Polyhedral Loop Parallelization: The Fine Grain

Peter Faber
Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany

Martin Griebl
Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany

Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany

Abstract

A safe basis for automatic loop parallelization is the polyhedron model which represents the iteration domain of a loop nest as a polyhedron in $\mathbb{Z}^n$. However, turning the parallel loop program in the model to efficient code meets with several obstacles, due to which performance may deteriorate seriously – especially on distributed memory architectures. We introduce a fine-grained model of the computation performed and show how this model can be applied to create efficient code.

1 Introduction

In contrast to traditional code analysis and transformations that view loops as unpredictable control structures, the polyhedron model considers different values of loop counters (or indices) in different loop iterations as a set with loop bounds as restrictions in the corresponding dimensions of $\mathbb{Z}^n$ [Fea92, Len93].

In this work, we describe a fine-grained model of computation that employs the polyhedron model in order to create a simple description of the calculation performed by a loop program and show how this fine-grained model can be used to create efficient code. Section 2 describes the basic model, Section 3 shows how to obtain a minimized representation of the computation performed, and Section 4 describes how to determine suitable transformations.

2 The Basics

In the polyhedron model, a loop nest is modelled as a multi-dimensional space: each loop spans one dimension and, for technical reasons, also every loop independent parameter, i.e., every symbolic constant in the program fragment considered, spans one dimension. In addition, we always add two artificial, program independent parameters:

- n_1 represents the constant 1; treating 1 as a parameter enables us to use so-called homogenous coordinates in order to represent affine expressions as linear expressions in a vector space that has an additional dimension for this parameter.
- n_∞ must be allowed to be arbitrarily large; especially, it is larger than any value of a loop counter in the program fragment.

Thus, the iteration domain – or index space of a loop nest – is expressed as a polyhedron in $\mathbb{Z}^{n+m}$, where n is the number of loops, and m is the number of all (program specific or program independent) parameters.

As usual, the dimensions of $\mathbb{Z}^{n+m}$ are enumerated as follows:

- The first n dimensions of $\mathbb{Z}^{n+m}$ correspond to the loop counters in the textual order in which the loops appear in the program text.
- The next $m-2$ dimensions correspond to the program specific parameters.
- The last two dimensions correspond to n_∞ and n_1, in this order.

The usual property in the polyhedron model is that the order in which points in a polyhedron are to be enumerated (according to the modelled code fragment) is the lexicographic order on $\mathbb{Z}^k$ (for a k-dimensional polyhedron). This is guaranteed by associating the i-th loop in the program text with the i-th dimension of the polyhedron.

In order to model the flow of data, and thus the computation performed, dependence analysis algorithms consider memory accesses during program execution. The basic constructs are the `read_A` and `write_A` operations – read and write accesses to an array `A`. As mentioned above, these accesses follow the form

$$\texttt{A}(f(i, j, n, m, n_\infty, n_1))$$

in the program text, where f is a linear function in the indices (loop counter variables) i, j, the (loop independent, but program specific) parameters n, m, and in n_∞ and n_1.

For an instancewise approach, we distinguish between different run-time instances, i.e., different executions of the same access for different values of surrounding loop indices.

2.1 Going Fine Grain

In this work, we aim at constructing a code fragment that can be executed as efficiently as possible on some given system. For this purpose, we eliminate as many redundant calculations as possible and then try to produce efficient code from the description of the remaining calculations to be performed. In order to obtain an accurate description of the calculations of the code fragment, we need a precise model of the program execution; i.e., our goal is to model the complete execution of a code fragment by polyhedra and dependence relations between those polyhedra. In our refined model, we want to maintain the property that the sequential program execution order corresponds to the enumeration of the polyhedron in lexicographic order.

The structures we use for this purpose are generalizations of the conventional polyhedron model. Linear expressions are considered atomic, i.e., they are computable at essentially no cost, and represent a well defined value that only depends on the place in the polyhedron where they are evaluated. We work on expressions, which are compositions of operators and function calls. Thus, we generalize the concept of read and write accesses to the execution of operators on a given system.

Definition 2.1 *Let $\mathcal{F}$ be a set of function symbols. In addition to procedure calls and user defined function names, the following symbols are elements of $\mathcal{F}$:*

`+,-,*,/,max,min,...`	*intrinsic functions.*
`LaExpr`	*A linear expression – considered atomic.*
`read_A`	*Read access to an element of array* `A`.
`write_A`	*Write access to an element of array* `A`.
`=`	*Assignment operator.*

Each function symbol op $\in \mathcal{F}$ represents an operator that can be executed by the given system. The function symbols are associated with an input arity by a function $\mathrm{a_i} : \mathcal{F} \to \mathbb{N}$ and an output arity by a function $\mathrm{a_o} : \mathcal{F} \to \mathbb{N}$, indicating the number of input, and output arguments, respectively. Thus the arity of a function symbol op is $\mathrm{a_i}(op) + \mathrm{a_o}(op)$. Without loss of generality, we assume all input arguments of op to be the first $\mathrm{a_i}(op)$ arguments and the output arguments to be arguments number $\mathrm{a_i}(op) + 1$ to $\mathrm{a_i}(op) + \mathrm{a_o}(op)$.

Next, we need to identify each textual occurrence of each function symbol in the program fragment to be considered. We use integer numbers for the identification, and we want these numbers to correspond to the sequential execution order of a single loop iteration in the program fragment.

Definition 2.2 (Occurrence) *With the operator symbols $\mathcal{F}$ defined in Definition 2.1 and the special operator symbol* **;** *as a sequence operator for statements, a code fragment without its control statements can be viewed as a term on $\mathcal{F} \cup \{\texttt{;}\}$. Each point s in the program text corresponds to a (sub)term and is assigned a unique integer number $\mathrm{OccNr}(s)$, its occurrence, such that:*

- *If $s \equiv \texttt{;}(i_1, i_2)$, then $\mathrm{OccNr}(i_1) < \mathrm{OccNr}(i_2)$, i.e., OccNr respects the order given by the sequential composition operator.*
- *If $s \equiv \mathtt{op}(i_1, \ldots, i_j, o_1, \ldots, o_k)$ with $\mathtt{op} \in \mathcal{F}$, input arguments i_m ($m \in \{1, \ldots, j\}$), and output arguments o_n ($n \in \{1, \ldots, k\}$), the occurrences are ordered in the following way:*
  - *$\mathrm{OccNr}(i_m) < \mathrm{OccNr}(s) < \mathrm{OccNr}(o_n)$, i.e., OccNr respects that an operator is executed only after its input arguments are evaluated, and before its output arguments can be written.*
  - *All o_n have the form $o_n \equiv \mathtt{write\_A}(o_{n,1}, \ldots, o_{n,l})$; the $o_{n,q}$ are enumerated such that: $\mathrm{OccNr}(o_{n,q}) < \mathrm{OccNr}(s)$, i.e., all memory addresses for the output variables are computed before the operator is executed.*

In order to re-establish the calculation to be performed, we define $\Gamma()$ *to be the inverse mapping of* $\mathrm{OccNr}()$.

For situations in which we are not interested in the subterms (arguments) of a point in the program, we define a mapping $\Xi : \mathbb{N} \rightarrow \mathcal{F}$ *that maps each occurrence to the corresponding function symbol in the program text, such that* $(\forall i : i \in \mathbb{N} : \ (\Gamma(i) \equiv op(e_1, \ldots, e_n)) \Rightarrow \ \Xi(i) \equiv op)$.

For the determination of suitable numbers for the occurrences, we use a syntax tree, in which control structures govern their bodies as well as the arguments that control the execution, while function and subroutine calls govern their arguments [ASU86]. Traversing this tree following an appropriate pattern, we can assign an increasing number as occurrence to every node so that the execution order of a single loop iteration is defined by the order on the occurrences.

Note that, for modelling the execution of a point in the program text itself, we still have to differentiate between different argument positions, since each argument position of an output argument represents another "currently calculated value". For this purpose, we introduce one more integer number:

Definition 2.3 *With each occurrence o, we associate the set of operand numbers*

$$\Psi(o) = \{-\,\mathrm{a_i}(\Xi(o)), \ldots, \mathrm{a_o}(\Xi(o)) - 1\}$$

The execution of a single occurrence corresponds to the enumeration of its operand numbers in increasing order:

1. *A negative operand number $-j$ corresponds to loading the j-th input argument of the operator to be executed ($\Xi(o)$) onto the stack – or into a register, if appropriate.*
2. *Operand number 0 represents the execution of the operator itself – by calling a function or a processor instruction –, and storing back the return value.*
3. *A positive operand number j represents the storing of the $(1 + j)$-th output value of the operator.*[1]

Now we can model all the operations the system has to perform as elements of a polyhedron. Therefore, we build polyhedra of *occurrence instances* which have two dimensions more than the index space. These dimensions represent variables for the occurrence and the operand number:

Definition 2.4 (Occurrence instance) *Let o be an occurrence surrounded by n loops within a code fragment containing m parameters (including the program independent parameters). We call any instance of o, enumerated by the loops, an occurrence instance.*

Let P be the $(n+m)$-dimensional polyhedron that represents the index space of the loop nest containing o. We represent an occurrence instance as a vector $\alpha = (\alpha_1, \ldots, \alpha_{n+m+2}) \in \mathbb{Z}^{n+m+2}$, such that

$$\begin{array}{rcl} (\alpha_1, \ldots, \alpha_n, \alpha_{n+3}, \ldots, \alpha_{n+m+2}) & \in & P \\ \alpha_{n+1} & = & o \\ \alpha_{n+2} & \in & \Psi(o) \end{array}$$

[1] The first output value is the one with operand number 0, which is the return value of the expression.

We denote the projection of an occurrence instance α to coordinates $1, \ldots, n$ (the loop index vector), $n+1$ (the occurrence) and $n+2$ (the operand number) with $Indcs(\alpha)$, $\mathrm{Occ}(\alpha)$, and $\mathrm{Op}(\alpha)$, respectively.
Additionally, we define the r-argument of α, $\rho_r(\alpha) = (\alpha_1, \ldots, \alpha_n, \alpha_{n+1}, r, \alpha_{n+3}, \ldots, \alpha_{n+m+2})$.

As described above, we assume that there is a distinct *large* parameter, n_∞, that represents an arbitrarily large value. This idea has been suggested by Feautrier for his tool PIP [Fea03]. We use this parameter in order to represent any occurrence instance α of a code fragment containing n loops and m parameters as an $(n+m)$-dimensional vector, no matter how many loops actually surround $o = \mathrm{Occ}(\alpha)$. If occurrence o is not surrounded by the i-th loop of the code fragment, we define the i-th component of α as

$$\alpha_i = \begin{cases} -n_\infty & \text{if } o \text{ is textually placed above } i\text{-th loop} \\ n_\infty & \text{if } o \text{ is textually placed below } i\text{-th loop} \end{cases}$$

Encoding the occurrence instances in that way ensures that the lexicographical order on occurrence instances corresponds exactly to the execution order of the occurrence instances in the (sequential) program.
We define OI as the set of all occurrence instances of the code fragment under consideration.

Example 2.5 *Figure 1 shows the set of occurrence instances for the following code fragment*

```
DO i=0,1
  B(i)=A+2*n
END DO
```

The parameter dimensions are ignored in this figure, the occurrence dimension reaches from bottom to top, the operand dimension from left to right, and the index dimension for `i`*, together with the operand dimension, build the ground plane (operand- and* `i`*-dimensions are shown slightly skewed in order to fit into the figure). The arrows indicate dependence relations between a term and its subterms (we will examine these dependences in greater detail in Section 3).*

2.2 A Word about Dependences

Dependences between memory accesses dictate the order in which the occurrence instances of a code fragment may be executed. We denote a flow dependence from occurrence instance α to β as $\alpha\,\delta^f\,\beta$, and input, anti and output dependences as $\alpha\,\delta^i\,\beta$, $\alpha\,\delta^a\,\beta$, and $\alpha\,\delta^o\,\beta$, respectively.

Dependence analysis algorithms, such as the one introduced by Feautrier [Fea91], access instances as linear functions – so called h-transformations. An h-transformation maps the index vector of a dependence's

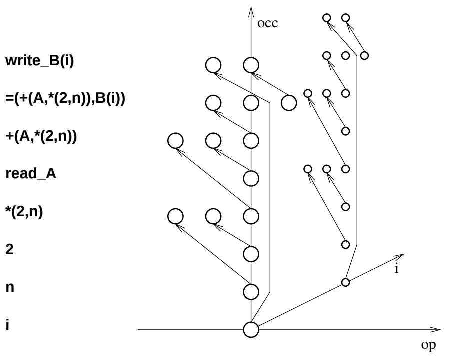

Figure 1: Occurrence instances of the code fragment of Example 2.5.

target to its source. In our framework, an h-transformation maps target occurrence instances to their respective souurce occurrence instances. In order to model the flow of data into and out of the code fragment considered, we suppose that, for each array `A`, there are dummy loops that assign each value `A(i)` to the array cell `A(i)` itself. These dummy loops precede and follow the considered program fragment. As a consequence, data flow to and from outside the considered program fragment can be modelled as ordinary dependence relations.

Example 2.6 *Consider the following code fragment (ignoring the dummy loops):*

```
        DO i = 1, min(10, m)
          DO j = i,i+n+5
!          [9]   [1]   [2]   [8] [6] [7] [5]   [3]   [4]
            a   (  i   ,  j  )  =   c   +   a  ( i-1 ,  j  )
          END DO
        END DO
```

Occurrences are written above the corresponding positions in the program text in brackets. We obtain the following flow dependence:

$$\begin{aligned}\delta^f &= \{((i,j,9\cdot n_1,0,n,m,n_1,n_\infty,n_1),(i+n_1,j,5\cdot n_1,0,n,m,n_\infty,n_1))\,|\\ &\qquad 1\le i\le \min(9\cdot n_1,m-n_1), i\le j\le i+n+5\cdot n_1, n_\infty=\infty, n_1=1\}\end{aligned}$$

This dependence is represented by an h-transformation (we omit the scope):

$$h_1 \;:\; (i,j,5\cdot n_1,0,n,m,n_\infty,n_1) \rightarrow (i-n_1,j,9\cdot n_1,0,n,m,n_1,n_\infty,n_1)$$

3 Alternative Program Description

Informally, the central question of code placement is where to put the computation of an expression in order to ensure that this computation is performed only if it is needed, and at the same time avoid recomputations of the same value. Relevant data flow information is on the one hand which computations have to be performed before an expression can be computed, and on the other hand which expressions share the same value. The technique we describe here, which we call loop-carried code placement (`LCCP`), is also described in some detail in earlier work together with an example and some timing results [FGL01].

We restrict ourselves to the purely syntactic – but loop-carried – equivalence of expressions. As a cost model, we employ the number of function calls and arithmetic operations executed. Then we minimize the amount of computations performed during the complete execution of the loop nest by replacing computations of already computed values by array lookups.

For the sake of simplicity, we assume that all variables that are written in the code fragment under consideration are live – i.e., may be read – after execution of that code fragment. Actually, live variables can be deduced from the fact that dependences to dummy loops originate from them.

Next, we define dependences between occurrence instances that do not just represent memory accesses.

For flow dependences, the idea is that the application of a function or operator depends on its input arguments, and the output arguments depend on the function application. Formally: Given an occurrence instance α_0 with $\mathrm{Op}(\alpha_0) = 0$ and $\Gamma(\mathrm{Occ}(\alpha_0)) \equiv f(e_{-1}, \ldots, e_{-l}, e_1, \ldots, e_m)$, where $e_{-1}, \ldots, e_{-l}$ are only read, while $e_1, \ldots, e_m$ are only written, and given $l{+}m$ occurrence instances $\alpha_{-1}, \ldots, \alpha_{-l}, \alpha_1, \ldots, \alpha_m$ with $\mathrm{Op}(\alpha_i) = 0$, $Indcs(\alpha_i) = Indcs(\alpha_0)$, and $\Gamma(\mathrm{Occ}(\alpha_i)) \equiv e_i$ for $i \in \{-l, \ldots, -1, 1, \ldots, m\}$, we define:

$$(\forall i : i \in \Psi(\mathrm{Occ}(\alpha_0)) : \ i < 0 \Rightarrow \rho_0(\alpha_i)\, \delta^f\, \rho_i(\alpha_0)) \tag{1}$$

$$(\forall i : i \in \Psi(\mathrm{Occ}(\alpha_0)) : \ i > 0 \Rightarrow \rho_i(\alpha_0)\, \delta^f\, \rho_0(\alpha_i)) \tag{2}$$

We call these textually carried dependences defined by (1) and (2) structural dependences and denote them by F. Using this definition, we create a graph that mimics the structure of the syntax tree, and enables us to argue about different loop iterations. A vertex in this occurrence instance graph (OIG) $G = (OI, E)$ is an occurrence instance α. In addition to the flow dependences that are extended by F, the anti and output dependences (written δ^a and δ^o, respectively) that are discovered by the access-based dependence test are included in the edge relation E:

$$E := {\delta^f}_{\mathrm{sup}} \cup {\delta^a}_{\mathrm{sup}} \cup {\delta^o}_{\mathrm{sup}}$$

The subscript sup denotes a pessimistic approximation of the subscripted dependence relation. Thus, the edges of the OIG are the "normal" dependences between occurrence instances: any order $\sqsubset$ on OI defines a valid execution of the program, if it is compatible with E and executes all operands of a given occurrence instance together ($\alpha, \beta \in OI, Indcs(\alpha) = Indcs(\beta), \mathrm{Op}(\alpha) = \mathrm{Op}(\beta) \Rightarrow \ (\alpha \not\sqsubset \beta \wedge \beta \not\sqsubset \alpha)$).

Example 3.1 *Let us review Example 2.5. The arrows in Figure 1 correspond to the structural dependences of the program, pointing from source to destination. The relationship between different instances for differing operand numbers could also be expressed by* implicit *dependences. However, differing operand numbers are a special case in that they* have *to be executed in an atomic fashion on a single processor on the sole ground that they represent different arguments to a single function call.*[2]

So far, we have not considered input dependences. In order to define equality on expressions, we generalize the definition of input dependences to occurrence instances. We define $\alpha\, \delta^i\, \beta$ iff for every input argument of α there is an input dependence to β.
More formally: For $\alpha, \beta \in OI$, with $\mathrm{Op}(\alpha) = \mathrm{Op}(\beta) = 0$, $\Xi(\mathrm{Occ}(\alpha)) = \Xi(\mathrm{Occ}(\beta))$

$$\begin{aligned}&(\forall i : i \in \Psi(\mathrm{Op}(\alpha)) : \ (\exists \gamma, \chi : \gamma, \chi \in OI : i < 0, (\gamma, \rho_i(\alpha)) \in F, \gamma\, \delta^i\, \chi, (\chi, \rho_i(\beta)) \in F)) \\ &\Rightarrow \alpha\, \delta^i\, \beta\end{aligned}$$

Note that write references cannot be input dependent on each other. Note further that we are only interested in *direct* input dependences, i.e., in input dependences which are executed without any write to the same memory cell executed in between. Formally:

$$\alpha\, \delta^i\, \beta \Longrightarrow \neg(\exists \gamma : \gamma \in OI : \alpha\, \delta^a\, \gamma \wedge \gamma\, \delta^f\, \beta)$$

[2] In functional languages such as Haskell, one could in principle use currying view *all* functions as unitary operators.

3.1 Code Placement by Affine Scheduling

Our aim is to perform code placement based on affine scheduling methods. The basic procedure here is as follows:

1. Construct the OIG (with control structures like loops being represented by sets of vertices).
2. Schedule equivalence classes of vertices in the OIG, and construct a placement function completing the schedule.
3. Generate the new syntax tree with loops governing assignment statements from the scheduled equivalence classes.

Step 1 is easily accomplished by a scan across the variable accesses. The determination of equivalence classes in Step 2 is based on the dependences computed by a dependence test such as Feautrier's [Fea91]. The subsequent calculation of a schedule in Step 2 is based on flow dependences resulting from the same dependence test. Step 3 generates the resulting code.

Since we work with equivalence classes wrt. input dependences, the actual representation is via a representative mapping $\mathrm{I} : OI \rightarrow OI$ to a representative of the equivalence class. For this representative, we choose the lexicographic minimum of the weakly connected component of the OIG that α belongs to. For each level of structural dependences, all remaining dependence relations are rewritten to $\mathrm{I}(\alpha)$ as dependence source instead of α, and correspondingly, OI is rewritten to $\mathrm{I}(OI)$. We call the resulting graph the condensed OIG.

As the number of occurrences in a program may get rather large, it is not feasible to compute space-time mappings for all sets of occurrence instances. However, this is only necessary for occurrence instances that build the source for several different targets – since then, the result has to be written out into memory in order to be consumed several times –, or if it is the source of a write reference to a live variable.

This technique effectively divides the computations into classes of different dimensionality, i.e., subexpressions that only depend on fewer index variables are extracted (similar to Loop Invariant Code Motion, with the exception that `LCCP` can generate its own loops for such purposes). Correspondingly, arrays have to be created that take up these values, and we have to decide where to place these arrays and the calculation for them. We will use a straight forward approach to tackle this problem in the next section.

4 Program Transformation

`LCCP` is able to remove redundant computations that are executed in different iterations of a loop. However, this improvement may come at the price of increased communication time. This is because intermediate results have to be stored in arrays which may introduce communications – actually a problem that occurs in any parallel program.

Communication time may depend on various factors. An effective placement method should take into account as many specific traits of the underlying system as possible. Such specific traits may include certain communication patterns for which efficient communication code can be generated and should have the ability to use replicated data storage in order to decrease communication time.

Our present aim is to model as many of these traits as flexibly as possible with regard to some underlying cost model. The idea is then to "plug in" a cost function that evaluates the current placements and to choose the placements that incur the least cost.

Example 4.1 *Let us consider the following code fragments:*

```
!HPF$ INDEPENDENT
     DO j=2,N-2
!HPF$ INDEPENDENT
        DO k=1,N
          L(k,j)= (y(j)   *2)*k&
                 -(y(j+2)*2)/k&
                 +(y(j+1)*2)+k&
                 +(y(j-1)*2)
        END DO
     END DO
```

```
!HPF$ INDEPENDENT
     DO j=1,N
       tmp(j)=y(j)*2
     END DO
!HPF$ INDEPENDENT
     DO j=2,N-2
!HPF$ INDEPENDENT
        DO k=1,N
          L(k,j)= tmp(j)   *k &
                 -tmp(j+2)/k &
                 +tmp(j+1)+k &
                 +tmp(j-1)
        END DO
     END DO
```

The (synthetic) code fragment on the left contains four computations of `y(j)*2`*. The same value is computed repeatedly for different iterations of both the* `k`*-loop and the* `j`*-loop.* `LCCP` *can transform this code fragment into the one on the right (the actual transformation depends on the scheduler used). In the transformed code,* `y(j)*2` *is calculated once and then assigned to a temporary.*

Let us suppose that `L` is `(BLOCK,BLOCK)`-distributed. Depending on the distribution of `y`, different distributions of `tmp` may be beneficial: most probably, an alignment with `y` is a good approach; if `y` is replicated, it is probably more useful to align with `L`.

4.1 The Model for Replicated Placements

In this section, we discuss our model in closer detail. We give a representation of replicated storage in our model, and finally show how to adapt dependence information to this representation.

4.1.1 Base Language

Our method depends on the presence of an initial data placement. We restrict ourselves to `HPF` programs, in which the user has already defined some data placement. The result of our method is a set of further placements that can be used in an `HPF` (or `HPF`-style) program for intermediate computations whose results are stored in arrays. Therefore, our method can be viewed as a compilation phase within an `HPF` compiler, or possibly as a preprocessor.

4.1.2 How to Model Replication

Our aim is to use replication (of data and computations) to reduce communication costs. The replicated mapping of a point $\alpha \in \mathbb{Z}^{n+m+2}$ to a subset of $\mathbb{Z}^p$ can be described as follows: we introduce a template T (corresponding to $\mathbb{Z}^p$) and a pair of affine functions $\Phi = (\Phi_V, \Phi_P)$. α is then mapped to T via Φ_V, while all the points of $\mathbb{Z}^p$ to which α should ultimately be mapped are themselves mapped to that same template element by Φ_P.

The set of processors on which an occurrence instance α is to be stored is:

$$\Phi(\alpha) = \Phi_P^{-1} \circ \Phi_V(\alpha)$$

A consistency condition is that the image of Φ_V is a subset of the image of Φ_P, which can safely be assumed in this context.

This approach corresponds to the description of replicated data in `HPF`. The functions Φ_P and Φ_V occur as their inverse in the corresponding `align` clause, and the vector space along which replication takes place is represented in `HPF` as a set of unit vectors: each align-target that does not correspond to an align-source – usually denoted by an asterisk – represents a unit vector of processors that store copies of the elements.

4.2 Dependences in the Presence of Replication

In order to estimate the costs for communication induced by the target program, we view the dependences with respect to space coordinates – i.e., after application of the placement relation $\Phi = (\Phi_V, \Phi_P)$. Since neither of the two functions defining the placement relation is necessarily invertible, it is not immediately clear how to represent dependences in the target program.

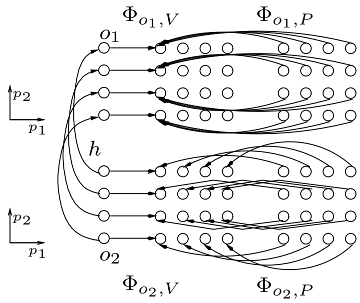

Figure 2: Dependence from instances of occurrence o_1 to instances of occurrence o_2.

Figure 2 shows an example of a dependence given by an h-transformation h as introduced by Feautrier [Fea92]. In the upper part of the figure, four instances of an occurrence o_1 are mapped to a virtual processor grid by $\Phi_{o_1,V}$; physical processors are mapped to the same grid by $\Phi_{o_1,P}$. The instances of occurrence o_1 are stored in a replicated fashion (all processors with the same p_1 coordinate own a copy of an occurrence instance).[3] The h-transformation maps a target occurrence instance α to its source occurrence instance $h(\alpha)$. Just as the mappings $(\Phi_{o_1,V}, \Phi_{o_1,P})$ define a placement relation for the instances of o_1, the mappings $(\Phi_{o_2,V}, \Phi_{o_2,P})$ define a placement for the instances of o_2, as depicted in the lower part of Figure 2. The instances of o_2 are only allocated on the first column of the processor space by their placement.

This means that all instances of occurrence o_2, each of which depends on a different instance of o_1, may "choose" the source processor from which to load the data needed for computation.

With the names taken from Figure 2, the new dependence relation is

$$h' = \Phi_{o_1,P}^{-1} \circ \Phi_{o_1,V} \circ h \circ \Phi_{o_2,V}^{-1} \circ \Phi_{o_2,P}$$

[3]Note that occurrence instances may represent computations as well as data.

. We observe that there are two sets involved in the computation of h':

- $\Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V}(\alpha)$: the possible sources of α (copies of the same value).
- $\Phi^{-1}_{o_2,V} \circ \Phi_{o_2,P}(\beta)$: the set of occurrence instances to be executed on the processor with space coordinates β.

If $\Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V}(\alpha)$ is a set, we may choose any point within that set for our dependence representation. Thus, we can account for $\Phi^{-1}_{o_1,P}$ by a generalized inverse that yields a single point.

However, $\Phi^{-1}_{o_2,V} \circ \Phi_{o_2,P}$ has to be represented as a set, e.g., by using two mappings to represent the relation h', just as with placement relations.

4.3 The Placement Method

With this description of replicated placements and resulting dependence relations, we can compute further placements. In order to allow almost any cost model to be "plugged in", we choose a naïve approach that basically considers all possible placements and selects the cheapest solution (names as in Figure 2):

1. For all sets of occurrence instances: propagate a placement candidate from source o_1 to target o_2 ($\Phi_{o_1} \circ h$) and from target to source ($\Phi_{o_2} \circ h^{-1}$).
2. For each combination of candidate placements for the occurrence instances:
   (a) Compute the image of the dependence information under the current placements. Candidate placements Φ_1, Φ_2 can be combined to a new placement Φ_3 placing all occurrence instances to both sets by asserting $\Phi^{-1}_{3,P} \circ \Phi_{3,V}(\alpha) \supseteq \Phi_1(\alpha) \cup \Phi_2(\alpha)$ for all α.
   (b) From each such image of the dependence information, compute the cost of the according placement; select the combination of placements that incurs the lowest cost.

As already noted, it is not necessary to compute a placement for *all* occurrence instances of a program. It can be deduced from the dependence information which sets need placements.

Placements are ultimately propagated from data. Therefore, we have to consider the transitive closure of the dependence relation in Step 1. Then we compute placement relations that can be expressed by two affine mappings. If an occurrence o_2 depends on an occurrence o_1 with h-transformation h, the placements Φ_{o_1}, Φ_{o_2} should satisfy

$$\Phi^{-1}_{o_2,P} \circ \Phi_{o_2,V}(\alpha) \quad \subseteq \quad \Phi^{-1}_{o_1,P} \circ \Phi_{o_1,V}(h(\alpha)) \tag{3}$$

If placement relations are propagated from source to target, this can be guaranteed by computing the Φ_{o_2} as: $\Phi_{o_2,P} := \Phi_{o_1,P}, \Phi_{o_2,V} := \Phi_{o_1,V} \circ h$.

If the placement is propagated from target to source, the computation is a bit more complex since we have to consider dependences to several different targets. However, we can compute a placement relation that ensures condition (3) by using replication: the smallest subspace for which replication has to occur to satisfy condition (3) can be computed as the solution of a linear equality system.

4.4 Propagating Placement Relations

In Section 4.3 we formulated our assumption that it is cheapest for an occurrence instance to be executed on the same processor as one of its sources or targets. When creating a list of possible placements for a given set O of occurrence instances, this leads to the conclusion that optimal placements can be calculated by placing O to the same processors as the occurrence instances it depends on or the ones that depend on O. Since all distributed occurrence instances ultimately depend on distributed array elements (if you consider the transitive closure of dependence relations) and – vice versa – ultimately are source for a write access to an array, these placements are given by the transitive closure of the dependence relation, i.e., they can be "propagated" from the array distribution at the "end" of each dependence chain. With this relation between instances of an occurrence o_1 and those of occurrence o_2 defined by an h-transformation, the possible placements for o_2 can be computed by a variant of the Floyd-Warshall algorithm on the finite representation of the condensed OIG.

Acknowledgements. This work is supported by the DAAD through project PROCOPE and by the DFG through project *LooPo/HPF*.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem. *Int. J. Parallel Programming*, 21(5/6), 1992. Two-part paper.

[Fea03] P. Feautrier. *Solving Systems of Affine (In)Equalities: PIP's User's Guide*, 2003. Additions by Jean-François Collard and C'edric Bastoul.

[FGL01] P. Faber, M. Griebl, and C. Lengauer. A closer look at loop-carried code replacement. In *Proc. GI/ITG PARS'01*, PARS-Mitteilungen Nr.18, pages 109–118. Gesellschaft für Informatik e.V., November 2001.

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