DE GRUYTER
OPEN

Int. J. Appl. Math. Comput. Sci., 2016, Vol. 26, No. 4, 919–939
DOI: 10.1515/amcs-2016-0065

amcs

TILING ARBITRARILY NESTED LOOPS BY MEANS OF THE TRANSITIVE CLOSURE OF DEPENDENCE GRAPHS

WŁODZIMIERZ BIELECKI [a], MAREK PAŁKOWSKI [a,*]

[a]Faculty of Computer Science
West Pomeranian University of Technology, Żołnierska 49, 71-210 Szczecin, Poland
e-mail: {wbielecki,mpalkowski}@wi.zut.edu.pl

A novel approach to generation of tiled code for arbitrarily nested loops is presented. It is derived via a combination of the polyhedral and iteration space slicing frameworks. Instead of program transformations represented by a set of affine functions, one for each statement, it uses the transitive closure of a loop nest dependence graph to carry out corrections of original rectangular tiles so that all dependences of the original loop nest are preserved under the lexicographic order of target tiles. Parallel tiled code can be generated on the basis of valid serial tiled code by means of applying affine transformations or transitive closure using on input an inter-tile dependence graph whose vertices are represented by target tiles while edges connect dependent target tiles. We demonstrate how a relation describing such a graph can be formed. The main merit of the presented approach in comparison with the well-known ones is that it does not require full permutability of loops to generate both serial and parallel tiled codes; this increases the scope of loop nests to be tiled.

Keywords: tiling, transitive closure, source-to-source compiler, polyhedral model, iteration space slicing.

1. Introduction

In this paper, we consider loop nest tiling techniques aimed at automatic generation of tiled code by means of optimizing compilers. Tiling (Irigoin and Triolet, 1988; Wolf and Lam, 1991; Ramanujam and Sadayappan, 1992; Xue, 1996; Bondhugula *et al.*, 2008a; Griebl, 2004; Lim *et al.*, 1999) is a very important iteration reordering transformation for both improving data locality and coarsening the granularity of parallelism.

Tiling for improving locality groups loop statement instances into smaller blocks (tiles) allowing reuse when the block fits in local memory. It partitions a loop nest iteration space into smaller blocks (tiles) so as to help ensure the data used in a loop nest stays in the cache until it is reused. In a parallel tiled code, tiles are considered indivisible macro statements. This coarsens the granularity of parallel applications, which often leads to improving the performance of an application running in parallel computers.

Loop tiling is beneficial for parallel computers with hierarchical memory: computers with both shared and distributed memory (Xue, 2012; Tang and Xue, 2000) as well as accelerators, for example, graphic cards (Grosser *et al.*, 2013). In this paper, we demonstrate how the introduced tiling approach enhances code locality and allows parallelism extraction for multiprocessor computers with shared memory.

Tiling can be used for the optimization of any numerical application provided that its code includes loop nests. This is particularly true for numerically intensive codes (Kowarschik and Weiß, 2003). Such codes occur in almost all science and engineering disciplines, e.g., computational fluid dynamics, computational physics, mechanical engineering. Almost all numerical algorithms can be tiled: linear algebra, image processing, combinatorial optimization, computational geometry, stencil algorithms, system identification, genetic and combinatorial algorithms (Jeffers and Reinders, 2015; Leader, 2004; Greenbaum and Chartier, 2012; Błaszczyk *et al.*, 2007; Campbell, 2001; Maciążek *et al.*, 2015; Zdunek, 2014).

To our best knowledge, well-known automatic tiling techniques are based on linear or affine transformations of program loops (Irigoin and Triolet, 1988; Wolf and Lam, 1991; Ramanujam and Sadayappan, 1992; Bondhugula *et al.*, 2008a; Griebl, 2004; Lim *et al.*, 1999; Xue, 1997;

*Corresponding author

Andonov *et al.*, 2001; Bastoul and Feautrier, 2003). To generate tiled code, first affine transformations, allowing for producing a band of fully permutable loops, are automatically formed, and then this band is transformed into tiled code.

Although state-of-the-art approaches are able to tile a number of loop nests, there are cases where they fail to generate any tiled code (Mullapudi and Bondhugula, 2014; Wonnacott *et al.*, 2015). The reason is applying conservative (sufficient but not necessary) conditions to guarantee the validity of tiled code. Automatic approaches for tiling have to guarantee that the tiling transformation respects all dependences in the original program. For this purpose, validity constraints are used.

The well-known validity constraint proposed by Irigoin and Triolet (1988) only allows for tiling bands of loops on which dependences have non-negative components, i.e., tiling can be applied only for bands of fully permutable loops. The validity condition presented by Xue (2012) checks for the lexicographic non-negativity of inter-tile dependences.

Applying these conservative conditions to guarantee tiled code validity may miss valid tiling transformations, which prevents tiling for some important loop nests (Mullapudi and Bondhugula, 2014; Wonnacott *et al.*, 2015).

In this paper, we discuss a way to improve current automatic tiling techniques. We demonstrate that applying the transitive closure of dependence graphs for tiling allows generating target tiles such that there is no cycle in the corresponding inter-tile dependence graph. It is well-known that, for such a case, a valid schedule of target tiles exists, i.e., a valid serial or parallel tiled code can be generated (Mullapudi and Bondhugula, 2014). Thus, we suggest a more general scheme of automatic tiling, allowing increasing the scope of loop nests to be tiled. Such tiling can be applied to bands of original loops not being fully permutable.

In our previous paper (Bielecki and Palkowski, 2015), we presented a novel approach to automatic generation of tiled code for perfectly nested loops in which all assignment statements are contained in the innermost loop. It is based on the transitive closure of a loop nest dependence graph and produces tiled code even when there does not exist any affine transformation allowing producing a band of fully permutable loops. According to that approach, we first form fixed rectangular original tiles and next examine whether all loop nest dependences are respected under the lexicographic order of tile enumeration. If so, this means that all original tiles are valid, and hence code generation is straightforward. Otherwise, we correct the original tiles so that all target tiles are valid, i.e., the lexicographic enumeration order of target tiles respects all dependences available in the original loop nest. The final step is code generation representing target (corrected) tiles.

In real programs, many important loops are imperfectly nested (that is, one or more assignment statements are contained in some but not all of the loops of the loop nest) (Ahmed *et al.*, 2000). According to a study by Sass and Mutka (1994), a majority of loops in scientific code are imperfectly nested.

In this paper, we present an extended approach which can be applied to tile both perfectly and arbitrarily nested loops. This allows us to considerably increase the scope of the approach applicability, because in practice, most loop nests are arbitrarily nested.

The contributions of this paper over previous work are as follows:

- an algorithm demonstrating how the iteration space slicing framework can be combined with the polyhedral model to improve the effectiveness (the scope of applicability) of tiling transformations for arbitrarily nested loops;
- clarification that this improvement is due to the fact that the presented algorithm can be directly applied to bands of original loops not being fully permutable, i.e., it does not require finding affine transformations to transform the original loop nest to a loop nest with a band(s) of fully permutable loops;
- demonstration of how the generated tiled code can be parallelized;
- development and presentation of the publicly available source-to-source TRACO compiler implementing the introduced algorithm;
- evaluation of the effectiveness of the introduced algorithm and the speed-up of tiled codes produced by means of the presented algorithm.

The rest of the paper is organized as follows. Section 2 contains background. Section 3 describes the concept of loop nest tiling and presents a formal algorithm to produce tiled code based on the transitive closure of a loop nest dependence graph. Section 4 clarifies how the generated tiled code can be parallelized. Section 5 shows how the introduced approach can be applied to a real-life code. Section 6 discusses the results of experiments. Section 7 presents related work. Section 8 concludes and introduces future work.

2. Background

In this paper, we deal with affine loop nests where, for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters (defining loop index bounds), and the loop steps are known constants.

A dependence analysis is required to carry out a valid loop transformation. Two statement instances I and J are dependent if both access the same memory location and if at least one access is a write. I and J are called the source and target of a dependence, respectively, provided that I is lexicographically less than J ($I \prec J$, i.e., I is executed before J).

The algorithm presented in this paper requires an exact representation of dependences and consequently an exact dependence analysis which detects a dependence if and only if it actually exists. To describe and implement the algorithm, we have chosen the dependence analysis proposed by Pugh and Wonnacott (1993), where dependences are represented by dependence relations.

A dependence relation is a tuple relation of the form [*input list*]→[*output list*]: *formula*, where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples, and *formula* describes the constraints imposed upon input and output lists and is a Presburger formula built of constraints represented by algebraic expressions, using logical and existential operators (Pugh and Wonnacott, 1993).

A dependence relation is a mathematical representation of a data dependence graph whose vertices correspond to loop statement instances while edges connect dependent instances. The input and output tuples of a relation represent dependence sources and destinations, respectively; the relation constraints point out instances which are dependent.

In the presented algorithm, standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference ($-$), domain (dom R), range (ran R), relation application ($S' = R(S) : e' \in S'$ iff exists e s.t. $e \to e' \in R, e \in S$). In detail, the description of these operations is presented by Kelly *et al*. (1995) as well as Pugh and Wonnacott (1993).

The positive transitive closure of a given relation R, R^+, is defined as follows (Kelly *et al*., 1995):

$$R^+ = \{e \to e' : e \to e' \in R \vee \exists e'' \text{ s.t. } e \to e'' \in R \wedge e'' \to e' \in R^+\}. \quad (1)$$

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

The transitive closure, R^*, is defined as

$$R^* = R^+ \cup I, \quad (2)$$

where I is 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.

Techniques aimed at calculating the transitive closure of a dependence graph, which in general is parametric, are presented by Kelly *et al*. (1996), Bielecki *et al*. (2010) and Verdoolaege *et al*. (2011), and they are beyond the scope of this paper. We would like to note that the existing algorithms return either exact transitive closure or its over-approximation. The former means that transitive closure represents only existing dependences in the original loop nest, while the latter implies that the representation of transitive closure includes both all existing and false (non-existing) dependences. Both representations can be used in the presented algorithm but, if we use an over-approximation of transitive closure, tiled code will be less optimal: it will allow less code locality and/or parallelization.

The paper by Bielecki *et al.* (2014) presents the time of transitive closure calculation for NPBs (NAS, 2015). It depends on the number of dependence relations extracted for a loop nest and can vary from milliseconds to several minutes (in very rare cases when the number of dependence relations is equal to hundreds or thousands).

The algorithm presented in this paper requires applying the union, composition, and application operations on dependence relations and the difference operation on sets. 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 can be violated. To allow applying the operations mentioned above on relations and sets, we have to preprocess them. Preprocessing makes the sizes of input and output tuples of each dependence relation the same by inserting the value -1 into the rightmost positions of the corresponding tuples as well as inserts identifiers of loop nest statements into the last positions of input and output tuples. Loop nest statement identifiers are necessary for code generation. The preprocessing procedure for relations is presented below. The preprocessing of a set differs from that of a relation by preprocessing only one

Procedure 1. Dependence relation preprocessing.

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

Output: Set S of preprocessed dependence relations.

Method:

foreach relation $R_{i,j} \in S$ **do**

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

Tiled code can be generated manually or automatically. The approach introduced in this paper is to generate tiled code automatically via its implementation in optimizing compilers. Below, we recall how tiled code can be generated automatically by means of affine transformations. For this purpose, let us consider the following example.

Example 1. Consider the code

```
for(i=0; i<=3; i++)
 for(j=0; j<=3; j++)
   a[i][j]=a[i][j+1]+a[i+1][j]
   +a[i+1][j-1];
```

In this paper, we use the syntax of the Barvinok tool (Verdoolaege, 2012) to present results of calculations on relations and sets.

The following three relations describe all the dependences in the working loop nest:

```
R1:={[i,j]->[i,j+1]:0<=i<=3 and
                    0<=j<=2};
R2:={[i,j]->[i+1,j]:0<=i<=2 and
                    0<=j<=3};
R3:={[i,j]->[i+1,j-1]: 0<=i<=2 and
                       1<=j<=3}.
```

This loop nest can be tiled by means of affine transformations. The classical way is to skew the loop nest iteration space and then generate tiled code. Applying the affine transformation presented with the matrix

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

to Example 1, we get a fully permutable loop nest, which next can be tiled to get the loop nest below, where the tiles are of size 2×2 (the code is produced by means of the optimizing compiler PLUTO (Bondhugula *et al.*, 2008a)):

```
for(t1=0;t1<=1;t1++)
 for(t2=t1;t2<=t1+2;t2++)
  for(t3=max(2*t1,2*t2-3);
         t3<=2*t1+1;t3++)
   for(t4=max(2*t2,t3);
       t4<=min(2*t2+1,t3+3); t4++)
    a[t3][-t3+t4]= a[t3][-t3+t4+1]
                  +a[t3+1][-t3+t4]
                  +a[t3+1][-t3+t4-1];
```

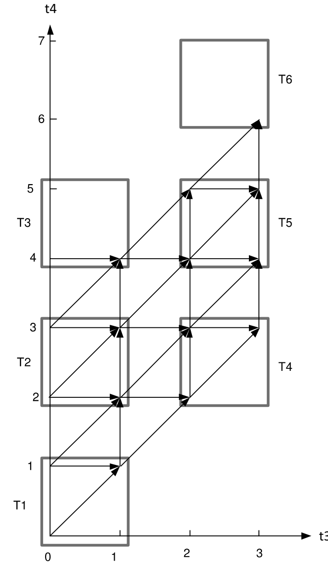

Fig. 1. Tiles generated by means of the affine transformation.

Figure 1 illustrates tiles represented with the code above in a graphical way. ♦

3. Tiling algorithm

In this section, we first present the section objective and basic concepts, then we discuss a tiling idea based on transitive closure, illustrate this idea by means of a working example, and finally introduce a formal tiling algorithm.

3.1. Section objective and basic concepts. The goal of this section is to demonstrate how the loop nest of depth d below:

```
for(i1=lb1; i1<=ub1; i1++){
  S1a(i1);
  for(i2=lb2; i2<=ub2; i2++){
    S2a(i1,i2);
    ........................
      for(id=lbd; id<=ubd; id++){
        Sda(i1,i2, ..., id);
      }
    ........................
  S2b(i1,i2);
 }
 S1b(i1);
}
```

where each statement S can be compound, i.e., it may consist of two or more loops, can be transformed to a valid tiled loop nest applying the transitive closure of a dependence graph. For the arbitrarily nested loop nest, a statement can be of type a or type b. Statement S_{ia}, which is surrounded by i loops and located before loops $i+1, i+2, \ldots, q$ is of type a, while statement S_{ib}, which is surrounded by i loops and located after loops $i+1, i+2, \ldots, q$ is of type b.

A tiled loop nest is valid if all original loop nest dependences are preserved under the lexicographic execution order of both tiles and statement instances within each tile, i.e., for any two dependent statement instances in the original loop nest, in the tiled loop nest, these statement instances are also dependent in the same order.

Let, for loop nest statement i, set $TILE_i(\boldsymbol{II}_i)$ include loop nest statement instances belonging to the original rectangular tile whose identifier is represented with parametric vector $\boldsymbol{II}_i$. The mathematical representation of this set is the following: $TILE_i(\boldsymbol{II}_i) = \{[\boldsymbol{I}_i] \mid \boldsymbol{B}_i * \boldsymbol{II}_i + \boldsymbol{LB}_i \leq \boldsymbol{I}_i \leq \min(\boldsymbol{B}_i * (\boldsymbol{II}_i + \boldsymbol{1}_i) + \boldsymbol{LB}_i - \boldsymbol{1}_i, \boldsymbol{UB}_i) \text{ AND } \boldsymbol{II}_i \geq 0\}$, where B_i is the diagonal matrix whose diagonal elements are constants $b_1, b_2, \ldots, b_{di}$ defining the rectangular tile size in the iteration space of statement S_i surrounded by d_i loops; $\boldsymbol{LB}_i$ and $\boldsymbol{UB}_i$ are the vectors whose elements are lower $lb_1, \ldots, lb_{di}$ and upper $ub_1, \ldots, ub_{di}$ bounds of indices $i_1, i_2, \ldots, i_{di}$ of the original loops, respectively.

We introduce the following definition.

Definition 1. If there exists a direct or transitive dependence whose target belongs to set $TILE_i(\boldsymbol{II}_i)$ and its source belongs to a tile with an identifier lexicographically greater than $\boldsymbol{II}_i$, then the target of this dependence is invalid within set $TILE_i(\boldsymbol{II}_i)$.

Further on, for brevity we will refer to an invalid dependence target as an invalid target.

Definition 2. A tile including one or more invalid targets is invalid.

To identify invalid original tiles, we suggest to form, for each loop nest statement $S_i, i = 1, 2, \ldots, q$, where q is the number of loop statements, set $TILE_GT_i(\boldsymbol{II}_i)$, including all the statement instances that are contained in the tiles whose identifiers are lexicographically greater than that of set $TILE_i(\boldsymbol{II}_i)$, i.e., $\boldsymbol{II}_i$. Given set $TILE_GT_i(\boldsymbol{II}_i)$, we can calculate the set $R^+(TILE_GT_i(\boldsymbol{II}_i))$ which includes all dependence targets whose sources belong to set $TILE_GT_i(\boldsymbol{II}_i)$. The intersection of the sets $TILE_i(\boldsymbol{II}_i)$ and $R^+(TILE_GT_i(\boldsymbol{II}_i))$ defines the set including all invalid targets within set $TILE_i(\boldsymbol{II}_i)$. If this set is empty for each $i = 1, 2, \ldots, q$, then all original tiles are valid.

To transform invalid tiles into valid ones, we will use set $TILE_LT_i(\boldsymbol{II}_i)$, including all the statement instances that are within the tiles whose identifiers are lexicographically less than that of set $TILE_i(\boldsymbol{II}_i)$. To calculate sets $TILE_GT_i(\boldsymbol{II}_i)$ and $TILE_LT_i(\boldsymbol{II}_i)$, we need to determine all the tile identifiers which are lexicographically greater and less, respectively, than identifier $\boldsymbol{II}_i$. For this purpose, we take into account that for the original loop nest, a statement can be of type a or type b (see the loop nest above). A statement of type a, S_{ia}, textually precedes statements: (i) $S_{jb}, 1 \leq j \leq q$; (ii) $S_{ja}, j > i$. A statement of type b, S_{ib}, textually precedes statements $S_{jb}, j > i$.

To allow lexicographic comparison of identifiers of tiles associated with different statements, we need to preprocess vector $\boldsymbol{II}_i$, including indices $ii_1, ii_2, \ldots, ii_{di}$ of the loops surrounding statement S_i, to get vector $\boldsymbol{II}_{iprep}$ according to the procedure below. Let us note that the value floor $((ub_i - lb_i - 1)/b_i)$ represents the upper bound for index ii_i.

Procedure 2. Preprocessing procedure of tile identifier vectors.

Input: A loop nest; vectors $\boldsymbol{II}_i$, $i = 1, 2, \ldots, q$, where q is the number of loop nest statements, representing identifiers of tiles formed for each loop nest statement instance $S_i, i = 1, 2, \ldots, q$; loop nest depth, d; the number of loops surrounding statement S_i, d_i, $i = 1, 2, \ldots, q$.

Output: Preprocessed vectors $\boldsymbol{II}_{iprep}$, $i = 1, 2, \ldots, q$.

Method:

foreach vector $\boldsymbol{II}_i, i = 1, 2, \ldots, q$ **do**

1. Insert '0' into the last $d - d_i$ positions of $\boldsymbol{II}_i$ if a corresponding statement S_i is of type a, S_{ia}, and the value equal to the value floor$((ub_i - lb_i - 1)/b_i)$ if a corresponding statement S_i is of type b, S_{ib}, where floor(x) is the function returning the largest integer no greater than x.

2. Before each element ii_j, $j = 1, 2, \ldots, d$, of the vector, obtained in Step 1, insert an additional element with the value equal to $n_j - 1$, where n_j is the number of loops, defined by index i_j and appearing before statement S_i.

3. Insert into the position $2d + 1$ of the vector, received in Step 2, the value equal to the loop nest statement number according to the textual order of statements in the loop nest.

It is worth noting that each vector $\boldsymbol{II}_{iprep}$, $i = 1, 2, \ldots, q$, is of length $2d + 1$. The application of the procedure above to the loop nest of the following structure:

```
for(i1=0; i1<4; ++i1) {
   S11a(i1);
   for(i2=0; i2<4; ++i2) {
       S21a(i1, i2);
   }
   S11b(i1);
   for(i2=0; i2<4; ++i2) {
      S22a(i1, i2);
   }
   S12b(i1);
}
```

provided that original tiles are of size 2×2, results in the following preprocessed vectors:

$$\boldsymbol{II}_{11aprep} = (0, ii1, 0, 0, 1)^T,$$
$$\boldsymbol{II}_{21aprep} = (0, ii1, 0, ii2, 2)^T,$$
$$\boldsymbol{II}_{11bprep} = (0, ii1, 0, 1, 3)^T,$$
$$\boldsymbol{II}_{22aprep} = (0, ii1, 1, ii2, 4)^T,$$
$$\boldsymbol{II}_{12bprep} = (0, ii1, 1, 1, 5)^T,$$

where "1" as the third elements of vectors $\boldsymbol{II}_{22aprep}$ and $\boldsymbol{II}_{12bprep}$ denotes the second appearance of the loop defined with index $i2$; "1" as the fourth elements of vectors $\boldsymbol{II}_{11bprep}$ and $\boldsymbol{II}_{12bprep}$ is equal to the value $floor((4-0-1)/2)$.

In general, we form sets $TILE_LT_i(\boldsymbol{II}_i)$ and $TILE_GT_i(\boldsymbol{II}_i)$ as the union of all the tiles whose identifiers are lexicographically less and greater than $\boldsymbol{II}_i$, respectively, as follows:

$TILE_LT_i(\boldsymbol{II}_i) = \{[\boldsymbol{I}_j] \mid \text{exists } \boldsymbol{II}_j{}' \text{ s.t. } \boldsymbol{I}_j \text{ in } TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_j) \text{ AND } \boldsymbol{II}_{jprep}{}' \prec \boldsymbol{II}_{iprep}\}$,

$TILE_GT_i(\boldsymbol{II}_i) = \{[\boldsymbol{I}_j] \mid \text{exists } \boldsymbol{II}_j{}' \text{ s.t. } \boldsymbol{I}_j \text{ in } TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_j) \text{ AND } \boldsymbol{II}_{jprep}{}' \succ \boldsymbol{II}_{iprep}\}$,

where $\boldsymbol{I}_i$ in $TILE_i$ $(\boldsymbol{II}_i{}', \boldsymbol{B}_i)$ means that vector $\boldsymbol{I}_i$ belongs to set $TILE_i$ $(\boldsymbol{II}_i{}', \boldsymbol{B}_i)$ whose identifier is $\boldsymbol{II}_i{}'$ and the corresponding tile is of the size defined by the diagonal matrix $\boldsymbol{B}_i$.

3.2. Tiling idea. The concept of the introduced approach is as follows. First, the loop nest iteration space is partitioned into smaller rectangular subspaces, i.e., tiles. If there are no dependences for this loop nest or all elements of dependence distance vectors are non-negative, we may immediately generate code scanning tiles in lexicographic order, and this code will be valid because all dependences of the original loop nest will be respected, i.e., each statement instance associated with a dependence source will be executed before the statement instance associated with the destination of this dependence.

However, when there exist dependence distance vectors whose elements are negative, scanning introduced rectangular tiles in the lexicographic order is invalid because dependences available in the original loop nest will not be respected.

Techniques based on affine transformations attempt to change the original loop nest iteration space so that the enumeration of rectangular tiles in the new iteration space is valid. But it is well known that this is not always possible.

Our idea to form valid target tiles is different from that based on affine transformations. We suggest to apply the transitive closure of the dependence graph, representing all the dependences available in the loop nest, first to check whether the original tiles are valid. Such a case is true when each original tile does not include any dependence destination whose corresponding dependence source belongs to a tile(s) whose identifier(s) is (are) greater than that of the tile including the dependence destination. This guarantees that a dependence destination will never be executed before the execution of the corresponding dependence destination. For such a case, tiled code can be generated directly without any changes of original rectangular tiles.

To verify whether this case is true, we apply the transitive closure of the dependence graph to the iteration sub-space including the tiles whose identifiers are greater than that representing a given tile. This will result in producing the sub-space of dependence destinations whose sources belong to the sub-space including the original tiles with the identifiers greater than that representing the given tile.

Next we calculate the intersection of that subspace with the subspace including the statement instances of the given tile. If the result is an empty set, this means that all original tiles are valid. Otherwise, we have to correct original tiles so that they do not include any invalid dependence destinations, i.e., remove those destinations whose sources belong to the tiles whose sources belong to the sub-space including tiles with the identifiers greater than that representing the given tile.

For this purpose, we remove from the set representing statement instances of the given tile all the destinations being comprised in the set calculated by applying the transitive closure of the dependence graph to the iteration sub-space, including the tiles whose identifiers are greater than that representing the given tile.

Finally, each invalid dependence target, which has been removed from some tile, say T, is added to exactly one tile whose identifier is lexicographically greater than that of T.

In this paper, we prove that all tiles produced in the way described above are valid and can be enumerated in lexicographic order.

To implement the presented concept and generate valid tiled code, we can apply the following four steps:

(i) form original fixed rectangular tiles for each loop nest statement;

(ii) check whether all dependences available in

the original loop nest are respected under the lexicographical order of the original tile enumeration, if so, the original tiles are valid, generate code representing original tiles, the end;

(iii) transform the invalid original tiles into valid target ones (tile correction);

(iv) generate tiled code enumerating valid target targets and iterations within each tile in lexicographical order.

Below, we explain how the concept above can be implemented mathematically to correct original invalid tiles in order to obtain valid target tiles represented with sets $TILE_VLD_i$, $i = 1, 2, \ldots, q$, where q is the number of loop nest statements. Further on, for brevity, we will skip vector $\boldsymbol{II}_i$, defining the tile identifier, in the set name.

For each $i = 1, 2, \ldots, q$, we will form set $TILE_VLD_i$ as the union of two sets, $TILE_ITR_i$ and $TVLD_LT_i$. Set $TILE_ITR_i$ includes only those iterations of set $TILE_i$ that are not invalid targets within $TILE_i$ (set $TILE_i$ from which all invalid targets are removed).

Set $TVLD_LT_i$ (targets valid to be put into set $TILE_ITR_i$ and contained within set $TILE_LT_i$) contains all the dependence targets such that each of them (i) has the corresponding source within set $TILE_ITR_i$, (ii) is valid to be put into set $TILE_ITR_i$, and (iii) is invalid for some tile with an identifier less than that of $TILE_i$.

To explain how set $TILE_ITR_i$ can be calculated, we first recall that the application of relation R to set S is defined as follows:

$$R(S) = \{[e'] : \text{there exists}\, e \text{ s.t. } e \to e' \in R, e \in S\},$$

i.e., $R(S)$ results in the range of relation R with domain S.

Now, we take into consideration that the application of relation R^+, representing the positive transitive closure of a loop dependence graph, to set $TILE_GT_i$, introduced in the previous subsection ($R^+(TILE_GT_i)$), results in a set comprising all the targets of dependences whose sources are within the tiles with the identifiers greater than that of $TILE_i$; i.e., this set includes all invalid targets for set $TILE_i$ and they have to be excluded from it, i.e., set $TILE_ITR_i$ is formed as follows:

$$TILE_ITR_i = TILE_i - R^+(TILE_GT_i).$$

To form set $TVLD_LT_i$, we note that the application of relation R^+ to set $TILE_ITR_i$ ($R^+(TILE_ITR_i)$) results in a set including all the targets of the dependences whose sources belong to set $TILE_ITR_i$.

The intersection of the sets $R^+(TILE_ITR_i)$ and $TILE_LT_i$ ($R^+(TILE_ITR_i) \cap TILE_LT_i$) yields a set, say $TILE_ITR_LT_i$, including the elements that (i) are the targets of the dependences whose sources are contained in set $TILE_ITR_i$ and (ii) belong to the tiles whose identifiers are lexicographically less than that of set $TILE_i$.

Set $TILE_ITR_LT_i$ comprises invalid targets to be put into set $TILE_ITR_i$ if their corresponding dependence sources belong not only to set $TILE_ITR_i$ but also to the tiles whose identifiers are greater than that of $TILE_i$, i.e., these sources are within set $TILE_GT_i$.

To form set $TVLD_LT_i$ comprising only valid targets to be put into set $TILE_ITR_i$, i.e., not including the targets of the dependences whose sources belong to set $TILE_GT_i$, we take into consideration that the set $R^+(TILE_GT_i)$ comprises all such invalid targets; hence set $TVLD_LT_i$ is calculated as follows:

$$TVLD_LT_i = TILE_ITR_LT_i - R^+(TILE_GT_i).$$

We form set $TILE_VLD_EXT_i$ to be used for producing tiled code by means of inserting (i) into the first positions of the tuple of set $TILE_VLD_i$ indices ii_1, ii_2, ..., ii_q, (ii) into the constraints of set $TILE_VLD_i$ the constraints of set, II_SET_i, defining tile identifiers: $II_SET_i = \{[\,\boldsymbol{II}_i] | \,\boldsymbol{II}_i \geq 0 \text{ AND } \boldsymbol{B}_i * \boldsymbol{II}_i + \boldsymbol{LB}_i \leq \boldsymbol{UB}_i\}$.

Any code generator allowing scanning elements of the union of sets $TILE_VLD_EXT_i$, $i = 1, 2, \ldots, q$, in lexicographic order can be applied to generate tiled code, for example, CLooG (Bastoul, 2004) or the codegen function of the Omega project (Kelly *et al.*, 1995).

3.3. Illustrating the tiling idea by means of a working example. To illustrate how the transitive closure of a dependence graph can be applied to produce valid tiled loop nests, let us consider the following working example.

Example 2.

```
for(i=0; i<=3; i++){
  b[i][0] = c[i][0];                    //S1
  for(j=0; j<=3; j++)
    a[i][j] = a[i+1][j-1]+b[i+1][j]
               +b[i][0]+a[i][j+1]; //S2
  d[i][3] = a[i+1][3]+a[i][3];      //S3
}
```

♦

We used the ISL library (Verdoolaege, 2011) to carry out all calculations necessary to generate tiled code. In this paper, we use the Barvinok tool syntax (Verdoolaege, 2012) to present results of calculations on relations and sets. The following preprocessed relations describe all the dependences in the working loop nest (extracted by means of Petit (Kelly *et al.*, 1995), the Omega project dependence analyzer):

```
R1:={[i,-1,1] -> [i,j,2] : 0 <= i <= 3
    && 0 <= j <= 3},
R2:={[i,j,2] -> [i+1,j-1,2] :
    0 <= i <= 2 && 1 <= j <= 3},
R3:={[i,0,2] -> [i+1,1] : 0 <= i <= 2},
```

```
R4:={[i,j,2] -> [i,j+1,2] : 0 <= i <= 3
     && 0 <= j <= 2},
R5:={[i,3,2] -> [i,-1,3] :
      0 <= i <= 3},

R6:={[i,-1,3] -> [i+1,3,2] :
      0 <= i <= 2}.
```

Let us recall that the last element of each tuple of a preprocessed dependence relation states for the identifier of a loop nest statement. Figure 2(a) shows the dependence graph for the working loop nest, where vertices represent loop statement instances; there exists an edge between two vertices if one defines the source of a dependence and the other defines the target of this dependence.

Figure 2(b) presents the original rectangular tiles. The numbers in the squared boxes show the order of tile execution. For statements $S1$ and $S3$, tiles are one-dimensional, while for statement $S2$ they are two-dimensional. Scanning those tiles and loop statement instances within each tile in lexicographic order is invalid because of the violation of the valid execution of dependent statement instances (to preserve a dependence, we should first execute the source of this dependence, then its destination). For example, the instance of statement $S1$ on iteration 1 (the destination of the dependence $S2(0,0) \rightarrow S1(1)$) will be executed before the execution of the instance of statement $S2$ on iteration (0, 0) (the source of this dependence). To cope with such a problem, we correct the content of the original tiles in the manner demonstrated in Fig. 2(e). Now scanning tiles *TILE_VLD*$_i$ and loop statement instances within each tile in lexicographic order is valid because all original loop nest dependences are preserved.

In order to carry out tile corrections in a formal way, we can proceed as follows. Let indices ii and jj define the identifier of an original preprocessed parametric rectangular tile, $TILE_i, i = 1, 2, 3$ (which is parametric with respect to indices ii, jj) represented below:

TILE$_1$
$= [ii] \rightarrow$ $\{[i, -1, 1] :$ $i \geq 2ii$ and $i \geq 0$ and $i \leq 3$ and $i \leq 1 +$ $2ii$ and $ii \geq 0\}$,

TILE$_2$
$= [ii, jj] \rightarrow$ $\{[i, j, 2] : i \geq 2ii$ and $i \geq 0$ and $i \leq 3$ and $i \leq 1 + 2ii$ and $j \geq 2jj$
and $j \geq 0$ and $j \leq 3$ and
$j \leq 1 + 2jj$ and $ii \geq 0$ and $jj \geq 0\}$,

TILE$_3$
$= [ii] \rightarrow$ $\{[i, -1, 3] :$ $i \geq 2ii$ and $i \geq 0$ and $i \leq 3$ and $i \leq 1 + 2ii$ and $ii \geq 0\}$,

where the notation $[x, y, z, \ldots] \rightarrow \{[\ldots] :$ constraints$\}$ means that $[x, y, z, \ldots]$ are parametric variables in the constraints of a set.

For each statement $S_i, i = 1, 2, 3$, we form two additional parametric sets, *TILE_LT*$_i$ and *TILE_GT*$_i$.

Figure 2(c) illustrates sets *TILE_LT*$_2$ and *TILE_GT*$_2$ for *TILE*$_2$($ii = 0$, $jj = 0$). To calculate these sets, we first preprocess tile identifiers according to the procedure presented in Section 3.1. For the working example, the preprocessed vectors, defining tile identifiers, are as follows:

$$\mathbf{II}_{1prep} = (0, ii, 0, 0, 1)^T,$$
$$\mathbf{II}_{2prep} = (0, ii, 0, jj, 2)^T,$$
$$\mathbf{II}_{3prep} = (0, ii, 0, 1, 3)^T.$$

Sets *TILE_LT*$_2$ and *TILE_GT*$_2$ calculated according to the formulas presented in Section 3.1 are of the following forms:

TILE_LT$_2$
$= [ii, jj] \rightarrow$ $\{\{[i, j, 2] : jj = 1$ and $ii \leq$ 1 and $i \geq 2ii$ and i $\geq$ and $i \leq 1 + 2ii$ and $j \leq 1$ and $j \geq 0; [i, -1, 3] : ii = 1$ and $i \leq 1$ and $i \geq$ $0; [i, -1, 7] :$ $ii \leq 1$ and $i \geq 0$ and $i \leq 1 + 2ii\}$,

TILE_GT$_2$
$= [ii, jj] \rightarrow$ $\{[i, j, 2] : jj = 0$ and $ii \leq 1$ and $ii \geq 0$ and $i \geq 2ii$ and $i \leq 1 + 2ii$ and $j \leq 3$ and $j \geq 2; [i, -1, 3] : (ii \leq 1$ and $i \geq 2ii$ and $i \geq 0$ and $i \leq 1 + 2ii)$ *or* $(ii = 0$ and $i \leq$ 3 and $i \geq$ $2); [i, -1, 1] : ii = 0$ and $i \leq$ 3 and $i \geq 2\}$.

Figure 2(d) illustrates sets *TILE_ITR*$_i$ and *TVLD_LT*$_i$ for various values of indices ii and jj calculated according to the formulas presented in Section 3.2.

Figure 2(e) illustrates sets *TILE_VLD*$_i$ for different values of indices ii and jj representing valid target tiles.

Figure 1 of Appendix presents tiled code for Example 2 when the upper bounds of i and j equal n and tiles are of size 32×32. The speed-up of this code is discussed in Section 6. Let us note that the optimizing compiler PLUTO, implementing affine transformations, cannot tile Example 2.

Applying the way discussed above to Example 1, we get target tiles presented in Fig. 3. It is worth noting that for Example 1, the tiled codes produced by means of the affine transformation and the way based on transitive closure are different. Comparing the tiles presented in Figs. 1 and 3, we may conclude that (i) applying the affine transformation results in 6 tiles while applying transitive

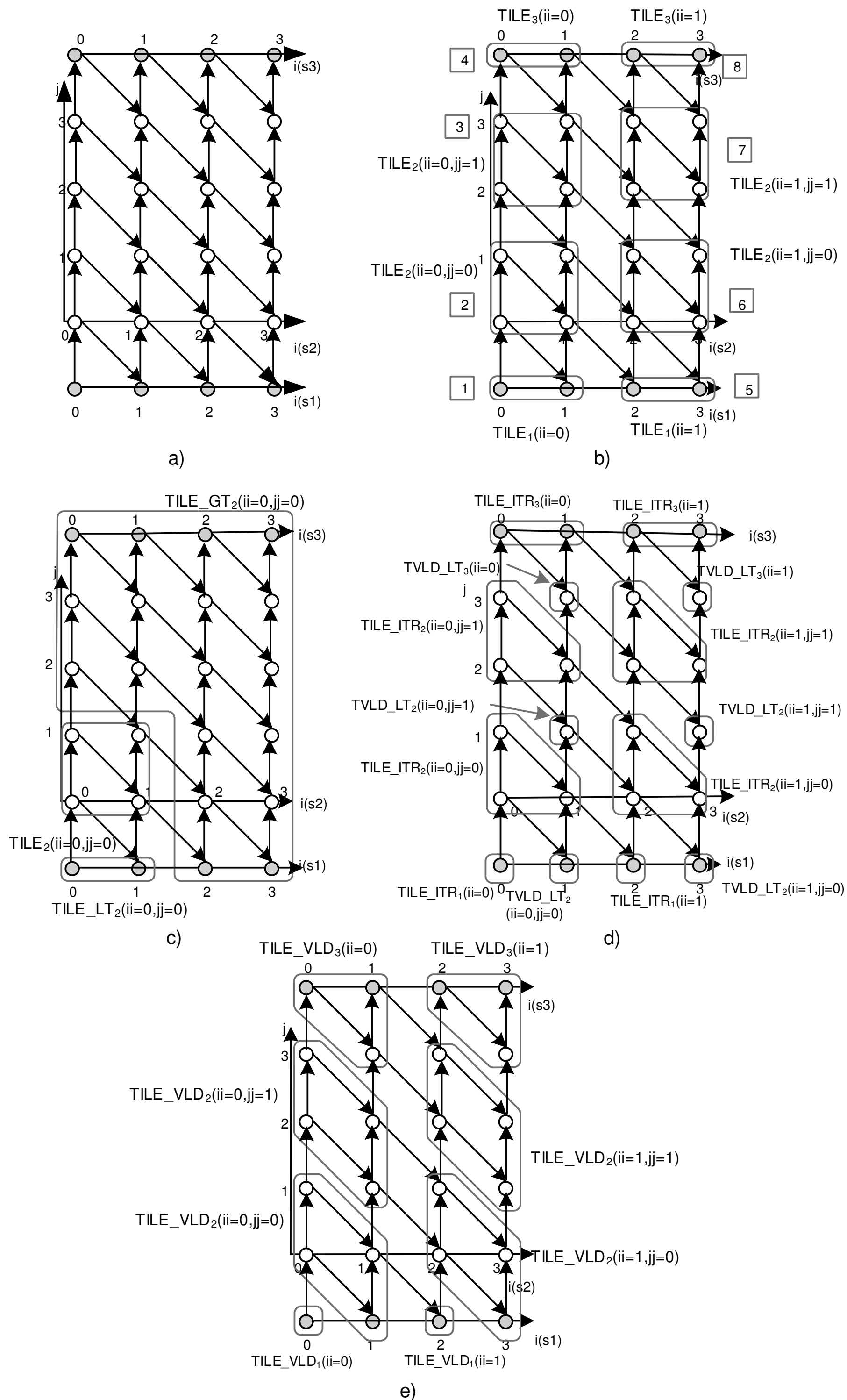

Fig. 2. Illustrations for the working loop nest: dependences (a), original tiles (b), sets *TILE_LT*$_2$ and *TILE_GT*$_2$ (c), sets *TILE_ITR*$_i$ and *TVLD_LT*$_i$ (d), target tiles (e).

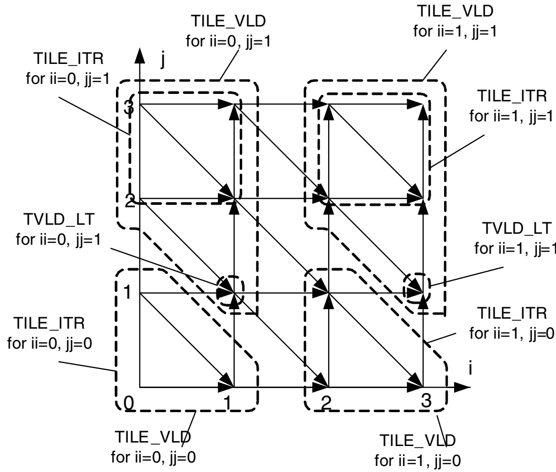

Fig. 3. Illustration of sets *TILE_ITR, TVLD_LT*, and *TILE_VLD* for Example 1.

closure generates 4 tiles, (ii) the structure and content of tiles are different, in Fig. 1 tiles include 1, 3, or 4 iterations while in Fig. 3 tiles comprise 3 or 5 iterations.

3.4. Formal algorithm and its correctness. Below, we present the formal algorithm, implementing the idea presented above and allowing the tiling transformation of arbitrarily nested loops of depth d. It includes four steps. The first one is preprocessing; it prepares input data and forms sets introduced in Section 3.1. The second step checks whether the original tiles are valid, if so, then the fourth step (code generation) is carried out. Otherwise, Step 3 transforms invalid original tiles into valid target ones.

To show the correctness of Algorithm 1, we have to prove that for each $i = 1, 2, \ldots, q$, (i) set $TILE_VLD_i$ does not include any invalid dependence target, (ii) each invalid dependence target, removed from $TILE_i$, is added to exactly one set $TILE_VLD_i$ whose identifier is lexicographically greater than that of $TILE_i$.

Proof. For each $i = 1, 2, \ldots, q$, set $TILE_VLD_i$ is the union of the two sets: $TILE_ITR_i$ and $TVLD_LT_i$. Set $TILE_ITR_i$ does not include any invalid dependence target because all invalid dependence targets are contained in the set $R^+(TILE_GT_i)$ and they are removed from set $TILE_i$ by applying the set difference operator: $TILE_ITR_i = TILE_i - R^+(TILE_GT_i)$.

Set $TVLD_LT_i$ also does not include any invalid dependence target because all invalid dependence targets are contained in the set $R^+(TILE_GT_i)$ and they are removed from the set $TILE_ITR_LT_i = R^+(TILE_ITR_i) \cap TILE_LT_i$ including the elements that i) are the targets of the dependences whose sources are contained in set $TILE_ITR_i$ and ii) belong to the tiles whose identifiers are lexicographically less than that of set $TILE_i$.

Because both sets $TILE_ITR_i$ and $TVLD_LT_i$ do not contain any invalid dependence target, set $TILE_VLD_i$ also does not include any invalid dependence target.

Each invalid dependence target, say t, belonging to the set $TILE_i$ with identifier ID, and having two or more associated dependence sources contained in the sets $TILE_j, j \neq i$ with identifiers $ID_1, ID_2, \ldots, ID_n$, $ID \prec ID_1 \prec ID_2 \prec \ldots \prec ID_n$, will be included into only one set, $TILE_VLD_n$, with identifier ID_n.

Indeed, the set $TILE_i$ with identifier ID_n is contained in the set $TILE_GT_i$ corresponding to the sets $TILE_i$ with identifiers $ID_1, ID_2, \ldots, ID_{(n-1)}$, hence target t is within the set $R^+(TILE_GT_i)$ for all those sets $TILE_i$ and it will be removed from all the sets $TVLD_LT_i$ with identifiers $ID_1, ID_2, \ldots, ID_{(n-1)}$. For the set $TILE_i$ with identifier ID_n, in the set $R^+(TILE_GT_i)$ there does not exist any source of the dependence whose target is t, hence target t will be added to exactly one set $TILE_VLD_i$ with identifier ID_n. ■

It is worth noting that Algorithm 1 produces target tiles (represented by set $TILE_VLD_i$) whose shapes in general are different from the rectangular shapes of the original tiles (represented by set $TILE_i$).

4. Tiled code parallelization

The goal of parallelization is to automatically generate code that executes tiles in parallel while loop nest statement instances within each tile serially. To automatically parallelize tiled code generated according to Algorithm 1, we have to realize the following steps.

First, we should form a relation that represents all dependences among tiles but ignores dependences available within each tile. Having such a relation, we can apply any known automatic parallelization technique to produce first parallel pseudo-code and next, by means of a postprocessor, convert such a code into parallel compilable code.

Automatic parallelization techniques are out of the scope of this paper. We only refer to techniques that were implemented in the optimizing compiler TRACO to generate parallel tiled code. The design of a postprocessor depends on the parallel computer architecture and a language, API, or a library to write parallel programs for the computer. TRACO, implementing Algorithm 1 and parallelization techniques, converts pseudo-code to OpenMP code (OpenMP Architecture Review Board, 2012), which next can be compiled by any appropriate compiler to generate parallel executable code. In this section, we show how a relation describing all dependences among tiles can be constructed.

Algorithm 1. Tiling transformation for arbitrarily nested loops.

Input: Arbitrarily nested d loops; constants $b_1, b_2, \dots, b_d$ defining the size of a rectangular input tile.
Output: Tiled code.
Method:

1. Data preparation. For each $i, i = 1, 2, \dots, q$ and d_i where q is the number of loop statements and d_i is the number of loops surrounding statements S_i, form the following data:
   - vector $\boldsymbol{I}_i$ whose elements are original loop indices $i_1, i_2, \dots, i_{di}$;
   - vector $\boldsymbol{II}_i$ whose elements $ii_1, ii_2, \dots, ii_{di}$ define the identifier of a tile for the iteration space of statement S_i;
   - vectors $\boldsymbol{LB}_i$ and $\boldsymbol{UB}_i$ whose elements are lower $(lb_1, \dots, lb_{di})$ and upper $(ub_1, \dots, ub_{di})$ bounds of indices $i_1, i_2, \dots, i_{di}$ of the original loops, respectively;
   - vector $\mathbf{1}_i$ and $\mathbf{0}_i$ whose all d_i elements are equal to 1 and 0, respectively;
   - diagonal matrix $\boldsymbol{B}_i$ whose diagonal elements are constants $b_1, b_2, \dots, b_{di}$ defining the rectangular tile size in the iteration space of statement S_i;
   - set $TILE_i(\boldsymbol{II}_i, \boldsymbol{B}_i) = \{[\boldsymbol{I}_i] \mid \boldsymbol{B}_i * \boldsymbol{II}_i + \boldsymbol{LB}_i \le \boldsymbol{I}_i \le \min(\boldsymbol{B}_i * (\boldsymbol{II}_i + \mathbf{1}_i) + \boldsymbol{LB}_i - \mathbf{1}_i, \boldsymbol{UB}_i) \text{ AND } \boldsymbol{II}_i \ge 0\}$ defining the original rectangular tiles;
   - preprocessed vector $\boldsymbol{II}_{iprep}$ of vector $\boldsymbol{II}_i$ according to the procedure presented in Section 3.1;
   - set $TILE_GT_i$ as the union of all the tiles whose identifiers are lexicographically greater than $\boldsymbol{II}_{iprep}$ as follows $TILE_GT_i = \{[\boldsymbol{I}_j] \mid \text{exists } \boldsymbol{II}_j{}' \text{ s. t. } \boldsymbol{I}_j \text{ in } TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_j) \text{ AND } \boldsymbol{II}_{jprep}{}' \succ \boldsymbol{II}_{iprep}\}$, where $\boldsymbol{I}_j$ in $TILE_i(\boldsymbol{II}_j{}'\ \boldsymbol{B}_i)$ means that vector $\boldsymbol{I}_j$ belongs to set $TILE_i(\boldsymbol{II}_j{}'\ \boldsymbol{B}_i)$.
2. Checking original tile validity.

   2.1. Carry out a dependence analysis to produce a set of relations describing all the dependences in the original loop nest; preprocess all dependence relations according to the procedure presented in Section 2.

   2.2. Calculate the positive transitive closure, R^+, of the union of all the preprocessed relations returned by Step 2.1.

   2.3. Calculate the following sets: $CHECK_VLD_i = TILE_i(\boldsymbol{II}_i, \boldsymbol{B}_i) \cap R^+(TILE_GT_i(\boldsymbol{II}_i))$, $i = 1, 2, \dots, q$. If each of these sets is empty, then $TILE_VLD_i = TILE_i(\boldsymbol{II}_i, \boldsymbol{B}_i)$, $i = 1, 2, \dots, q$; go to Step 4.

3. Forming valid target tiles, represented with set $TILE_VLD_i$. For each $i, i = 1, 2, \dots, q$, calculate

   3.1. Set $TILE_LT_i$ as the union of all the tiles whose identifiers are lexicographically less than $\boldsymbol{II}_{iprep}$, as follows:

   $TILE_LT_i = \{[\boldsymbol{I}_j] \mid \text{exists } \boldsymbol{II}_j{}' \text{ s. t. } \boldsymbol{I}_j \text{ in } TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_j) \text{ AND } \boldsymbol{II}_{jprep}{}' \prec \boldsymbol{II}_{iprep}\}$, where $\boldsymbol{I}_j$ in $TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_i)$ means that vector $\boldsymbol{I}_j$ belongs to set $TILE_i(\boldsymbol{II}_j{}', \boldsymbol{B}_i)$.

   3.2. Set $TILE_ITR_i$ not including any invalid dependence target as below:

   $TILE_ITR_i = TILE_i - R^+(\ TILE_GT_i\ )$.

   3.3. Set $TVLD_LT_i$ including all the iterations that (i) belong to the tiles whose identifiers are lexicographically less than that of set $TILE_ITR_i$, (ii) are the targets of the dependences whose sources are contained in set $TILE_ITR_i$, and (iii) are not any target of a dependence whose source belong to set $TILE_GT_i$ as follows:

   $TVLD_LT_i = (\ R^+(TILE_ITR_i) \cap TILE_LT_i) - R^+(\ TILE_GT_i\ )$.

   3.4. Set $TILE_VLD_i$ representing target tiles as the union of sets $TILE_ITR_i$ and $TVLD_LT_i$

   $TILE_VLD_i = TILE_ITR_i \cup TVLD_LT_i$.

4. Code generation.

   4.1. For each $i = 1, 2, \dots, q$, form set $TILE_VLD_EXT_i$ by means of inserting (i) into the first positions of the tuple of set $TILE_VLD_i$ indices $ii_1, ii_2, \dots, ii_{di}$; (ii) into the constraints of set $TILE_VLD_i$ the constraints defining tile identifiers:

   $\boldsymbol{II}_i \ge 0$ and $\boldsymbol{B}_i * \boldsymbol{II}_i + \boldsymbol{LB}_i \le \boldsymbol{UB}_i$.

   4.2. Generate tiled code by means of applying any code generator scanning elements of the union of sets $TILE_VLD_EXT_i$ in lexicographic order, for example, CLooG (Bastoul, 2004) or the codegen function of the Omega project (Kelly *et al.*, 1995).

The idea of constructing such a relation, say *R_TILE*, is the following. We take into account that if some target tile includes the source of a dependence (available in the original loop nest) whose destination belongs to another target tile, then there exists a dependence between these tiles. To form relation *R_TILE*, we use a relation, say R, representing all the dependences of the original loop nest. The first tuple of relation *R_TILE* defines the identifiers of the target tiles (represented with set *TILE_VLD*) including dependence sources (represented with relation R), while the second tuple defines the identifiers of the tiles comprising the corresponding dependence destinations. The constraints of relation *R_TILE* have to include (i) the constraint defining a set including all target tile identifiers (they are the same as those of the original tiles); (ii) existential vectors, say $\boldsymbol{I}, \boldsymbol{J}$, representing a pair of the identifiers of dependent target tiles; (iii) the constraint defining that vector $\boldsymbol{J}$ represents the destination of a dependence whose source is represented with vector $\boldsymbol{I}$, i.e., $\boldsymbol{J} = \boldsymbol{R}(\boldsymbol{I})$, where $R(\boldsymbol{I})$ means the operator of the application of relation R to $\boldsymbol{I}$.

Below, we present mathematically relation *R_TILE* which describes dependences among all target tiles but ignores dependences available within each tile, i.e., it describes inter-tile dependences:

R_TILE := {[***II***] $\to$ [***JJ***]: ***II***, ***JJ*** in $\cup_{i=1}^{q}$ ($\boldsymbol{II}_i$) AND $\boldsymbol{II}_i \geq 0$ AND $\boldsymbol{B}_i$*$\boldsymbol{II}_i$ +$\boldsymbol{LB}_i \leq \boldsymbol{UB}_i$ AND exist ***I***, ***J*** s.t. ***I*** in $\cup_{i=1}^{q}$ (*TILE_VLD*$_i$(***II***)) AND ***J*** in $\cup_{i=1}^{q}$ (*TILE_VLD*$_i$(***JJ***)) AND ***J*** in *R*(***I***)},

where $\boldsymbol{II}$, $\boldsymbol{JJ}$ are vectors representing the sources and destination of inter-tile dependences, respectively; q is the number of loop nest statements; $\boldsymbol{II}_i$ is the vector representing tile identifiers for the i-th loop nest statement; $\boldsymbol{B}_i$ is the diagonal matrix whose diagonal elements are constants $b_1, b_2, \ldots, b_{di}$ defining the original rectangular tile size in the iteration space of statement S_i; $\boldsymbol{LB}_i$ and $\boldsymbol{UB}_i$ are vectors whose elements are lower $lb_1, \ldots, lb_{di}$ and upper $ub_1, \ldots, ub_{di}$ bounds of indices $i_1, i_2, \ldots, i_{di}$ of the original loops, respectively; $TILE_VLD_i, i = 1, 2, \ldots, q$ are the sets returned by Algorithm 1 and representing target tiles; R is the relation describing all the dependences in the original loop nest.

Below, we present the meaning of the particular parts of the constraints of relation *R_TILE*: ***II***, ***JJ*** in $\cup_{i=1}^{q}$ ($\boldsymbol{II}_i$) means that vectors ***II***, ***JJ*** belong to the union of all the vectors representing tile identifiers of all loop nest statements; $\boldsymbol{II}_i \geq 0$ AND $\boldsymbol{B}_i$*$\boldsymbol{II}_i$+$\boldsymbol{LB}_i \leq \boldsymbol{UB}_i$ are the constraints imposed on tile identifiers of loop nest statements; exist ***I***, ***J*** s.t. ***I*** in $\cup_{i=1}^{q}$ (*TILE_VLD*$_i$(***II***)) AND ***J*** in $\cup_{i=1}^{q}$ (*TILE_VLD*$_i$(***JJ***)) means that there exist vectors ***I***, ***J*** representing loop nest statement instances such that they belong to sets *TILE_VLD*$_i$(***II***) and *TILE_VLD*$_i$(***JJ***), respectively, returned by Algorithm 1; ***J*** in *R*(***I***) denotes that elements of vector ***J*** are the targets of the dependences whose sources are elements of vector ***I***.

It is worth noting that relation *R_TILE* represents only cycle-free inter-tile dependence graphs because according to Algorithm 1, all target tiles are valid; i.e., in those graphs, each tile with identifier $\boldsymbol{I}$ includes the dependence targets whose sources belong to the tiles with identifiers which are lexicographically less than $\boldsymbol{I}$, this disables any cycles in the dependence graph whose vertices are target tiles.

It is well know that, for the cycle-free graph, a legal schedule for vertices of this graph can be found (Feautrier, 1992a; 1992b). In other words, applying Algorithm 1 makes possible to use any known algorithm, aimed at extracting a legal schedule, for paralllelization of target tiles represented by tiled code.

Tiled code generated by means of Algorithm 1 and relation *R_TILE* can be used as input data for any known algorithm for automatic loop nest parallelization. Techniques based on affine transformations and/or the transitive closure can be used to generate parallel tiled code. In our implementation, we applied the techniques presented by Beletska *et al.* (2011) and Bielecki *et al.* (2012) to generate synchronization-free tiled code and tiled code based on the free schedule, respectively.

5. Applying the approach to real-life code: General linear recurrence equations

In this section, we present the application of Algorithm 1 to Kernel 6 of the Livermore Loops (`http://www.netlib.org/benchmark/livermore.`)

The general linear recurrence equations is a fundamental numerical computation that can be applied in many important scientific applications: partial differential equations, tridiagonal linear systems, polynomial evaluations, eigenvalue (eigenvector) problems, and digital signal processing. All these problems are computationally intensive and require high-speed computations.

Kernel 6 of the Livermore Loops is presented with the following loop nest.

```
for ( l=1 ; l<=loop ; l++ )
   for ( i=1 ; i<n ; i++ )
      for ( k=0 ; k<i ; k++ )
        w[i] += b[k][i] * w[(i-k)-1];
```

It is known that this loop nest cannot be tiled by means of affine transformations. The optimizing compiler PLUTO (Bondhugula *et al.*, 2008b), implementing affine transformations, does not generate any tiled code for this loop nest.

Below, we demonstrate how tiled code can be generated for the two inner loops i and k by means of Algorithm 1.

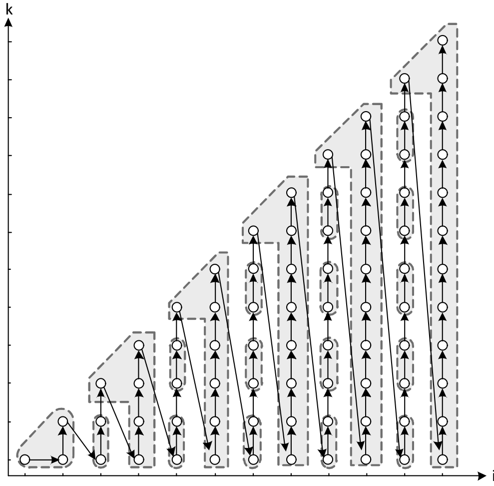

Fig. 4. Target tiles for Kernel 6 with all dependences.

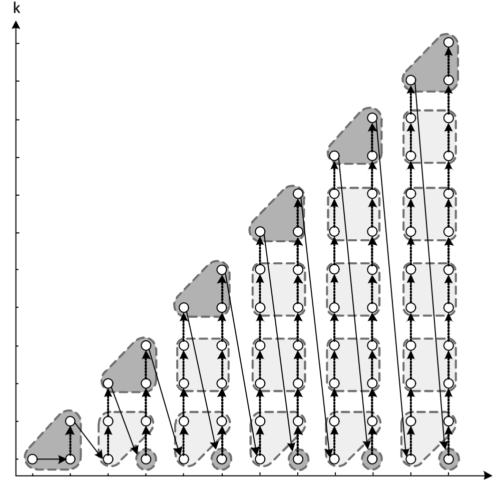

Fig. 5. Target tiles for Kernel 6 without considering reduction dependences.

First, we extract relation R which represents all dependences available in Kernel 6 and next remove all transitive dependences by means of applying the well-known formula $R = R - (R^+ \circ R)$, where $R^+$is the positive transitive closure of R, "$\circ$" is the relation composition operator (Kelly *et al.*, 1996).

The following two relations represent all direct dependences in the two inner loops after removing all transitive dependences:

$$R1 := [n] \to \{[i, -1+i] \to [1+i, 0] : i \geq 1 \text{ and } i \leq -2+n\},$$

$$R2 := [n] \to \{[i, k] \to [i, k'] : i \leq -1+n \text{ and } k \geq 0 \text{ and } k' \geq 1+k \text{ and } k' \leq -1+i\}.$$

Relation $R1$ represents data flow dependences while $R2$ describes reduction dependences.

Traditional data dependence analysis detects flow, output and anti-dependences. If we take into consideration associative and commutative updates, we can change the order in which those updates are done to discover parallelism; for example, each thread can compute a local sum, and then the local sums are summed up (Pugh and Wonnacott, 1994). For this purpose, we can either automatically or manually recognize a commutative and associative update. The dependence between two such updates is termed a reduction dependence.

If we take into consideration both types of dependences represented with relations $R1$ and $R2$, the application of Algorithm 1 to Kernel 6 results in target tiles shown in Fig. 4, provided that the tile size is 2×2. There is no parallelism in the corresponding code because target tiles should be executed serially to respect all dependences.

We may treat reduction dependences in a special way. When we wish to generate serial tiled code, we may ignore all reduction dependences because the order of array elements summation by means of one thread can be arbitrary.

Figure 5 shows target tiles generated with Algorithm 1 when reduction dependences are ignored; the execution of those tiles in serial order respects all dependences represented with relation $R1$. From Fig. 5, we can see that there are the two types of target tiles: rectangular and triangular.

Serial tiled code for Kernel 6 when the tile size is 32×32 is as follows:

```
for(c0=0;c0<=floord(n- 2, 32);c0 += 1)
for(c1=0;c1<= c0;c1 += 1)
 for(c2=32*c0+1;
     c2<=min(n1,32*c0+32);c2++){
    // conditions for triangular tiles
    if(c1==0&&c0==0&&c2>=2)
      w[c2]=w[c2]+b[0][c2]*w[c2-0-1];
    else if(c1==c0&&c0>=1&&c2>=32*c0+2)
      w[c2]=w[c2]+b[0][c2]*w[c2-0-1];
    // rectangular tiles
    for(c3=max(32*c1,-c0+floord(-c0+
        c2-2,31)+1);
        c3<=min(32*c1+31,c2-1);c3+=1)
          w[c2]=w[c2]
                +b[c3][c2]*w[c2-c3-1];
 }
```

To generate parallel tiled code, we can apply tile reduction—an OpenMP tile aware parallelization technique that allows parallel reduction to be performed on multi-dimensional arrays (Gan *et al.*, 2009). OpenMP

(open multi-processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most platforms, processor architectures and operating systems, including Solaris, AIX, HP-UX, Linux,OS X, and Windows (OpenMP Architecture Review Board, 2012).

Hovewer, the approach presented by Gan *et al.* (2009) can deal only with rectangular tiles. We extend that approach to cope with target tiles of arbitrary shapes. This extension divides all target tiles among all threads in an OpenMP parallel region, each thread computes a local sum of elements of array w for assigned rectangular tiles, and then those local sums are summed up in a critical section. Finally, elements of triangular tiles are added to the sum of elements comprised into rectangular tiles. The part of the parallel tiled code for Kernel 6, generated according to the way above, is presented in Fig. 2 of Appendix. Comments in that code clarify the role of particular constructions. The whole source programs used in experiments are presented in the TRACO repository (`https://sourceforge.net/p/traco/code/HEAD/tree/trunk/examples/tile_amcs/`).

PLUTO is not able to produce any tiled code even after removing reduction dependences. The explanation of this fact is the following. The time partition constraint for relation $R1$ is (Lim *et al.*, 1999)

$$C_{11}(i+1) - C_{11}i - C_{12}(i-1) \geq 0$$

or

$$C_{11} - C_{12}(i-1) \geq 0,$$

where C_{11}, C_{12} are the unknown coefficients defining affine transformations. For i satisfying the inequalities $i \geq 1$ and $i \leq -2+n$, there exists single linear independent solution to the constraint above: $C_{11} = 1, C_{12} = 0$. It is well known that the dimension of tiles generated with affine transformations is equal to the number of linear independent solutions to corresponding time partition constraints (Lim *et al.*, 1999). Accordingly, there does not exist any affine transformations allowing even 2-D tiling.

K6 tiled code speed-up is discussed in the following section.

6. Experimental study

The presented algorithm has been implemented in the optimizing compiler TRACO, publicy available at `traco.sourceforge.net`. It includes Petit, a dependence analyser (Kelly *et al.*, 1995) whose output is converted by a preprocessor to the format acceptable by the Barvinok tool (`http://garage.kotnet.org/~skimo/barvinok/barvinok.pdf`), which in turn offers an interface to the functionality provided by the ISL library (`www.kotnet.org/~skimo/isl/manual.pdf`).

TRACO uses this library to apply operations on sets and relations, employed in the presented algorithm, to produce parametric sets defining target tiles. Next, these sets are passed to the CLooG tool to generate tiled code. Finally, a postprocessor forms compilable code in the OpenMP C/C++ standard (OpenMP Architecture Review Board, 2012).

TRACO uses the ISL library function `isl_map_transitive_closure` to calculate the transitive closure of a loop nest dependence graph.

To evaluate the effectiveness of TRACO and the efficiency of tiled code generated with it, we experimented with the NAS Parallel Benchmarks 3.2 (NPB) (NAS, 2015), Polyhedral Benchmarks (PolyBench) (Pol, 2012), and K6 Kernel of the Livermore Loops (McMahon, 1986). The NAS benchmarks are derived from computational fluid dynamics (CFD) applications. The Polybench benchmarks include linear algebra kernels and solvers, data mining, dynamic programming, regularity detection, and a stencil algorithm.

From 431 programs of the NAS benchmark suite, Petit (the Omega project dependence analyzer) is able to analyse 257 ones, and dependences are available in 134 programs (the other 123 ones do not expose any dependence). 60 programs are represented by perfectly nested loops and 74 are described by arbitrarily nested loops. For the Polybench suite, there exist 48 loop nests exposing dependences of which 14 are perfectly nested and 34 are arbitrarily nested.

To compare results produced by means of TRACO with those produced with affine transformations, we chose the state-of-the-art optimizing compiler PLUTO (Bondhugula *et al.*, 2008a), which implements most advanced affine transformation techniques. By means of PLUTO we generated tiled code for all programs under experiments.

Tiled TRACO and PLUTO codes can be found at `http://sourceforge.net/p/traco/code/HEAD/tree/trunk/examples/`.

To asses the effectiveness of the proposed approach, we generated tiled code for all programs presented in both NAS and Polybench benchmarks.

To evaluate the efficiency of tiled programs, we classified for experiments 5 computative intensive programs from the NAS benchmarks and 5 computative intensive programs from the Polybench benchmarks. The program names are given in Figs. 6 and 7. The following criteria were taken into account for choosing those programs: (i) loop nests have to be of depth 2 or more; (ii) the upper loop index bounds have to be parametric; (iii) the loop nest body has to include non-trivial assignment statements such that computation time has to be considerably increased with increasing the

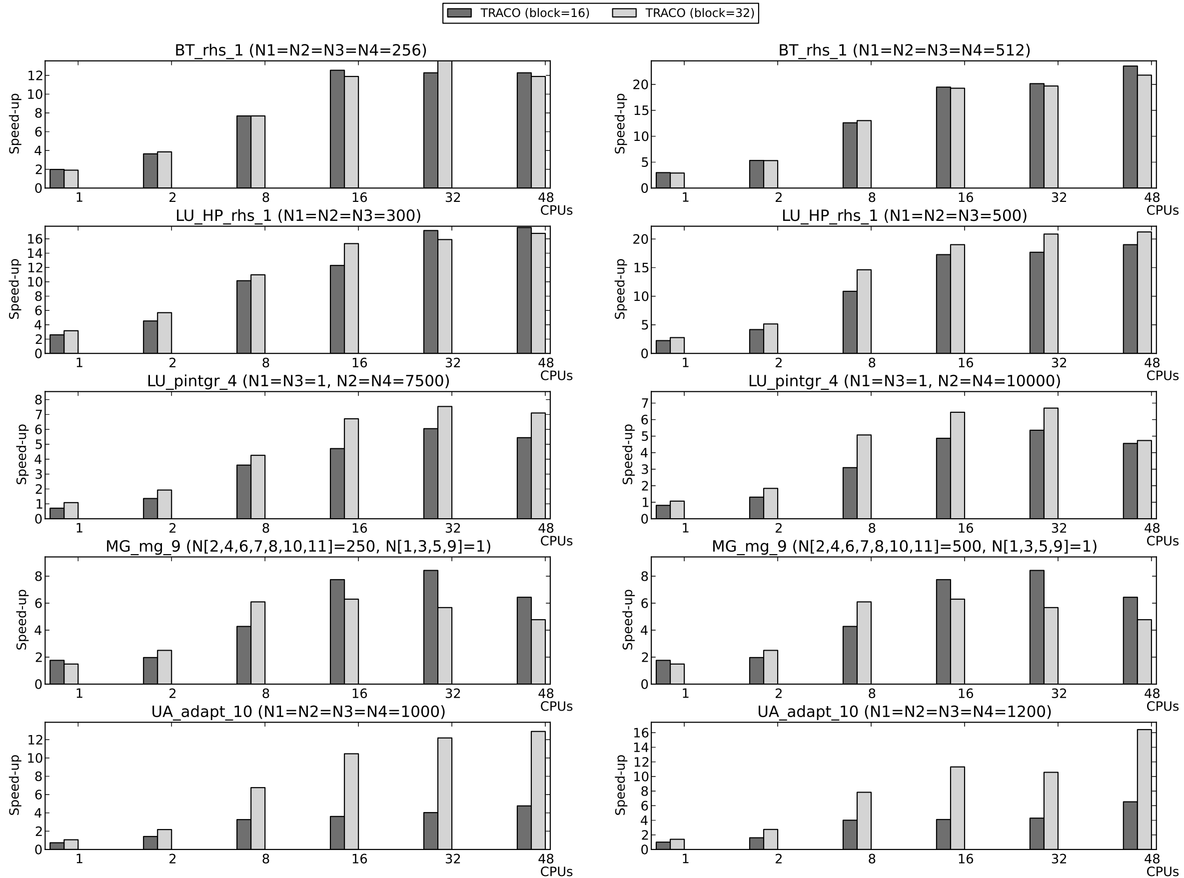

Fig. 6. Speed-up for tiled synchronization-free NBP programs.

values of the upper loop index bounds.

Experiments were carried out by means of a parallel computer with the following specification: 2 × Intel Xeon CPU E5-2695 v2, 2.40GHz, 12 cores, 24 Threads, 30 MB Cache, 16 GB RAM. All programs were compiled with the Intel C Compiler (icc 15.0.2) and optimized at the $-O3$ level.

Analysing the results of experiments for serial tiled code, we may conclude that for NPB and Polybench benchmarks the effectiveness of the presented approach is the same as that of affine transformation techniques implemented in PLUTO, but there exist loop nest samples which can be tiled only by means of the presented approach, for example, K6 Kernel of the Livermore Loops and Example 2 presented in Section 3.

The numbers of tiles in PLUTO and TRACO tiled codes in general are different. TRACO always produces the same number of target tiles as that of the original ones and does not change the loop nest iteration space. In general, for a loop nest, PLUTO tiled code represents more tiles than those generated by TRACO due to the fact that PLUTO can change the original loop nest iteration space skewing it. Despite differences in the examined tiled codes produced with PLUTO and TRACO, we observed similar code performance for 10 computatively intensive programs chosen for experiments.

As far as loop nest transformation time is concerned, we may conclude that for each examined loop nest the PLUTO code generation time is less than the TRACO one. PLUTO takes less than one second to produce tiled code, while TRACO takes from hundred milliseconds to several seconds to return tiled code. TRACO takes the most time for calculating transitive closure and code generation by means of Barvinok function calls. There is a strong need to reduce the computative complexity of transitive closure calculation algorithms. In the future, we plan to use ISL functions directly instead of the Barvinok tool; this will allow us to reduce code generation time.

To generate synchronization-free parallel tiled code, TRACO applies the algorithms presented by Beletska *et al*. (2011), for which input data is tiled code returned by Algorithm 1 and relation *R_TILE* formed as described in Section 4. Experiments carried out with synchronization-free tiled codes produced by TRACO allow us to derive the following conclusions. TRACO is able to generate tiled synchronization-free parallel code

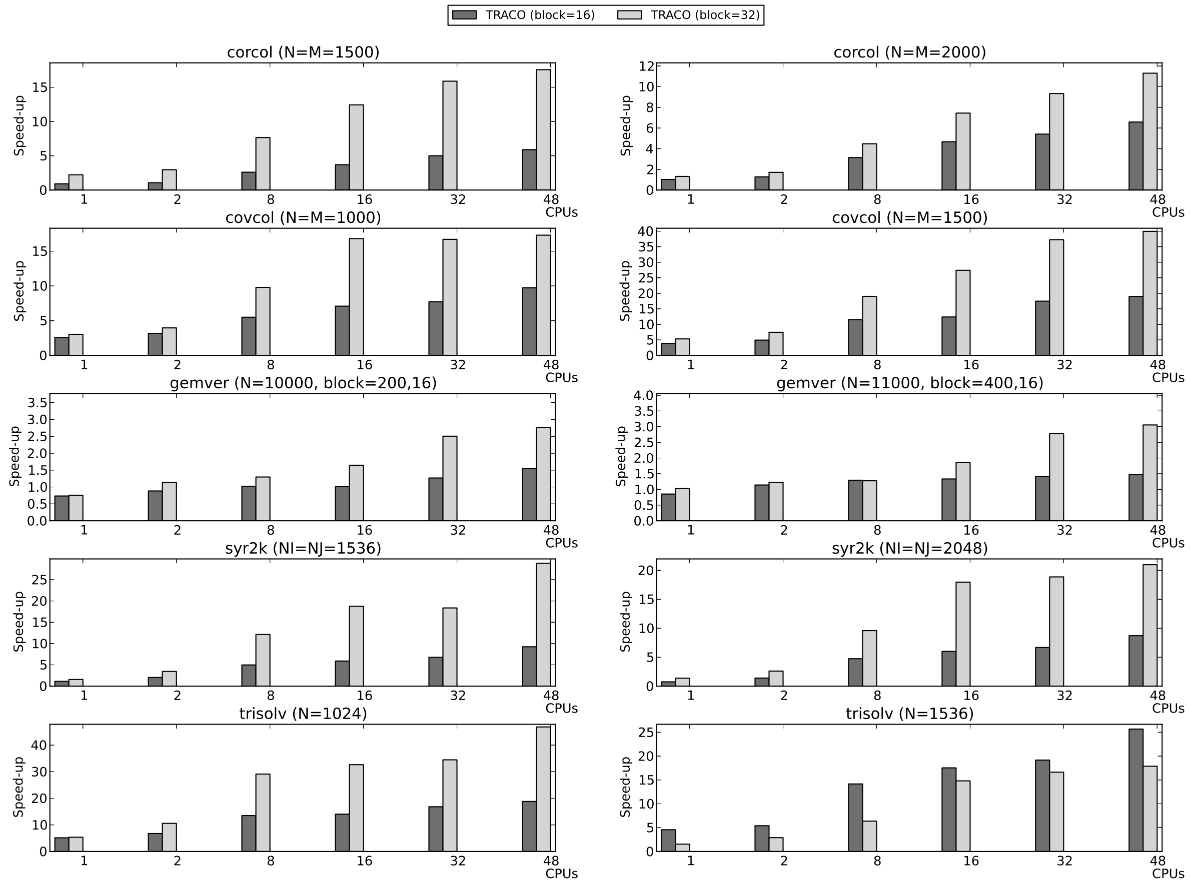

Fig. 7. Speed-up for tiled synchronization-free Polybench programs.

for 43 (32%) of 134 NBP loops and for 16 (50%) of 32 PolyBench loop nests. TRACO and PLUTO generate tiled synchronization-free parallel code for the same NPB and Polybench loop nests; i.e., the effectiveness of TRACO and PLUTO for these benchmark suites is the same.

Figures 6 and 7 present the speed-up of the five tiled synchronization-free Polybench programs and the five synchronization-free NBP programs.

For each benchmark chosen for experiments, we measured execution time for the original and tiled codes produced with TRACO, then speed-up was calculated as the ratio of the execution time of an original code and that of a corresponding (parallel) tiled code. TRACO tiled codes were produced for the various sizes of a problem and the various sizes of the original tile. CPU $= 1$ means that data correspond to the time received for a serial tiled program.

It is worth noting that for the four Polybench programs: corcol, covcol, gemver, and trisolv, even serial tiled codes expose positive speed-up (> 1) while the tiled syr2k program demonstrates positive speed-up only for parallel code (CPUs > 1). All NBP serial tiled codes expose positive speed-up (> 1).

Figure 8 present the speed-up of serial tiled code for Example 2. There is no synchronization-free parallelism for this code.

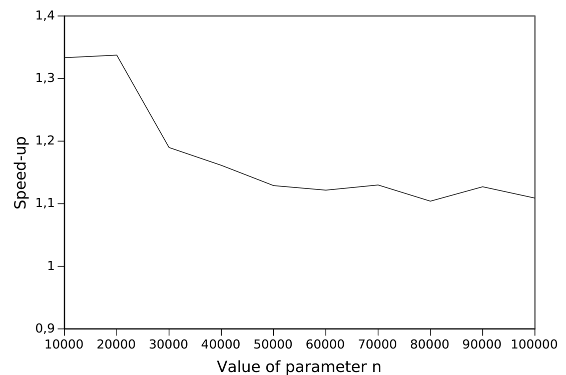

Fig. 8. Speed-up of tiled code for Example 2.

Figure 9 shows speed-up received for both tiled serial and parallel Kernel 6 of the Liveromore Loops, which cannot be tiled with PLUTO. As we can see, parallel tiled code demonstrates high speed-up, which depends on the upper bound values of loop indices and the number of

threads in a parallel region.

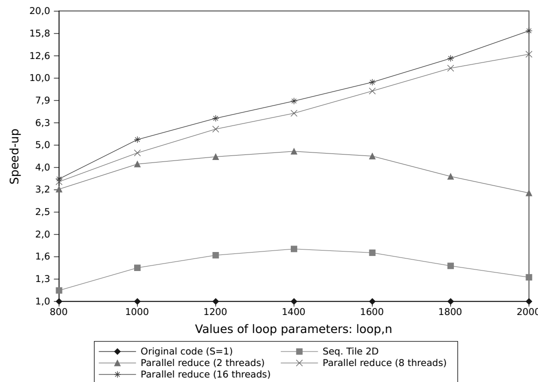

Fig. 9. Speed-up of Kernel 6 codes.

So, we may conclude that, for the examined benchmarks, both serial and parallel tiled codes, generated by means of the presented approach, expose speed-up.

7. Related work

The presented approach is to automatically generate tiled code by means of optimizing compilers. This is why we restrict related work only to techniques aimed at automatic generation of tiled code. There has been a considerable amount of research into tiling, demonstrating how to aggregate a set of loop nest iterations into tiles with each tile as an atomic macro statement, from pioneer papers (Irigoin and Triolet, 1988; Wolf and Lam, 1991; Ramanujam and Sadayappan, 1992) to those presenting advanced techniques (Bondhugula *et al.*, 2008a; Griebl, 2004; Lim *et al.*, 1999; Wonnacott and Strout, 2013). Several popular frameworks are used to produce tiled code automatically: the classic polyhedral model (Feautrier, 1992a; 1992b; Lim and Lam, 1994; Bondhugula *et al.*, 2008a), the sparse polyhedral model (Strout *et al.*, 2004), the non-polyhedral model (Kim and Rajopadhye, 2009), and iteration space slicing (Pugh and Rosser, 1997; 1999).

One of the most advanced reordering transformation frameworks is based on the polyhedral model (Feautrier, 1992a; 1992b; Ramanujam and Sadayappan, 1992; Lim and Lam, 1994; Bondhugula *et al.*, 2008a). Let us recall that this approach includes the following three steps: (i) program analysis aimed at translating high level codes to their polyhedral representation and providing data dependence analysis based on this representation, (ii) program transformation with the aim of improving program locality and/or parallelization, (iii) code generation.

All the above three steps are available in the approach presented in this paper. But there exists the following difference in step (ii): in the polyhedral model a (sequence of) program transformation(s) is represented by a set of affine functions, one for each statement, while the presented approach does not find or use any affine function for program transformation(s). It applies the transitive closure of a program dependence graph to transform invalid original tiles into valid target ones. From this point of view the program transformation step is rather within the iteration space slicing framework introduced by Pugh and Rosser (1997): *Iteration Space Slicing takes dependence information as input to find all statement instances from a given loop nest which must be executed to produce correct values for the specified array elements*. The key step in iteration space slicing is calculating the transitive closure of a loop nest dependence graph.

Summing up, we may conclude that Algorithm 1 is based on a combination of the polyhedral and iteration spacing Slicing frameworks. Such a combination allows improving loop nest tiling transformation effectiveness due to the fact that iteration space slicing allows us to automatically form target tiles so that all original loop nest dependences are respected under the lexicographic order of target tiles.

Transformations based on the polyhedral model produce code at compile-time, while the sparse polyhedral framework (Strout *et al.*, 2004) extends the polyhedral model by using uninterpreted function call abstraction for the compile-time specification of run-time reordering transformations. The approach presented in this paper aims at producing code at compile-time, hence we compare it only with techniques producing tiled code at compile time.

The works of Bondhugula *et al.* (2008a), Griebl (2004) and Lim *et al.* (1999) generalize pioneer techniques and present an advanced theory on tiling, implying that, given a loop nest, first “time-partition constraints” are to be formed, then a solution to them has to be found. The “time-partition constraints” (Feautrier, 1992a; 1992b; Lim *et al.*, 1999) represent the condition that if one iteration is dependent upon another, then the first one must be assigned to a time that is no earlier than that of the second; if they are assigned to the same time, then the first has to be executed after the second. If there exists more than one linearly independent solution to the time-partition constraints of a loop nest, then it is possible to apply a tiling transformation to this loop nest (Lim *et al.*, 1999). Algorithms implemented in PLUTO (Bondhugula *et al.*, 2008a), allow for combining tiling together with the fusion and SCC graph splitting techniques to improve program locality.

Index set splitting is presented by Griebl *et al.* (2000); this approach does not make tiling valid where it is invalid.

Pugh and Rosser (1999) demonstrate by means of several examples how iteration space slicing can be applied to improve program locality due to forming

slices (not fixed tiles). Each slice is composed of dependent statement instances. This improves code locality: while executing each slice, a value produced with some statement instance is immediately consumed by the following statement instances. But the authors do not provide any formal algorithm allowing for extracting slices and code generation.

Beletska *et al*. (2011) and Bielecki *et al*. (2012) demonstrate how to extract coarse- and fine-grained parallelism applying different iteration space slicing algorithms, however, they do not consider any tiling transformation.

Bielecki and Palkowski (2015), Bielecki *et al*. (2015) or Palkowski *et al*. (2015) deal with applying transitive closure to only perfectly nested loops.

Summing up, we may conclude that the approach presented in this paper is the first attempt to demonstrate how iteration space slicing (instead of a set of affine functions, one for each statement, to allow tiling validity) can be used to restructure arbitrarily nested loops in the program transformation step of the polyhedral model to produce valid tiled code.

8. Conclusion

In this paper, we presented a novel approach based on a combination of the polyhedral model and iteration space slicing frameworks that allows automatic generation of tiled code by means of optimizing compilers. The popular affine transformation framework allows tiling only when it is able to convert an original nest loop to a band of fully permutable loops. We suggested to apply the transitive closure of a loop nest dependence graph instead of affine transformations to generate both serial and parallel tiled codes. This allows us to enlarge the scope of loop nests which can be tiled because applying the transitive closure of a dependence graph does not require full permutability of loops to generate tiled code.

The main idea of the approach is to first introduce original rectangular tiles in the loop nest iteration space, then recognize invalid tiles whose enumeration in lexicographic order does not respect the original loop nest dependences, and finally, by means of the transitive closure of a dependence graph, correct (change) those invalid tiles so that their enumeration in lexicographic order is valid. We illustrated this idea by means of a working example and presented a formal algorithm implementing this idea.

We demonstrated by means of Kernel 6 of the Livermore Loops how tiled code can be generated with the presented approach and showed that the affine transformation framework fails to generate any tiled code for this kernel.

The introduced tiling algorithm was implemented in the TRACO optimizing compiler developed by us and made available to the public at the TRACO repository (`https://sourceforge.net/p/traco/code/HEAD/tree/trunk/examples/tile_amcs/`.)

Using TRACO, we carried out an experimental study to evaluate the effectiveness of the approach and the efficiency of tiled code generated by means of this approach. We compared the obtained results with those achieved by means of PLUTO, the most advanced optimizing compiler based on the affine transformation framework. We observed similar tiled code performance for 10 computatively intensive programs chosen for experiments and compiled by means of TRACO and PLUTO.

For Kernel 6 of the Livermore Loops, which can be tiled with TRACO but cannot be tiled by means of PLUTO, we get speed-up for both serial and parallel tiled codes.

One of the limitations of the introduced approach is that original tiles should be only rectangular; this can reduce parallelism degree of tiled code. In the future, in order to increase the tiled code parallelism degree, we plan to present an extended approach allowing tiling arbitrarily nested loops with arbitrary shapes of original tiles.

References

Ahmed, N., Mateev, N. and Pingali, K. (2000). Tiling imperfectly-nested loop nests, *ACM/IEEE 2000 Conference on Supercomputing, Dallas, TX, USA*, Article No. 31.

Andonov, R., Balev, S., Rajopadhye, S. and Yanev, N. (2001). Optimal semi-oblique tiling, *IEEE Transactions on Parallel and Distributed Systems* **14**(9): 940–966.

Bastoul, C. (2004). Code generation in the polyhedral model is easier than you think, *PACT'13, IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France*, pp. 7–16.

Bastoul, C. and Feautrier, P. (2003). Improving data locality by chunking, *International Conference on Compiler Construction, Warsaw, Poland*, pp. 320–335.

Beletska, A., Bielecki, W., Cohen, A., Palkowski, M. and Siedlecki, K. (2011). Coarse-grained loop parallelization: Iteration space slicing vs affine transformations, *Parallel Computing* **37**(8): 479–497.

Bielecki, W., Kraska, K. and Klimek, T. (2014). Using basis dependence distance vectors to calculate the transitive closure of dependence relations by means of the Floyd–Warshall algorithm, *Journal of Combinatorial Optimization* **30**(2): 253–275.

Bielecki, W., Klimek, T., Palkowski, M. and Beletska, A. (2010). An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations, *in* W. Wu and O. Daescu (Eds.), *COCOA 2010: Fourth International Conference on Combinatorial Optimization and Applications*, Lecture Notes in Computer Science, Vol. 6508, Springer, Berlin/Heidelberg, pp. 104–113.

Bielecki, W. and Palkowski, M. (2015). Perfectly nested loop tiling transformations based on the transitive closure of the program dependence graph, *in* A. Wilinski *et al.* (Eds.), *Soft Computing in Computer and Information Science*, Advances in Intelligent Systems and Computing, Vol. 342, Springer International Publishing, Cham, pp. 309–320.

Bielecki, W., Palkowski, M. and Klimek, T. (2012). Free scheduling for statement instances of parameterized arbitrarily nested affine loops, *Parallel Computing* **38**(9): 518–532.

Bielecki, W., Palkowski, M. and Klimek, T. (2015). Free scheduling of tiles based on the transitive closure of dependence graphs, *in* R. Wyrzykowski (Ed.), *11th International Conference on Parallel Processing and Applied Mathematics*, Part II, Lecture Notes in Computer Science, Vol. 9574, Springer, Berlin/Heidelberg, pp. 133–142.

Błaszczyk, J., Karbowski, A. and Malinowski, K. (2007). Object library of algorithms for dynamic optimization problems: Benchmarking SQP and nonlinear interior point methods, *International Journal of Applied Mathematics and Computer Science* **17**(4): 515–537, DOI: 10.2478/v10006-007-0043-y.

Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A. and Sadayappan, P. (2008a). Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model, *in* L. Hendren (Ed.), *Compiler Constructure*, Lecture Notes in Computer Science, Vol. 4959, Springer, Berlin/Heidelberg, pp. 132–146.

Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P. (2008b). A practical automatic polyhedral parallelizer and locality optimizer, *ACM SIGPLAN Notices* **43**(6): 101–113.

Campbell, S.L. (2001). Numerical analysis and systems theory, *International Journal of Applied Mathematics and Computer Science* **11**(5): 1025–1034.

Feautrier, P. (1992a). Some efficient solutions to the affine scheduling problem, I: One-dimensional time, *International Journal of Parallel Programming* **21**(5): 313–348.

Feautrier, P. (1992b). Some efficient solutions to the affine scheduling problem, II: Multidimensional time, *International Journal of Parallel Programming* **21**(6): 389–420.

Gan, G., Wang, X., Manzano, J. and Gao, G.R. (2009). Tile reduction: The first step towards tile aware parallelization in openmp, *in* M.S. Muller *et al.* (Eds.), *Evolving OpenMP in an Age of Extreme Parallelism*, Springer, Berlin/Heidelberg, pp. 140–153.

Greenbaum, A. and Chartier, T.P. (2012). *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms*, Princeton University Press, Princeton, NJ.

Griebl, M. (2004). *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*, D.Sc. thesis, University of Passau, Passau.

Griebl, M., Feautrier, P. and Lengauer, C. (2000). Index set splitting, *International Journal of Parallel Programming* **28**(6): 607–631.

Grosser, T., Cohen, A., Kelly, P.H., Ramanujam, J., Sadayappan, P. and Verdoolaege, S. (2013). Split tiling for GPUS: Automatic parallelization using trapezoidal tiles, *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, Houston, TX, USA*, pp. 24–31.

Irigoin, F. and Triolet, R. (1988). Supernode partitioning, *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'88, San Diego, CA, USA*, pp. 319–329.

Jeffers, J. and Reinders, J. (2015). *High Performance Parallelism Pearls, Volume Two: Multicore and Many-Core Programming Approaches*, Morgan Kaufmann, Burlington, MA.

Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T. and Wonnacott, D. (1995). The omega library interface guide, *Technical report*, University of Maryland at College Park, MD.

Kelly, W., Pugh, W., Rosser, E. and Shpeisman, T. (1996). Transitive closure of infinite graphs and its applications, *International Journal of Parallel Programming* **24**(6): 579–598.

Kim, D. and Rajopadhye, S.V. (2009). Parameterized tiling for imperfectly nested loops, *Technical Report CS-09-101*, Colorado State University, Fort Collins, CO.

Kowarschik, M. and Weiß, C. (2003). An overview of cache optimization techniques and cache-aware numerical algorithms, *in* U. Meyer *et al.* (Eds.), *Algorithms for Memory Hierarchies*, Springer, Berlin/Heidelberg, pp. 213–232.

Leader, J.J. (2004). *Numerical Analysis and Scientific Computation*, Pearson Addison/Wesley Boston, MA.

Lim, A., Cheong, G.I. and Lam, M.S. (1999). An affine partitioning algorithm to maximize parallelism and minimize communication, *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, Rhodes, Greece*, pp. 228–237.

Lim, A.W. and Lam, M.S. (1994). Communication-free parallelization via affine transformations, *in* K. Pingali *et al.* (Eds.), *24th ACM Symposium on Principles of Programming Languages*, Springer-Verlag, Berlin/Heidelberg, pp. 92–106.

Maciążek, M., Grabowski, D. and Pasko, M. (2015). Genetic and combinatorial algorithms for optimal sizing and placement of active power filters, *International Journal of Applied Mathematics and Computer Science* **25**(2): 269–279, DOI: 10.1515/amcs-2015-0021.

McMahon, F.H. (1986). The Livermore Fortran kernels: A computer test of the numerical performance range, *Technical Report UCRL-53745*, Lawrence Livermore National Laboratory, Livermore, CA.

Mullapudi, R.T. and Bondhugula, U. (2014). Tiling for dynamic scheduling, *IMPACT 2014, 14th International Workshop on Polyhedral Compilation Techniques, Vienna, Austria*.

NAS (2015). NAS benchmarks suite, `http://www.nas.nasa.gov`.

OpenMP Architecture Review Board (2012). OpenMP application program interface version 4.0, http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf.

Palkowski, M., Klimek, T. and Bielecki, W. (2015). TRACO: An automatic loop nest parallelizer for numerical applications, *Federated Conference on Computer Science and Information Systems, Łódź, Poland*, pp. 681–686

Pol (2012). The Polyhedral benchmark suite, http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

Pugh, W. and Rosser, E. (1997). Iteration space slicing and its application to communication optimization, *International Conference on Supercomputing, Vienna, Austria*, pp. 221–228.

Pugh, W. and Rosser, E. (1999). Iteration space slicing for locality, *in* L. Carter and J. Ferrante (Eds.), *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Vol. 1863, Springer, Berlin/Heidelberg, pp. 164–184.

Pugh, W. and Wonnacott, D. (1993). An exact method for analysis of value-based array data dependences, *6th Annual Workshop on Programming Languages and Compilers for Parallel Computing, Portland, OR, USA*, pp. 546–566.

Pugh, W. and Wonnacott, D. (1994). Static analysis of upper and lower bounds on dependences and parallelism, *ACM Transactions on Programming Languages and Systems* **16**(4): 1248–1278.

Ramanujam, J. and Sadayappan, P. (1992). Tiling multidimensional iteration spaces for multicomputers, *Journal of Parallel and Distributed Computing* **16**(2): 108–120.

Sass, R. and Mutka, M. (1994). Enabling unimodular transformations, *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Washington, DC, USA*, pp. 753–762.

Strout, M.M., Carter, L., Ferrante, J. and Kreaseck, B. (2004). Sparse tiling for stationary iterative methods, *International Journal of High Performance Computing Applications* **18**(1): 2004.

Tang, P. and Xue, J. (2000). Generating efficient tiled code for distributed memory machines, *Parallel Computing* **26**(11): 1369–1410.

Verdoolaege, S. (2011). Integer set library—manual, http://www.kotnet.org/~skimo//isl/manual.pdf.

Verdoolaege, S. (2012). Barvinok: User guide, Barvinok-0.36, www.garage.kotnet.org/~skimo/barvinok/barvinok.pdf.

Verdoolaege, S., Cohen, A. and Beletska, A. (2011). Transitive closures of affine integer tuple relations and their overapproximations, *in* E. Yahav (Ed.), *Proceedings of the 18th international Conference on Static analysis, SAS'11*, Springer-Verlag, Berlin/Heidelberg, pp. 216–232.

Wolf, M.E. and Lam, M.S. (1991). A data locality optimizing algorithm, *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, Toronto, Canada*, pp. 30–44.

Wonnacott, D.G. and Strout, M.M. (2013). On the scalability of loop tiling techniques, *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT), Berlin, Germany*.

Wonnacott, D., Jin, T. and Lake, A. (2015). Automatic tiling of mostly-tileable loop nests, *IMPACT 2015, 5th International Workshop on Polyhedral Compilation Techniques, Amsterdam, The Netherlands*.

Xue, J. (1996). Communication-minimal tiling of uniform dependence loops, *Languages and Compilers for Parallel Computing*, Springer, Berlin/Heidelberg, pp. 330–349.

Xue, J. (1997). On tiling as a loop transformation, *Parallel Processing Letters* **7**(4): 409–424.

Xue, J. (2012). *Loop Tiling for Parallelism*, Springer Science & Business Media, Springer-Verlag, New York, NY.

Zdunek, R. (2014). Regularized nonnegative matrix factorization: Geometrical interpretation and application to spectral unmixing, *International Journal of Applied Mathematics and Computer Science* **24**(2): 233–247, DOI: 10.2478/amcs-2014-0017.

Włodzimierz Bielecki is a full professor, the head of the Software Technology Department of the West Pomeranian University of Technology in Szczecin, Poland. His research interest includes parallel and distributed computing, optimizing compilers, and techniques of extraction of both fine- and coarse-grained parallelism available in program loop nests based on the transitive closure of dependence graphs.

Marek Pałkowski has graduated and obtained his Ph.D. degree in computer science from the Technical University of Szczecin, Poland. The main goal of his research is extraction of parallelism available in program loop nests using the transitive closure of dependence graphs, and development of the publicly available TRACO compiler implementing parallelization techniques based on the transitive closure of dependence graphs.

Appendix

Figure A1 presents tiled code for Example 2. Figure A2 presents parallel tiled code for K6 Kernel of the Livermore Loops.

```
for (c1 = 0; c1 <= floord(n, 32); c1 += 1)
  for (c2 = 0; c2 <= 2; c2 += 1) {
    if (c2 == 2) {
      for (c5 = 32 * c1; c5 <= min(n, 32 * c1 + 31); c5 += 1)
        for (c6 = max(1, 32 * c1 - c5 + 2); c6 <= 2; c6 += 1) {
          if (c6 == 2) {
            d[c5][n]=a[c5+1][n]+a[c5][n];
          } else
            for (c7 = max(n + 32 * c1 - c5 + 1, -(n % 32) + n); c7 <= n; c7 += 1)
              a[c5][c7]=a[c5+1][c7-1]+b[c5+1][c7]+b[c5][0]+a[c5][c7+1];
        }
    } else if (c2 == 1) {
      for (c3 = 0; c3 <= n / 32; c3 += 1)
        for (c5 = 32 * c1; c5 <= min(n, 32 * c1 + 31); c5 += 1) {
          if (c3 == 0 && c5 >= 32 * c1 + 1)
            b[c5][0]=c[c5][0];
          for (c7 = max(0, 32 * c1 + 32 * c3 - c5);
               c7 <= min(n + 32 * c1 - c5, 32 * c1 + 32 * c3 - c5 + 31); c7 += 1)
            a[c5][c7]=a[c5+1][c7-1]+b[c5+1][c7]+b[c5][0]+a[c5][c7+1];
        }
    } else
      b[32*c1][0]=c[32*c1][0];
  }
```

Fig. A1. Tiled code for Example 2.

```
omp_set_num_threads(kind);  // sets the number of threads in a parallel region
int btile=32;               // btile defines the value of tile side
long double w_[48][32];     // array w_ is to be declared as shared in the parallel region
                            // 32 is the maximal tile side; 48 is the maxiaml number of threads

int lb_c0,thread_id,block,lb,ub,num_threads;
num_threads = kind;         //num_threads defines the number of threads in the parallel region

for ( l=1 ; l<=loop ; l++ ) {                              // serial no tiled loop
 for (c0 = 0; c0 <= floord(n - 2, 32); c0 += 1){          // defines tile id along axis i
  lb_c0 = 32*c0+1;                                         // offset for the value of index i

  // Pragma omp parallel defines a parallel region, which is code that will be
  // executed by multiple threads in parallel.
  // 'private' specifies that each thread should have its own instance of a variable.
  // 'shared' specifies that the variables should be shared among all threads.

 #pragma omp parallel  private(i,w_,c1,c2,c3,lb,ub,thread_id) shared(c0,b,lb_c0,num_threads,btile)
 {
   thread_id = omp_get_thread_num();                      //returns the thread number
   block = c0/num_threads ;          // specifies the number of tiles along axis i for a given c0

   lb = thread_id * block;        //lower bound of c1 for the  tread with the identifier thread_id
   ub = min((thread_id+1) * block, c0-1);                 // upper bound of c1 for the thread with
                                                          // the identifier thread_id

   for(i=0; i<btile; i++)              // zeroing elements of array w_
     w_[thread_id][i] = 0;

   for (c1 = lb; c1 <= ub; c1 += 1)      //  this and the next two loops enumerate iterations
                                         //  of 2-d tiles assigned to the thread with
                                         //  the identifier  thread_id
     for (c2 =32*c0+1; c2 <= min(n - 1, 32 * c0 + 32); c2 += 1) {
       if (c1 == 0 && c0 == 0 && c2 >= 2)
         w_[thread_id][c2-lb_c0] += b[0][c2]*w[c2-0-1];
       for (c3 = max(32*c1, -c0 + floord(-c0 + c2-2, 31) + 1); c3 <= min(32*c1 + 31, c2-1); c3++)
         w_[thread_id][c2-lb_c0] += b[c3][c2]*w[c2-c3-1];
     }

   #pragma omp critical //specifies that the code below is only executed on one thread at a time.
   {
     for(i=0; i<btile; i++)   //sums all local w_[thread_id][i] calculated for  rectangular tiles
       w[lb_c0+i] += w_[thread_id][i];
   }

  }

  // taking into account elements of triangular tiles
  for (c2 = 32 * c0 + 1; c2 <= min(n - 1, 32 * c0 + 32); c2 += 1) {
   // c1 = c0  block
   if (c0 >= 1 && c2 >= 32 * c0 + 2)
     w[c2]=w[c2]+b[0][c2]*w[c2-0-1];
   for (c3 = max(32 * c0, -c0 + floord(-c0 + c2 - 2, 31) + 1); c3 <= min(32*c0 + 31, c2-1); c3++)
     w[c2]=w[c2]+b[c3][c2]*w[c2-c3-1];
  }
 }
}
```

Fig. A2. Main part of parallel tiled code for Kernel 6 of the Livermore Loops.

Received: 3 November 2015
Revised: 12 April 2016
Re-revised: 5 June 2016
Accepted: 9 August 2016