


**AUTHOR QUERY FORM**

	<b>Journal: PARCO</b>  <b>Article Number: 2081</b>	<b>Please e-mail or fax your responses and any corrections to:</b>  <b>E-mail: <a href="mailto:corrections.esch@elsevier.sps.co.in">corrections.esch@elsevier.sps.co.in</a></b>  <b>Fax: +31 2048 52799</b>
---	--	---

Dear Author,

Please check your proof carefully and mark all corrections at the appropriate place in the proof (e.g., by using on-screen annotation in the PDF file) or compile them in a separate list. Note: if you opt to annotate the file with software other than Adobe Reader then please also highlight the appropriate place in the PDF file. To ensure fast publication of your paper please return your corrections within 48 hours.

For correction or revision of any artwork, please consult <http://www.elsevier.com/artworkinstructions>.

Any queries or remarks that have arisen during the processing of your manuscript are listed below and highlighted by flags in the proof. Click on the 'Q' link to go to the location in the proof.

<b>Location in article</b>	<b>Query / Remark: <a href="#">click on the Q link to go</a></b> <b>Please insert your reply or correction at the corresponding line in the proof</b>
<a href="#"><u>Q1</u></a>  <a href="#"><u>Q2</u></a>	<p>Please confirm that given names and surnames have been identified correctly.</p> <p>Please check whether the designated corresponding author is correct, and amend if necessary.</p> <div data-bbox="400 1789 963 1888"> <p>Please check this box if you have no corrections to make to the PDF file</p> <input data-bbox="851 1805 921 1864" type="checkbox"/> </div>

Thank you for your assistance.

---

**Highlights**

---

► Free scheduling represents maximal fine-grained parallelism. ► The power  $k$  of a dependence relation is used to form free scheduling. ► Free-scheduling permits for generating effective parallel code for graphical cards.

---



Contents lists available at SciVerse ScienceDirect

## Parallel Computing

journal homepage: [www.elsevier.com/locate/parco](http://www.elsevier.com/locate/parco)

## Free scheduling for statement instances of parameterized arbitrarily nested affine loops

Włodzimierz Bielecki\*, Marek Palkowski, Tomasz Klimek

West Pomeranian University of Technology, Szczecin, Poland

## ARTICLE INFO

## Article history:

Received 26 April 2011

Received in revised form 20 May 2012

Accepted 2 June 2012

Available online xxxx

## Keywords:

Affine loops

Free scheduling

Fine-grained parallelism

Transitive closure

## ABSTRACT

An approach is presented permitting us to build free scheduling for statement instances of affine loops. Under the free schedule, loop statement instances are executed as soon as their operands are available. This allows us to extract maximal fine-grained loop parallelism and minimize the number of synchronization events. The approach is based on calculating the power  $k$  of a relation representing exactly all dependences in a loop. In general, such a relation is a union of simpler relations. When there are troubles with calculating free scheduling due to the large number of simpler dependence relations, another technique is discussed allowing for extracting free scheduling in an iteration subspace defined by indices of inner nests of this loop. We demonstrate that if we are able to calculate the power  $k$  of a dependence relation describing all dependences in the loop, then we are able also to produce free scheduling. Experimental results exposing the effectiveness, efficiency, and time complexity of the algorithms are outlined. Problems to be resolved in the future to utilize the entire power of the presented techniques are discussed.

© 2012 Published by Elsevier B.V.

## 1. Introduction

A legal schedule of loop statement instances is a function that assigns a time of execution to each loop statement instance preserving all dependences in this loop.

Scheduling permits for extracting fine-grained parallelism available in loops. Let us remind that fine-grained parallelism means tasks are relative small in terms of code size and execution time, data is transferred among processors frequently, while coarse-grained parallelism means data is transferred infrequently after larger amounts of computation.

Under the free schedule, loop statement instances are executed as soon as their operands are available that permits us to extract all fine-grained parallelism available in the loop. From a theoretical point of view, free-scheduling permits for answering the question: how much fine-grained parallelism is not extracted by well-known techniques.

There have been developed numerous approaches to form loop statement instances scheduling, for example [1–7]. But well-known techniques based on linear or affine schedules do not guarantee finding free scheduling for statement instances of affine loops.

In this paper, we present a novel algorithm for calculating free scheduling for statements instances of affine loops. It is based on calculating the power  $k$  of a relation  $R$ ,  $R^k$ , where  $R$  represents exactly all dependences in a loop. We demonstrate that the algorithm produces the free-schedule when well-known techniques fail to produce it. When there are troubles with calculating  $R^k$ , for example, the number of simpler dependence relations composing relation  $R$  is large enough (hundreds and

\* Corresponding author.

E-mail addresses: [wlodzimierzb@gmail.com](mailto:wlodzimierzb@gmail.com), [WBielecki@wi.zut.edu.pl](mailto:WBielecki@wi.zut.edu.pl) (W. Bielecki), [MPalkowski@wi.zut.edu.pl](mailto:MPalkowski@wi.zut.edu.pl) (M. Palkowski), [TKlimek@wi.zut.edu.pl](mailto:TKlimek@wi.zut.edu.pl) (T. Klimek).

thousands), we consider another algorithm that iteratively tries to extract free scheduling in an iteration subspace defined by indices of inner nests of the loop. We also demonstrate that free scheduling can be applied for producing effective parallel programs for multi-core graphic cards.

## 2. Background

In this paper, we deal with *static-control loop nests*, where lower and upper bounds as well as conditionals and array subscripts are affine functions of symbolic parameters and surrounding loop indices. A *statement instance*  $s(I)$  is a particular execution of a statement  $s$  of the loop for some loop iteration  $I$ .

**Definition 1** [8]. The *iteration vector* of a statement  $s$  is the vector consisting of values of the *loop indices* of all *loops* surrounding the statement  $s$ , from outermost to innermost.

Two statement instances  $s_1(I)$  and  $s_2(I)$  are dependent if both access the same memory location and if at least one access is a write. Provided that  $s_1(I)$  is executed before  $s_2(I)$ ,  $s_1(I)$  and  $s_2(I)$  are called the *source* and *destination* of the dependence, respectively. The sequential execution ordering of statement instances, denoted as  $s_1(I) \prec s_2(J)$ , is induced by the lexicographic ordering of iteration vectors and the textual ordering of statements when the instances share the same iteration vector.

**Definition 2** [6]. Loops are called perfectly nested if all statements are surrounded by the same loops, otherwise they are imperfectly nested.

**Definition 3** ([6,9]). The *free schedule* is the function that assigns discrete time of execution to each loop statement instance as soon as its operands are available, that is, it is mapping  $\sigma: LD \rightarrow \mathbb{Z}$  such that

$$\sigma(p) = \begin{cases} 0 & \text{if there is no } p_1 \in LD \text{ s.t. } p_1 \rightarrow p, \\ 1 + \max(\sigma(p_1), \sigma(p_2), \dots, \sigma(p_n)); & p, p_1, p_2, \dots, p_n \in LD, \\ p_1 \rightarrow p, p_2 \rightarrow p, \dots, p_n \rightarrow p, \end{cases}$$

where  $p, p_1, p_2, \dots, p_n$  are loop statement instances,  $LD$  is the loop domain,  $p_1 \rightarrow p, p_2 \rightarrow p, \dots, p_n \rightarrow p$  mean that the pairs  $p_1$  and  $p$ ,  $p_2$  and  $p$ ,  $\dots$ ,  $p_n$  and  $p$  are dependent,  $p$  represents the destination and  $p_1, p_2, \dots, p_n$  represent the sources of dependences,  $n$  is the number of operands of statement instance  $p$  (the number of dependences whose destination is statement instance  $p$ ).

The free schedule is the fastest legal schedule [6].

The approach to find free scheduling, presented in this paper, requires an exact representation of dependences.

**Definition 4** [10]. A dependence analysis is exact if it detects a dependence if and only if one exists.

To describe the approach and carry out experiments, we have chosen the dependence analysis proposed by Pugh and Wonnacott [10], where dependences are represented with relations. This analysis is implemented in Petit [11] (the Omega library dependence analyzer) which returns a set of dependence relations describing all dependences in a loop. A dependence relation is a tuple relation of the form  $\{[input\ list] \rightarrow [output\ list]; constraints\}$ , where *input list* and *output list* are the lists of variables used to describe input and output tuples and *constraints* is a Presburger formula describing the constraints imposed upon *input list* and *output list*.

The general form of a dependence relation is as follows [10]:

$$\left\{ [s_1, \dots, s_k] \rightarrow [t_1, \dots, t_k] \mid \bigvee_{i=1}^n \exists \alpha_{i1}, \dots, \alpha_{im_i} \text{ s.t. } F_i \right\},$$

where  $F_i$ ,  $i = 1, 2, \dots, n$  are represented by Presburger formulas [12], i.e., they are conjunctions of affine equalities and inequalities on the input variables  $s_1, \dots, s_k$ , the output variables  $t_1, \dots, t_k$ , the existentially quantified variables  $\alpha_{i1}, \dots, \alpha_{im_i}$ , and symbolic constants.

An *ultimate dependence source* (resp. *destination*) is a source (resp. *destination*) that is not the destination (resp. source) of another dependence. A set, *UDS*, comprising all ultimate dependence sources can be found as  $\text{domain}(R) - \text{range}(R)$ , where  $R$  represents all dependences in a loop.

Algorithms, presented in this paper, require applying the union, composition, and application operations on dependence relations and the difference operation on sets [11]. Applying these operations is possible when the size of tuples (the number of elements representing a tuple) of different *relations(sets)* is the same. This condition is always true for relations describing dependences in perfectly nested loops, but for imperfectly nested loops it is not always true.

To permit for applying the operations mentioned above on relations and sets, we have to preprocess them. Preprocessing makes the sizes of input and output tuples of dependence relations to be the same by inserting the value “−1” at the rightmost positions of correspondent tuples as well as inserts identifiers of loop statements in the last position of input and output tuples. Inserting “−1” does not introduce any *false (not existing)* dependence after preprocessing because existing

programming languages suppose that loop indices cannot be negative. Inserting loop statement identifiers makes clear which statements originate sources and destinations of dependences.

The preprocessing procedure is presented below. Step one of the preprocessing procedure is obviously omitted when we deal with perfectly nested loops. Step two of the preprocessing procedure can be skipped when the loop body comprises the only statement. Preprocessing a set is carried out in the same manner as preprocessing a relation except that it is applied to the single tuple of this set.

### Dependence relations preprocessing procedure [13]

**Input:** Set  $S$  of dependence relations  $R_{ij}$ , where values of  $i, j \in [1, q]$ , represent the statement identifiers numbered in the textual order (the order in which statements appear in the source text),  $q$  is the number of statements nested within an  $m$ -depth loop. Each  $R_{ij}$  denotes the union of all the relations describing dependences between instances of statements  $i$  and  $j$ .

**Output:** Set  $S$  of preprocessed dependence relations.

**foreach** relation  $R_{ij} \in S$  **do**

1. Transform relation  $R_{ij}$  so that each input and output tuple of  $R_{ij}$  has exactly  $m$  elements by inserting the value  $-1$  at the rightmost positions of those tuples whose number of elements is less than  $m$ , e.g., replace the tuple  $e = [e_1 \ e_2 \ \dots \ e_{m-k}]$ , where  $k$  is some integer, for the tuple  $e = [e_1 \ e_2 \ \dots \ e_{m-k} \ \underbrace{-1 \ \dots \ -1}_{k \text{ times}}]$ .
2. Extend the input and output tuples of  $R_{ij}$  with identifiers of statements  $i$  and  $j$ , respectively, i.e., transform  $R_{ij} = \{[e] \rightarrow [e']\}$  to  $R_{ij} = \{[e, i] \rightarrow [e', j]\}$ .

Positive transitive closure for a given relation  $R$ ,  $R^+$ , is defined as follows [14]  $R^+ = \{e \rightarrow e' : e \rightarrow e' \in R \vee \exists e'' \text{ s.t. } e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+\}$ .

It describes which vertices  $e'$  in a dependence graph (represented by relation  $R$ ) are connected with vertex  $e$ .

Transitive closure,  $R^*$ , is defined as follows [14]:  $R^* = R^+ \cup I$ , where  $I$  the identity relation. It describes the same connections in a dependence graph (represented by  $R$ ) that  $R^+$  does plus connections of each vertex with itself.

### 3. Finding free scheduling for loop statement instances

The idea of the algorithm presented in this section is the following. Given preprocessed relations  $R_1, R_2, \dots, R_m$ , we first calculate  $R = \bigcup_{i=1}^m R_i$  and then  $R^k$ , where  $R^k = R \circ R \circ \dots \circ R$ , “ $\circ$ ” is the composition operation. Techniques of calculating the power  $k$  of relation  $R$  are presented in the following publications [14–18] and are out of the scope of this paper.

Let us only note that given transitive closure  $R^*$ , we can easily convert it to the power  $k$  of  $R$ ,  $R^k$ , and vice versa, for details see [15,17].

A set of vertices representing sources and destinations of loop statement instance dependences and a set of edges from sources to destinations of those form a dependence graph.

Given a dependence graph represented by relation  $R$  (the tuples of  $R$  represent vertices while the constraints of  $R$  define edges), the length of the path connecting a pair of vertices in this graph is equal to  $n - 1$ , where  $n$  is the number of vertices laying on this path. Each incoming edge of a vertex stands for an operand of the statement instance defined by this vertex. The number of incoming edges of a vertex is equal to the number of operands of the statement instance represented by this vertex.

Relation  $R^k$  describes those connected vertices in the dependence graph, represented by dependence relation  $R$ , for which the length of each path connecting them is equal to  $k$ , i.e., the length of each path between a vertex represented by the input tuple of  $R^k$  and a correspondent vertex(s) described by the output tuple of  $R^k$  is equal to  $k$ . This path length defines the time when a correspondent operand (defined by an incoming edge) becomes available, i.e., if the path length is equal to  $k$ , then the correspondent operand is available at time  $k$  also.

Given a set of ultimate dependence sources ( $UDS$ ) and provided that  $R^k$  is calculated exactly, each vertex represented by the set  $R^k(UDS)$ , produced as the application of relation  $R^k$  to set  $UDS$ , is connected in the dependence graph, described by relation  $R$ , with a correspondent vertex(s) belonging to set  $UDS$ . The length of each path between these vertices is equal to  $k$ . But not all the statement instances defined by the set  $R^k(UDS)$  may have available operands because some operands may be available at time more than  $k$ . Each vertex represented by the set  $R^+ \circ R^k(UDS)$  is also connected in the dependence graph with some vertex(s) of set  $UDS$  by a path of a length more than  $k$ . It is worth to note that the sets  $R^k(UDS)$  and  $R^+ \circ R^k(UDS)$  may include the same elements. These are elements(statement instances) that do not have all available operands. Indeed, if some vertex represented by the set  $R^k(UDS)$  has a non-available operand, this means that this vertex is connected in the dependence graph with some vertex(s) defined by set  $UDS$  by different paths, at least one of them is of length  $k$  while one or more other paths are of lengths more than  $k$ .

Hence, each vertex, represented by the set  $S(k) = R^k(UDS) - R^+ \circ R^k(UDS)$ , is connected in the dependence graph, defined by relation  $R$ , with some vertex(s) represented by set  $UDS$ . The length of each path between these vertices is equal to  $k$  and there does not exist a shorter path in the dependence graph between them; if there exist two or more paths between these vertices, then their lengths are the same. This means that each vertex (loop statement instance) represented by the set  $S(k)$  has all available operands that are formed as soon as possible because the length of all paths connecting a vertex represented

by  $\underline{S}(k)$  and a corresponding vertex represented by set  $UDS$  is the same and equal to  $k$ . Hence at time  $k$ , all the statement instances belonging to the set  $\underline{S}(k)$  can be scheduled for execution and it is guaranteed that  $k$  is as few as possible.

Finally, we expose independent statement instances, that is, those that do not belong to any dependence and generate code enumerating them. According to the free schedule, they are to be executed at time  $k = 0$ .

To illustrate the presented idea, let us consider the following dependence relations, describing dependences in a loop with a single statement.

$$R1 = \{[i,j] \rightarrow [i,j+1]: 1 \leq i \leq n \ \&\& \ 1 \leq j < n\};$$

$$R2 = \{[i,j] \rightarrow [i+1,j]: 1 \leq i < n \ \&\& \ 1 \leq j \leq n\};$$

$$R3 = \{[i,j] \rightarrow [i+1,j+1]: 1 \leq i,j < n\}.$$

Relation  $R^k$  for  $n = 3$  and the sets  $UDS$ ,  $R^k(UDS)$ , where  $R = R1 \cup R2 \cup R3$ , are as follows.

$$R^k = \{[i,j] \rightarrow [i',j']: i' \leq 3 \ \&\& \ j' \leq 3 \ \&\& \ 1 \leq i \ \&\& \ 1 \leq k \ \&\& \ k+i+j \leq i'+j' \ \&\& \ 1 \leq j \ \&\& \ j' \leq k+j \ \&\& \ i' \leq k+i\},$$

$$UDS(R) = \{[1,1]\},$$

$$R^k(UDS) = \{[1,1] \rightarrow [i',j']: i' \leq k+1, 3 \ \&\& \ j' \leq k+1, 3 \ \&\& \ 1 \leq k \ \&\& \ 2+k \leq i'+j'\}.$$

Fig. 1 demonstrates vertices(statement instances) belonging to the set  $R^k(UDS)$  for particular values of  $k$ . The set  $R^+ \circ R^k(UDS)$  is the following.

$$R^+ \circ R^k(UDS) = \{[1,1] \rightarrow [i',j']: i' \leq 3 \ \&\& \ j' \leq 3 \ \&\& \ 3+k \leq i'+j' \ \&\& \ 1 \leq k\}.$$

Fig. 2 illustrates vertices fitting to the set  $R^+ \circ R^k(UDS)$ .

The set  $\underline{S}(k)$  is presented below.

$$\underline{S}(k) = R^k(UDS) - R^+ \circ R^k(UDS) = \{[i,k-i+2]: k-1, 1 \leq i \leq k+1, 3 \ \&\& \ 1 \leq k\}.$$

Fig. 3 shows vertices belonging to the set  $\underline{S}(k)$  for particular values of  $k$ .

We may conclude that the set  $\underline{S}(k)$  stands for the free schedule of loop statement instances represented with vertices of the dependence graph defined by relation  $R$ .

Below we present the algorithm that realizes the presented above idea in a formal way.

**Algorithm 1.** Finding free scheduling for loop statement instances.

**Input:** a source loop; set,  $S$ , of  $m$  preprocessed dependence relations  $R_1, R_2, \dots, R_m$  representing all dependences in the loop.

**Output:** code representing (free) scheduling for statement instances of the input loop, variable *free* informing what is the kind of scheduling returned by the algorithm (free or not free), *free* is TRUE for free scheduling, and FALSE otherwise.

**Method:**

1. *free* = TRUE, form relation  $R$  as the union of all the relations  $R_i$ ,  $1 \leq i \leq m$ , fitting to set  $S$ ,  $R = R_1 \cup R_2 \cup \dots \cup R_m$ .
2. Calculate set,  $UDS$ , containing ultimate dependence sources  $UDS = \text{domain}(R)$ .
3. Form relation  $R^k$ ; for this purpose use any known algorithm, for example, one from those published in [14–18]. If the constraints of  $R^k$  are affine, then go to step 4, otherwise, compute an over-approximation of  $R^k$  with affine constraints using one of techniques presented in [11,16], *free* = FALSE.
4. Calculate parameterized set,  $\underline{S}(k)$ , containing loop statement instances to be executed at time  $k$  under the free-schedule

$$\underline{S}(k) = R^k(UDS) - R^+ \circ R^k(UDS), \text{ note that for } k = 0, \underline{S}(k) = UDS$$

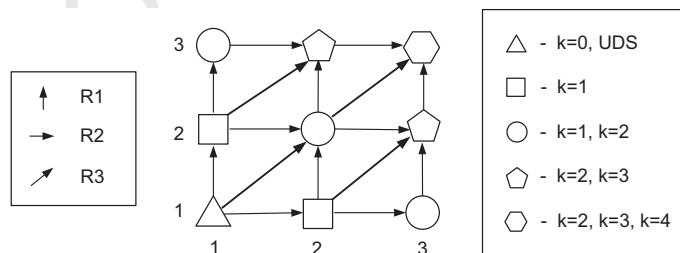


Fig. 1. Illustrating the set  $R^k(UDS)$ .

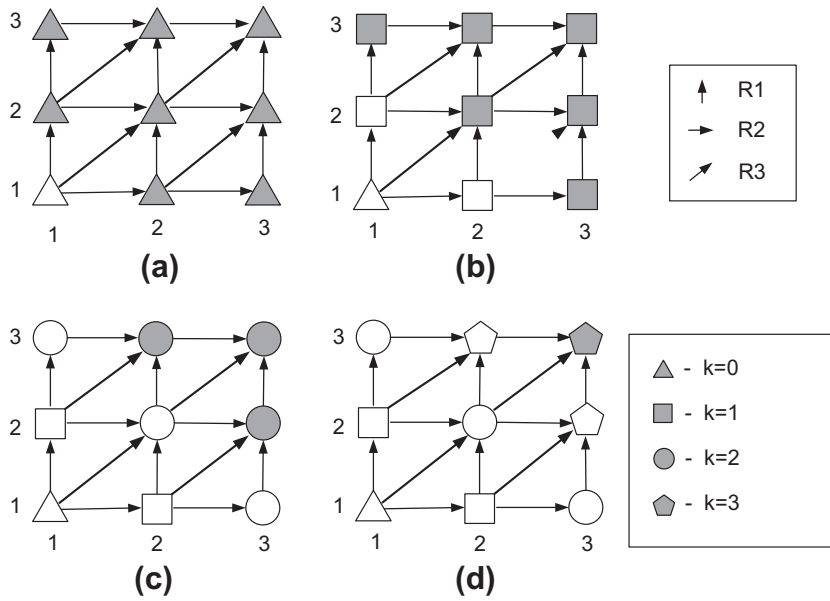


Fig. 2. Illustrating the set  $R^+ \circ R^k(UDS)$ .

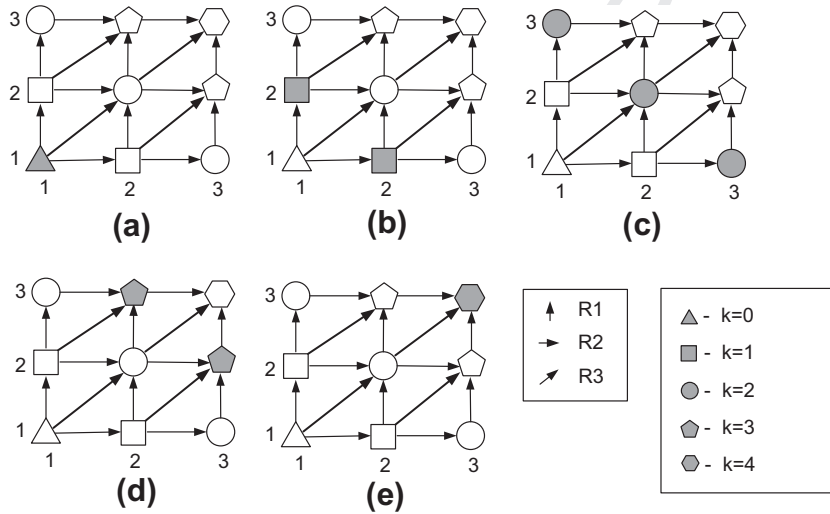


Fig. 3. Illustrating the set  $S(k)$ .

5. Insert into the first position of the tuple representing set  $S(k)$  symbolic variable  $k$  responsible for representing time under (free) scheduling.
6. Generate pseudo-code scanning elements of set  $S(k)$  produced in step 5, using any well-known technique, for example one of those presented in [19–21]. In such code, the outermost loop is sequential, it enumerates values of  $k$  (time) while inner loops are parallel to scan pseudo statement instances to be executed at time  $k$ ; a pseudo statement is represented as  $S(e_1, e_2, \dots, e(n+2))$ , where  $n$  is the number of loop indices,  $e_i$  are expressions,  $e_1$  represents time  $k$ ,  $e_2, e_3, \dots, e(n+1)$  stand for array index expressions,  $e(n+2)$  represents the identifier of a statement (it is inserted by the preprocessing procedure).
7. In the code produced by step 6, replace each pseudo statement  $S(e_1, e_2, \dots, e(n+2))$  for an appropriate statement of the source loop; for this purpose use the element  $e(n+2)$  to recognize the statement and the elements  $e_2, e_3, \dots, e(n+1)$  to insert in this statement proper array index expressions.
8. Find set,  $IND$ , containing independent statement instances as the difference between the preprocessed loop iteration space,  $IS$ , represented by the union of preprocessed iterations sets of all loop statements and the set



including all dependent statement instances, i.e.,

$$IND = IS - (\text{domain}(R) \cup \text{range}(R)).$$

9. Generate code scanning elements of set  $IND$  in the same manner as it is presented by steps 6 and 7 for set  $S(k)$ . This code is to be executed at time  $k = 0$ .

In the presented algorithm, step 4 is the most important. Let us analyse what a set  $S(k)$  represents. Prior to present Algorithm 1, we justified that at time  $k$  all the statement instances belonging to the set  $S(k)$  can be scheduled for execution and it is guaranteed that  $k$  is as few as possible.

The following conditions are also satisfied for elements of a set  $S(k)$ . For a given  $k$ , statement instances represented by the set  $S(k)$  are independent, i.e., there does not exist any path in the dependence graph connecting any pair of vertices represented by the set  $S(k)$ . The execution of time partitions formed by sets  $S(k)$ ,  $k = 0, 1, \dots, k_{max}$ , sequentially, (where  $k_{max}$  is the last time under the free schedule), while the execution of elements(statement instances) of each set  $S(k)$  in parallel honors all the dependences represented by relation  $R$ . Indeed, for a given  $k$ , elements of  $S(k)$  are independent and it is guaranteed that for each dependence in the dependence graph represented by  $R$ , the dependence source,  $ds$ , is executed before the corresponding dependence destination,  $dd$ , because  $ds \in S(k_1)$ ,  $dd \in S(k_2)$  and  $k_2 > k_1$ .

Reminding the definition of free scheduling (see Section 2), we can state that sets  $S(k)$ ,  $k = 0, 1, 2, \dots, k_{max}$  form the free schedule for statement instances represented with vertices of the dependence graph defined by relation  $R$ .

Step 5 in Algorithm 1 modifies a set  $S(k)$  by introducing time  $k$  as the first element of the tuple representing  $S(k)$ . This allows us to generate an outermost loop scanning sequentially values of  $k$ , i.e., time partitions.

The rest steps in the presented algorithm are to permit for calculating a set  $S(k)$  and generate code applying well-known techniques.

It is worth to note that when the following conditions are true: (i) an over-approximation of  $R^k$  has a parametrized upper bound of  $k$  (for example  $k \leq 2n$ , where  $n$  is the parameter) and (ii) transitive closure of  $R$ ,  $R^+$ , is derived from  $R^k$ , that satisfies condition (i), by making  $k$  to be existentially quantified in  $R^k$  [14], then applying such  $R^k$  and  $R^+$  in Algorithm 1 leads to producing a legal schedule (not always the free-schedule). Indeed, in such a case,  $R^k$  and  $R^+$  may represent false dependences (not existing in a given loop) in a parametrically bounded domain (it is bounded due the fact that  $k$  is limited by an upper bound, see condition (i)). If a false dependence is a transitive dependence, that can be represented as a composition of true direct dependences, then Algorithm 1 will still produce the free schedule because such a false dependence does not impact set  $S(k)$ , it will be removed in step 4. If a false dependence is a direct dependence, then Algorithm 1 does not guarantee producing free scheduling, but guarantees that returned scheduling is legal. In fact a direct false dependence may cause that according to a schedule, produced by Algorithm 1, particular statement instances may be executed at later time than that under the free schedule. However, all dependences (both true and false ones) will be respected, i.e., the source of each dependence will be executed at some time  $k$  while the corresponding dependence destination at time  $k'$ ,  $k' > k$ , and each statement instance will be executed only one time.

Let us illustrate the presented algorithm by means of the following example.

Example 1.

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    a[i][j] = a[i-2][j] + a[i-1][j+3];
```

Fig. 4 presents the free schedule for the loop of Example 1 when  $n = 7$ .

For this loop, there are the two dependence relations returned by Petit.

$$R1 = \{[i, j] \rightarrow [i+2, j]: 1 \leq i \leq n-2 \ \&\& \ 1 \leq j \leq n\};$$

$$R2 = \{[i, j] \rightarrow [i+1, j-3]: 1 \leq i < n \ \&\& \ 4 \leq j \leq n\}.$$

The relations above do not require preprocessing. For this and the following examples, we use the Omega calculator to carry out steps of the presented algorithms. Applying Algorithm 1, we get the following results.

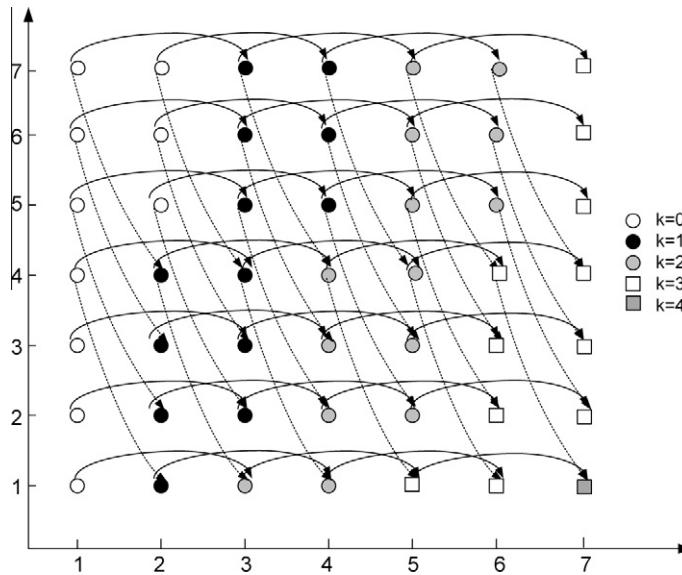
1.  $free = TRUE$ ,  $R = R1 \cup R2 = \{[i, j] \rightarrow [i+2, j]: 1 \leq i \leq n-2 \ \&\& \ 1 \leq j \leq n\} \cup \{[i, j] \rightarrow [i+1, j-3]: 1 \leq i < n \ \&\& \ 4 \leq j \leq n\}$ .
2.  $Domain(R) = \{[i, j]: 1 \leq i \leq n-2 \ \&\& \ 1 \leq j \leq n\} \cup \{[i, j]: 1 \leq i < n \ \&\& \ 4 \leq j \leq n\}$ .  $Range(R) = \{[i, j]: 3 \leq i \leq n \ \&\& \ 1 \leq j \leq n\}$ .

$$UDS = \{[i, j]: i+2, j \leq n \leq j+2 \ \&\& \ 1 \leq i \leq 2\} \cup \{[1, j]: 1 \leq j \leq n-3\}.$$

3. Using the technique presented in [17], we get:

$$R^k = \{[i, j] \rightarrow [k+i, j-3k]: 3k+1 \leq j \leq n \ \&\& \ 1 \leq k \ \&\& \ 1 \leq i \ \&\& \ k+i \leq n\} \cup \{[i, j] \rightarrow [2k+i, j]: 1 \leq j \leq n \ \&\& \ 1 \leq k \ \&\& \ 1 \leq i \ \&\& \ 2k+i \leq n\} \cup \{[i, j] \rightarrow [i', j-6k-3i+3i']: j \leq n \ \&\& \ i' \leq n \ \&\& \ i' < 2k+i \ \&\& \ k+i < i' \ \&\& \ 1 \leq i \ \&\& \ 6k+3i < j+3i'\} \cup \{[i, j] \rightarrow [i, j]: k=0 \ \&\& \ i+2, j \leq n \leq j+2 \ \&\& \ 1 \leq i \leq 2\} \cup \{[1, j] \rightarrow [1, j]: k=0 \ \&\& \ 1 \leq j \leq n-3\}.$$





**Fig. 4.** The free schedule for Example 1 when  $n=7$ . The solid lines represent dependencies described with relation R1, the dotted lines represent dependencies described with relation R2.

4.  $S(k) = \{[k+1, j]: 3 \leq n \text{ \& \& } 0 \leq k \text{ \& \& } 3k+j \leq n \text{ \& \& } 1 \leq j\} \cup \{[i, j]: k+2 \leq i \leq n \text{ \& \& } 1 \leq j \leq n \text{ \& \& } 4 \leq n \text{ \& \& } n+3i \leq 8+6k+j \text{ \& \& } 3+6k+j \leq n+3i\} \cup \{[3, j]: n=3 \text{ \& \& } k=1 \text{ \& \& } 1 \leq j \leq 3\}.$
5.  $S(k) = \{[k, k+1, j]: 3 \leq n \text{ \& \& } 0 \leq k \text{ \& \& } 3k+j \leq n \text{ \& \& } 1 \leq j\} \cup \{[k, i, j]: 1+2 \leq i \leq n \text{ \& \& } 1 \leq j \leq n \text{ \& \& } 4 \leq n \text{ \& \& } n+3i \leq 8+6k+j \text{ \& \& } 3+6k+j \leq n+3i\} \cup \{[k, 3, j]: n=3 \text{ \& \& } k=1 \text{ \& \& } 1 \leq j \leq 3\}.$
6. The loop scanning elements of set  $S(k)$  for  $k \geq 0$  and being produced by the codegen function of the Omega library is as follows.

```

if (n >= 3) {
  for (t1 = 0; t1 <= intDiv (2*n - 2, 3); t1++) {
    for (t3 = 1; t3 <= -3*t1 + n; t3++) {
      sl(t1, t1 + 1, t3);
    }
  }
  if (n >= 4) {
    for (t2 = max (intDiv (-n + 6*t1 + 4 + 2, 3), t1 + 2); _ t2 <= min (n, 2*t1 + 2); t2++) {
      for (t3 = max (-6*t1 + n + 3*t2 - 8, 1); _ t3 <= min (n, -6*t1 + n + 3*t2 - 3); t3++) {
        sl(t1, t2, t3);
      }
    }
  }
  if (n <= 3 \& \& t1 >= 1) {
    for (t3 = 1; t3 <= 3; t3++) {
      sl(1, 3, t3);
    }
  }
}

```

7. The final code scanning elements of set  $S(k)$  for  $k \geq 0$  (generated by the Omega codegen function) is of the form.

```

if (n >= 3) {
  for (k = 0; k <= intDiv (2*n - 2, 3); k++) {
    for (t2 = 1; t2 <= -3*k + n; t2++) { //parallel loop

      a[k+1][t2] = a[k-1][t2] + a[k][t2+3];
    }
    if (n >= 4) {
      for (t1 = max (intDiv (-n + 6*k + 6, 3), k + 2); //parallel loop
           _ t1 <= min (n, 2*k + 2); t1++) {
        for (t2 = max (-6*k + n + 3*t1 - 8, 1); //parallel loop
             _ t2 <= min (n, -6*k + n + 3*t1 - 3); t2++) {
          a[t1][t2] = a[t1-2][t2] + a[t1-1][t2+3];
        }
      }
    }
    if (n <= 3 \& \& k >= 1) {
      for (t2 = 1; t2 <= 3; t2++) { //parallel loop
        a[3][t2] = a[1][t2] + a[2][t2+3];
      }
    }
  }
}

```

8.  $IND = \{[i, j]: 1, n-1 \leq i \leq n \ \&\& \ 1 \leq j \leq n \ \&\& \ i \leq 2\}$ .

9. The loop scanning elements of set IND and being produced by the codegen function of the Omega library is as follows:

```
for (t1 = max(n-1, 1); t1 <= min(n, 2); t1++) { //parallel loop
  for (t2 = 1; t2 <= n; t2++) { //parallel loop
    a[t1][t2] = a[t1-2][t2] + a[t1-1][t2+3];
  }
}
```

Analysing the code produced by Algorithm 1, we can conclude the following. For  $n \geq 3$ , the number of time partitions under the free schedule is equal to  $\lfloor (2n-2)/3 \rfloor$  (see the outermost loop of the code scanning elements of set  $S(k)$ ). Following the well-known Affine Transformation Framework (ATF) [1,2,6] to derive an affine mapping permitting for scheduling, we first form the following legality constraints on the basis of relations R1 and R2:  $c1(i+2) + c2*j - c1*i - c2*j \geq 0$ ,  $c1(i+1) + c2*(j-3) - c1*i - c2*j \geq 0$ .

They can be simplified to the form.

$$2c1 \geq 0, c1-3c2 \geq 0.$$

The best legal linear schedule  $k = c1*i + c2*j$  (minimizing the number of time partitions) for Example 1 occurs when  $c1 = 1$ ,  $c2 = 0$ , i.e.,  $k = i$ .

This means that the best linear schedule for this example results in  $n$  time partitions and it is not the free schedule. The difference  $n - \lfloor (2n-2)/3 \rfloor$  demonstrates how many synchronization events free scheduling yields less than those implied by the best linear schedule. For example, for  $n = 1000$ , this difference is equal to 334 and it increases with increasing  $n$ .

Let us apply now Algorithm 1 to the following imperfectly nested loop.

Example 2.

```
for (col = 1; col <= N1; col++){
  temp[col] = tx[col][1][1];
  for (i = 2; i <= N2; i++){
    temp[col] = temp[col] + qbnew[i-1][1][1]*tx[col][i][1];
  }
}
```

There are the two dependence relations yielded by Petit

$R1 = \{[col] \rightarrow [col, i']: 1 \leq col \leq N1 \ \&\& \ 2 \leq i' \leq N2\}$ ;

$R2 = \{[col, i] \rightarrow [col, i']: 2 \leq i < i' \leq N2 \ \&\& \ 1 \leq col \leq N1\}$ .

After preprocessing, these relations are of the form.

$R1 = \{[col, -1, 1] \rightarrow [col, i', 2]: 1 \leq col \leq N1 \ \&\& \ 2 \leq i' \leq N2\}$ ;

$R2 = \{[col, i, 2] \rightarrow [col, i', 2]: 2 \leq i < i' \leq N2 \ \&\& \ 1 \leq col \leq N1\}$ .

The results produced by Algorithm 1 are as follows.

1.  $free = TRUE$ ,  $R = \{[col, -1, 1] \rightarrow [col, i', 2]: 1 \leq col \leq N1 \ \&\& \ 2 \leq i' \leq N2\} \cup \{[col, i, 2] \rightarrow [col, i', 2]: 2 \leq i < i' \leq N2 \ \&\& \ 1 \leq col \leq N1\}$ .

2.  $UDS = \{[col, -1, 1]: 1 \leq col \leq N1 \ \&\& \ 2 \leq N2\}$ .

3. Applying the algorithm, published in [17], we yield:

$R^k = \{[col, -1, 1] \rightarrow [col, -1, 1]: k = 0 \ \&\& \ 1 \leq col \leq N1 \ \&\& \ 2 \leq N2\} \cup \{[col, -1, 1] \rightarrow [col, i, 2]: 1 \leq k < i \leq N2 \ \&\& \ 1 \leq col \leq N1\}$ .

4.  $S(k) = \{[col, k+1, 2]: 1 \leq k < N2 \ \&\& \ 1 \leq col \leq N1\} \cup \{[col, -1, 1]: k = 0 \ \&\& \ 1 \leq col \leq N1 \ \&\& \ 2 \leq N2\}$ .

5.  $S(k) = \{[k, col, k+1, 2]: 1 \leq k < N2 \ \&\& \ 1 \leq col \leq N1\} \cup \{[0, col, -1, 1]: 1 \leq col \leq N1 \ \&\& \ 2 \leq N2\}$ .

6. The loop scanning elements of set  $S(k)$  for  $k \geq 0$  and being produced by the codegen function of the Omega library is as follows.

```
if (N2 >= 2) {
  for (t1 = 1; t1 <= N1; t1++) {
    sl(0, t1, -1, 1);
  }
}
if (N1 >= 1) {
  for (k = 1; k <= N2-1; k++) {
    for (t1 = 1; t1 <= N1; t1++) {
      sl(k, t1, k+1, 2);
    }
  }
}
```

7. The final code scanning elements of set  $S(k)$  for  $k \geq 0$  is of the form.

```
if (N2 >= 2) {
  for (t1 = 1; t1 <= N1; t1++) { //parallel loop
```

```

temp[t1] = tx[t1][1][1];
}}
if (N1 >= 1) {
for (k = 1; k <= N2-1; k++) {
for (t1 = 1; t1 <= N1; t1++) { //parallel loop
temp[t1] = temp[t1] + gbnew[k][1]*tx[t1][k+1];
}}}

```

8.  $IND = \{[t1, -1, 1]: 1 \leq t1 \leq N1 \ \&\& \ N2 \leq 1\}$ .

9. The loop scanning elements of set IND and being produced by the codegen function of the Omega library is as follows:

```

if (N2 <= 1) {
for (t1 = 1; t1 <= N1; t1++) { //parallel loop
temp[t1] = tx[t1][1];
}}

```

#### 4. Finding free scheduling in a subspace represented by inner loop nests

For a given set of dependence relations whose union results in relation  $R$  representing all dependences in a loop, it is not always possible to calculate  $R^k$  with affine constraints. The possible reasons are (i) the exact transitive closure of a parameterized affine relation is not affine in general [14] and consequentially  $R^k$  is not affine either; (ii) an algorithm used fails to calculate  $R^k$  (although  $R^k$  exists); (iii) the number of dependence relations is large (for some NAS benchmarks, the number of dependence relations returned by Petit equals to several hundreds and even several thousands) that prevents calculating  $R^k$  in reasonable time due to limited resources of a computer. In such a case, we can apply the algorithm presented below. It extracts free scheduling in a subspace defined by indices of inner nests of a loop (although for the whole iteration space, this algorithm produces some legal scheduling not being the free one).

**Algorithm 2.** Finding free scheduling in a subspace represented by indices of inner loop nests

**Input:** A loop.

**Output:** Code scanning statement instances according to free scheduling in a subspace represented by indices of inner loop nests when appropriate.

**Method:**

1.  $i = 0$ .
2. Extract dependence relations for the iteration subspace being defined by the last  $n-i$  loop indices, where  $n$  is the number of loop nests (loop indices), provided that the first  $i$  indices are represented as symbolic constants.
3. Try to calculate  $R^k$ , where  $R$  is the union of relations returned by step 2. If this attempt results in success, then call Algorithm 1 for the iteration subspace being defined by the last  $n-i$  loop indices provided that the first  $i$  indices are represented as symbolic constants, else go to step 5.
4. Form resulting code as the composition of the first  $i$  sequential nests of the source loop and  $n-i$  nests returned by Algorithm 1, where nests  $1, \dots, i-1$  are sequential while nests  $i+1, \dots, n$  are parallel, the end.
5. If  $n-i \geq 2$ , then  $i = i+1$ , go to step 2; else the end, the algorithm fails to produce any code.

Code produced by Algorithm 2 is legal because it honors all the dependences in the dependence graph described by relation  $R$ . Indeed, dependences within each subspace defined by the last  $n-i$  loop indices are respected by code returned by Algorithm 1 while dependences among those subspaces are honored due to the sequential execution of the first  $i$  outer nests of the source loop.

Let us illustrate Algorithm 2 by means of the following example.

Example 3.

```

for (l = 1; l <= n; l++){
a[l][1] = b[l]; //s1
for (i = 1; i <= n; i++){
b[i] = 0; //s2
for (j = 1; j <= n; j++){
a[i][j] = a[i][j-1] + b[i]; //s3
}}}

```

Petit forms 16 dependence relations for this loop. Algorithm 1 fails to generate parallel code for this example because step 3 does not return  $R^k$  (all techniques known to us fail to produce  $R^k$ ).

Applying Algorithm 2, we yield the following results.

1.  $i = 1$  (for  $i = 0$ , step 3 of Algorithm 1 does not return  $R^k$ ).
2. There are the two dependence relations produced by Petit for the iteration subspace being defined by indices  $i, j$  provided that index  $l$  is the symbolic constant  
 $R1 = \{[i] \rightarrow [i, j]: 1 \leq i \leq n \ \&\& \ 1 \leq j' \leq n\};$   
 $R2 = \{[i, j] \rightarrow [i, j+1]: 1 \leq i \leq n \ \&\& \ 1 \leq j < n\}.$   
 After preprocessing these relations, we have.  
 $R1 = \{[i, -1, 2] \rightarrow [i, j', 3]: 1 \leq i \leq n \ \&\& \ 1 \leq j' \leq n\};$   
 $R2 = \{[i, j, 3] \rightarrow [i, j+1, 3]: 1 \leq i \leq n \ \&\& \ 1 \leq j < n\}.$
3. Using the technique, presented in [17], we yield:  
 $R^k = \{[i, -1, 2] \rightarrow [i, j', 3]: 2 \leq k \leq j' \leq n \ \&\& \ 1 \leq i \leq n\} \cup \{[i, -1, 2] \rightarrow [i, j', 3]: k = 1 \ \&\& \ 1 \leq i \leq n \ \&\& \ 1 \leq j' \leq n\} \cup \{[i, j, 3] \rightarrow [i, k+j, 3]: 1 \leq i \leq n \ \&\& \ 1 \leq k \ \&\& \ k+j \leq n \ \&\& \ 1 \leq j\} \cup \{[i, -1, 2] \rightarrow [i, -1, 2]: k = 0 \ \&\& \ 1 \leq i \leq n\}.$   
 Calling Algorithm 1, we get:  
 $S(k) = \{[k, i, -1, 2]: k = 0 \ \&\& \ 1 \leq i \leq n\} \cup \{[k, i, 1, 3]: k = 1 \ \&\& \ 1 \leq i \leq n\} \cup \{[k, i, k, 3]: 2 \leq k \leq n \ \&\& \ 1 \leq i \leq n\};$ 

```

for (t2 = 1; t2 <= n; t2++) { //parallel loop
  sl(0, t2, -1, 2);
}
for (t2 = 1; t2 <= n; t2++) { //parallel loop
  sl(1, t2, 1, 3);
}
for (t1 = 2; t1 <= n; t1++) {
  for (t2 = 1; t2 <= n; t2++) { //parallel loop
    sl(t1, t2, t1, 3);
  }
}

```
4. The final code is the following.

```

for (l = 1; l <= n; l++){
  a[l][1] = b[l]; // s1
  for (t2 = 1; t2 <= n; t2++) { //parallel loop
    b[t2] = 0; // s2
  }
  for (t2 = 1; t2 <= n; t2++) { //parallel loop
    a[t2][1] = a[t2][0] + b[t2]; // s3
  }
  for (t1 = 2; t1 <= n; t1++) {
    for (t2 = 1; t2 <= n; t2++) { //parallel loop
      a[t2][t1] = a[t2][t1-1] + b[t2]; // s3
    }
  }
}

```

## 5. Related work

The approaches presented in [22–25] build an explicit graph of a subset of the iteration space, with each node representing the instance of a statement. Free scheduling can be found by searching the graph or using the transitive closure of the graph, but dependences are restricted to uniform ones and the problem regarding boundary cases exists.

For the case of a single loop statement with uniform dependences, linear scheduling (optimized Lamport's hyperplane method) is asymptotically optimal for loops with **specific (fat)** domains [9], i.e., the difference between the number of times under the best linear schedule and that under the free schedule is bounded by an expression including in general symbolic constants.

For the case of multiple statements with uniform dependences, the previous result has been extended in work [4] to show that a linear schedule plus shifts leads to asymptotically optimal schedule. For the case of polyhedral approximations of dependences (including direction vectors), Darte and Vivien's algorithm is optimal [26].

For affine dependences, the most powerful algorithm for building scheduling is Feautrier's one based on multi-dimensional affine schedules [2]. But as mentioned by Feautrier, it is not optimal for all codes with affine dependences. However, the dimension of the schedules built by Feautrier's algorithm is minimal for each statement of the loop nest [27].

The approaches published in [5,7,28] present different ways of building affine partition mappings, but none of them guarantees producing free scheduling for the general case of loops with affine dependences.

The approaches that consider loop parallelization with non-uniform dependences [29–31] extract the different amount of loop parallelism, but none of them ensures building the free schedule.

Index splitting [32] is a heuristic procedure for loop parallelization, but it does not answer how much index splitting is necessary to reach free scheduling.

Paper [33] presents a technique permitting for building free scheduling but only for non-parameterized loops.

Paper [13] presents approaches that are also based on using the transitive closure of a relation representing all dependences in a loop. However those techniques extract independent slices comprising loop statement instances (space parti-

tions), i.e., they extract coarse-grained parallelism available in loops, while the algorithms, presented in this paper, aim at extracting fine-grained parallelism (time partitions) available in loops.

Our contribution (Algorithm 1) consists in demonstrating how we can form free-scheduling for loop statement instances provided that the exact power  $k$  of a dependence relation,  $R$ , describing all dependences in a loop,  $R^k$ , can be calculated. The presented approach permits for forming free scheduling even when there does not exist any affine transformation allowing for deriving it (see Example 1). We also show that Algorithm 1 can be applied for building legal scheduling (not always free-scheduling) when on its input we use an over-approximation of  $R^k$  with a parametrized upper bound of  $k$ . Such scheduling also can be used for loop parallelization but with a larger number of synchronization events.

## 6. Experimental results

The goals of our experiments were estimating: (i) the effectiveness of the algorithms, (ii) the efficiency of parallel code produced by the presented algorithms, and (iii) the time complexity of the algorithms. In order to achieve them, we have implemented the presented algorithms in the form of a tool. For this purpose, we applied the Omega library [34] extended by additional functions permitting for preprocessing relations and sets, calculating  $R^k$  on the basis of converting  $R^+$  to  $R^k$  (using the way published in [17]), calculating set  $S(k)$ , and generating C-like pseudo-code scanning loop statement instances according to (free) scheduling. The tool can be downloaded from <http://sourceforge.net/projects/issf>.

Using this tool, we have experimented with loops of the NAS 3.2 [35] benchmark suite. We have studied only those loops for which Omega's dependence analyzer Petit is able to carry out exact dependence analysis (Petit fails to analyse loops containing the "break", "goto", "continue", and "exit" statements). From 431 loops of the NAS benchmark suite, Petit is able to carry out dependence analysis for 257 loops and dependences are present in 133 loops only.

### 6.1. Effectiveness of the algorithms

For 133 loops qualified for experiments, the tool is able to calculate  $R^k$  for 67 ones (for 61 loops, it produces exact  $R^k$  while for 6 ones it returns an over-approximation). For the rest of 66 loops, the tool fails to calculate  $R^k$ . Applying Algorithm 2, we are able to calculate  $R^k$  in a subspace formed by inner loops. Columns 1 and 2 of Table 1 present the details, where  $i = 0, 1, 2, 3$  means that  $R^k$  is calculated for an iteration subspace defined by  $n-i$  indices of inner loops.

Columns 3 and 4 of Table 1 show the number and percentage of loops exposing parallelism. For 21 from 61 loops, Algorithm 1 produces free scheduling but does not expose any parallelism (there exists a single statement instance for each time partition) despite the fact that it is possible to calculate  $R^k$  for these loops.

On the basis of experiments carried out with NAS benchmarks, we can derive the following conclusions. The most important step in Algorithms 1 and 2 is calculating  $R^k$ . When we are able to calculate  $R^k$  exactly, we are able also to form free-scheduling. Calculating the power  $k$  of a dependence relation  $R$ ,  $R^k$ , is presented in papers [14–18]. Some of known algorithms to calculate  $R^k$  are implemented in the Omega and ISL libraries [17,34]. Our experiments with those algorithms and libraries on NAS benchmarks permit us to derive the following conclusions about their limitations.

1. For some relations, the constraints of  $R^k$  are non-linear despite the fact that the constraints of  $R$  are affine, for details see [14]. As a consequence, the algorithms presented in the paper do not permit for producing free-scheduling (because all well-known techniques of generating code are based on scanning sets with affine constraints, for example [19–21]), however they can produce legal scheduling after converting exact non-linear  $R^k$  to an over-approximation of  $R^k$  with affine constraints where the upper value of  $k$  is parametrically bounded (for example,  $k \leq n_1 * n_2$ , where  $n_1, n_2$  are parameters).
2. Applying some algorithms of calculating the power  $k$  of a dependence relation  $R$ ,  $R^k$ , results in a huge number of conjunctions representing  $R^k$  that may require a lot of memory and/or processor resources. This leads to the interruption of running a function implementing an algorithm of calculating  $R^k$  because of memory/time limitations. Usually such an effect takes place for dependence relations being represented by unions of many single conjunct relations (several thousand), several tens of NAS loops expose such relations.
3. Omega and ISL are academic software not being well tested. Sometimes they return assertions while executing functions implementing algorithms of calculating  $R^k$ .

**Table 1**  
Effectiveness of the algorithms.

Number of nests	$R^k$ , number (%)	Algorithm 1, number (%)	Algorithm 2, number (%)
$n$	61, 45.9 (exact)	40, 30.1 (exact)	–
$n$	6, 4.5 (overapproximation)	6, 4.5 (overapproximation)	–
$n - 1$	2, 1.5	–	2, 1.5
$n - 2$	19, 14.3	–	19, 14.3
$n - 3$	28, 21.1	–	28, 21.1

Summing up, we may conclude that there is a strong need in improving existing algorithms and libraries permitting for calculating relation  $R^k$ .

## 6.2. Efficiency of parallel code

To assess the efficiency of code produced by Algorithm 1, the following criteria were taken into account for choosing NAS loops: (i) a loop must be computatively **heavy** (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified), (ii) code produced by Algorithm 1 must be parallel (there are NAS loops for which there exists a single statement instance for each time partition), (iii) structures of chosen loops must be different (there are many NAS loops of a similar structure). Applying these criteria, we selected the following NAS loops: *FT\_auxfnct\_2*, *UA\_diffuse\_4*, *UA\_transfer\_4*, *MG\_mg\_3*, and *UA\_setup\_16*.

Codes, produced by Algorithm 1 for these loops, were manually converted by us to parallel programs to be executed on a graphic card: NVIDIA 8800 GTS, 96 GPU 1.6 GHz, GDDR3 512 MB. For this purpose, we have used the NVIDIA CUDA library [36].

Table 2 presents results of time measuring (in seconds) for executing the chosen loops on the graphic card. The execution time of a loop consists of the time of data transfer to/from the graphic card and the time of calculations. Experiments were carried out for two different values of the upper bounds of loop indices (see column 2). The time of data transfer (see columns 3–6) comprises the times of [36]: allocation, sending data to the graphic card, and fetching data memory of the graphic card. Column 6 presents the sum of those times as the time of data transfer. It is worth to note that the time of data transfer does not depend on the number of GPU cores [36]. Columns 7–10 show the time of calculations (not including the time of data transfer) for 1, 2, 8, 32, 64, and 96 GPU cores.

Table 3 exhibits the execution time (the sum of the time of data transfer and the time of calculations), speed-up, and efficiency for different numbers of GPU cores. Fig. 5 illustrates the data presented in Table 3 in a graphical way. The results in Table 3 demonstrate that parallel loops formed on the basis of parallel code produced by Algorithm 1: (1) permit for utilizing many GPU cores (up to 96 under our experiments); (2) speed-up increases with increasing the number of GPU cores (up to 96 under our experiments).

Summing up, we may conclude that extracting free scheduling has not only theoretical importance permitting for answering the question: how much fine-grained parallelism does not extracted by well-known approaches, but also practical significance allowing for producing parallel programs for graphic cards. Such programs demonstrate satisfactory efficiency when each thread, running a part of a parallel program, gets work whose execution requires more time than that required for thread **management** (creating, initializing, synchronizing, terminating, etc.).

## 6.3. Evaluating the time complexity of the algorithms

To evaluate the time complexity of the algorithms presented in this paper, for each loop of the NAS suite qualified for experiments, we measured the time that takes the tool, implementing the presented algorithms, from the beginning of a dependence analysis to the end of code generation on a machine with the following features: Intel Core 2 Duo 2.34 GHz, 2 GB RAM, Ubuntu Linux.

This time is composed of the times of: a dependence analysis, calculating transitive closure of  $R$ ,  $R^+$ , and the power  $k$  of  $R$ ,  $R^k$ , calculating a set  $S(k)$ , and generating final code. The time of calculating  $R^+$  for different relations may differ considerably: from several milliseconds to several hours. This depends on the number of single conjunct relations composing relation  $R$  (some NAS loops expose several hundreds and even thousands such relations) as well as on the kind of relation constraints defining: (i) whether dependences are uniform or not; (ii) what are elements of distance vectors (when all elements are positive, there is no problem with calculating the transitive closure of a relation); (iii) what are the domain and range of depen-

**Table 2**  
Results of time measuring for NAS benchmarks.

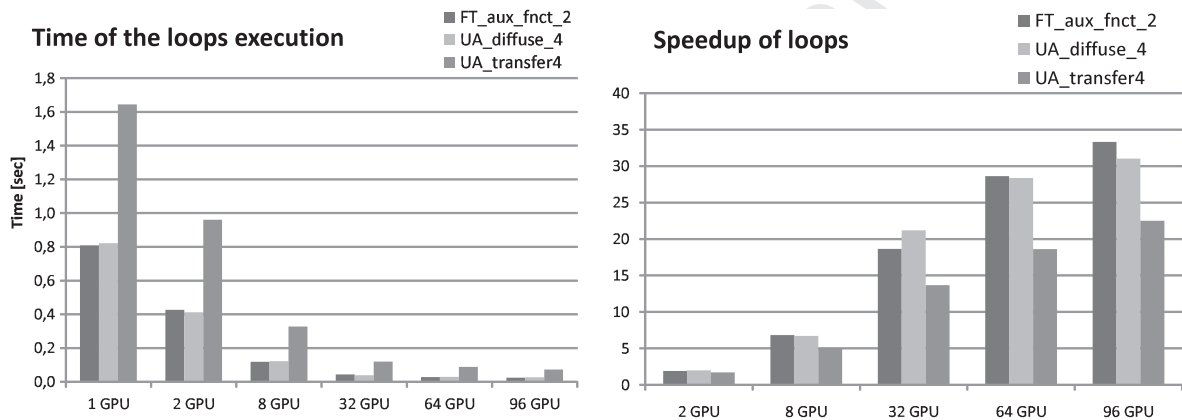
Loop	Upper bounds	Time of data transfer (s)				Time of calculations (s)					
		Alloc	Send	Fetch	Total	1 GPU	2 GPUs	8 GPUs	32 GPUs	64 GPUs	96 GPUs
FT_auxfnct_2	N1, N2, N3 = 100	0.00035	0.00620	0.00690	0.01345	0.7950	0.4130	0.1050	0.0299	0.0148	0.0108
	N1, N2, N3 = 200	0.00044	0.04800	0.05100	0.09944	6.7070	3.4560	0.8710	0.2241	0.1217	0.1050
UA_diffuse_4	N1, N2 = 64; N3, N4 = 10	0.00030	0.00318	0.00172	0.00520	0.1959	0.0959	0.0264	0.0163	0.0119	0.0083
	N1, N2 = 128; N3, N4 = 10	0.00037	0.01030	0.00600	0.01667	0.8050	0.3960	0.1060	0.0221	0.0123	0.0098
UA_transfer_4	N1 = 1, N2 = 1000	0.00032	0.00629	0.00035	0.00696	0.4050	0.2330	0.0620	0.0182	0.0154	0.0120
	N1, N2 = 2000	0.00038	0.02231	0.00041	0.02310	1.6210	0.9380	0.3040	0.0972	0.0652	0.0500
MG_mg_3	N1 = 1, N2 = 1,000,000	0.04439	0.02785	0.02932	0.10156	1.0675	0.5418	0.1361	0.0307	0.0233	0.0201
	N1, N2 = 500,000	0.04442	0.05129	0.06355	0.15926	2.1324	1.0834	0.2718	0.1219	0.0725	0.0500
UA_setup_16	N1, N2, N3 = 100	0.04462	0.00012	0.00050	0.04523	0.4276	0.2454	0.0639	0.0202	0.0132	0.0116
	N1, N2, N3 = 200	0.04393	0.00042	0.00077	0.04512	3.4016	1.9251	0.4917	0.1388	0.0801	0.0721



**Table 3**

Time, speed-up, and efficiency.

Loop	Upper bounds	1 GPU	2 GPUs			8 GPUs			32 GPUs			64 GPUs			96 GPUs		
		Time (s)	Time (s)	S	E	Time (s)	S	E	Time (s)	S	E	Time (s)	S	E	Time (s)	S	E
FT2	N1, N2, N3 = 100	0.808	0.426	1.90	0.95	0.118	6.83	0.85	0.043	18.65	0.19	0.028	28.62	0.30	0.024	33.34	0.35
	N1, N2, N3 = 200	6.806	3.555	1.91	0.96	0.970	7.01	0.88	0.324	21.04	0.22	0.221	30.78	0.32	0.204	33.29	0.35
UAd4	N1, N2 = 64; N3, N4 = 10	0.201	0.101	1.99	0.99	0.032	6.36	0.80	0.022	9.35	0.10	0.017	11.76	0.12	0.014	14.90	0.16
	N1, N2 = 128; N3, N4 = 10	0.822	0.413	1.99	0.99	0.123	6.70	0.84	0.039	21.19	0.22	0.029	28.36	0.30	0.026	31.04	0.32
UAt4	N1, N2 = 1000	0.412	0.240	1.72	0.86	0.069	5.97	0.75	0.025	16.37	0.17	0.022	18.42	0.19	0.019	21.73	0.23
	N1, N2 = 2000	1.644	0.961	1.71	0.86	0.327	5.03	0.63	0.120	13.67	0.14	0.088	18.62	0.19	0.073	22.49	0.23
MG3	N1 = 1, N2 = 1,000,000	1.169	0.643	1.82	0.91	0.238	4.92	0.61	0.132	8.84	0.09	0.125	9.36	0.10	0.122	9.61	0.10
	N1 = 1, N2 = 500,000	2.292	1.243	1.84	0.92	0.431	5.32	0.66	0.281	8.15	0.08	0.232	9.89	0.10	0.209	10.95	0.11
UAs16	N1, N2, N3 = 100	0.473	0.291	1.63	0.81	0.109	4.33	0.54	0.065	7.23	0.08	0.058	8.09	0.08	0.057	8.32	0.09
	N1, N2, N3 = 200	3.447	1.970	1.75	0.87	0.537	6.42	0.80	0.184	18.74	0.20	0.125	27.53	0.29	0.117	29.41	0.31

**Fig. 5.** Time of execution and speed-up of the *FT\_auxfnct\_2*, *UA\_diffuse\_4* and *UA\_transfer\_4* loops using 1, 2, 8, 32, 64, and 96 GPU cores.

dence relations. Under our experiments, we have limited the time of calculating  $R^+$  to 10 s, if during this time  $R^+$  is not returned by a correspondent function, the tool terminates calculations. Table 1 presents statistics demonstrating how many loops can be parallelized under such a limitation (maximum 10 s for calculating transitive closure).

When the tool is able to produce final code under the above limitation (from 133 loops qualified for experiments), the time of producing this code varies from 0.02 to several seconds. Table 4 presents the results for the five computationally heavy NAS loops. The first column contains the name of a loop while the remaining columns expose time measurement results (in seconds): of a dependence analysis, calculating  $R^+$  and  $R^k$  ( $R^+$  and  $R^k$ ), calculating set  $S(k)$ , final code generation, and the total time.

In our implementation, Petit was used as the dependence analyzer.  $R^+$  and  $R^k$  were calculated by functions written by us (these functions are added to the Omega + library, see [37]). Calculating a set  $S(k)$  was carried out by a function written by us, and final code generation was carried out by an Omega library function.

**Table 4**

Time complexity of the algorithms.

Loop	Dep. anal. (s)	$R^+$ and $R^k$ (s)	$S(k)$ (s)	Final code (s)	Total time (s)
FT_auxfnct_2	0.01	0.02	0.01	0.01	0.05
UA_diffuse_4	0.01	0.03	0.02	0.01	0.08
UA_transfer_4	0.01	0.07	0.03	0.01	0.12
MG_mg_3	0.01	0.02	0.02	0.02	0.07
UA_setup_16	0.01	0.03	0.01	0.01	0.6



Carried out experiments indicate that prior to calculate  $R^+$  it is strongly recommended to remove redundant dependence relations and/or redundant dependences represented by a particular dependence relation exposed for a loop. This will permit us: (i) to reduce the time of calculating  $R^+$ ; (ii) to calculate  $R^+$  by means of known techniques because the tool often fails to calculate  $R^+$  due to the huge number of single conjunct relations composing a dependence relation describing all the dependences in a loop.

Summing up, we may conclude that extracting free scheduling has not only theoretical meaning (it represents all fine-grained loop parallelism) but also has practical importance allowing for producing effective parallel programs for multi-core graphic cards.

## 7. Conclusion

In this paper, we presented the algorithms that permit us to build (free) scheduling for parameterized arbitrarily nested loops in the whole iteration space or in a subspace defined by indices of inner loop nests. The necessary condition to apply them is the possibility of the calculation of the exact power  $k$  of relation  $R$  describing all dependences in a loop or its over-approximation with affine constraints. Algorithm 1 produces the free schedule when it is possible to compute exact  $R^k$  with affine constraints otherwise it returns a legal schedule.

Our contribution consists in demonstrating how we can form free-scheduling for loop statement instances provided that the exact power  $k$  of a dependence relation,  $R$ , describing all dependences in a loop,  $R^k$ , can be calculated (Algorithm 1). The presented approach permits for forming free scheduling even when there does not exist any affine transformation allowing for deriving it (see Example 1). We also show that Algorithm 1 can be applied for building legal scheduling (not always free-scheduling) when we use an over-approximation of  $R^k$  with a parametrized upper bound of  $k$ . Such scheduling also can be used for loop parallelization but with a larger number of synchronization events.

There are tasks to be resolved in the future to strengthen the power of the presented algorithms and justify their application for parallelizing real-life codes: (1) developing advanced techniques for calculating the power  $k$  of relation  $R$  being a union of affine relations representing dependences in the loop; (2) when the free schedule is represented by non-linear forms, techniques should be developed to generate code enumerating statement instances under the free schedule; (3) effective techniques aimed at reducing redundant dependences should be defined and applied to the loop prior to apply the algorithms presented in this paper.

## References

- [1] P. Feautrier, Some efficient solutions to the affine scheduling problem: I. one-dimensional time, *Int. J. Parallel Prog.* 21 (5) (1992) 313–348.
- [2] P. Feautrier, Some efficient solutions to the affine scheduling problem: II. multi-dimensional time, *Int. J. Parallel Prog.* 21 (5) (1992) 389–420.
- [3] A. Darte, Y. Robert, Constructive methods for scheduling uniform loop nests, *IEEE Trans. Parallel Distrib. Syst.* 5 (1994) 814–822.
- [4] P. Le Gouëslier d'Argence, Affine scheduling on bounded convex polyhedric domains is asymptotically optimal, *Theor. Comput. Sci.* 196 (1998) 395–415.
- [5] A.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*, ACM Press, 1999, pp. 228–237.
- [6] A. Darte, Y. Robert, F. Vivien, *Scheduling and Automatic Parallelization*, Birkhäuser, 2000.
- [7] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: *Conference on Programming Language Design and Implementation*, ACM, 2008, pp. 101–113.
- [8] K. Kennedy, J.R. Allen, *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*, Morgan Kaufman Publishers Inc., San Francisco, CA, USA, 2002.
- [9] A. Darte, L. Khachiyan, Y. Robert, Linear scheduling is nearly optimal, *Parallel Proc. Lett.* 1 (2) (1991) 73–81.
- [10] W. Pugh, D. Wonnacott, An exact method for analysis of value-based array data dependences, in: *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Springer-Verlag, 1993.
- [11] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, The omega library interface guide, Tech. rep., College Park, MD, USA (1995).
- [12] S.H.L.M. Surhone, M.T. Tennoe, Presburger Arithmetic, VDM Verlag Dr. Mueller AG & Co. Kg, 2010, ISBN: 6133083557.
- [13] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski, K. Siedlecki, Coarse-grained loop parallelization: iteration space slicing vs affine transformations, *Parallel Comput.* 37 (2011) 479–497.
- [14] W. Kelly, W. Pugh, E. Rosser, T. Shpeisman, Transitive closure of infinite graphs and its applications, *Int. J. Parallel Prog.* 24 (6) (1996) 579–598.
- [15] A. Beletska, D. Barthou, W. Bielecki, A. Cohen, Computing the transitive closure of a union of affine integer tuple relations, in: *COCOA 2009: Third International Conference on Combinatorial Optimization and Applications*, vol. 5573 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 98–109.
- [16] W. Bielecki, T. Klimek, M. Palkowski, A. Beletska, An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations, in: *COCOA 2010: Fourth International Conference on Combinatorial Optimization and Applications*, *Lecture Notes in Computer Science*, Vol. 6508/2010, 2010, pp. 104–113.
- [17] S. Verdoolaege, Integer set library – manual, Tech. rep., 2011, <[www.kotnet.org/skimo/jisl/manual.pdf](http://www.kotnet.org/skimo/jisl/manual.pdf)>.
- [18] W. Bielecki, T. Klimek, K. Trifunovic, Calculating exact transitive closure for a normalized affine integer tuple relation, *Electron. Notes Disc. Math.* 33 (2009) 7–14.
- [19] C. Bastoul, Code generation in the polyhedral model is easier than you think, in: *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, 2004, pp. 7–16.
- [20] F. Quilleré, S. Rajopadhye, D. Wilde, Generation of efficient nested loops from polyhedra, *Int. J. Parallel Prog.* 28 (5) (2000) 469–498.
- [21] N. Vasilache, C. Bastoul, A. Cohen, Polyhedral code generation in the real world, in: *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, Springer-Verlag, Vienna, Austria, 2006, pp. 185–201.
- [22] D.-K. Chen, Compiler optimizations for parallel loops with fine-grained synchronization, Ph.D. thesis, Champaign, IL, USA, 1994, uMI Order No. GAX95-12325.
- [23] V. Krothapalli, P. Sadayappan, Removal of redundant dependences in doacross loops with constant dependences, *IEEE Trans. Parallel Distrib. Syst.* 2 (1991) 281–289.
- [24] S.P. Midkiff, D.A. Padua, A comparison of four synchronization optimization techniques, *ICPP* (2) (1991) 9–16.
- [25] S.P. Midkiff, D.A. Padua, Compiler algorithms for synchronization, *IEEE Trans. Comput.* 36 (1987) 1485–1495.

- [26] A. Darte, F. Vivien, Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs, in: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 281–291.
- [27] F. Vivien, On the optimality of Feautrier's scheduling algorithm, in: Euro-Par 2002: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer-Verlag, 2002, pp. 299–308.
- [28] W. Kelly, W. Pugh, A framework for unifying reordering transformations, Tech. rep., Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-92-126.1, College Park, MD, USA, 1993.
- [29] S. Punyamurtula, V. Chaudhary, J. Ju, S. Roy, Compile time partitioning of nested loop iteration spaces with non-uniform dependences, J. Parallel Algor. Appl. 13 (1996) (special issue on Optimizing Compilers for Parallel Languages).
- [30] T.H. Tzen, L.M. Ni, Dependence uniformization: a loop parallelization technique, IEEE Trans. Parallel Distrib. Syst. 4 (1993) 547–558.
- [31] A. Zaafrani, M.R. Ito, Parallel region execution of loops with irregular dependencies, in: Proc. Int. Conf. Parallel Processing ICPP 1994, vol. 2, 1994, pp. 11–19.
- [32] M. Griebl, P. Feautrier, C. Lengauer, Index set splitting, Int. J. Parallel. Programming 28 (2000) 607–631.
- [33] V. Beletsky, K. Siedlecki, Finding free schedules for non-uniform loops, in: Euro-Par 2003 Parallel Processing, Lecture Notes in Computer Science, vol. 2790/2003, 2003, pp. 297–302.
- [34] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, The Omega project, <<http://www.cs.umd.edu/projects/omega/release-1.0.html>>.
- [35] NAS benchmarks suite, <<http://www.nas.nasa.gov>>.
- [36] NVIDIA, NVIDIA CUDA Programming Guide 2.0, 2008, <[http://www.nvidia.pl/object/cuda\\_home\\_new\\_pl.html](http://www.nvidia.pl/object/cuda_home_new_pl.html)>.
- [37] C. Chen, School of Computing University of Utah, February 2011, <<http://www.cs.utah.edu/~chunchen/omega/>>.