4.7 are used for both transformed source and destination index spaces of the dependence. This results in following matrix $M_{BufferBase}$, with each matrix M, V, P and C representing the corresponding sub-matrices for variables, parameters and constants, respectively:

$$M_{BufferBase} := \left(\begin{array}{cc|cc|cc} V_{SrcISPC} & 0 & 0 & & P & C \\ \hline 0 & V_{DestISPC} & 0 & & P & C \\ \hline -I & M_{hT} & 0 & & P & C \\ \hline I & -M_{hT} & 0 & & P & C \\ \hline V_{SrcTrafo} & 0 & 0 & -I \; 0 & P & C \\ \hline 0 & V_{DestTrafo} & 0 & 0 \; -I & P & C \\ \hline 0 & & V_{SrcTile} & 0 & P & C \\ \hline 0 & & 0 & V_{DestTile} & P & C \\ \hline V_{SrcTrc} & 0 & 0 & & P & C \\ \hline 0 & V_{DestTrc} & 0 & & P & C \end{array}\right).$$

Again, $-I$ and $V_{SrcTile}$ ($V_{DestTile}$) represent the same sub-matrices as in matrix M_{Comp} for computation statements.

The corresponding inequality system provides more information than needed for the domain description of the buffer management statements. The communication structure described in Section 3.4 requires no information about the destination time coordinate of the corresponding dependency, because the unpacking of buffers is always performed on the global time step that follows the communication. Thus, the global time dimensions for the destination of a dependence can be projected away. In the case that time tiling is used, the corresponding time tile dimensions of the $V_{DestTile}$ matrix are projected away, whereas in the other case, where no time tiling is applied, the corresponding time dimensions of the destination TISPC are projected away.

The resulting inequality system for the destination side leaves only spacial coodinates that describe the destination processor. In the case that no space tiling is used, these spacial coordinates consist of transformed index space coordinates:

$$M_{BufferBase} := \left(\begin{array}{cc|cc|cc} V_{SrcISPC} & 0 & 0 & & P & C \\ \hline 0 & V_{DestISPC} & 0 & & P & C \\ \hline -I & M_{hT} & 0 & & P & C \\ \hline I & -M_{hT} & 0 & & P & C \\ \hline V_{SrcTrafo} & 0 & -I & 0 & P & C \\ \hline 0 & V_{DestTrafo} & 0 & -I_{proc} & P & C \\ \hline V_{SrcTrc} & 0 & 0 & & P & C \\ \hline 0 & V_{DestTrc} & 0 & & P & C \end{array}\right),$$

where $-I_{proc}$ is a negated unit matrix with the size of the number of processor dimensions in the target index space.

For the case that space tiling is used, logical transformed index space coordinates are only needed for the source inequality system variables of the dependence, because they can be reused at both sides of the communication for enumeration of data elements that are accessed in the same order. However, we can project away all logical space-time coordinates of the TISPC for the destination of the dependence, thus leaving only a description of processor tile coordinates for the destination processors of the corresponding dependence.

Also, the destination ISPC dimensions $M_{DestISPC}$, the corresponding space-time transformation $M_{DestTrafo}$ and restrictions from $M_{DestTrc}$ can be projected away, if no transformed destination coordinates are needed in the domain description.

The resulting matrix M_{Buffer} for the case that tiling is applied for time and space dimensions, looks as follows:

$$M_{Buffer} := \left(\begin{array}{|c|ccc|c|c|} \hline V_{SrcISPC} & \multicolumn{3}{c|}{0} & P & C \\ \hline V_{SrcTrafo} & 0 & -I & 0 & P & C \\ \hline 0 & V_{SrcTile} & & 0 & P & C \\ \hline \multicolumn{2}{|c}{0} & & V_{DestProcTile} & P & C \\ \hline V_{SrcTrc} & \multicolumn{3}{c|}{0} & P & C \\ \hline \end{array}\right),$$

defining the corresponding inequality system

$$M_{Buffer} \cdot \begin{pmatrix} \vec{v}_{srcIspc} \\ \vec{v}_{srcTile} \\ \vec{v}_{srcTispc} \\ \vec{v}_{destProc} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.16}$$

Here, the corresponding inequalities for the h-transformation equation are not shown explicitly, because they are implicitly given in the remaining relation between source ISPC coordinates and destination tile coordinates.

Note that the inequalities for ISPC coordinates are only needed for the case that non-unimodular space-time transformations are used. In this case, inequalities for the original index space are used in combination with the equations from the space-time transformation to obtain an exact definition of the transformed index space, that can be used by **CLooG**.

The dimensionality of the index space defined by Inequality system 4.16 is $dim_{ispc} + dim_{tile} + dim_{tispc} + dim_{destProc}$, where dim_{ispc} and dim_{tispc} are the number of dimensions of the corresponding index spaces, dim_{tile} is the total number of tiled dimensions (if tiling is used) and $dim_{destProc}$ is the number of processor tile dimensions (or logical processor dimensions, if tiling is not used).

Communication statement description

For the domain description of the communication statement, it is possible to reuse the corresponding domain descriptions for all computation statements.

The resulting domain description leads to the execution of a communication statement at each global time step that contains computation statements, as described in Section 3.4.2.

For that purpose, the inequality system from Inequality 4.9 is used for each corresponding computation statement. In order to obtain a description of all global time coordinates that contain computation operations, this inequality system is projected to the global time coordinate.

However, if time tiling is applied, the required *skewing* of the index space leads to complications in this simple approach. So far, skewing has been ignored for the domain description of computation and buffer management statements, because it can easily be implemented by the scatter dimensions described in Section 4.1.3. However, if we project away all processor dimensions for the domain description of the communication statement, the skewing function cannot be realized, because it requires the time tile dimension to be skewed by the sum of all entries in the corresponding processor tile dimensions, as described in Section 3.2.4.

In order to avoid this problem, skewing is implemented directly in the domain description by applying an affine function for the skewing transformation on the inequality system for each computation statement, *before* projecting onto the global time coordinates.

The affine skewing function uses only global time and processor tile coordinates (logical TISPC processor coordinates in the case that space tiling is not used). In the case that tiling of time and space dimensions is used, it is defined by the following matrix M_{Skew}:

$$M_{Skew} := \left(\begin{array}{c|c|c|c} I_{time} & M_{rp} & 0 & 0 \\ \hline 0 & I_{rp} & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right),$$

where I_{time} is an n-dimensional unit matrix and M_{rp} is an $n \times m$-matrix, with n and m being the number of time and processor dimensions respectively. Likewise, I_{rp} is a m-dimensional unit matrix. Therefore, M_{rp} is constructed as follows:

$$M_{rp} := \begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \\ -1 & \dots & -1 \\ 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}.$$

The non-zero entries in row t of M_{rp} indicate that the corresponding time dimension t is skewed by the sum of all processor dimension entries.

In the case of the input program of Figure 4.1, tiling of both space and time dimensions results in an one-dimensional global time coordinate and an one-dimensional processor tile coordinate. This leads to following skewing matrix:

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

The inequality system for which this skewing function is later used, is generated for each computation statement from Inequality 4.9 and projected to the global time ($timeTile$) and processor coordinate ($procTile$):

$$\left(\begin{array}{c|c|c|c} V_{timeTile} & 0 & P_{timeTile} & C_{timeTile} \\ \hline 0 & V_{procTile} & P_{procTile} & C_{procTile} \end{array}\right) \geq \vec{0}. \tag{4.17}$$

The final index space IS_{Comm} corresponding to the inequality system for the communication statement domain description is then derived by applying the skewing function f_{Skew} (defined by matrix M_{Skew}) on each index vector $\vec{i}$ that satisfies Inequality 4.17, thus obtaining the image of the index space defined by the same inequality under f_{Skew}.

Because the communication statement is implemented using collective *MPI* operations (`MPI_Alltoall, MPI_Alltoallv`), this statement is implicitly executed on all available real processors in parallel, thus the processor dimension is redundant for the domain description and can be safely projected away.

In the case of our input program of Figure 4.1, the domain description for IS_{Comm} is defined by following inequality system for the global time dimension $timeTile$:

$$\begin{pmatrix} 1 & 0 & 1 \\ -3 & 3 & -5 \end{pmatrix} \cdot \begin{pmatrix} timeTile \\ m \\ 1 \end{pmatrix} \geq \vec{0}. \tag{4.18}$$

4.1.3 Scatter functions

Using domain descriptions, the index spaces for each type of statement in the target program are specified separately. In order to guarantee that all statement types will intertwine correctly in the target program, additional schedule information is given by the means of **CLooG**'s *scatter functions*.

It is also necessary to reorder dimensions from the domain description, in order to specify a correct enumeration order of the loop nests for each statement type. Furthermore, scatter functions can be used to implement affine transformations on the corresponding index space defined by each domain description, in order to realize the skewing transformation for time tiling and to implement array access functions for the buffer management statements.

Each scatter function defines an equation $M \cdot \vec{d} = \vec{s}$, where $\vec{d}$ is the index vector for all dimensions from the domain description of the corresponding statement and $\vec{s}$ is a vector of additional index variables that define loop dimensions outside the loop nest that is defined by the domain description. These additional dimensions are called *scatter dimensions* in **CLooG**.

If scatter functions are used in **CLooG**, all statements have to use the same number of scatter dimensions, because they define a common order on all statements.

For this project, the communication structure from Section 3.4 requires the following order in which statements are executed relative to each other:

1. For each logical time step within a global time step, the dependence structure can lead to at most three different types of statements being executed in the following order:
   (a) *Unpack-buffer* statement (reads the received value from the receive-buffer)
   (b) *Compute* statement (computes the new value)
   (c) *Write-buffer* statement (writes the new value to the send-buffer)
2. At the end of each global time step, the *communication* statement is executed on each processor.

This execution order can be implemented by the use of two additional scatter dimensions, that are inserted after the scatter dimensions that represent the global time and logical time dimension, respectively. The following corresponding scatter dimensions for the input program from Figure 4.1 that are given to **CLooG**:

`glT` describing the global time dimension.

`bl` an additional scatter dimension `bl` for the distinction of the "block" of computation and buffer management statements from the single communication statement.

`rp` describing the processor tile dimension.

`vT` describing the logical time dimension.

`stmtType` scheduling the three types of compute and buffer management statements in the correct order (*(a)* ... *(b)*), as mentioned above.

`local1, local1, local1` remaining scatter dimensions that are used differently by each statement type, but are necessary for the scheduling of the separate domain descriptions.

The resulting order of scatter dimensions is:

```
glT bl rp vT stmtType otherP local1 local2 local3
```

The corresponding scatter functions for each statement type are used as follows.

Computation statements

For each computation statement, the `bl` dimension is always set to 0, thus scheduling it in the first "block" of statements, and before the communication statement.

The global time dimensions (`glT` in our example) are given by the time tile dimensions from the domain description, but in order to implement skewing, the corresponding processor tile dimensions are added to the global time as well, which leads to following simple equation in the case of our input program: $glT = timeTile + procTile$.

The next scatter dimensions (`rp` and `vT`) are given by a direct equation from the corresponding dimensions for processor tiles and logical time in the domain description.

In order to schedule the computation statement between the other buffer management statement types, the `stmtType` dimension is set to a value of 1.

The last relevant dimension of the domain description, the logical processor dimension, is given next by the `otherP` dimensions. Finally, the remaining scatter dimensions are only needed to achieve a common number of scatter dimensions for all scatter functions and are thus set to 0.

In the case that no tiling is used, the glT and rp dimensions are given directly by the logical time and logical processor dimensions from the domain description.

Buffer management statements

Because the same domain description is used for both types of buffer management statements, it is necessary to implement the differences in the ordering of loop indices in the corresponding scatter functions for unpack-buffer and write-buffer statements.

For the global time dimensions (`glT`), both statements use the time tile coordinates of the source of the corresponding dependence, skewed by the processor tile coordinate, also of the source of the dependence. For the unpack-buffer statements, an additional constant value of 1 is added to the global time coordinate, thus scheduling the unpacking for the next global time step that follows the write-buffer statement of the corresponding dependence.

Again, `bl` is set to 0, in order to schedule both buffer management statements before the communication statement.

In order to realize the different processor locations on which the unpack-buffer and write-buffer are executed, the `rp` dimensions are set to the processor tile dimensions of the corresponding processor tile dimensions from the domain description. Here, unpack-buffer statements are placed on the destination processor tile coordinates, whereas write-buffer statements are placed on the source processor tile coordinate. The corresponding logical processor equivalents are used instead of tile coordinates, if no space tiling is applied.

For the next logical time dimensions (`vT`), both buffer management statements use the corresponding logical time coordinate of the source of the dependence, in order to guarantee a correct order of the unpacking and writing

of buffer elements.

The `stmtType` is used to schedule unpack-buffer statements *before* computation statements, by using the constant value of 0, whereas write-buffer statements are scheduled *after* computation statements, by using a greater value of 2.

In the next dimension (`otherP`), the corresponding communication partner processor is described, which is the source processor tile coordinate for the unpack-buffer statement and the destination processor tile coordinate for the write-buffer statement, respectively. Again, if no space tiling is used, the corresponding logical processor coordinates from the domain description are used.

The next dimensions are used for defining the scatter dimensions for accesses to array elements in the target program. These are realized by an affine function that takes transformed, logical space-time coordinates from the domain description and returns the coordinates for array accesses. Thus, the logical coordinates at the source side of the dependence can be taken for both types of buffer management statements, because the the dependence implies a common access to the same array index for both involved communication partners. In the example of the input program from Figure 4.1, the identity function is used as array access function: $(local1, local2) = \overrightarrow{srcTispc}$.

Finally, all occurring arrays in the target program are enumerated by the last scatter dimension in order to again maintain a correct order, in which the buffer elements are accessed.

Communication statements

For the communication statement, the only relevant dimensions are the `glT` dimensions and the `bl` dimension, because the statement is only executed once for each global time step on each processor. Thus, no further dimensions for logical space-time or processor tile coordinates needs to be specified.

The global time coordinate is again specified via the corresponding time tile coordinate from the domain description, whereas the `bl` dimension is always set to 1, in order to schedule the communication statement *after* the block of computation and buffer management statements, at the end of each global time step.

4.2 Post-processing the generated target program

The loop nest that is generated by running **CLooG** with the constructed domain descriptions and scatter functions consists of loop bodies that contain placeholders for each statement of the domain description in the input file. Each placeholder represents a statement number, for which the corresponding code has to be inserted.

For each computation statement, the original statement code from the input program is inserted without any changes.

For each write-buffer statement, code is inserted that performs a lookup in the generated processor map in order to prevent data elements to be communicated unnecessarily, if the source and destination processor tile number are both mapped to the same real processor. Otherwise, the required data is written into the local buffer for the corresponding target processor.

The corresponding code for each unpack-buffer statement also checks the processor map to assure that the corresponding source processor tile coordinate is mapped to a remote real processor. In case that non-local data is required, the next buffer element is read from the communication buffer and written to the corresponding local memory cell.

In the case that no space tiling is used, this lookup operation is performed for each logical processor destination coordinate, instead of each processor tile coordinate, which results in a larger overhead for time spent in managing the buffers.

Finally, the code for the communication statement is inserted. Here, the communication buffer is constructed by merging the separate buffers for each destination processor. In this way, the size of the resulting communication buffer is determined, which is then communicated among all processors by a collective `MPI_Alltoall` operation.

Note that this merging operation can be costly, when no space tiling is used, because in that case, the number of seperate buffers (for each logical processor coordinate) is significantly larger than in the case that only one seperate buffer for each processor tile is used.

Each processor then readjusts the size for the receive buffer to the required size. Finally, the collective `MPI_Alltoallv` operation is called in order to communicate the actual data and each processor receives the corresponding data elements in its receive buffer.

As mentioned in Section 3.5.4, a separate post-processing step is also necessary to insert an additional `if` statement that restricts the execution of each iteration of the processor tile loop nest to the processor with the corresponding real processor number.

Chapter 5

Conclusions

5.1 Future work

This section briefly introduces some extensions to the approach that is described in this thesis.

5.1.1 Fine-tuning

Although this implementation of code generation works automatically, there are some optional settings that can be fine-tuned manually by the user, mostly concerning various aspects of *tiling*:

- Tiling can be adjusted to be used for time or space dimensions.
- It is also possible to switch off tiling completely, generating only logical space-time dimensions, as obtained from the scheduler/allocator.
- It is possible to choose different tile shapes in order to reduce communication cost between processor tiles. These tile shapes can also be generated automatically, using a heuristics to minimize communication overhead.
- Choosing an arbitrary tile size (in combination with the cyclic processor mapping) allows to fine-tune the distribution of logical processors onto physical processors, using various forms of block, cyclic or even block-cyclic distributions.

In order to find a reasonable configuration, it will be necessary to compare the execution time of several generated target programs for a corresponding input program. However, this process tends to be extremely time-consuming. Most of the tools involved for the parallelization and code generation use mathematical methods (e.g. Fourier-Motzkin elimination) that are very complex (e.g. double-exponential complexity: $O(2^{2^n})$).

A solution to this optimization problem could be to develop a cost model for the desired target architecture and use mathematical tools for analyzing the expected run time efficiency of the target program. This way, some of

the more time-consuming processes in the later code generation stages could be avoided by rather analyzing the polytope descriptions of the parallelized program instead.

As an example, methods for counting integral points within polytopes could be used, as described by Rabl [Rab], in order to evaluate the communication overhead, when applied to the polytope representations of buffer management or communication statements.

5.1.2 Implementing hierarchical parallelism

The implementation realized for this thesis automatically generates parallel target code for *distributed memory architectures*, using *C+MPI* as target language. The resulting target program consists of a number of parallel processes, running on several processors, usually connected by a high-speed network.

Although this architecture is very common in the field of high performance computing, today many (if not most) modern CPUs also feature multiple processor cores, integrated in one physical unit (e.g. *dual-core* architecture), using a shared memory model for accessing memory. These architectures provide different levels of parallelism, forming a hierarchical structure. Normally, this hierarchy includes local, fine-grained parallelism (in the form of parallel threads on shared memory multiprocessor CPUs) and coarse-grained parallelism between distributed memory cluster nodes. Other levels of parallelism are possible, e.g. processors using Internet connections for communication (the *GRID* [TL03]).

As indicated in Section 2.3, target program based on a combination of synchronous and asynchronous parallelism could be used for this hierarchical target architecture, e.g.:

```
for t1=1..n
  parfor p1=1..m
    for t2=1..i
      parfor p2=1..n
        ...
      end
    end
  end
end
```

Here, the first spatial dimension $p1$ could be implemented by communicating distributed memory processors, whereas each of these processors uses local shared memory parallelism for executing the $p2$ dimension on multiple, parallel running threads (cf. the approach described by Quinn [Qui04]). This way, $t2$ and $p2$ can be viewed as a local, synchronous, parallel shared memory program.

For this purpose, the implementation covered by this thesis could be extended to generate parallel target code that uses a combination of communicating *MPI* processes and shared memory parallelism using *OpenMP*. Space

tiling can be used to implement these two levels of parallelism, whereas processor tiles are executed by parallel *MPI* processes, with each of these processes executing a number of parallel threads for each logical processor coordinate within its corresponding processor tile. Thus, the logical processor loop is executed as a number of separate threads, using the *OpenMP* compiler directive (`#omp parallel for`) to distribute the iterations across all available threads, as illustrated figuratively in this code example:

```
for globalTime=..
  for procTile=..
    if (lookup(procTile) == rank)
      // iterations executed as MPI processes
      ...
      for logicalTime=..
        #omp parallel for
        for logicalProc=..
          // iterations executed as OpenMP threads
          ...
        end
      end
    end if
  end
end
```

The buffer management from Section 3.4.2 can be used with one modification: in order to keep the buffer elements ordered, the number of separate buffers is extended from one buffer per processor tile to one buffer per logical processor coordinate. This could theoretically introduce some additional overhead, when the separate buffers are combined into one contiguous communication buffer (as used by the *MPI_Alltoallv* command). Because the number of logical processor coordinates usually exceeds the number of processor tiles significantly, this combination process can be time-consuming. However, the overall amount of data that is copied into the buffer remains the same, so the actual amount of additional overhead for using more separate buffers can only be demonstrated by execution time results.

Remark. *Note that this approach could theoretically also be used to generate pure shared memory target programs as a byproduct, when no space tiling is used and all logical processor coordinates are executed using* OpenMP *threads.*

5.1.3 An alternative approach to processor mapping

As described in Section 3.5, this implementation uses a map created at run time, that can be used to lookup the real processor number of each processor tile coordinate used in the target program. In theory, this run time mapping should not be required, because space tiling already aggregates several logical processors into processor tiles at compile time, which in theory should be

constructed choosing an appropriate tile size (at compile time!) to adapt the number of resulting processor tiles to the number of available real processors.

However, because the number of real processors is only known as a run time parameter, this approach leads to mathematical problems, because the resulting system of inequalities and equalities includes a multiplication of a parameter (determining the tile size) with a variable (for the processor tile coordinate). This occurrence of so called non-linear parameters can be avoided by using only constants for tile size and tile shape.

Great efforts have been made to implement mathematical tools [LW93, Grö03] that can solve non-linearly parameterized inequality systems. These tools use real quantifier elimination methods [Wei88] to extend existing methods and have been used successfully in the **LooPo** project for different tasks. For example, Fourier-Motzkin elimination and tiling can now be applied to systems with non-linear parameters.

For the project covered by this thesis, however, the already high run time complexity of the code generation tool lead to the decision to restrict to inequality systems with linear parameters only, because the more powerful non-linear tools may introduce additional overhead for the generation of the input data that is used by **CLooG**.

Also, the processing of the input data by **CLooG** is already very time consuming in some of the larger examples and is expected to be even worse in the non-linear case. Furthermore, **CLooG** would have to be extended to handle the occuring case differentiations.

5.1.4 Adaptive Tiling

Even the extended tiling approach, briefly discussed in Section 5.1.3, still can yield suboptimal load balance between involved real processors. Consider Figure 5.1, which illustrates the result of a space tiling, that is performed at compile time (statically). In this example, poor processor utilization results, because tiling considers only the total number of logical processors, viewing the index space as a whole.

Obviously, a better mapping strategy for the logical processors in this index space can be found theoretically, e.g. illustrated in Figure 5.2. However, so far this adaptive tiling approach, as described by Seidel [Sei04], is implemented only for simple examples, using manual methods for finding optimal tile sizes and determining the intervals where a re-adaptation of space tiling is reasonable. Also, integrating time tiling into this approach is also a quite complex task.

5.2 Related work

There are other projects that also address automatic code generation for distributed memory architectures. As mentioned briefly in Section 3.6.1, Faber [Fab97, FGL01] describes an approach that is also based on the polytope model, by which *HPF* target code is generated that includes specifications

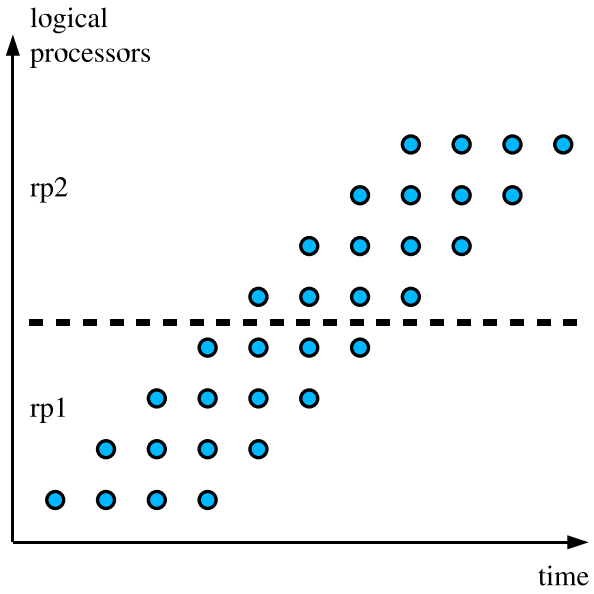

Figure 5.1: load balance problem with static tiling

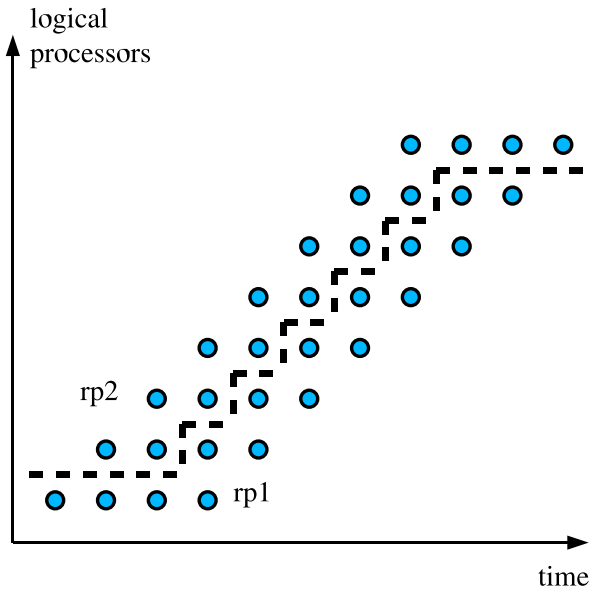

Figure 5.2: better load balance using adaptive tiling

of how data is distributed among processors. The *HPF* compiler then generates the required communication from this distribution annotations. However, common *HPF* compilers often fail to generate the corresponding communication code for cases that require very irregular communications.

Ferner [Fer] also describes a method of automatically deriving communication code for distributed memory architectures. This method is based on a parallelization technique of Lim and Lam [LL97] that generates so-called *partitions* of the index space from an input program. These partitions can be executed in parallel on logical processors. Ferner proposes a mapping algorithm that uses affine mappings from logical processors (partitions) to real processors and illustrates how the corresponding communication code loop nests can be obtained.

However, as opposed to the original parallelization technique described by Lim and Lam, he restricts the partitions to be one-dimensional, which limits the degree of parallelism obtained in the target program, but simplifies the code generation.

He also restricts the target programs to asynchronous parallel programs, whereas Lim and Lam's method often results in synchronous parallel target programs that feature an outer sequential loop for time steps.

In the approach described, the target program uses separate loop nests for sending, computing and receiving data, where for each partition, all required data has to be received first, then all computations of the entire partition are executed, followed by the sending of updated data to the depending partitions. The execution of overlapped polytopes for sending and receiving of data is not possible during the execution of computation operations within a partition (no *overlapping* of communication with computation). This leads to target programs that are sequential even in relatively simple examples. It also prevents the type of target programs that use pipelined parallelism as described by Lim and Lam.

5.3 Summary

The code generation method described in this thesis uses **CLooG** to generate automatically efficient target code for programs that are analyzed and transformed in the polytope model, with the goal of efficient parallel target programs for distributed memory architectures.

As described in Section 3.2, the tiling method can be used to aggregate operations into larger chunks in order to achieve a coarser grained parallelism in the target program. The described communication structure uses time tiling to obtain a block structure of global time steps, during which data is aggregated in communication buffers and only communicated at the end of each global time step, thus implementing message vectorization as opposed to the communication of data at each logical time step. Because the number of global time steps can be adjusted to be significantly lower than the number of logical time steps, the overhead from startup costs can be reduced.

This communication structure makes it necessary to restrict the space-time

transformation that is used for the target program, as described in Section 3.3 and Section 3.4.3.

The implementation uses collective operations in the synchronous parallel target program for synchronization and communication of the buffered data. A combination of *C* and *MPI* is used as target language, especially the collective operations `MPI_Alltoall` and `MPI_Alltoallv`.

For the generation of the target loop nest of computation and communication statements, a polytope model representation of the input program is used, in order to construct the domain descriptions required by **CLooG**, where scatter dimensions are used to implement additional scheduling and transformation tasks.

This approach has been successful for the automatical generation of target programs for small input programs, which can be executed correctly. However, when using larger sized input programs, **CLooG** fails to produce the corresponding target code, because of too much consumed memory, although newer versions of **CLooG**, which are still in development, may solve that problem.

Currently, the scanning algorithm used in **CLooG** leads to integer overflows for large tile sizes, which restricts the use of space tiling in the tested example programs. This restriction leads to a large overhead for buffer management, because, without space tiling, the destination processor (without later mapping to real processors) is given as logical processor coordinates, instead of processor tile coordinates.

Because space tiling is not used to aggregate the large number of logical processor coordinates into fewer, larger processor tile coordinates, the larger number of logical processor coordinates leads to two problems:

- In the statements for writing and unpacking of buffers, a check is performed for each destination processor coordinate, whether the corresponding real processor number is equal to the local real processor number, in order to prevent unnecessary copying of data to and from buffers.

  Normally, this check is only performed for each processor tile number, but without space tiling, every single logical processor coordinate is tested, thus increasing the computation overhead, because of the far larger number of logical processor coordinates.

  Although this check can be omitted, this would result in copying of every data element for every logical processor coordinate that leads to a memory access, regardless, whether the corresponding data element is locally available or not.

- In the communication statement, all seperate buffers are combined in one contiguous communication buffer that is used in the collective send operation. This merging of buffers also introduces a large overhead, if the number of seperate buffers is increased to the number of logical processor coordinates.

In order to deal with these efficiency problems, Cédric Bastoul has developed a testing version of **CLooG** that uses multiple precision integers to avoid

the integer overflows caused by large space tile sizes. However, correctness and complexity problems with this version of **CLooG** (when used for the comparatively large input files generated during code generation) prevented the use for the examples that were tested for this thesis.

At the moment, great efforts are made by the developer of **CLooG** to improve its run time and memory efficiency and to avoid correctness problems when using multiple precision integer computations.

Bibliography

[Ban92] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1992.

[Bas] Cédric Bastoul. CLooG - a loop generator for scanning Z-polyhedra.

[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, Ljubjana, October 2003.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multidimensional scheduling, 1994.

[Ell04] Nils Ellmenreich. *PolyAPM: Comparative Parallel Programming with Abstract Parallel Machines*. PhD thesis, FMI, 2004.

[Fab97] Peter Faber. Transformation von shared-memory-programmen zu distributed-memory-programmen, 1997. Diploma thesis.

[Fea92] Paul Feautrier. Some efficient solution to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, 1992.

[Fer] Clayton Ferner. Revisiting communication code generation algorithms for message-passing systems. *JPEDS*. Submitted.

[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)*, Lecture Notes in Computer Science 2017, pages 359–362. Springer-Verlag, 2001.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proc. IEEE*, 54(12):1901–1909, December 1966.

[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*, Lecture Notes in Computer Science 2481. Springer-Verlag, 2002. To Appear.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[Grö03] Armin Größlinger. Extending the polyhedron model to inequality systems with non-linear parameters using quantifier elimination, 2003. Diploma thesis.

[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.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, February 1974.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[LL97] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, 1997.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying Linear Quantifier Elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[Mes94] MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[MPI96] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[Ope] OpenMP - open specifications for multi processing. http://www.openmp.org/drupal/.

[PL01] Cherri M. Pancake and Christian Lengauer. High-performance java - introduction. *Commun. ACM*, 44(10):98–101, 2001.

[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.

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[Qui04] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[Rab] Tilman Rabl. Volume calculation and estimation of parameterized integer polytopes. Diploma thesis. To Appear.

[Sei04] Georg Seidel. Methods for adaptive tiling in the polyhedron model, 2004. Diploma thesis.

[TL03] Douglas Thain and Miron Livny. Building reliable clients and servers. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[Wei88] Volker Weispfenning. The Complexity of Linear Problems in Fields. *Journal of Symbolic Computation*, 5(1&2):3–27, February–April 1988.

[Wet95] Sabine Wetzel. Automatic code generation in the polytope model, November 1995. Diploma thesis.

[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.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind und dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 30. September 2005

Michael Claßen