Proceedings of the 5th WSEAS Int. Conf. on APPLIED INFORMATICS and COMMUNICATIONS, Malta, September 15-17, 2005 (pp369-376)

Finding Free Schedules for Parameterized Loops with Affine Dependences Represented with a Single Dependence Relation

WLODZIMIERZ BIELECKI, ROBERT DRAZKOWSKI
Faculty of Computer Science
Technical University of Szczecin
Zolnierska 49 st., 71-210 Szczecin
POLAND

Abstract: - An approach, permitting us to build free schedules for affine loops with affine dependences represented with a single dependence relation, is described. The iterations of each time under the free schedule can be executed as soon as their operands are available. This allows us to extract maximal fine-grained loop parallelism. The approach requires an exact dependence analysis. To describe the approach and carry out experiments, the dependence analysis by Pugh and Wonnacott has been chosen where dependences are represented in the form of tuple relations. The approach can be applied to both non-parameterized and parameterized loops. Problems to be resolved in the future to utilize the entire power of the presented technique are discussed.

Key-Words: - Affine Loops, Loop Parallelization, Free Schedules

1 Introduction

Numerous transformations have been developed to expose parallelism in loops, for example, [2,4-9, 11-14,16,17]. Most of those transformations permit us to extract parallelism available in both uniform and non-uniform loops. But the question is how much loop parallelism these approaches extract.

The general problem of the loop parallelization is the following. Dependences in loops can be represented with approximations in general, or with an exact representation when possible. For each dependence representation based on approximations (level of dependences, uniform dependences, polyhedral representation of dependences), optimal algorithms exist, but for exact affine dependences, it is not known what are the loop transformations that extract maximal parallelism [9]. So, some of the main questions arising at the loop parallelization are the following. Can we extract maximal parallelism for loops with affine dependences? If so, how can we generate code representing maximal parallelism? The only attempt in this direction is index splitting [13]. But it is a heuristic procedure, and it does not answer how much index splitting is necessary, how many different code structures must be generated to reach optimality.

This paper is to contribute in answering the above questions.

Following Vivien [20], we imply that an algorithm extracting parallelism is optimal if it finds all parallelism: 1) that can be extracted in its framework; or 2) that is contained in the representation of the dependences it handles; or 3) that is contained in the program to be parallelized (not taking into account the dependence representation used nor the transformations allowed).

Free schedules [9] permit us to extract all parallelism available in loops, but well-known techniques based on linear or affine schedules sometimes fail to find free schedules for non-uniform loops.

This paper presents an approach permitting us to find free schedules for parameterized loops whose dependences are represented with a single dependence relation. This approach is optimal and finds all parallelism available in the affine loop provided that problems discussed in this paper will be resolved in the future.

The main objective of this paper is to focus on the free schedule as a very important approach for understanding the parallelism contained in loops, not necessarily for its run-time execution.

2 Background

In this paper, we deal with affine loop nests where lower and upper bounds, array subscripts, and conditionals are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known positive constants.

Following work [20], we refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation. Two operations I and J are dependent if both access the same memory location and if at least one access is a write. We refer to I and J as the source and destination of the dependence, respectively, provided that operation I accesses the same memory location earlier than operation J.

The approach, presented in this paper, requires an exact representation of dependences, and hence an exact dependence analysis that detects a dependence if and only if it exists. In general, any known technique, extracting exact dependences, can be used for our approach, for example, [10,18]. However, describing the approach depends on the format of the presentation of exact dependences and carrying out experiments depends on the availability of tools permitting us to find exact dependences.

To describe the approach and carry out experiments, we have chosen the dependence analysis proposed by Pugh and Wonnacott [18] where dependences are represented with dependence relations. This analysis is implemented in Petit, a research tool for doing dependence analysis and program transformations. A dependence relation is a mapping from one iteration space to another, and it is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of the dependence and the values of the symbolic constants. Each dependence relation represents a dependence class (flow, anti, output).

Given a dependence relation R, our approach requires calculating relation R^k, where

$R^k=R\circ R^{k-1}$

$R^{k-1}=R\circ R^{k-2}$

.....

$R^2=R\circ R$

$R^1=R$

" ° " is the denotation of the composition operation.

Dependence relations, found by Petit, usually contain more variables than there exist index variables in the loop. Forming and resolving a recurrence equations system to obtain relation R^k is more convenient when the number of variables is equal or less than the number of the loop index variables (the number of loop nests) in a given loop. We call a procedure of reducing the number of variables in the dependence relation constraints as normalization and it is performed as follows.

1. Resolve all equations being contained in the constraints of a dependence relation, let $x=a$ be the solution to these equations, substitute all the appearances of x in the dependence relation and its constraints for a.

2. Resolve all inequalities in the dependence relation constraints.

For example, the relation

R:={[i,j]->[i',j']: 2i-i'=1 && -j-1=0 &&
-j'-1=0 && 2<=i<=25}

after the normalization is transformed to the relation

R:={[i,1]->[2i-1,1]: 2<=i<=25}.

We refer to the source (destination) of a dependence as the ultimate dependence source (destination) if it is not a destination (source) of any other dependence. Ultimate dependence sources and destinations represented with relation R can be found by means of the following calculations: (domain R – range R) and (range R – domain R), respectively.

Definition [5]. A schedule is a mapping, which assigns a time of execution to each operation of the loop in such a way that all dependences are preserved, that is, mapping $s:I\rightarrow Z$ such that for any arbitrary dependent operations op1 and op2 $\in$ I, $s(op1)<s(op2)$ if op1 $\prec$ op2.

Assertion [5]. A free schedule assigns operations as soon as their operands are available, that is, it is mapping $\sigma:I\rightarrow Z$ such that

$$\sigma(p)=\begin{cases} 0 \text{ if there is no } p'\in I \text{ s.t. } p'\prec p \\ 1+\max(\sigma(p'), p'\in I, p'\prec p) \end{cases}.$$

The free schedule is the "fastest" schedule possible. Its total execution time is

$$T_{free}=1+\max(\sigma_{free}(p), p\in I).$$

The idea of finding the free schedule to execute operations of a loop with a single dependence relation is the following. Since all dependences are represented by a single dependence relation, the iterations to be executed at time k, k=1,2... can be calculated as range(R^k(UDS)), where UDS is a set being comprised of ultimate dependence sources to be executed at time 0 under the free schedule. Figures 1, 2 illustrate this fact. Hence, we need an approach to calculate relation R^k. The next section presents a technique to calculate R^k.

To follow the material of this paper, the reader should be familiar with the iterations on tuple relations such as: union, difference, range, domain, composition, application as well as existentially quantified variables explained in [15].

3 Calculating relation R^k

In this paper, we deal with a loop of the form

for $i_1=l_1$ **to** u_1 **do**
for $i_2=l_2$ **to** u_2 **do**
...
for $i_n=l_n$ **to** u_n **do**
$V[p_{11}i_1+p_{12}i_2+...+q_1 ,...,p_{n1}i_1+p_{n2}i_2+...+q_n]= ...$
$...=V[r_{11}i_1+r_{12}i_2+...+s_1 ,...,r_{n1}i_1+r_{n2}i_2+...+s_n]$
endfor
...
endfor
endfor.

Let set *UDS* represent ultimate dependence source and be of the form

UDS:= $\{[t_1, t_2,. . . , t_n]$: constraints imposed on $t_1,t_2,...,t_n\}$.

To find relation R^k, we construct and resolve a system of recurrence equations using the fact that dependent iterations create chains where in each of them a dependence destination is the dependence source for the consecutive dependence except from the ultimate dependence destination. This property can be described with a system of recurrence equations. For the loop whose dependence relation is written as follows

$$R=\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}.\begin{bmatrix} i_1 \\ i_2 \\ ... \\ i_n \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}\rightarrow\begin{bmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ & ... & & \\ c_{n1} & c_{n2} & ... & c_{nn} \end{bmatrix}.\begin{bmatrix} i_1 \\ i_2 \\ ... \\ i_n \end{bmatrix}+\begin{bmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{bmatrix}$$

the appropriate system of recurrence equations may be represented in the form

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}.\begin{bmatrix} i_1^{k+1} \\ i_2^{k+1} \\ ... \\ i_n^{k+1} \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}=\begin{bmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ & ... & & \\ c_{n1} & c_{n2} & ... & c_{nn} \end{bmatrix}.\begin{bmatrix} i_1^k \\ i_2^k \\ ... \\ i_n^k \end{bmatrix}+\begin{bmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{bmatrix}. \quad (1)$$

Obtaining a solution to that system requires initial values of variables, which are provided with ultimate dependence sources being represented by the following constraint

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}.\begin{bmatrix} i_1^0 \\ i_2^0 \\ ... \\ i_n^0 \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}=\begin{bmatrix} t_1 \\ t_2 \\ ... \\ t_n \end{bmatrix}, \quad (2)$$

where $t_1,t_2,\ldots,t_n$ satisfy constraints of set *UDS* introduced above.

Recurrence equations (1) with constraints (2) may be resolved by means of one of numerous commercial and academic solvers, for example Maple [21], Mathematica [22], Maxima [23], MuPAD [24], PURRS [25]. For our experiments, we have chosen Mathematica, which offers enough flexibility for describing and resolving systems of recurrence equations with initial values of variables.

Having a solution to system (1) with constraints (2), we can form relation R^k as follows

$$R^k=\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}.\begin{bmatrix} i_1^0 \\ i_2^0 \\ ... \\ i_n^0 \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}\rightarrow\begin{bmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ & ... & & \\ c_{n1} & c_{n2} & ... & c_{nn} \end{bmatrix}.\begin{bmatrix} i_1^k \\ i_2^k \\ ... \\ i_n^k \end{bmatrix}+\begin{bmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{bmatrix},$$

where $i_1^k,i_2^k,...,i_n^k$ are solutions to an appropriate system of recurrence equations, $t_1,t_2,\ldots,t_n$ (presented in the solution $i_1^k,i_2^k,...,i_n^k$) satisfy constraints of ultimate dependence sources, k satisfies the condition $1 \le k \le k_{max}$, where k_{max} is the number of times under the free schedule.

The number of times under the free schedule, k_{max}, can be obtained taking into account the constraints of dependence relation R^k and the upper bounds of the loop indices. A solution standing for R^k must also preserve the lexicographical order of operations.

For an n-nested loop, the system of inequalities permitting us to find the number of times under the free schedule, k_{max}, is of the form

$$\begin{bmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ & ... & & \\ c_{n1} & c_{n2} & ... & c_{nn} \end{bmatrix}.\begin{bmatrix} i_1^k \\ i_2^k \\ ... \\ i_n^k \end{bmatrix}+\begin{bmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{bmatrix}\le\begin{bmatrix} u_1 \\ u_2 \\ ... \\ u_n \end{bmatrix}, \quad (3)$$

where $u_1,u_2,...,u_n$ are the upper bounds of the loop index variables, and $i_1^k,i_2^k,...,i_n^k$ are the solutions to a system of recurrence equations (1) with constraints (2).

The lexicographical order of iterations being executed at times *k-1* and *k* is preserved when a solution satisfies the following constraint

$$\exists p:\left(\sum_{j=1}^{n} a_{pj}\left(i_j^k-i_j^{k-1}\right)>0\right)\wedge\left(\forall q\ s.t.\ 1\le q<p:\sum_{j=1}^{n} a_{qj}\left(i_j^k-i_j^{k-1}\right)=0\right). \quad (4)$$

Resolving inequalities (3) and (4), we find k that is usually in the domain of real numbers. As the number of times under the free schedule we take $\max_{t_1,t_2,...,t_n}\lfloor k \rfloor$.

A solution to systems (1) and (2) can be in the domain of integers and/or real numbers. There exist numerous techniques permitting generating code scanning elements of a set being represented with affine constraints, but to our knowledge, there is no technique permitting generating code scanning elements of a set being represented with non-linear constraints.

To verify that solutions to a system of recurrence equations are integers, we can transform the system of recurrence equations

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & ... & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}.\begin{bmatrix} i_1^{k+1} \\ i_2^{k+1} \\ ... \\ i_n^{k+1} \end{bmatrix}+\begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix}=\begin{bmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ & ... & & \\ c_{n1} & c_{n2} & ... & c_{nn} \end{bmatrix}.\begin{bmatrix} i_1^k \\ i_2^k \\ ... \\ i_n^k \end{bmatrix}+\begin{bmatrix} d_1 \\ d_2 \\ ... \\ d_n \end{bmatrix}$$

into the form

$$\begin{bmatrix} i_1^{k+1} \\ i_2^{k+1} \\ \dots \\ i_n^{k+1} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \dots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}^{-1} \cdot \left(\begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ & \dots & & \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{bmatrix} \cdot \begin{bmatrix} i_1^k \\ i_2^k \\ \dots \\ i_n^k \end{bmatrix} + \begin{bmatrix} d_1 \\ d_2 \\ \dots \\ d_n \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \right). \quad (5)$$

If all of the coefficients of the inverted matrix on the left-hand side of (5) are integers, then the solutions represented with $i_1^k, i_2^k, \dots, i_n^k$ are also integers; otherwise they could be both integer and/or real numbers.

4 Finding Free Schedules

The idea of the algorithm presented in this section is as follows. Given relation *R*, we find ultimate dependence sources to be executed at time 0 and next form and resolve a correspondent recurrence equation(s) to build R^k. On the basis of relation R^k, we form and resolve appropriate inequalities to find the upper bound of *k*, k_{max}, which represents the maximal number of times under the free schedule. Using relation R^k, we form set *S(k)* comprising iterations to be executed at time *k*, $k=1,2,\dots,k_{max}$, under the free schedule. When all the elements of *S(k)* are integers and described by affine forms, then any well-known technique to generate code scanning elements of *S(k)* can be applied. Finally, we expose independent iterations, that is, those that do not belong to any dependence and generate code enumerating these iterations. According to the free schedule, they are to be executed at time 0.

Algorithm. Finding the free schedule for the loop with a single dependence relation

Input: dependence relation *R* represented in the normalized form.

Output: R^k; the number of times under the free schedule, k_{max}; set of operations to be executed at time *k*, *S(k)*, $k=1,2,\dots, k_{max}$; code representing the free schedule provided that all the elements of set *S(k)* are integers and represented by affine constraints.

1) Find ultimate dependence sources (operations to be executed at time 0), *UDS*, as
UDS = domain R – range R.

2) Taking into account set *UDS*, form appropriate recurrence equations in such a manner as described in Section 3.

3) Resolve all the equations formed at step 2 and build R^k*(UDS)* as described in Section 3.

4) Find the upper bound of *k*, k_{max}, on the basis of constraints imposed on input and output variables of R^k and the upper bounds of the loop indices by means of constructing and resolving appropriate inequalities and constraints honoring the fact that the operations to be executed at time *k* must be lexicographically greater than those executed at time *k-1* (see Section 3).

5) Form set *S(k)* as
S(k)= { range R^k (UDS): constraint(elements of S(k) ∈ Integers)}.

6) When the elements of set *S(k)* are integers and represented by only affine constraints (do not include non-linear expressions), generate code scanning operations of set *S(k)* applying any well-known technique, for example [1,3,4,19].

7) Find a set, *IND*, representing independent operations, that is, ones that are no source or destination of any dependence as follows
IND= LD – (domain R ∪ range R)
where *LD* is a set representing the loop domain.

8) Generate code scanning iterations of set *IND* applying any well-known technique, for example [1,3,4,19].

It is worth to note that in the general case, set *S(k)* yielded by the algorithm above, can comprise elements that are not integers and/or are represented not only by affine constraints but also by non-linear expressions. For example, consider the following loop

Example 1
for i=1 **to** 1000 **do**
for j=1 **to** 1000 **do**
  a(2*j+3,i+1)=a(2*i+j+1,i+j+3)
endfor
endfor.

Petit finds the following dependence relation for this loop
R={[-x1+2x2+4,2x1-2x2-6]->[x1,x2]:
1<=-x1+2x2+4<=1000 && 1<=2x1-2x2-6<=1000 && 1<=x1<=1000 && 1<=x2<=1000}.

Ultimate dependence sources are contained in the following set
UDS = {[t1,t2]: Exists(alpha: 2alpha=t2 && 2t2+4<=t1<=-t2+997 && 2<=t2) OR Exists(alpha: 2alpha=t2 && 1<=t1<=-t2+997 && t1<=t2+3 && 2<=t2)}.

The solution to an appropriate system of recurrence equations yielded with Mathematica is as follows

$$x1[k] \to \frac{1}{4\sqrt{17}} \left(-4 \left(\left(\frac{1}{4} \left(3 - \sqrt{17} \right) \right)^k - \left(\frac{1}{4} \left(3 + \sqrt{17} \right) \right)^k \right) t1 - 2^{1-2k} \left(\left(3 - \sqrt{17} \right)^{1+k} - \left(3 + \sqrt{17} \right)^{1+k} \right) (2 + t1 + t2) \right),$$

$$x2[k] \to \frac{1}{17} 4^{-1-k} \left(-17\, 2^{3+2k} + 34 \left(3 - \sqrt{17} \right)^k - 14\sqrt{17} \left(3 - \sqrt{17} \right)^k + 34 \left(3 + \sqrt{17} \right)^k + 14\sqrt{17} \left(3 + \sqrt{17} \right)^k + \left(\left(34 - 6\sqrt{17} \right) \left(3 - \sqrt{17} \right)^k + 2 \left(3 + \sqrt{17} \right)^k \left(17 + 3\sqrt{17} \right) \right) t1 + \left(\left(17 - 7\sqrt{17} \right) \left(3 - \sqrt{17} \right)^k + \left(3 + \sqrt{17} \right)^k \left(17 + 7\sqrt{17} \right) \right) t2 \right)$$

For times 0,1,2, we yield the following sets.

The elements to be executed at time 0 are comprised in the set $S(0) = \{[t_1, t_2]\}$.

The elements to be executed at time 1 are represented with the set

$$S(1)=\left\{\left[t_1 + t_2 + 2, t_1 + \frac{1}{2} t_2 - 1 \right]\right\}.$$

The elements to be executed at time 2 are contained in the set

$$S(2)=\left\{\left[2t_1 + \frac{3}{2} t_2 + 3, \frac{3}{2} t_1 + \frac{5}{4} t_2 + \frac{1}{2} \right]\right\}.$$

Elements of set *S(0)* are integers (they are represented with ultimate dependence sources). Since dependence ultimate sources are all even numbers, elements of *S(1)* are also integer numbers. But there are both integer and non-integer numbers in *S(2)*, for example when t_1=1 and t_2=2 (an element of the ultimate dependence sources set), the correspondent element is [8, 4.5], which does not belong to the loop domain. For t_1=1 and t_2=4, the correspondent element is [11, 7], which describes the operation in the loop domain. Hence, there is the need for developing techniques permitting us to generate code when set *S(k)* can comprise elements that are not integers and/or are represented not only with affine constraints but also with non-linear expressions.

5 Examples

In this section, we attach two examples illustrating applying the approach presented in the previous section. For the loop below

Example 2

```
for i=1 to n do
for j=1 to n do
  a(i,j)=a(i,2*j)
endfor
endfor
```

Petit finds the dependence relation

R:= {[i,j] -> [i,2j]: 1<=i<=n && 1<=2j<=n}

which does not require the normalization. The loop domain is defined as follows

LD:={[i,j]:1<=i,j<=n}.

Independent operations are represented with the following set

IND:= LD – (domain(R) union range(R)) = {[i,j]: Exists(alpha: 2alpha=1+j && i,j<=n<=2j-1 && 1<=i)}.

Ultimate dependence sources are the result of the following calculation

UDS:= domain(R) – range(R) = {[t1,t2]: Exists(alpha: 2alpha=1+t2 && 1<=t1<=n && 1<=t2 && 2t2<=n)}.

The system of recurrence equations to find relation R^k (written using the Mathematica syntax) is of the form

RSolve[{x1[k+1]==x1[k],x2[k+1]==2x2[k], x1[0]==t1,x2[0]==t2}, {x1[k],x2[k]},k] .

The solutions to this system are

x1[k]→t1,

x2[k]→t2*2^k.

They both are represented with positive integers, and the upper bound of *k* can be obtained from the inequality

$t_2*2^k \le n$,

that yields

$$k \le \frac{Log\left(\frac{n}{t_2}\right)}{Log(2)},$$

and k_{max} is

$$k_{\max} = \max_{t_2} \left\lfloor \frac{Log\left(\frac{n}{t_2}\right)}{Log(2)} \right\rfloor = \left\lfloor \frac{Log(n)}{Log(2)} \right\rfloor.$$

Relation R^k may be written as follows

R^k:= {[t1,t2]->[t1,t2*2k]: Exists(alpha: 2alpha=1+t2 && 1<=t1<=n && 1<=t2 && 2t2<=n) && t2*2^(k)<=n},

and set *S(k)* is of the form

S(k) := range R^k = {[t1,t2*2k]: Exists(alpha: 2alpha=1+t2 && 1<=t1<=n && 1<=t2 && 2t2<=n) && t2*2^(k)<=n}.

The elements of set *S(k)* can be scanned by the code

```
for ( k=1; k<=trunc(log(n)/log(2))+1; k++) {
parfor( t1=1; t1<= n; t1++) {
parfor(  t2=2*intDiv(intDiv(n+1+1,2),2)-1;  t2<=n; t2 += 2) {
if( t2*2^(k)<=n) {
  a(t1,t2*2^(k))=a(t1,2t2*2^(k));
}
}
}
}.
```

This loop was built manually because of limitations of existing tools.

Independent operations and ultimate dependence sources can be enumerated with the loop

```
parfor( t1 = 1; t1<=n; t1++) {
parfor( t2=1; t2<=intDiv(n,2); t2+=2) {
  a(t1,t2)=a(t1,2t2);
}
parfor(  t2=2*intDiv(intDiv(n+1+1,2),2)-1;  t2<=n;
t2+=2) {
  a(t1,t2)=a(t1,2t2);
}
}.
```

This loop was generated with the codegen function of the Omega library.

Figure 1 presents the loop domain and dependences for the loop of Example 2 when n=10.

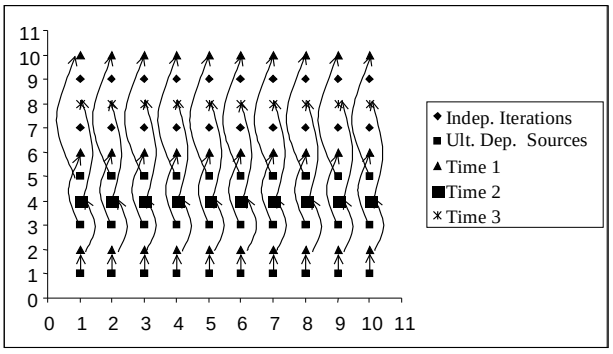

Fig. 1. Loop domain with dependences for the loop of Example 2 for n=10.

For the loop below

Example 3

```
for i=1 to m do
for j=1 to m do
  a(i,j)=a(i,2*n-j)
endfor
endfor
```

Petit finds the dependence relation

R := {[i,j] -> [i,2n-j]: 1<=i<=m && 1<=j<n && 2n<=m+j}

which does not require the normalization.

The loop domain is defined as follows

LD := {[i,j]: 1<=i,j<=m}.

Independent operations are calculated as follows

IND := LD – (domain(R) union range(R)) = {[i,j]: 2n,1<=j<=m && 1<=i<=m} union {[i,j]: 1<=i<=m && 1<=j<=m && m+j<2n} union {[i,n]: 1<=n<=m && 1<=i<=m}.

Ultimate dependence sources are contained in the following set

UDS := domain(R) – range(R) = {[t1,t2]: 1<=t1<=m && 1<=t2<n && 2n<=m+t2}.

The system of recurrence equations to find relation R^k (written using the Mathematica syntax) is of the form

RSolve[{x1[k+1]==x1[k],x2[k+1]==2n-x2[k], x1[0]==t1,x2[0]==t2}, {x1[k],x2[k]},k].

The solutions to this system are

$x1[k] \rightarrow t1$,

$x2[k] \rightarrow (-1)^k (n(-1)^k - n + t2)$.

x2[k] can be simplified to the form

x2[k]→t2, when k is an even number

x2[k]→2n-t2, when k is an odd number.

Because x2[k] is periodic, k_{max}=1. This means that there are only two times under the free schedule: time 0 to execute independent iterations and ultimate dependence sources and time 1 to execute dependence destinations.

Relation R^k can be written as follows

R^k:={[t1,t2]->[t1,t2]: k=0 && 1<=t1<=m && 1<=t2<n && 2n<=m+t2} union {[t1,t2]->[t1,2n-t2]: k=1 && 1<=t1<=m && 1<=t2<n && 2n<=m+t2}.

and set *S(k)* is of the form

S(k):= range R^k = {[t1,t2]: k=0 && 1<=t1<=m && 1<=t2<n && 2n<=m+t2} union {[t1,t2]: k=1 && n<t2<=2n-1,m && 1<=t1<=m}.

The elements of set *S(k)* can be scanned by the code

```
if( k ==0 && n>=2 && n<=m-1) {
parfor( t1=1; t1<=m; t1++) {
parfor( t2=max(2*n-m,1); t2 <= n-1; t2++) {
  a(t1,t2)=a(t1,2*n-t2);
}
}
}
if(k == 1 && n >= 2 && n <= m-1) {
parfor(t1 = 1; t1 <= m; t1++) {
parfor(t2 = n+1; t2 <= min(m,2*n-1); t2++) {
  a(t1,t2)=a(t1,2*n-t2);
}
}
}.
```

Independent operations and ultimate dependence sources can be enumerated with the loop

```
parfor( t1=1; t1<=m; t1++) {
parfor( t2=1; t2<=min(-m+2*n-1,m); t2++) {
  a(t1,t2)=a(t1,2*n-t2);
}
parfor (t2=max(-m+2*n,1); t2<=n-1; t2++) {
  a(t1,t2)=a(t1,2*n-t2)
}
if (m>=n && n>=1) {
  a(t1,t2)=a(t1,n);
}
parfor (t2 = max(2*n,1); t2 <= m; t2++) {
  a(t1,t2)=a(t1,2*n-t2);
}
}.
```

The both above codes were genererated with the codegen function of the Omega library.

Figure 2 presents the loop domain and dependences for the loop of Example 3 when m=10 and n=5.

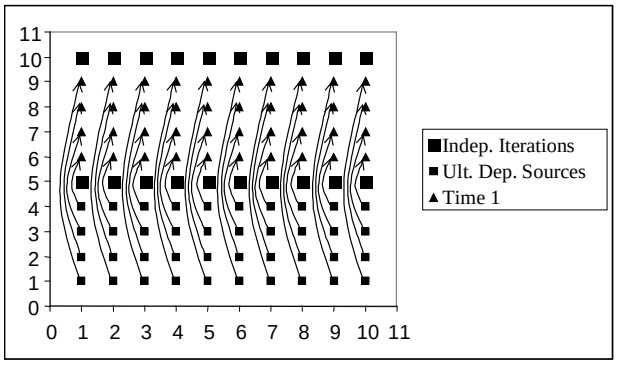

Fig. 2. Loop domain for the loop of Example 3 when 2n=m=10.

6 Related work

Concerning the loop parallelization, the following facts are known.

If the level of dependences is the only available representation, then Allen and Kennedy's algorithm (loop distribution) is known to be optimal [6] with respect to its input, that is, it finds maximal parallelism that is contained in the representation of the dependences it handles.

For the case of a single statement with uniform dependences, linear scheduling (optimized Lamport's hyperplane method) is asymptotically optimal [8].

For the case of several statements with uniform dependences, the Lamport's hyperplane method has been extended in work [14] where it is shown that a linear schedule plus shifts leads to finding maximum parallelism.

For the case of polyhedral approximations of dependences (including direction vectors), Darte and Vivien's algorithm is optimal [7].

For affine dependences, the most powerful algorithm is Feautrier's one based on multi-dimensional affine schedules [12]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, among all possible affine schedules, it is optimal [20].

Since the approach presented in [16,17] (affine time partition mappings) is based on affine schedules, it does not allow us to detect maximal loop parallelism for affine dependences in the general case.

The technique presented in this paper is to contribute to understanding the problem of free schedules for loops with affine dependences and it is optimal for both non-parameterized and parameterized loops whose dependences can be represented with a single dependence relation.

7 Conclusion

In this paper, we have presented the algorithm that permit us to build free schedules for both non-parameterized and parameterized loops whose dependences are represented with a single dependence relation. If a free schedule is represented by affine forms, code can be generated easily by means of well-known techniques and public available tools, for example, the Omega project software, available on the Internet site ftp://ftp.cs.umd.edu/pub/omega or the Polylib code generator, available on the Internet site http://www.irisa.fr/cosi/ALPHA/welcome.html.
When a free schedule is represented by non-linear forms, techniques should be developed to generate code enumerating iterations to be executed at correspondent times under the free schedule.

We have focused on the free schedule as a very important approach for understanding the parallelism contained in a loop, not necessarily for its run-time execution.

The task for further research is to develop techniques permitting us to build free schedules for loops whose dependences are represented by multiply dependence relations. The challenge here is to be able to compute a schedule in a symbolic way and to generate the corresponding code with a complexity that depends on the loop structure but not on the volume of the computation it describes. This challenge deserves more attention from the research community.

References:

[1] C. Ancourt and F. Irigoin, Scanning polyhedra with DO loops, In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991, pp. 39-50.

[2] D. Bacon, S. Graham, and O. Sharp, Compiler transformations for high-performance computing, *Computing Surveys*, Vol. 26(4), 1994, pp. 345-420.

[3] C. Bastoul, Code Generation in the Polyhedral Model Is Easier Than You Think, In *Proceedings of the PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, 7-16, 2004.

[4] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien, Loop paralellelization algorithms: from parallelism extraction to code generation. *Technical Report 97-17*, LIP, ENS-Lyon, France, June , 1997.

[5] A. Darte and Y. Robert, Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, 1994, pp. 814-822.

[6] A. Darte, F. Vivien, On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops, *Journal of Parallel Algorithms and Applications, Special issue on "Optimizing Compilers for Parallel Languages"*, Vol. 12, 1997, pp. 83-112.

[7] A. Darte, F. Vivien, Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs, *International Journal of Parallel Programming*, Vol. 25(6), 1997, pp. 447-496

[8] A. Darte, L. Khachiyan, Y. Robert, Linear Scheduling is Nearly Optimal, *Parallel Processing Letters*, Vol. 1(2), 1991, pp. 73-81.

[9] A. Darte, Y. Robert, F. Vivien, *Scheduling and Automatic Parallelization*, Birkhäuser Boston, 2000.

[10] P. Feautrier, Dataflow analysis for array and scalar references. *International Journal of Parallel Programming*, 20(1), 1991, pp. 23-53.

[11] P. Feautrier, Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *International Journal of Parallel Programming*, Vol. 21(5), 1992, pp. 313-348.

[12] P. Feautrier, Some efficient solutions to the affine scheduling problem, Part II, multidimensional time. *International Journal of Parallel Programming*, Vol. 21(6), 1992.

[13] P. Feautrier, M. Griebl, C. Lengauer, Index Set Splitting, *International Journal of Parallel Programming*, Vol. 28(6), 2000, pp. 607-631.

[14] P. Le Gouëslier d'Argence, Affine Scheduling on Bounded Convex Polyhedric Domains is Asymptotically Optimal, *TCS* 196(1-2), 1998, pp. 395-415.

[15] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, The Omega Library Interface Guide, *Technical Report CS-TR-3445*, Dept. of Computer Science, University of Maryland, College Park, March 1995.

[16] W. Lim, G.I. Cheong, M.S. Lam, An affine partitioning algorithm to maximize parallelism and minimize communication, In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, 1999.

[17] W. Lim, M.S. Lam, Maximizing parallelism and minimizing synchronization with affine transforms, In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[18] W. Pugh, D. Wonnacott, An Exact Method for Analysis of Value-based Array Data Dependences, *Workshop on Languages and Compilers for Parallel Computing*, 1993.

[19] F. Quilleré, S. Rajopadhye, D. Wilde, Generation of Efficient Nested Loops from Polyhedra, *International Journal of Parallel Programming*, 28(5), 2000.

[20] F. Vivien, On the optimality of Feautrier's scheduling algorithm, In *Proceedings of the EUROPAR'2002*, 2002.

[21] http://www.maplesoft.com

[22] http://www.wolfram.com

[23] http://maxima.sourceforge.net

[24] http://www.mupad.com

[25] R. Bagnara et al., The automatic solution of recurrence relations. I. Linear recurrences of finite order with constant coefficients, *Quaderno 334*, Dipartimento di Matematica, Università di Parma, Italy, 2003