DOI: 10.15439/2015F34
ACSIS, Vol. 5

# TRACO: An Automatic Loop Nest Parallelizer for Numerical Applications

Marek Palkowski, Tomasz Klimek, Wlodzimierz Bielecki
West Pomeranian University of Technology in Szczecin
ul. Zolnierska 49, 71-210 Szczecin, Poland
Email: mpalkowski@wi.zut.edu.pl, tklimek@wi.zut.edu.pl, wbielecki@wi.zut.edu.pl

***Abstract*—We present the source-to-source TRACO compiler allowing for increasing program locality and parallelizing arbitrarily nested loop sequences in numerical applications. Algorithms for generation of tiled code and extracting synchronization-free slices composed of tiles are presented. Parallelism of arbitrary nested loops is obtained by creating a kernel of computations represented in the OpenMP standard to be executed independently on many CPUs. We consider benchmarks, typical from compute-intensive sequences of algebra operations or numerical computation from industry and engineering. The speed-up of programs generated by TRACO are discussed. Related compilers and techniques are considered. Future work is outlined.**

## I. Introduction

EFFICIENT parallel numerical algorithms for commonly occurring problems in scientific computing are more difficult to write than sequential ones. Developers must analyze their performance, granularity and scalability. Optimizing and parallelizing compiler research with empirical evaluation is significant for an efficient usage of widely available multi-core systems.

Because for many numerical kernels and solvers most computations are contained in program loop nests, automatic extraction of parallelism available in loop nests is extremely important for multi-core processing. However, there is a lack of automated and completed tools permitting for exposing parallelism in serial programs. The most advanced approach to improve program locality and parallelization is based on the Affine Transformation Framework (ATF). Unfortunately, this approach can fail to parallelize loop nests exposing storage-related dependences, and as consequence potential parallelism is left unexploited in some cases [1]. This paper presents an alternative approach to increasing program locality and parallelization which is implemented in the open source tool, TRACO [2], based on calculating the **TRA**nsitive **C**l**O**sure of dependence graphs. It currently parallelizes loop nests being written in the C language.

The purpose of this source-to-source compiler is to automatically convert existing serial numerical applications to parallel ones to be run on multicore systems and high performance computers. It produces parallel target code that is semantically identical with original source code.

## II. Background

The source-to-source TRACO compiler implements Iteration Space Slicing (ISS) techniques together with the free-scheduling, variable privatization and parallel reduction techniques. Output code, produced by TRACO, is compilable and contains OpenMP directives [3]. TRACO is available at the website http://traco.sourceforge.net.

ISS was introduced by Pugh and Rosser [4]. It takes dependence information as input to find all statement instances that must be executed to produce the correct values for the specified array elements. Dependences available in a loop nest are described by dependence relations with constraints presented by means of the Presburger arithmetic that is the first-order theory of the integers in the language $L$ having 0, 1 as constants, +,- as binary operations, and equality =, order $<$ and congruences $\equiv_n$ modulo all integers $n \geq 1$ as binary relations.

Coarse-grained code is presented with synchronization-free slices. TRACO uses the dependence analysis [5] proposed by Pugh and Wonnacott where dependences are represented by dependence relations. This analysis is implemented in Petit [6].

Standard operations on relations and sets are used, such as intersection (∩), union (∪), difference (-), domain (dom $R$), range (ran $R$), relation application ($S'= R(S)$: $e' \in S'$iff exists $e$ s.t. $e \rightarrow e' \in R, e \in S$), positive transitive closure of relation $R$, $R+$ = $\{[e] \rightarrow [e'] : e \rightarrow e' \in R \vee \exists\ e'', e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R+\}$, transitive closure $R* = R+ \cup I$. In detail, the description of these operations is presented in papers [5], [7].

The positive transitive closure for a given relation $R$, $R^+$, is defined as follows [7]
$R^+ = \{e \rightarrow e' :\ e \rightarrow e' \in R \vee \exists e'' s.t.\ e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+\}$.
It describes which vertices $e'$ in a dependence graph (represented by relation $R$) are connected directly or transitively with vertex $e$.

Transitive closure, $R^*$, is defined as follows [8]: $R^* = R^+ \cup I$, where $I$ the identity relation. It describes the same connections in a dependence graph (represented by $R$) that $R^+$ does plus connections of each vertex with itself.

To perform operations on sets and relations as well as calculating transitive closure, TRACO uses the ISL library [9].

## III. Iteration Space Slicing for Parallelism Extraction

ISS algorithms, presented in paper [1], allow us to generate parallel code representing synchronization-free slices. An (iteration-space) slice is defined as follows.

**Definition 1**. Given a dependence graph defined by a set of dependence relations, a slice $S$ is a weakly connected component of this graph, i.e., a maximal sub-graph such that for each pair of vertices in the sub-graph there exists a forward or backward path.

**Definition 2**. An ultimate dependence source is a source that is not the destination of another dependence. Given a dependence relation *R,* describing all the dependences in a loop, set, $S_{UDS}$, including all ultimate dependence sources can be calculated as domain(*R*) - range(*R*).

**Definition 3**. The representative of a slice is its lexicographically minimal ultimate source.

An approach to extract synchronization-free slices takes two steps [1]. First, for each slice, a representative statement instance is defined (it is the lexicographically minimal statement instance from all the sources of a slice). Next, slices are reconstructed from their representatives and code scanning these slices is generated.

Given a dependence relation *R,* describing all the dependences in a loop, we calculate set of all ultimate dependence sources of slices , $S_{UDS}$, as follows

$$S_{UDS} = domain(R) - range(R). \tag{1}$$

In order to find elements of $S_{UDS}$ that are representatives of slices, we build a relation, $R_{USC}$, that describes all pairs of the ultimate dependence sources being transitively connected in a slice, as follows:

$$R_{USC} = \{[e] \to [e'] : e, e' \in S_{UDS}, e \prec e', (R^*(e) \cap R^*(e'))\}. \tag{2}$$

The condition ($e \prec e'$) in the constraints of relation $R_{USC}$ above means that *e* is lexicographically smaller than *e*′. Such a condition guarantees that the lexicographically smallest element from *e* and *e*′ will always appear in the input tuple of $R_{USC}$ (its representative source), i.e., it can never appear in the output tuple. The intersection (*R**(*e*)∩*R**(*e*′)) in the constraints of $R_{USC}$ guarantees that vertices *e* and *e*′ are transitively connected, i.e., they are the sources of the same slice.

Next, set, $S_{repr}$, containing representatives of each slice is found as $S_{repr}$ = $S_{UDS}$ - range($R_{USC}$). Each element *e* of set $S_{repr}$ is the lexicographically minimal statement instance of a synchronization-free slice. If *e* is the representative of a slice with multiple sources, then the remaining sources of this slice can be found applying relation ($R_{USC}$)* to *e*, i.e., ($R_{USC}$)*(*e*). If a slice has the only source, then ($R_{USC}$)*(*e*)=*e*. The elements of a slice represented with *e* can be found applying relation *R** to the set of sources of this slice:

$$S_{slice} = R^*((R_{USC})^*(e)). \tag{3}$$

To generate code, we insert in the first positions of the tuple of set $S_{slice}$ the elements of the tuple of set $S_{repr}$ (together with corresponding constraints) and apply the CLooG library [10] to so extended set $S_{slice}$ .

To parallelize loop nests which expose a single synchronization-free slice, time partitioning can be applied. The algorithm, presented in our paper [11], allows us to generate time partitions and corresponding fine-grained parallel code on the basis of the free schedule; all statement instances of a time partition can be executed in parallel, while partitions are enumerated sequentially. The free schedule function is defined as follows.

**Definition 4** [12]. The *free schedule* is the function that assigns discrete time of execution to each loop statement instance as soon as its operands are available, that is, it is mapping $\sigma$:*LD*→ $\mathbb{Z}$ such that

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

where $p, p_1, p_2, ..., p_n$ are loop statement instances, *LD* is the loop domain, $p_1 \to p, p_2 \to p, ..., p_n \to p$ mean that the pairs $p_1$ and $p$, $p_2$ and $p$, ...,$p_n$ and $p$ are dependent, *p* represents the dependence destination, while $p_1, p_2, ..., p_n$ represent the sources of dependences, *n* is the number of operands of statement instance *p* (the number of dependences whose destination is statement instance *p*). The free schedule is the fastest legal schedule [12]. In paper [11] we presented fine-grained parallelism extraction based on the power *k* of relation *R*.

The idea of the algorithm is the following [11]. Given relations $R_1$, $R_2$, ..., $R_m$, representing all dependences in a loop nest, we first calculate $R = \bigcup_{i=1}^{m} R_i$ and then $R^k$, where $R^k = \underbrace{R \circ R \circ ...R}_{k}$, "∘" is the composition operation. Techniques of calculating the power *k* of relation *R* are presented in the following publications [8], [9], [14] and they are out of the scope of this paper. Let us only note that given transitive closure $R^+$, we can easily convert it to the power *k* of *R*, $R^k$, and vice versa, for details see [9].

Given set *UDS* comprising all loop nest statement instances that are ready to execution at time *k*=0, each vertex, belonging the set $S_k$ = $R^k(UDS)$ - $R^+ \circ R^k(UDS)$, is connected in the dependence graph, defined by relation *R*, with some vertex(ices) represented by set *UDS* with a path of length *k* . Hence at time *k*, all the statement instances belonging to the set $S_k$ can be scheduled for execution and it is guaranteed that *k* is as few as possible.

## IV. Loop Tiling

Tiling is a very important iteration reordering transformation for both improving data locality and extracting loop nest parallelism. TRACO allows users generate parallel tiled code by means of algorithms based on the transitive closure of a dependence graph [13].

First, we form set *TILE(**II**, **B**)* including iterations belonging to a parametric tile as follows

*TILE*(***II**, **B***) = {[***I***] | ***B*****II*** +***LB*** $\leq$ ***I*** $\leq$ min( ***B****(***II*** +**1**) + ***LB*** -**1**, ***UB***) AND ***II*** $\geq$ 0}, where vectors ***LB*** and ***UB*** include the lower and upper loop index bounds of an original loop nest, respectively; diagonal matrix ***B*** defines the size of a rectangular original tile; elements of vectors ***I*** and ***II*** represent the original loop nest indices and the identifiers of tiles, respectively; **1** is the vector whose all elements have value 1.

*TILE*(***II**, **B***) represents a tile of the rectangular shape with a fixed size defined by the user.

Sets *TILE_LT* and *TILE_GT* are the unions of all the tiles whose identifiers are lexicographically less and greater than that of *TILE*(***II***, ***B***), respectively.

*TILE_LT* ={[***I***] | exists ***II***$'$ s. t. ***II***$' \prec$ ***II*** AND ***II***, ***II***$'$ in *II_SET* AND ***I*** in *TILE*(***II***$'$, ***B***)},

*TILE_GT* ={[***I***] | exists ***II***$'$ s. t. ***II***$' \succ$ ***II*** AND ***II***, ***II***$'$ in *II_SET* AND ***I*** in *TILE*(***II***$'$, ***B***)}.

Set *TILE_ITR* = *TILE* - $R^+$( *TILE_GT* ) does not include any invalid dependence target.

Set *TVLD_LT* = ( $R^+$(*TILE_ITR*) $\cap$ *TILE_LT*) - $R^+$(*TILE_GT*) includes all the iterations that i) belong to the tiles whose identifiers are lexicographically less than that of set *TILE_ITR*, ii) are the targets of the dependences whose sources are contained in set *TILE_ITR*, and iii) are not any target of a dependence whose source belong to set *TILE_GT*.

Set *TILE_VLD* = *TILE_ITR* $\cup$ *TVLD_LT* represents target tiles.

To generate code, we form set *TILE_VLD_EXT* by means of inserting i) into the first positions of the tuple of set *TILE_VLD* indices $ii_1, ii_2, ..., ii_d$; ii) into the constraints of set *TILE_VLD* the constraints defining tile identifiers ***II*** $\geq$ 0 and ***B*****II***+***LB*** $\leq$ ***UB***.

The resulting code can be produced by means of applying any code generator to scan elements of set *TILE_VLD_EXT* in the lexicographic order, for example, CLooG [10]. TRACO implements an extended approach allowing for tiling imperfectly nested loops also.

All the presented algorithms implemented in the TRACO compiler are based on the transitive closure of a dependence relation representing all dependences in a loop nest. Loop nest tiling and iteration space slicing can be combined in order to increase program locality and the grain size of parallel code. For this purpose, we form relation *R_TILE* that describes dependences among all tiles but ignores dependences within each tile as follows.

*R_TILE*:={[***II***]->[***JJ***]: exist ***I***, ***J*** s.t. (***II***, ***I***) in *TILE_VLD_EXT*(***II***) *AND* (***JJ***, ***J***) in *TILE_VLD_EXT*$_i$(***JJ***) *AND* ***J*** in *R*(***I***)},

where ***II***, ***JJ*** are the vectors representing tile identifiers. Such a relation can be used to extract slices comprising tiles or free-scheduling in the same way as it is described in Section III except from instead of relation $R$ relation *R_TILE* has to be used.

## V. TRACO USAGE FOR THE HYDRO-FRAGMENT CODE

In this section, we present the usage of the TRACO compiler to optimize the code of the hydrodynamics fragment of the first kernel of the Livermoore Loops suite (*k1*) [15].

```
for ( l=1 ; l<=loop ; l++ ) {
 for ( k=0 ; k<n ; k++ ) {
  x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
 }
}
```

The following relation describes dependences in this program.

*R* := [n,loop] -> { [l,k,13] -> [l',k',13] :( k' = k and 1 <= l < l' <= loop and 0 <= k < n and l < loop and 1 <= loop and 2 <= l' ) },

where here and further on "13"states for the loop nest statement identifier, defined by the line number of this statement in the source code.

First of all, we generate parallel synchronization-free code. For this purpose, we calculate the transitive closure of relation *R*, $R^*$:

$R^*$ := [n, loop] -> { [l, k, 13] -> [l', k, 13] : l >= 1 and k >= 0 and k <= -1 + n and l' >= 1 + l and l' <= loop; [l, k, v] -> [l, k, v] }.

Next, we expose slice representatives. Because $R_{UCS}$ = $\oslash$, we have

S$_{REPR}$ =*UDS*:= {[n, loop] -> [0, i1, 13] : loop >= 17 and i1 >= 0 and 16i1 <= -1 + n }.

Now, we form set, $S_{slice}$, representing slices:

$S_{slice}$ := [n, loop, c1] -> ] [i0, c1, 13] : c1 >= 0 and c1 <= -1 + n and i0 <= loop and loop >= 2 and i0 >= 1 },

and insert in the first positions of the tuple of set $S_{slice}$ the elements of the tuple of set $S_{REPR}$(together with corresponding constraints) and apply to so extended set $S_{slice}$ CLooG to get the following parallel code.

```
if (n + loop >= 3 && loop >= 1)
#pragma omp parallel for
  for (c1 = 0; c1 < n; c1 += 1)
      if (c1 >= 0 && loop >= 2 && n >= c1 + 1)
          for (c0 = 1; c0 <= loop; c0 += 1)
          x[c1]=q+y[c1]*(r*z[c1+10]+t*z[c1+11]);
```

Below, we demonstrate how tiled code can be generated for the same example. First, we define a rectangular parametric tile of the size 16x16 as follows.

*TILE*:= [ll, kk, n, loop] -> { [l, k, 13] : l >= 1 + 16ll and l >= 1 and l <= loop and l <= 16 + 16ll and k >= 16kk and k >= 0 and k <= -1 + n and k <= 15 + 16kk and ll >= 0 and kk >= 0 and loop >= 1 and n >= 1 }.

Next, we calculate the following sets:

*TILE_LT*:= [ll, kk, n, loop] -> { [l, k, 13] : l >= 1 + 16ll and l >= 1 and l <= loop and l <= 16 + 16ll and k >= 0 and k <= -1 + 16kk and n >= 1 + 16kk and ll >= 0 and loop >= 16ll and kk >= 0 },

*TILE_GT*:= [ll, kk, n, loop] -> { [l, k, 13] : l >= 1 + 16ll and l >= 1 and l <= loop and l <= 16 + 16ll and k >= 16 + 16kk and k <= -1 + n and ll >= 0 and loop >= 16ll and kk >= 0 and n >= 1 + 16kk , *TILE_ITR*:= [n, loop, ll, kk] -> [l, k, 13] : ll >= 0 and kk >= 0 and l >= 1 + 16ll and l <= 16 + 16ll and l <= loop and k >= 16kk and k <= 15 + 16kk and k <= -1 + n },

*TVLD_LT* = ∅, *TILE_VLD* = *TILE_ITR*,

*TILE_VLD_EXT*:= [n, loop] -> { [i0, i1, i2, i3, 13] : i0 >= 0 and i1 >= 0 and i2 >= 1 + 16i0 and i2 <= 16 + 16i0 and i2 <= loop and i3 >= 16i1 and i3 <= 15 + 16i1 and i3 <= -1 + n }.

Applying CLooG to set *TILE_VLD_EXT,* we generated the following tiled code (without parallelism).

```
for(c0=0; c0 <= (loop-1)/16; c0 += 1)
 for(c1=0; c1 <= (n-1)/16; c1 += 1)
  for(c2=16*c0+1; c2<=min(loop,16*c0+16); c2++)
   for(c3=16*c1; c3<=min(n-1,16*c1+15); c3++)
    x[c3]=q+y[c3]*(r*z[c3+10]+t*z[c3+11]);
```

To generate parallel synchronization-free tiled code, we calculate tile representatives (the lexicographically minimal tiles of slices) comprised in set *TILE_SOUR* and a relation *R_TILE* that describes dependences among all tiles but ignores dependences within each tile as follows. Next, the transitive closure of that relation, $R\_TILE^*$, is calculated, and finally set, *SLICE*, representing slices comprising tiles are extracted. The corresponding sets and relations for the loop nest above are as follows.

*TILE_SOUR*:= [n, loop] -> { [ i1, i3, 13] : loop >= 17 and i1 >= 0 and i3 >= 16i1 and i3 <= 15 + 16i1 and i3 <= -1 + n }, where “13”is the statement identifier,

*R_TILE:*= [n, loop] -> { [i0, i1, 13] -> [o0, i1, 13] : i0 >= 0 and i1 >= 0 and 16o0 <= -1 + loop and o0 >= 1 + i0 and 16i1 <= -1 + n and 16i0 <= -2 + loop },

$R\_TILE^*$:= [n, loop] -> { [i0, i1, 13] -> [o0, i1, 13] : i0 >= 0 and i1 >= 0 and 16o0 <= -1 + loop and o0 >= 1 + i0 and 16i1 <= -1 + n and 16i0 <= -2 + loop },

*SLICE:*= [n, loop, c1] -> { [i0, i1, i2, i3, 13] : loop >= 17 and i2 <= 16 + 16i0 and i2 <= loop and i3 >= 16c1 and i3 >= 0 and i3 <= 15 + 16c1 and i3 <= -1 + n and i3 <= 15 + 16i1 and i3 >= 16i1 and i2 >= 1 and i2 >= 1 + 16i0 }.

Applying CLooG to set *SLICE* and preprocessing the code returned by CLooG, we get the following OpenMP C parallel code, where line 1 represents the OpenMP *parallel for* directives pointing out that the *for* loop in line 2 can be executed in parallel; line 2 and line 4 include the *for* loops enumerating tile identifiers whereas line 5 and line 6 present *the for* loops scanning statement instances within a tile whose identifier is defined by the indices of the loops in lines 2 and 4.

```
1.#pragma omp parallel for
2.for (c1 = 0; c1 <= floord(n - 1, 16); c1++)
3. if (loop >= 17)
4.  for (c0 = 0; c0 <= (loop-1)/16; c0 += 1)
5.   for(c2=16*c0+1;c2<=min(16*c0+16,loop);c2++)
6.    for(c3=16*c1; c3<=min(n-1,16*c1+15); c3++)
7.      x[c3]=q+y[c3]*(r*z[c3+10]+t*z[c3+11]);
```

## VI. Related work

Well-known automatic parallelization of numerical algorithms is based on the polyhedral model, which provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests. The polyhedral source to source tools: Pluto [16], POCC and PTile [17] transform C programs to expose parallelism and improve data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient loop nest tiling and fusion, but not limited to those.

TABLE I
Numerical programs

| Benchmark | Description |
|---|---|
| advect3d | Advection Kernel for Weather Modeling |
| correl | Correlation Computation |
| dct | Discrete Cosinus Transform |
| doitgen | Multi resolution Analysis Kernel |
| dsyr2k | Symmetric rank-2k operations |
| gemver | Vector Multiplication and Matrix Addition |
| k6 | General Linear Recurrence Equations |
| tce | Tensor Contraction Expressions |

The polyhedral model approach includes the following three steps: i) program analysis aimed at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation, ii) program transformation with the aim of improving program locality and/or parallelization, iii) code generation [18], [19], [20], [21], [22]. All 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*” [23] while the presented approach does not find and use any affine function. It applies the transitive closure of a program dependence graph to specific subspaces of the original loop nest iteration space. At this point of view the program transformation step is rather within the Iteration Space Slicing Framework introduced by Pugh and Rosser [4]: “*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.

To extract affine transformations, the polyhedral model assumes that first “time-partition constraints” are to be formed, then a solution to them has to be found. The “time-partition constraints” [18], [19], [24] represent the condition that if one iteration is depend upon the other, then the first 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 exist more than one linearly independent solutions to the time-partition constraints formed for a loop nest, then it is possible to derive affine transformations allowing for loop nest parallelization and program locality improvement. Otherwise, the polyhedral model fails to expose parallelism and improve program locality.

Some limitations of affine transformation are considered in paper [1]. The main drawback of the affine transformation framework is that there does not always exist two or more

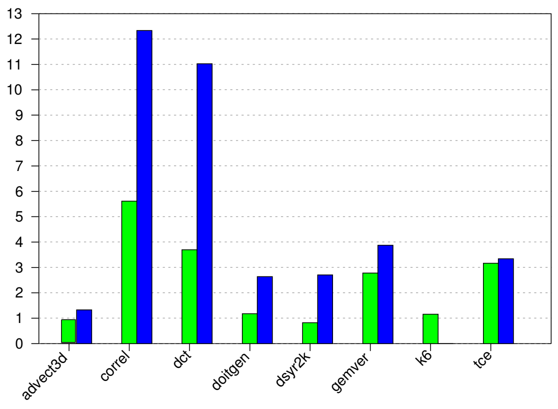

Fig. 1. Speed-up1 for tiled codes (the green bars) and speed-up2 for parallel tiled codes (the blue bars) of the studied numerical programs.

independent solutions to the time-partition constraints. For example, the studied numerical algorithm *k6* (General Linear Recurrence Equations) cannot be tiled or parallelized by PLUTO or other polyhedral tools because there exists the only solution to corresponding time-partition constraints. To extract parallelism, TRACO needs only the transitive closure of a dependence graph, so it lacks the drawbacks inherent for affine transformations.

## VII. Numerical programs

In this section, we present numerical programs chosen for carrying out experiments (Table I). We consider applications of numerical algorithms in industry and engineering: *advect3d*, *tce*, and *dct*; typical programs from compute-intensive sequences of algebra operations: *correl*, *doitgen*, *dsyr2k*, *gemver*, and *k6*. Source codes of these numerical programs are available at the TRACO website [2].

*advect3d* is the Runga-Kutta advection core from the NCOMMAS code for mesoscale weather modeling [25]. *tce* is a sequence of four nested loops, occurring in Tensor Contraction Expressions that appear in computational quantum chemistry problems [26]. Discrete cosinus transform (*dct*) is important to numerous applications in science and engineering, from lossy compression of audio and images to spectral methods for the numerical solution of partial differential equations.

The reminding benchmarks are linear algebra programs: *gemver* is a composition of BLAS operations used for householder bidiagonalization, *doitgen* is an in-place 3D-2D matrix product, and symmetric rank-2k operations *dsyr2k*. General linear recurrence equations *k6* is a kernel from the Livermore loops [15]. *correl* creates a correlation matrix and is used also in data mining [27].

The computations of *k6*, *dsyr2k*, *advect3d*, *tce*, and *gemver* are represented by perfectly nested loops, while *doitgen*, *dct*, and *correl* are represented by imperfectly nested loops.

## VIII. Experiments

We have applied TRACO to each of the benchmarks, presented in Table 1, to generate serial tiled and parallel synchronization-free tiled code. For the *k6* program, TRACO is able to generate only serial tiled code (for this benchmark, PLUTO fails to generate tiled and any parallel code).

To run programs, we have used a computer with Intel i5-4670 3.40 GHz processors (Haswell, 2013), 6MB cache and 8GB RAM. Table II presents execution time for original (*t_untiled*), serial tiled (*t_tiled*) and parallel tiled (*t_par_tiled*) applications (for 4 CPUs) for the different problem size and tile (block) size as well as the speed up of tiled code, where *speed-up1=t_untiled/t_tiled*, *speed-up2=t_untiled/t_par_tiled*, *speed-up3=t_tiled/t_par_tiled.*

Analysing the data in Table II, we may conclude that serial tiled code for each program, except from *advect3d* and *dsyr2k*, demonstrates positive speed-up (see speed-up1), for the *correl* program, speed-up is equal about 5,5. This fact can be explained by increasing tiled code locality in comparison with that of untiled code.

Serial tiled code does not increase the locality of the original *advect3d* and *dsyr2k* programs, so the corresponding serial tiled programs are not faster than the original ones.

All parallel synchronization-free tiled programs expose positive speed-up (speed-up2 and speed-up3). Two parallel programs, *dct* and *corcol,* demonstrate super linear speed-up2 ( for four CPUs, the speed-up of those programs is about 12 and 11, respectively). The reason for super-linear speed-up is that the working set of a problem is greater than the cache size when executed sequentially, but can fit nicely in each available cache when executed in parallel.

The best program speed-ups (speed-up2) are presented in a graphical way in Figure 1. All parallel tiled programs, generated by TRACO, are faster than serial tiled code – see speed-up3.

## IX. Conclusion

In this paper, we presented applying the transitive closure of dependence graphs, implemented in TRACO, for automatic producing both serial and parallel tiled code for chosen numerical applications. Loop nest computations are divided into multiple slices which are mapped to processors as threads. TRACO allows users to achieve significant speed-up of parallel numerical algorithms on shared memory machines with multi-core processors. The effectiveness of applying TRACO is comparable or better than that of other well-known optimizing compilers.

In the future, we plan to implement in TRACO techniques allowing for tiled code scalability and tile shapes different from the rectangular one, first of all parallelepiped tiles that will allow us to increase parallelism degree of parallel tiled code.

## References

[1] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski, K. Siedlecki, “Coarse-grained loop parallelization: Iteration space slicing vs affine transformations”. *Parallel Computing*, vol. 37, pp. 479–497, 2011.
[2] The TRACO Compiler, http://traco.sourceforge.net, 2015.
[3] OpenMP Specification, version 3.1, http://www.openmp.org, 2014.

TABLE II
EXECUTION TIMES OF ORIGINAL, TILED AND PARALLEL TILED CODE, SPEED-UP FOR THE STUDIED NUMERICAL PROGRAMS.

| loop | size | block | t_untiled | t_tiled | t_par_tiled | speed-up1 | speed-up2 | speed-up3 |
|---|---|---|---|---|---|---|---|---|
| advect3d | 200 | 16 | 0.144 | 0.185 | 0.112 | 0.778 | 1.286 | 1.652 |
| | | 32 | | 0.164 | 0.098 | 0.878 | 1.469 | 1.673 |
| | 300 | 16 | 0.421 | 0.496 | 0.356 | 0.849 | 1.183 | 1.393 |
| | | 32 | | 0.445 | 0.317 | 0.946 | 1.328 | 1.403 |
| correl | 1000 | 16 | 0.730 | 0.383 | 0.222 | 1.906 | 3.288 | 1.725 |
| | | 32 | | 0.364 | 0.164 | 2.005 | 4.451 | 2.220 |
| | 1200 | 16 | 3.553 | 0.728 | 0.320 | 4.880 | 11.103 | 2.275 |
| | | 32 | | 0.633 | 0.288 | 5.613 | 12.337 | 2.220 |
| dct | 512 | 16 | 0.277 | 0.205 | 0.116 | 1.351 | 2.388 | 1.776 |
| | | 32 | | 0.174 | 0.060 | 1.592 | 4.617 | 2.900 |
| | 1024 | 16 | 4.884 | 1.520 | 0.655 | 3.213 | 7.456 | 2.320 |
| | | 32 | | 1.321 | 0.443 | 3.697 | 11.025 | 2.981 |
| doitgen | 250 | 32 | 3.369 | 2.689 | 2.420 | 1.253 | 1.392 | 1.111 |
| | | 64 | | 2.868 | 1.278 | 1.175 | 2.636 | 2.244 |
| | 350 | 32 | 12.806 | 10.797 | 8.050 | 1.186 | 1.591 | 1.341 |
| | | 64 | | 10.777 | 5.604 | 1.188 | 2.285 | 1.923 |
| dysr2k | 1024 | 16 | 2.037 | 2.484 | 0.753 | 0.820 | 2.705 | 3.298 |
| | | 32 | | 1.977 | 1.347 | 1.030 | 1.512 | 1.467 |
| | 1536 | 16 | 6.702 | 6.341 | 4.529 | 1.057 | 1.480 | 1.400 |
| | | 32 | | 7.523 | 2.933 | 0.891 | 2.285 | 2.565 |
| gemver | 6000 | 200 | 0.645 | 0.254 | 0.190 | 2.539 | 3.395 | 1.336 |
| | | 400 | | 0.260 | 0.224 | 2.481 | 2.879 | 1.160 |
| | 10000 | 200 | 2.038 | 0.734 | 0.526 | 2.777 | 3.875 | 1.395 |
| | | 400 | | 1.041 | 0.689 | 1.958 | 2.958 | 1.510 |
| k6 | 1500 | 16 | 12.456 | 10.778 | - | 1.156 | - | - |
| | | 64 | | 11.166 | - | 1.116 | - | - |
| | 2000 | 16 | 35.723 | 34.189 | - | 1.045 | - | - |
| | | 64 | | 34.21 | - | 1.044 | - | - |
| tce | 200 | 32 | 0.293 | 0.226 | 0.189 | 1.296 | 1.550 | 1.195 |
| | | 64 | | 0.256 | 0.166 | 1.145 | 1.765 | 1.542 |
| | 300 | 32 | 16.779 | 8.518 | 4.944 | 1.970 | 3.394 | 1.722 |
| | | 64 | | 5.304 | 5.019 | 3.163 | 3.343 | 1.056 |

[4] W. Pugh, E. Rosser, "Iteration space slicing and its application to communication optimization". *In International Conference on Supercomputing*, pp. 22–228, 1997.

[5] W. Pugh, D. Wonnacott, "An exact method for analysis of value-based array data dependences". In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, Springer-Verlag, 1993.

[6] W. Kelly et. al., "New User Interface for Petit and Other Extensions", User Guide, 1996.

[7] W. Kelly et. al., "The omega library interface guide". Technical report, College Park, MD, USA, 1995.

[8] W. Kelly, W. Pugh, E. Rosser, T. Shpeisman, "Transitive closure of infinite graphs and its applications", *Int. J. Parallel Programming*, vol. 24 (6), pp. 579–598, 1996.

[9] S. Verdoolaege, "Integer Set Library - Manual", isl.gforge.inria.fr/manual.pdf, 2015.

[10] C. Bastoul, "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, 2004.

[11] W. Bielecki, M. Palkowski, T. Klimek, "Free scheduling for statement instances of parameterized arbitrarily nested affine loops", *Parallel Computing*, vol. 38 (9), pp. 518–532, 2012.

[12] A. Darte, Y. Robert, F. Vivien, "Scheduling and Automatic Parallelization", Birkhauser, 2000.

[13] W. Bielecki, M. Palkowski, "Perfectly nested loop tiling transformations based on the transitive closure of the program dependence graph", Soft Computing in Computer and Information Science Advances in Intelligent Systems and Computing, vol. 342, pp. 309-320, 2015.

[14] W. Bielecki, T. Klimek, M. Palkowski, A. Beletska, "An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations", COCOA 2010: Fourth International Conference on Combinatorial Optimization and Applications, Lecture Notes in Computer Science, vol. 6508, pp. 104–113, 2010.

[15] T. Peters, "Livermore Loops coded in C", Kendall Square Res. Corp., http://www.netlib.org/benchmark/livermorec, 1992.

[16] U. Bondhugula, et al., "A practical automatic polyhedral parallelizer and locality optimizer", SIGPLAN Not., vol. 43 (6), pp. 101–113, urlhttp://pluto-compiler.sourceforge.net, 2008.

[17] PoCC the Polyhedral Compiler Collection, pocc.sourceforge.net, 2014.

[18] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. One-dimensional time", *Int. J. Parallel Program.*, Kluwer Academic Publishers, vol. 21, pp. 313–348, 1992.

[19] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time", *International Journal of Parallel Programming*, vol. 21, pp. 389–420, 1992.

[20] J. Ramanujam, P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers", *Journal of Parallel and Distributed Computing*, Volume 16, Issue 2, pp. 108–120, 1992.

[21] A. W. Lim, M. S. Lam, "Communication-free parallelization via affine transformations 24th ACM Symp. on Principles of Programming Languages, Springer-Verlag, pp. 392–106, 1994.

[22] U. Bondhugula, et. al., "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model Compiler Constructure", In Proceedings of the CC'08/ETAPS'08, Springer, pp. 132–146, 2008.

[23] M. W. Benabderrahmane, L. N. Pouchet, A. Cohen, C. Bastoul, "The polyhedral model is more widely applicable than you think". Proceedings of the 19th joint European conference on Theory and Practice of Software, International Conference on Compiler Construction, Springer-Verlag, pp. 283–303, 2010.

[24] A. Lim, G. I. Cheong, M. S. Lam, "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication", In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, ACM Press, pp. 228–23, 1999.

[25] L. J. Wicker and R. B. Wilhelmson, "Simulation and analysis of tornado development and decay within a three-dimensional supercell thunderstorm". *J. Atmos. Sci.*, vol. 52, pp. 2675–2703, 1995.

[26] The Tensor Contraction Engine ,http://www.csc.lsu.edu/~gb/TCE/, 2014.

[27] The Polyhedral Benchmark suite, *http://www.cse.ohio-state.edu/ pouchet/software/polybench/*, 2014.