Computing and Informatics, Vol. 35, 2016, 1277–1306

TRACO: SOURCE-TO-SOURCE PARALLELIZING COMPILER

Marek Palkowski, Wlodzimierz Bielecki

Faculty of Computer Science
West Pomeranian University of Technology
Zolnierska 49, 70-210 Szczecin, Poland
e-mail: {mpalkowski, wbielecki}@wi.zut.edu.pl

Abstract. The paper presents a source-to-source compiler, TRACO, for automatic extraction of both coarse- and fine-grained parallelism available in C/C++ loops. Parallelization techniques implemented in TRACO are based on the transitive closure of a relation describing all the dependences in a loop. Coarse- and fine-grained parallelism is represented with synchronization-free slices (space partitions) and a legal loop statement instance schedule (time partitions), respectively. TRACO enables also applying scalar and array variable privatization as well as parallel reduction. On its output, TRACO produces compilable parallel OpenMP C/C++ and/or OpenACC C/C++ code. The effectiveness of TRACO, efficiency of parallel code produced by TRACO, and the time of parallel code production are evaluated by means of the NAS Parallel Benchmark and Polyhedral Benchmark suites. These features of TRACO are compared with closely related compilers such as ICC, Pluto, Par4All, and Cetus. Feature work is outlined.

Keywords: Source-to-source parallelizing compiler, loop parallelization, iteration space slicing, fine- and coarse-grained parallelism, free scheduling, transitive closure

Mathematics Subject Classification 2010: 68N20, 65Y05, 52Bxx, 97E60, 05-XX

1 INTRODUCTION

Parallel computer programs are more difficult to write than sequential ones. Exposing parallelism in serial programs and writing parallel programs without applying

parallelizing compilers decreases the productivity of programmers and increases the time and cost of producing parallel programs. Because for many applications, most computations are contained in program loops, automatic extraction of parallelism available in loops is extremely important for multi-core systems.

There is a large volume of published studies describing techniques and tools for automatic parallelism extraction, for example [16, 17, 22, 23, 24, 29, 30, 31, 35, 37, 41, 43]. Most researchers have studied how affine transformations can be applied to extract both coarse- and fine- grained parallelism available in loops. However, far too little attention has been paid to apply the transitive closure of a program dependence graph (hereafter we refer to it simply as transitive closure) to extract parallelism, only few studies have been carried out in this field [1, 2, 3, 7, 10, 15, 21]. Those studies demonstrate that parallelization techniques based on transitive closure open new possibilities in extracting parallelism and building optimizing compilers: they permit us to parallelize more loops than those parallelized by means of affine transformations providing similar parallel program performance. But the main limitations of those studies are an unsatisfactory evaluation of effectiveness and computational complexity of techniques based on transitive closure, an insufficient evaluation of performance of parallel programs produced by means of those techniques, the lack of a detail comparison with techniques implemented in such popular compilers as, for example, Pluto, Par4all, Cetus, ICC, and PPCG.

There are still many questions to be answered: whether well-known techniques and tools to calculate transitive closure are mature enough to be used in parallelizing compilers; what are strong and weak features of techniques based on transitive closure; what is performance of parallel programs generated by means of techniques based on transitive closure; how techniques based on affine transformations and ones based on transitive closure behave on the same benchmarks; what are well-known complementary techniques that can be integrated with those based on transitive closure to improve compiler effectiveness and parallel program performance.

The goal of this paper is to answer some questions above. For this purpose, we have reviewed published studies dealing with the transitive closure of a program dependence graph as well as those presenting loop parallelization approaches based on transitive closure. We have chosen most advanced published techniques and implemented them in a novel source-to-source compiler called TRACO.

The input of TRACO is a C program, while the output is an OpenMP C/C++ or OpenACC C/C++ program. TRACO extracts both coarse- and fine-grained parallelism. It also uses variable privatization and parallel reduction techniques to reduce the number of dependence relations; this leads to reducing parallelization time and extending the scope of the TRACO applicability. The compiler includes a preprocessor of the C program, data dependence analyzer, parallelization engine, code generator, and post-processor.

Applying TRACO, an experimental study has been carried out to evaluate the effectiveness of algorithms implemented as well as the performance of parallel programs produced by means of TRACO. A comparative analysis of results yielded

with TRACO and those demonstrated by Pluto, Par4all, Cetus, ICC, and PPCG has been fulfilled.

The contributions of this paper over related work are as follows.

- An overview of parallelizing techniques based on the transitive closure of a program dependence graph.
- A novel algorithm integrating privatization and reduction with techniques extracting synchronization-free slices.
- Presentation of the open source TRACO compiler implementing chosen loop parallelization approaches based on transitive closure.
- An evaluation of the effectiveness of TRACO for the NAS and PolyBench benchmarks.
- An evaluation of the performance of parallel programs produced by TRACO for the NAS and PolyBench benchmarks.
- A comparative analysis of the TRACO effectiveness and that demonstrated by Pluto, Par4all, Cetus, and ICC.
- Defining strong and weak features of techniques based on transitive closure and pointing out research directions to enhance the power of techniques based on transitive closure.

The rest of the paper is organized as follows. Section 2 introduces background. Section 3 presents iteration space slicing algorithms. Section 4 considers combining variable privatization and parallel reduction with iteration space slicing. Section 5 discusses (free) scheduling approaches. Section 6 introduces details of the TRACO implementation. Section 7 presents results of an experimental study. Section 8 describes related work. Section 9 draws conclusions and briefly outlines future research to enhance the power of TRACO.

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 indices bounds), and the loop steps are known constants.

A dependence analysis is required for loop parallelization. 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 *destination* of a dependence, respectively, provided that I is lexicographically smaller than J ($I \prec J$, i.e., I is executed before J).

Algorithms implemented in TRACO require an exact representation of loop-carried dependences and consequently an exact dependence analysis which detects

a dependence if and only if it actually exists. To describe and implement parallelization algorithms, we chose the dependence analysis proposed by Pugh and Wonnacott [8], where dependences are represented with dependence relations.

A dependence relation is a tuple relation of the form [*input list*] $\rightarrow$ [*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 list* and *output list* and it is a Presburger formula built of constraints represented with algebraic expressions and using logical and existential operators [8].

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. $e \rightarrow e' \in R$, $e \in S$). In detail, the description of these operations is presented in [8, 9].

The positive transitive closure for a given relation R, R^+, is defined as follows [9]:

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

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 [10]: $R^* = R^+ \cup I$, 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.

To facilitate the exposition and implementation of TRACO algorithms, we have to preprocess dependence relations making their input and output tuples to be of the same dimension and to contain the identifiers of statements responsible for the source and destination of each dependence. The preprocessing algorithm is presented in paper [1].

Given a relation R, found as the union of all (preprocessed) dependence relations extracted for a loop, the iteration space, S_{DEP}, including dependent statement instances is formed as domain(R) $\cup$ range(R). A set, S_{IND}, comprising independent statement instances is calculated as the difference between the set of all statement instances, S_{SI}, and the set of all dependent statement instances, S_{DEP}, i.e. $S_{IND} = S_{SI} - S_{DEP}$. To scan elements of sets S_{DEP} and S_{IND} in the lexicographic order, we can apply any well-known code generation technique [9, 16].

3 COARSE-GRAINED PARALLELISM EXTRACTION USING ITERATION SPACE SLICING

Algorithms presented in papers [1, 11, 12, 13, 14] are based on transitive closure and allow us to generate parallel code representing synchronization-free slices or slices requiring occasional synchronization. Below, we present basic definitions and a way to extract synchronization-free parallelism implemented in TRACO. A (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 inverse path.

If there exist two or more slices in a dependence graph, the above definition guarantees that all these slices are independent when executed as concurrent threads (there is no dependence among them).

Definition 2. An ultimate dependence source is a source that is not the destination of another dependence. Ultimate dependence sources and destinations represented by relation R can be found by means of the following calculations: $\mathrm{domain}(R) - \mathrm{range}(R)$. The set of ultimate dependence sources of a slice forms the set of its sources.

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

An approach to extract synchronization-free slices implemented in TRACO 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 the code scanning these slices is generated.

Given a dependence relation R describing all the dependences in a loop, a set of statement instances, S_{UDS}, is calculated. It describes all ultimate dependence sources of slices as

$$S_{UDS} = \mathrm{domain}(R) - \mathrm{range}(R). \tag{2}$$

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:

$$R_{USC} = \{[e] \to [e'] : e, e' \in S_{UDS}, e \prec e', (R^*(e) \cap R^*(e'))\}. \tag{3}$$

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 can be represented by the input tuple of R_{USC} only. The intersection $(R * (e) \cap R * (e'))$ 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 $S_{repr} = S_{UDS} - \mathrm{range}(R_{USC})$. 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:

$$S_{slice} = R^*((R_{USC})^*(e)). \tag{4}$$

Any tool to generate code for scanning polyhedra can be applied to produce parallel pseudo-code, for example, the CLOOG library [16] or the *codegen* function of the Omega project [17].

The presented technique is illustrated by means of the following imperfectly nested loop.

```
for (i=1; i<=n; i++)
s1: b[i][i]=a[i-3][i];
    for (j=1; j<=n; j++)
s2:  a[i][j]=a[i][j-1]+b[i][i];
```

There are the following three dependence relations returned by the Omega dependence analyzer, Petit [18].

```
R12 = {[i]->[i,j]: 1<=i<n & 1<=j<=n};
R21 = {[i,i+3]->[i+3]: 1<=i<=n-3};
R22 = {[i,j]->[i,j+1]: 1<=i<=n & 1<=j<n}.
```

Dependences are illustrated in Figure 1. The dependence relations after preprocessing are as follows:

```
R12 = {[i,-1,1]->[i,j,2]: 1<=i<n & 1<=j<=n};
R21 = {[i,i+3,2]->[i+3,-1,1]: 1<=i<=n-3};
R22 = {[i,j,2]->[i,j+1,2]: 1<=i<=n & 1<=j<n}.
```

For these relations, the Omega calculator [9] produces $R_{USC} = \varnothing$ and $S_{repr} = \{[i,-1,1] : 1 \leq i \leq \min(n,3)\}$.

Set S_{slice} is of the form: $S_{slice} = \{[i,-1,1] : \exists(\alpha : i = t1 + 3\alpha \ \& \ 1 \leq t1 \leq i-3 \ \& \ i \leq n)\} \cup \{[t1,j,2] : 1 \leq t1 \leq n \ \& \ 1 \leq j \leq n\} \cup \{[i,j,2] : \exists(\alpha : i = t1 + 3\alpha \ \& \ 1 \leq t1 \leq i-3 \ \& \ 1 \leq j \leq n \ \& \ i \leq n)\} \cup \{[i,-1,1] : i = t1\}$.

Finally, applying CLOOG and inserting the *parallel* and *for* OpenMP directives [4], we obtain the following parallel code

```
#pragma omp parallel for
for (int t1 = 1; t1 <= min(n, 3); t1 += 1){
  b[t1][t1]=a[t1-3][t1];  //s1(t1, -1, 1);
    for (int c1 = 1; c1 <= n; c1 += 1)
       a[t1][c1]=a[t1][c1-1]+b[t1][t1];  //s1(t1, c1, 2);
    for (int c0 = t1 + 3; c0 <= n; c0 += 3) {
      b[c0][c0]=a[c0-3][c0];    // (c0, -1, 1);
      for (int c1 = 1; c1 <= n; c1 += 1)
        a[c0][c1]=a[c0][c1-1]+b[c0][c0];  // s1(c0, c1, 2);
}}
```

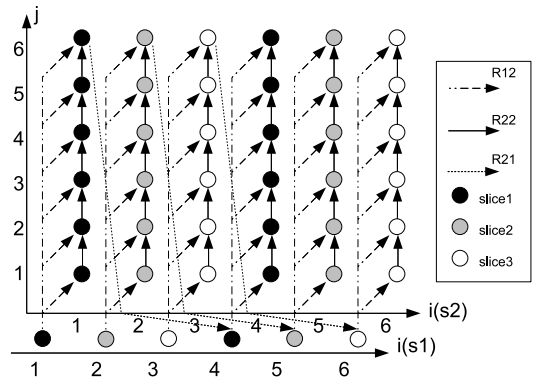


Figure 1. Dependences for the loop example, $n = 6$

4 USING VARIABLE PRIVATIZATION AND PARALLEL REDUCTION

Extracting parallelism by means of TRACO relies on the transitive closure of an affine dependence relation describing all the dependences in a loop. Algorithms aimed at calculating transitive closure are presented in papers [10, 19, 20, 21] and they are out of the scope of this paper. The time and memory complexities of calculating transitive closure depend considerably on the number of dependence relations representing all the dependences in the loop. In many cases, when the number of dependence relations is more than several hundreds, known algorithms [10, 19, 20, 21] fail to produce transitive closure due to limited resources of computers or because time of calculating transitive closure is not acceptable in practice (from several hours to several days). That is why reducing the number of dependence relations is very important prior to calculate transitive closure.

In this section, we present a technique implemented in TRACO that automatically defines loop variables that can be safely privatized and/or can be used for parallel reduction. Applying this technique permits us to reduce the number of dependence relations.

Privatization is a technique that allows each concurrent thread to allocate a variable in its private storage such that each thread accesses a distinct instance of a variable.

Definition 4. A scalar variable x defined within a loop is said to be privatizable with respect to that loop if and only if every path from the beginning of the loop body to a use of x within that body must pass through a definition of x before reaching that use [22].

Definition 5. Given n inputs $x_1, x_2, \ldots, x_n$ and an associative operation $\otimes$, a parallel *reduction* algorithm computes the output $x_1 \otimes x_2 \otimes \ldots \otimes x_n$ [23].

Parallel reduction is a frequent operation in parallel algorithms. It is applied for parallel execution of: dot products, matrix multiplications, counting, etc.

The idea of the algorithm presented in this section is the following. The first step of the algorithm is to search for scalar or one dimensional array variables for privatization. A variable can be privatized if the lexicographically first statement in the loop body referring to this variable does not read its value, i.e., the first access to this variable is a write operation [22].

Next, we seek for variables that are involved in reduction dependences [18] only (they cannot be involved in other types of dependences). Then we check whether there exist dependence relations referring to variables which cannot be privatized or used for parallel reduction. If no, this means that privatization and parallel reduction eliminate all the dependences in the loop, thus its parallelization is trivial. Otherwise, we form a set including

1. dependence relations not being eliminated by means of variable privatization and reduction and
2. dependence relations describing dependences not carried by loops and referring to variables to be privatized. Finally, we generate output code using the set mentioned above and a set including variables to be used for parallel reduction.

Below, we present the algorithm that implements the idea above in a formal way.

Algorithm 1 Extracting synchronization-free-slices applying variable privatization and/or reduction

Input: Set of relations $S = \{R_i, 1 \leq i \leq n\}$ whose union describes all the dependences in a loop, where n is the number of relations.
Output: Code representing synchronization-free slices with variables to be privatized and/or used for parallel reduction.

1. For each scalar/one-dimensional array variable X, originating dependences, find the lexicographically first statement, referring X. If the statement only writes a value to X, put X into set *PrivSet*.
2. Put each relation R_i describing dependences involving variable X, $X \in$ *PrivSet*, into set *PC*.
3. Put each relation R_i involving a shared variable that is not involved in dependences other than reduction, into set *RED* and put this variable together with its associative operations into set D.
4. $R_{slice} = S - PC - RED$.

5. **If** $R_{slice} = \varnothing$, **then** privatize all variables in set *PrivSet* and make the outermost loop to be parallel. **Goto** step 7.
6. **For each** variable X in set *PrivSet* **do**

   (a) $z =$ the minimal number of inner loops surrounding all statements referring X

   (b) $SET\_X = \varnothing$.

   (c) **For each** relation R_q, $1 \le q \le m$, from set PC **do**

   (d) Form new relation P_q in the following way:
   $P_q = \{[e_1, e_2, \ldots, e_k] \to [e'_1, e'_2, \ldots, e'_k] : \text{constraints}(R_q) \bigwedge_{j=1}^{z} (e_j = e'_j)\}$
   where $e_1, e_2, \ldots, e_k$ and $e'_1, e'_2, \ldots, e'_k$ are the variables of the input and output tuples of R_q, respectively; k is the number of loop indices; constraints(R_q) are the constraints of relation R_q.
   / the constraints $\bigwedge_{j=1}^{z} (e_j = e'_j)$ mean that relation Pq does not describe dependences carried by z inner loop nests. */*

   (e) $SET\_X = SET\_X \cup P_q$
   end for each

   (f) $R_{slice} = R_{slice} \cup SET\_X$;

   end for each

7. Apply any technique presented in [1] to set R_{slice} to extract synchronization-free slices and generate code as below:

```
parallel block shared(D_s_1,D_s_2,...) private(D_p_1,D_p_2,...){
  D_p_1 = 0; D_p_2 = 0; ...

  parallel loop scanning slices private(X){
    ... }
 crtical{
   D_s_1 = D_s_1 op_1 D_p_1;
   D_s_2 = D_s_2 op_2 D_p_2;
   ...
 }
}
```

where `parallel block` is a part of code that may be executed by multiple threads; each parallel reduction for a variable in set D is represented by shared variable D_s_i, private variable D_p_i and operation, op_i; `private(X)` means that all threads have their own copies of all variables contained in set X; `critical` specifies a region of code that must be executed by only one thread at a time.

Let us illustrate the presented algorithm by means of the following loop:

```
1: for(i=1; i<=n; i++){
2:  c = 0;
3:   for(j=1; j<=n; j++){
4:    a[i][j] = a[i][j-1] + c;
5:    b += a[i][j];
6:  }
7: }
```

The set of dependence relations for this loop is the following:

$$R1 = \{[i,-1,2] \rightarrow [i,j',4] : 1 \leq i \leq n \;\&\&\; 1 \leq j' \leq n\};$$
$$R2 = \{[i,-1,2] \rightarrow [i',j',4] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j' \leq n\};$$
$$R3 = \{[i,-1,2] \rightarrow [i',-1,2] : 1 \leq i < i' \leq n\};$$
$$R4 = \{[i,j,4] \rightarrow [i',-1,2] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j \leq n\};$$
$$R5 = \{[i,j,4] \rightarrow [i,j+1,4] : 1 \leq i \leq n \;\&\&\; 1 \leq j < n\};$$
$$R6 = \{[i,j,4] \rightarrow [i,j,5] : 1 \leq i \leq n \;\&\&\; 1 \leq j \leq n\};$$
$$R7 = \{[i,j,5] \rightarrow [i,j',5] : 1 \leq j < j' \leq n \;\&\&\; 1 \leq i \leq n\};$$
$$R8 = \{[i,j,5] \rightarrow [i',j',5] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j \leq n \&\& 1 \leq j' \leq n\},$$

where relations $R1$, $R2$, $R3$, $R4$ describe dependences involving variable c, relations $R5$, $R6$ involve variable a, $R7$, $R8$ (represent reduction dependences) involve variable b. Applying the algorithm, we get:

1. PrivSet $= \{c\}$,
2. PC $= \{R1, R2, R3, R4\}$,
3. RED $= \{R7, R8\}, D = \{b, +\}$,
4. $R_{slice} = \{R5, R6\}$,
5. $R_{slice} \neq \varnothing$.
6. **For** variable c **do**

   (a) $z = 1$,
   (b) SET_X $= \varnothing$,
   (c) PC $= \{R1, R2, R3, R4\}$,
   (d) $P1 = R1$; $P2 = P3 = P4 = \varnothing$,
   (e) SET_X $= \{P1\}$,
   (f) $R_{slice} = \{R1, R5, R6\}$, $R = R1 \cup R5 \cup R6$.

   end for
7. $S_{repr}(R) = \{[i,-1,2] : 1 \leq i \leq n \;\&\&\; 2 \leq n\}$.

   Applying algorithm ***Gen_affine*** [1], we extract n slices represented with the following loop in the OpenMP standard [4].

```
#pragma omp parallel for private(c) reduction(+:b) shared(a)
for(c0 = 1; c0 <= n; c0 += 1)
  if (c0 >= 1 && n >= c0) {
    c = 0 ;                              // s1(c0,-1,2);
    for(c1 = 1; c1 <= n; c1 += 1){
        a[c0][c1] = a[c0][c1-1] + c;  // s1(c0,c1,4);
        b += a[c0][c1];               // s1(c0,c1,5);
    }
  }
```

5 FINDING (FREE) SCHEDULING FOR PARAMETRIZED LOOPS

The algorithm, presented in our paper [3], allows us to generate fine-grained parallel code based on the free schedule representing time partitions; 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 6 ([24, 25])**.** 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 $\sigma : LD \to \mathbb{Z}$ such that

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

where $p, p_1, p_2, \ldots, p_n$ are loop statement instances, LD is the loop domain, $p_1 \to p, p_2 \to p, \ldots, p_n \to p$ mean that the pairs p_1 and p, p_2 and p, $\ldots$, p_n and p are dependent, p represents the destination while $p_1, p_2, \ldots, p_n$ 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 [24].

The idea of the algorithm to extract time partitions is as follows [3]. Given preprocessed relations R_1, R_2, $\ldots$, R_m, we firstly calculate $R = \bigcup_{i=1}^{m} R_i$. Next we create a relation R' by inserting variables k and $k+1$ into the first position of the input and output tuples of relation R; variable k is to present the time of a partition (a set of statement instances to be executed at time k). Next, we calculate the transitive closure of relation R', $R'*$, and form the following relation:

$$FS = \{[X] \to [k, Y] : X \in UDS(R) \wedge (k, Y) \in \text{Range}((R')^* \setminus \{[0, X]\}) \\ \wedge \neg(\exists k' > k \text{ s.t. } (k', Y) \in \text{Range}(R')^+ \setminus \{[0, X]\})\},$$

where $UDS(R)$ is a set of ultimate dependence sources calculated as $\text{Domain}(R) - \text{Range}(R)$; $(R')^* \setminus \{[0, X]\}$ means that the domain of relation $R'*$ is restricted to the set including ultimate dependences sources only (elements of this set belong to the

first time partition); the constraint $\neg(\exists k' > k \text{ s.t. } (k', Y) \in \text{Range}(R')^+ \setminus \{[0, X]\})$ guarantees that partition k includes only those statement instances whose operands are available, i.e., each statement instance will belong to one time partition only.

It is worth to note that the first element of the tuple representing the set Range(FS) points out the time of a partition while the last element of that exposes the identifier of the statement whose instance (iteration) is defined by the tuple elements 2 to $n-1$, where n is the number of the tuple elements of a preprocessed relation. Taking the above consideration into account and provided that the constraints of relation FS are affine, the set Range(FS) is used to generate parallel code applying any well-known technique to scan its elements in the lexicographic order, for example, the techniques presented in papers [9, 16].

The outermost sequential loop of such code scans values of variable k (representing the time of partitions) while inner parallel loops scan independent instances of partition k.

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

Let us illustrate the presented technique by means of the following imperfectly nested loop.

```
for(i=1; i<=n; i++){
s1: a[i][0] = 1;
    for(j=1; j<=n; j++){
s2:   a[i][j] = a[i-1][j] + a[i][j-1];
  }
}
```

There are the three dependence relations returned by Petit

$$R1 = \{[i, -1, 1] \to [i, 1, 2] : 1 \leq i \leq n\};$$
$$R2 = \{[i, j, 2] \to [i + 1, j, 2] : 1 \leq i < n \;\&\&\; 1 \leq j \leq n\};$$
$$R3 = \{[i, j, 2] \to [i, j + 1, 2] : 1 \leq i \leq n \;\&\&\; 1 \leq j < n\}.$$

Figure 2 presents the free schedule for the loop when $n = 5$, where the number into a circle indicates the time when a correspondent statement instance has to be executed.

Applying the presented technique, we get the following results being produced by means of the Omega calculator.

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

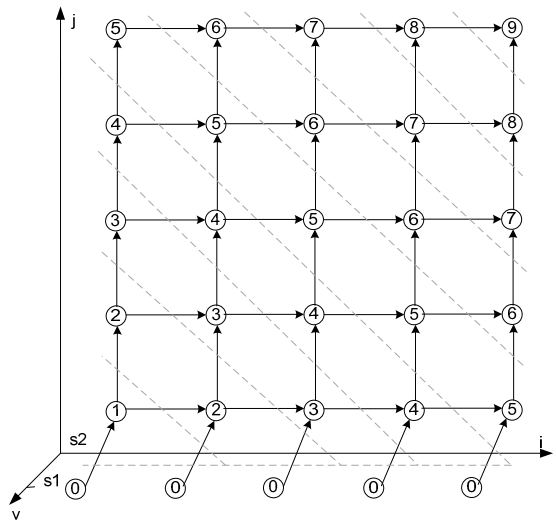


Figure 2. The free schedule for the loop when $n = 5$. The solid lines represent dependences, the dotted lines represent synchronization barriers between time partitions

3. $FS = \{[1,-1,1] \to [k, i', k-i'+1, 2] : 1 \le i' \le k, n \;\&\&\; k < n+i'\} \cup \{[i,-1,1] \to [0, i, -1, 1] : 1 \le i \le n\}$.
4. $\text{Range}(FS) = \{[k, i, k-i+1, 2] : 1 \le i \le k, n \;\&\&\; k < n+i\} \cup \{[0, i, -1, 1] : 1 \le i \le n\}$.

   The loop scanning elements of the set $\text{Range}(FS)$ for $k \ge 0$ and being produced by TRACO is as follows.

```
if (n >= 1) {
  for(k=0;k<=2*n-1;k++) {  //serial loop
    if (k == 0) {
      #pragma omp parallel for // parallel loop
      for(i=1;i<=n;i++) {
        a[i][0]=1;  //s1(0,i,-1,1);
    }}
    #pragma omp parallel for  //parallel loop
    for(i=max(1,k-n+1);i<=min(k,n);i++) {
      a[i][k-i+1]=a[i-1][k-i+1]+a[i][k-i+1-1];
                    // s1(k,i,k-i+1,2);

}}}
```

5. IND = $\varnothing$. There are no independent statement instances in the loop.

The calculation of the transitive closure for relation R' can be impossible for some cases of loops [10]. TRACO can be configured to enable another approach dedicated for fine-grained parallelism extraction based on the power k of relation R.

The idea of the algorithm is the following [2]. Given preprocessed relations R_1, R_2, ..., R_m, we first calculate $R = \bigcup_{i=1}^{m} R_i$ and then R^k, where $R^k = \underbrace{R \circ R \circ \ldots R}_{k}$, "$\circ$" is the composition operation. Techniques of calculating the power k of relation R are presented in the following publications [10, 26, 20, 19, 27] 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 [19, 26].

Each vertex, represented with the set $S(k) = Rk(UDS) - R^+ \circ R^k(UDS)$, is connected in the dependence graph, defined by relation R, with some vertex(s) represented by set UDS. 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. Examples illustrating this approach can be found in the paper [2]. TRACO implements both techniques mentioned above.

6 IMPLEMENTATION

Figure 3 shows the details of the TRACO implementation. Currently, it supports C/C++ programs on its input. A preprocessor, written in the Python language, recognizes loops in a source program and converts them to the format acceptable by the Omega dependence analyzer, Petit [18], which returns a set of dependence relations representing all dependences in a loop. Then TRACO recognizes variables to be privatized and/or used for parallel reduction. If privatization and/or reduction remove all dependence relations, parallelization is trivial, all loops can be made parallel. For such a case, TRACO makes the outermost loop to be parallel while the remaining loops it makes serial to produce coarse-grained code.

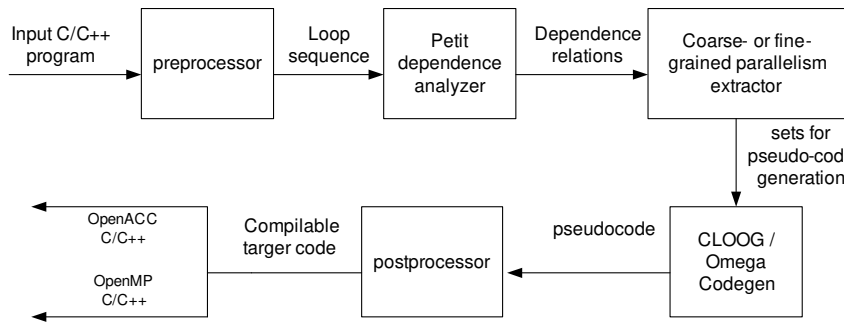


Figure 3. TRACO organization

When a set of dependence relations after applying privatization and/or parallel reduction is not empty, the number of synchronization-free slices is calculated. If this number is not equal to one, then data necessary for generating pseudo-code representing slices are calculated and forwarded to a pseudo-code generator. Otherwise, data necessary for extracting (free) scheduling are prepared and directed to the pseudo-code generator. A post-processor generates parallel code in OpenMP/OpenACC. Below, we present some details concerned code generation. TRACO permits also for a manual choice of a parallelization technique.

a)
```
S := {[i,j] : 1<=i<=n &&
1 <= j <= n && 2 <= n };
```

b)
```
if (n >= 2)
  for (int c0 = 1; c0 <= n; c0 += 1)
    for (int c1 = 1; c1 <= n; c1 += 1)
      s1(c0, c1);
```

c)
```
if (n >= 2)
  #pragma omp parallel for
  for (int c0 = 1; c0 <= n; c0 += 1)
    for (int c1 = 1; c1 <= n; c1 += 1)
      a[c0][c1] = a[c0][c1-1];
```

d)
```
S := {[k,i,j] : 2+k = i+j &&  1, 3-i <= j <= n
 && 1 <= i <= n && 2 <= n }
```

e)
```
for (int c0 = 1; c0 < 2 * n - 1; c0 += 1)
 for(int c1=max(-n+c0+ 2,1); c1<=min(c0+1,n); c1+=1)
    s1(c0, c1, c0 - c1 + 2);
```

f)
```
for (int c0 = 1; c0 < 2 * n - 1; c0 += 1)
 #pragma omp parallel for private(c1)
 for(int c1=max(-n+c0+ 2,1); c1<=min(c0+1,n); c1+=1)
   a[c1][c0-c1+2]=a[c1][c0-c1+2+1]+a[c1+1][c0-c1+2];
```

Figure 4. Code generation details: a), b), c) synchronization-free slices; d), e), f) free-scheduling

6.1 Parallel Pseudo-Code Generation

Input for the pseudo-code generator is a set representing slices or scheduling. For the first case, the first element of the set states for slice representatives, all the following elements, but the last one, describe statement instances of a parametrized slice, and the last one represents a statement identifier, which may be skipped when all dependent statement instances are originated by the same statement. An example set is illustrated in Figure 4 a). In this set, the first element is responsible for slice representatives while the second one together with the first one presents statement instances of a slice. There is no element describing a statement identifier.

Taking such a set as input, CLOOG generates pseudo-code (Figure 4 b)), where by default the outermost loop is to scan slice representatives (this loop is parallel), while the inner loop (serial) enumerates statement instances of the slice with a representative presented by the outermost loop.

Any other code generator, permitting to scan set elements in the lexicographic order, can be applied in TRACO, for example, the codegen function of Omega or Omega+ [28].

When a set S represents scheduling, then the first element of the set is responsible for the time partition representation, all the following elements, but the last

one, describe statement instances of a parametrized time partition, and the last one represents a statement identifier, which may be absent when all statement instances are originated from the same statement. An example set is given in Figure 4 d), where the first element represents time partitions, while the second and third ones are to enumerate statement instances of a particular time partition defined by the first element.

Taking such a set as input, CLOOG generates pseudo-code (Figure 4 e)), where by default the outermost loop scans times (this loop is serial), while the remaining loops (parallel) enumerate statement instances of the time partition for a time represented by the outermost loop.

6.2 Parallel Compilable Code Generation

Compilable OpenMP/OpenACC code is produced by means of the post-processor written in Python. It inserts source loop statements with proper index expressions into pseudo-code. Original index variables are replaced with variables represented with the tuple elements of a set representing polyhedra taking into account the role of particular tuple elements (see Section 6.1). For example, provided that the set S in Figure 4 a) is associated with the statement $a[i][j] = a[i][j-1]$, in the pseudo statement $s1(c0, c1)$ in Figure 4 b), variables $c0$, $c1$ correspond to variables i, j which are substituted for $c0$, $c1$ in the source statement (Figure 4 c)).

Given the set S in Figure 4 d) is associated with the source statement $a[i][j] = a[i][j+1] + a[i+1][j]$, the code generator recognizes that in the pseudo code in Figure 4 e) $c0$ states for time of partitions, $c1$ corresponds to variable i, while $c0 - c1+2$ corresponds to variable j. So it generates the following statement in the output loop (see Figure 4 f)) $a[c1][c0-c1+2] = a[c0][c0-c1+2+1]+a[c0+1][c0-c1+2]$.

Depending on whether pseudo-code represents slices or scheduling, the post-processor inserts proper OpenMP pragmas such as *Parallel, For, Critical* and proper clauses to define private and/or reduction variable or OpenACC pragmas such as *Kernel, Data, Loop.*

The source repository of the TRACO compiler is available on the website `http://traco.sourceforge.net`.

7 RELATED WORK

TRACO implements algorithms based on the calculation of the transitive closure of a relation describing all the dependences in the loop. These algorithms can be divided into the two classes: the first one is to extract coarse-grained parallelism represented with synchronization-free slices while the second one is to extract fine-grained parallelism represented with time partitions.

Papers [7, 10] demonstrate how transitive closure can be used for building optimizing compilers, namely how to optimize inter-processor communication, remove dependences, finding induction variables; however, it does not clarify how to extract

synchronization-free slices and the free schedule. The paper [1] presents different algorithms to extract synchronization-free slices but without applying variable privatization and parallel reduction as well as without any details of the TRACO implementation.

Algorithms presented in the papers [1, 2, 3] produce pseudo-code only, while TRACO generates compilable code.

Variable privatization and parallel reduction techniques are the well-known popular techniques used for dependence elimination [22]. Algorithm 1, presented in this paper, is based on these techniques, it includes the following novel elements:

1. integrates both the variable privatization and parallel reduction techniques to eliminate loop carried dependences;
2. demonstrates how to eliminate dependences carried by private and reduction variables from dependence relations (step 6 (d) in Algorithm 1);
3. clarifies how to manage private and reduction variables in techniques aimed at extracting synchronization-free slices (step 7 of Algorithm 1).

The affine transformation framework (ATF), considered in the papers [29, 30, 31] unifies a large number of previously proposed loop transformations. The paper [1] demonstrates that ATF does not exploit all synchronization-free parallelism available in affine loops, while the techniques implemented in TRACO really do. The paper [2] shows that for particular loops algorithms based on calculating a transitive closure (those implemented in TRACO) do expose the free schedule while ATF fails to extract such parallelism.

Different source-to-source compilers have been developed to extract coarse-grained parallelism available in loops. To choose compilers to be compared with TRACO, we have applied the following criteria: it has to

1. be a source-to-source compiler;
2. support the C language;
3. produce compilable code in OpenMP/ACC C/C++.

The following compilers were chosen to be compared with TRACO: ICC (XE 2013), Pluto (0.10), Cetus (1.3.1), and Par4All (1.4.5). We have omitted other projects, for example, Bones [32] that is not a fully automated compiler, it inserts special directives (skeletons) into input code; Kremlin [33] is a profiler and provides only a list of regions that have to be parallelized.

Polyhedral Parallel Code Generator (PPCG) is a source-to-source compiler based on the polyhedral model and affine transformations to extract loop parallelism [44]. It produces host and GPU code for any static control loop nest with affine loop bounds and affine subscripts. PPCG implements tiling algorithms that are not implemented in the current version of TRACO, this makes impossible a correct comparison of codes produced with TRACO and PPCG. We intend to compare them in another paper after implementing tiling in TRACO.

Below, we shortly describe compilers being classified for comparison with TRACO.

ICC, [34]. The Intel Compilers enable threading through automatic parallelization and OpenMP support. With automatic parallelization, the compilers detect loops that can be safely and efficiently executed in parallel and generate multithreaded code. They search for loops that are candidates for parallel execution and perform data-flow analysis to verify the correctness of parallel execution. The Intel Compilers support variable privatization, loop distribution, and permutation. Unfortunately, they can only effectively analyze loops with a relatively simple structure [34].

Pluto, [35]. An automatic parallelization tool based on the polyhedral model [37]. Pluto transforms C programs from source to source for coarse- or fine-grained parallelism and data locality simultaneously. The core transformation framework mainly works to find affine transformations for efficient tiling and fusion, but not limited to those [37]. Though the tool is fully automatic (C to OpenMP C), a number of options are provided (both command-line and through meta files) to tune aspects like tile sizes, unroll factors, and outer loop fusion structures. A version with support for generating CUDA code is also available. However, Pluto does not support variable privatization and reduction recognition.

Par4All, [38]. A project aims at easing code generation for parallel architectures from sequential source code written in C or Fortran with almost no manual code modification required. Par4All is currently composed of different components: the PIPS tool [39], the Polylib library [40], and internal parsers. Program transformations available by the compiler include loop distribution, scalar and array privatization, atomizers (reduction of statements to a three-address form), loop unrolling (partial and full), stripmining, loop interchanging and others.

Cetus, [41]. It provides an infrastructure for research on multi-core compiler optimizations that emphasizes automatic parallelization by means of the Java API. The compiler targets C programs and supports source-to-source transformations. The tool performs a loop dependence analysis and generates parallel loop annotations. The tool is limited only to basic transformations: induction variable substitution, reduction recognition, array privatization, pointers, alias and range analysis. However, the tool is able to parallelize a wide range of program loops.

PIT, [45]. It combines polyhedral model with iterative compilation for loop transformations. PIT finds the optimal sequence of non-parametric loop transformations in the first phase. Next, it defines the perfomance model using hardware monitoring tools and search transformation parameters by means of the Nelder-Mead simplex algorithm in the three-phase loop optimization engine. The paper [46] describes the iterative compilation algorithm for parameter search in detail. Hardware performance counters based model of the PIT compiler reduces also the set of useless loop transformations.

8 EXPERIMENTAL STUDY

The goals of experiments were to evaluate such features of TRACO as: effectiveness, the kind of parallelism extracted (coarse- or fine-grained), efficiency of parallel loops produced, and the time of parallel code generation. Another goal was to compare these features of TRACO with those demonstrated by the compilers classified for comparison (see Section 7).

To evaluate the effectiveness of TRACO, we have experimented with NAS Parallel Benchmarks 3.3 (NPB) [5] and Polyhedral Benchmarks 3.2 (PolyBench) [6]. NPB are developed at the NASA Ames Research Centre to study performance of parallel supercomputers. Benchmarks are derived from computational fluid dynamics (CFD) applications, they consist of five kernels and three pseudo-applications [5]. PolyBench are developed at the Ohio State University and its benchmarks are derived from linear-algebra kernels and solvers, datamining, medley, and stencils applications.

From 431 loops of the NAS benchmark suite, Petit is able to analyse 257 loops, and dependences are available in 134 loops (the rest 123 loops do not expose any dependence). For these 134 loops, TRACO is able to extract: synchronization-free parallelism for 109 (81 %) loops and fine-grained parallelism for 22 (16 %) loops when it fails to extract synchronization-free parallelism.

Table 1 presents techniques used by TRACO which acts as follows. First of all, it tries to extract coarse-grained parallelism by applying privatization only, for 39 NAS loops, variable privatization eliminates all dependences, hence loop parallelization is trivial. Next to the remanding benchmarks, the technique presented in Section 4 is applied, this results in parallelization of 70 NAS loops. Finally, to the rest benchmarks, techniques extracting (free) scheduling are applied that yields 22 NAS loops representing fine-grained parallelism. TRACO fails to extract parallelism for the three loops for which each iteration (except the first one) depends on the previous one: *CG_cg_6*, *CG_cg_8*, and *MG_mg_4*.

Technique	NAS	PolyBench
Privatization only	39	0
Slicing with privatization and reduction	70	30
Free scheduling	22	15
All techniques	131	45
All loops	134	48

Table 1. Techniques of loop parallelization

For the PolyBench suite, there exist 48 loops exposing dependences. TRACO is able to parallelize 45 (94 %) loops. One of the LU decomposition loops (*ludcmp_3*) is serial (each iteration depends on the previous one). For the Seidel-2D and Floyd-Warshall loops, TRACO fails to extract any parallelism because all known to us tools permitting for calculating the transitive closure of a dependence representing all the dependences in a loop [10, 20, 21] are not able to produce transitive closure

for these loops. There exists a strong need to improve the existing algorithms for calculating the transitive closure to enhance their effectiveness. 30 PolyBench loops were parallelized by applying algorithms of synchronization-free slices extraction [1]. For 15 PolyBench loops, only the fine-grained parallelism was found (the outermost loop is serial).

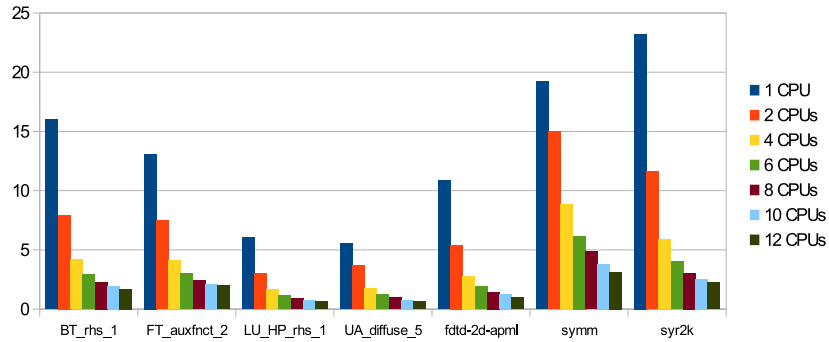


Figure 5. Times of program loops execution for various numbers of threads

To check the performance of coarse-grained parallel code, produced with TRACO, the following criteria were taken into account for choosing NAS and PolyBench loops:

1. a loop must be computatively heavy (there are many NAS benchmarks with constant upper bounds of loop indices, hence their parallelization is not justified),
2. structures of chosen loops must be different (there are many loops of a similar structure).

Applying these criteria, we have selected the following four NAS loops: *BT_rhs_1* (Block Tridiagonal Benchmark), *FT_auxfnct.f2p_2* (Fast Fourier Transform Benchmark), *LU_HP_rhs_1* (Lower-Upper symmetric Gauss-Seidel Benchmark), *UA_diffuse_5* (Unstructured Adaptive Benchmark) and the three PolyBench loops: *fdtd-2d-apml* (FDTD using Anisotropic Perfectly Matched Layer), *symm* (Symmetric Matrix-multiply), and *syr2k* (Symmetric Rank-2k Operations).

For each loop qualified for experiments, we have measured execution time, then speed-up and efficiency have been calculated. Speed-up is a ratio of sequential time and parallel time, $S = T(1)/T(P)$, where P is the number of processors. Efficiency, $E = S/P$, tells us about usage of available processors while parallel code is executed. Table 2 shows time (in seconds), speed-up, and efficiency for 2, 6, and 12 processors.

Loop	Parameters	1 CPU	2 CPUs			6 CPUs			12 CPUs		
		Time	Time	S	E	Time	S	E	Time	S	E
BT_rhs_1	N1, N2, N3 = 200	3.937	2.142	1.84	0.92	0.874	4.50	0.75	0.489	8.05	0.67
	N1, N2, N3 = 300	15.98	7.938	2.01	1.01	2.933	5.45	0.91	1.689	9.46	0.79
FT_auxfnct_2	N1, N2, N3 = 200	0.309	0.176	1.76	0.88	0.159	1.94	0.32	0.133	2.32	0.19
	N1, N2, N3 = 500	13.085	7.454	1.76	0.88	2.999	4.36	0.73	2.004	6.53	0.54
LU_HP_rhs_1	N1, N2, N3 = 100	0.446	0.241	1.85	0.93	0.12	3.72	0.62	0.07	6.37	0.53
	N1, N2, N3 = 200	6.043	3.02	2.00	1.00	1.177	5.13	0.86	0.642	9.41	0.78
UA_diffuse_5	N1, N2, N3, N4 = 100	1.391	0.875	1.59	0.79	0.308	4.52	0.75	0.171	8.13	0.68
	N1, N2, N3, N4 = 200	5.548	3.704	1.50	0.75	1.25	4.44	0.74	0.653	8.50	0.71
fdtd-2d-apml	CZ, CXM, CYM = 256	1.355	0.683	1.98	0.99	0.25	5.42	0.90	0.138	9.82	0.82
	CZ, CXM, CYM = 512	10.88	5.425	2.01	1.00	1.912	5.69	0.95	1.006	10.82	0.90
symm	NI, NJ = 1 024	19.199	14.999	1.28	0.64	6.182	3.11	0.52	3.129	6.14	0.51
	NI, NJ = 2 048	171.459	131.18	1.31	0.65	53.763	3.19	0.53	28.912	5.93	0.49
syr2k	NI, NJ = 1024	2.782	1.406	1.98	0.99	0.501	5.55	0.93	0.26	10.70	0.89
	NI, NJ = 2 048	23.241	11.623	2.00	1.00	4.003	5.81	0.97	2.24	10.38	0.86

Table 2. Time (in seconds), speed-up and efficiency

Experiments were carried out on an Intel Xeon Processor E5645, 12 threads, 2.4 GHz, 12 MB Cache and 16 GB RAM.

Loop	Parameters	CUDA cores			
		1	2	8	32
CG_cg_4	$N = 100\,\mathrm{K}$	0.488	0.269	0.077	0.019
	$N = 200\,\mathrm{K}$	1.036	0.519	0.158	0.039
LU_pintgr_4	$N2 = N4 = 1\,024,\ N1 = N3 = 1$	1.082	0.561	0.199	0.046
	$N2 = N4 = 2\,048,\ N1 = N3 = 1$	4.466	2.256	0.866	0.174
adi	$n = 500$, tsteps $= 10$	9.89	4.97	1.24	0.31
	$n = 100$, tsteps $= 100$	3.50	1.88	0.45	0.11
jacobi-2D	$n = 500$, tsteps $= 10$	2.82	1.62	0.39	0.09
	$n = 100$, tsteps $= 100$	1.06	0.64	0.16	0.04
reg-detect	length $= 256$, maxgrid $=$ iter $= 64$	26.58	16.41	5.41	1.89
	length $= 128$, maxgrid $=$ iter $= 32$	1.77	1.05	0.38	0.16

Table 3. Time (in seconds) of fine-grained loops execution for 1, 2, 8, and 32 CUDA cores

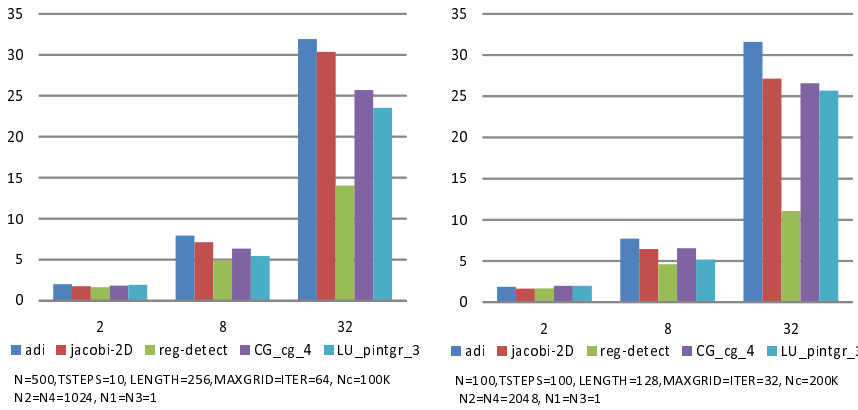


Figure 6. Speed-up of loops representing fine-grained parallelism for various numbers of CUDA cores and loop upper bounds

Figure 5 illustrates the times presented in Table 2 in a graphical way. Analysing the data in Table 2, we may conclude that for all coarse-grained parallel loops, positive speed-up is achieved. Efficiency depends on the problem size defined by index loop upper bounds and the number of CPUs used for parallel program execution. For most cases, efficiency increases with increasing the problem size for the same number of processors used.

To check the efficiency of fine-grained parallel code, we have selected (considering the previously mentioned criteria) two NBP loops: *CG_cg_4* (Conjugate Gradient

Benchmark), *LU_pintgr_4* (Lower-Upper symmetric Gauss-Seidel Benchmark) and three PolyBench loops: *adi* (Alternating Direction Implicit solver), *jacobi-2D* (2-D Jacobi stencil computation), and *reg-detect* (2-D Image processing).

Let us cite the the following fragment of paper [42]:

> *"Relatively small tasks do not always boost OpenMP programs for multi-core CPUs, because thread creation and scheduling are computationally heavy, since they can involve the saving and restoration of the processor state and expensive calls to the operating system kernel. In contrast, the parallel threading model of GPU can provide high scalability of fine-grained programs, when a set of operations are performed on data in the form of a pipeline and thread scheduling is essentially without cost."*

I.e., if fine-grained code demonstrates poor performance on multi-core CPUs, it may still provide sufficient performance on multiple GPUs. This is why we have studied speed-up of GPU codes for above mentioned loops by means of 2, 8, and 32 scalar processor cores of a multi-core graphic card, NVIDIA Tesla S1070 with 16 GB RAM.

Table 3 and Figure 6 present the experimental results: time and speed-up, respectively. There exist $\log_2 N$, $\log_2(N2-N1)$, $6*TSTEPS$, $2*STEPS$, and $4*ITER$ synchronization points for the fine-grained versions of the *CG_cg_4*, *LU_pintgr_4*, *adi*, *jacobi-2D*, and *reg-detect* benchmarks, respectively. But despite numerous synchronization points, for studied parallel fine-grained loops, positive speed-up is achieved.

Next, we present the comparison of TRACO features with those of the compilers chosen (see Section 7).

Benchmark	Parallelism	TRACO	Pluto	Par4All	Cetus	ICC
NAS	synchronization-free	109	35	79	107	45
	fine-grained	22	7	25	19	9
	total	131	42	104	126	56
PolyBench	synchronization-free	30	29	30	29	28
	fine-grained	15	10	10	8	9
	total	45	39	39	38	37

Table 4. Number of NPB and PolyBench loops parallelized by various compilers

Table 4 presents the effectivenesses of the studied compilers. TRACO is able to parallelize 131 NAS loops and 45 PolyBench loops. Pluto exposes parallelism for 42 NAS and 39 PolyBench loops, it does not support variable privatization and parallel reduction, whereas Cetus and Par4All support these transformations and parallelize more NAS loops. ICC parallelizes 56 NAS loops only.

Table 4 shows also what kind of parallelism the compilers extract. All of the studied tools extract more synchronization-free parallelism than the fine-grained one.

Table 5 presents the time of code production by means of the studied compilers. The tools were written in C++ with the exception of Cetus, which was developed

by means of Java. Each of them generates parallel OpenMP code. For time measuring, the Linux/Unix *time* command was used. The fastest tool is ICC, but it fails to transform loops with relatively simple structures, for example (*BT_rhs_1*, *LU_HP_rhs_1*, and *UA_setup_16*). The times of code generation for Pluto and TRACO are comparable. Cetus is written in Java and it is a much slower tool than Traco and Pluto, particularly when it parallelizes PolyBench loops. The most time consuming is Par4All, it takes more than one second to produce code for most of the benchmarks.

The most time consuming part in TRACO is transitive closure calculation. There is a strong need in improving algorithms for calculating transitive closure to reduce their time complexity and enhance their effectiveness.

Loop	TRACO	Pluto	Par4All	Cetus	ICC
BT_rhs_1	0.344	0.428*	1.533	1.003	0.233
CG_cg_4	0.221	0.191*	0.883*	0.608	0.073
FT_auxfnct_2	0.154	0.109	1.158	0.882	0.213
LU_HP_rhs_1	0.188	0.219*	1.481	1.058	0.189**
LU_pintgr_3	0.158	0.115*	1.158*	0.588	0.212*
UA_diffuse_5	0.289	0.113	1.176**	1.062**	0.254
adi	1.562	1.392	3.805	7.472	0.444
fdtd-2d-apml	1.625	8.630*	0.360*	10.416	0.631**
jacobi-2d	0.489	0.434	2.437	7.108	0.309
reg-detect	1.990	2.048	2.804	7.468	0.368
symm	0.590	0.361*	3.895**	9. 405	1.600**
syr2k	0.285	0.129	3.986	9.295	0.329

Table 5. Time of parallelization (in seconds) for studied loops by means of the related compilers. (*) means that output code is serial, (**) means that the number of synchronization events in a correspondent parallel code is greater than that in the code produced with TRACO.

We have compared speed-up of parallel code produced by means of TRACO with that of Pluto, Cetus, Par4all, and ICC. The results are presented in Figure 7. Analyzing these results, we can derive the following conclusions. For all benchmarks, except from *jacobi-2d*, TRACO produces parallel code where speed-up is the same or comparable with that demonstrated by the best code among codes produced by means of Pluto, Cetus, Par4all, and ICC. For the *jacobi-2d* benchmark, TRACO produces fine-grained code with more number of synchronization events than that of code produced by Pluto. This is due to the fact that any known tool fails to calculate transitive closure for the entire iteration space for this benchmark, hence TRACO seeks for parallelism only in a loop iteration sub-space being defined with inner loops only; i.e., this weakness is caused by the immaturity of existing transitive closure calculation algorithms.

Summing up, we may derive the following conclusions. For NAS and PolyBench benchmarks, the strong feature of techniques implemented in TRACO is high ef-

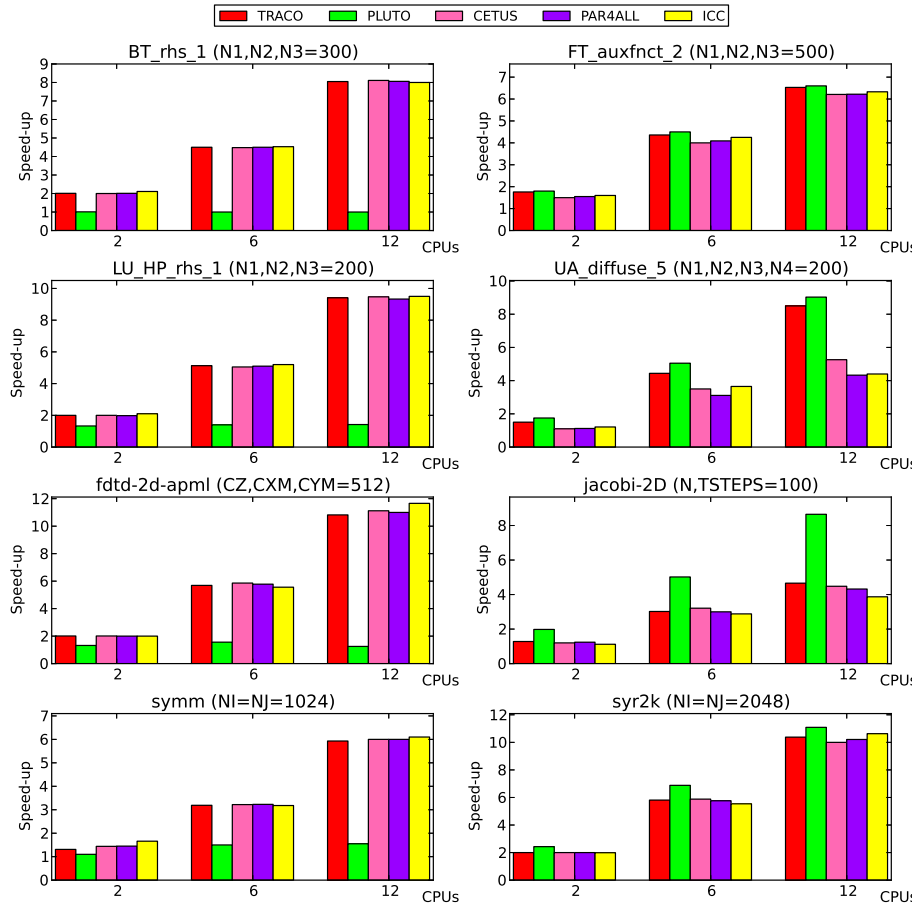


Figure 7. Speed-up of parallel codes produced by TRACO and other compilers

fectiveness: they extract more parallelism than that extracted by means of Pluto, Par4All, Cetus, ICC; performance (speed up and efficiency) of TRACO parallel code is the same or comparable to that of parallel code generated by means of the compilers mentioned above. For most benchmarks, the time complexity of techniques implemented in TRACO is less or comparable with that of techniques implemented in Pluto, Par4All, Cetus, ICC. The weaknesses of techniques implemented in TRACO are the following:

1. the immaturity of algorithms of calculating the transitive closure of a parameterized loop dependence graph: the time complexity of calculating transitive closure for some loops is very high (the time of calculation takes several hours),

there is a strong need in answering the following questions: why for these loops it is extremely high and how it can be reduced;

2. the lack of tiling algorithms implemented in TRACO: tiling algorithms based on transitive closure have to be developed and implemented in TRACO to permit for a correct comparison with techniques and compilers implementing such techniques (Pluto, PPCG).

9 CONCLUSION

We have presented a source-to-source compiler, TRACO, permitting for extracting both coarse- and fine-grained parallelism available in loops represented in the C/C++ language. It produces compilable parallel OpenMP C/C++ or OpenACC C/C++ code. TRACO implements parallelization algorithms based on the transitive closure of a relation describing all the dependences in the loop. Coarse- and fine-grained parallelism is represented with synchronization-free slices (space partitions) and a legal loop statement instance schedule (time partitions), respectively.

An experimental study carried out demonstrates a wide scope of the TRACO applicability and good speed-up of parallel applications (produced with TRACO) on memory-shared machines with multi-core processors as well as on graphic cards. Generated code is competitive with that generated by related tools. The time of compilable parallel code generation by TRACO is comparable or less than that yielded by related tools.

In the future, we plan to add the following features to TRACO: induction variable recognition and substitution; locality enhancement techniques such as tiling, array contraction, interleaving inner loops and statements in a parallel loop; while loop parallelization; handling loops with pointers.

We plan to combine iteration space slicing [1] and free scheduling [2] in one framework to permit users to manage the granularity and/or parallelism degree of target code by means of iterative compilation. We intend also to adjust TRACO to produce code for embedded and mobile platforms.

REFERENCES

[1] Beletska, A.—Bielecki, W.—Cohen, A.—Palkowski, M.—Siedlecki, K.: Coarse-Grained Loop Parallelization: Iteration Space Slicing vs. Affine Transformations. Parallel Computing, Vol. 37, 2011, pp. 479–497.

[2] Bielecki, W.—Palkowski, M.—Klimek, T.: Free Scheduling for Statement Instances of Parameterized Arbitrarily Nested Affine Loops. Parallel Computing, Vol. 38, 2012, No. 9, , pp. 518–532.

[3] Bielecki, W.—Palkowski, M.: Using Free Scheduling for Programming Graphic Cards. In: Keller, R., Kramer, D., Weiss, J.-P. (Eds.): Facing the Multicore – Challenge II. Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7174, 2012, pp. 72–83.

[4] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 4.0, 2012. Available on: `http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf`.

[5] NAS Benchmarks Suite, 2013. Available on: `http://www.nas.nasa.gov`.

[6] The Polyhedral Benchmark Suite, 2012. Available on: `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

[7] Pugh, W.—Rosser, E.: Iteration Space Slicing and Its Application to Communication Optimization. International Conference on Supercomputing, 1997, pp. 221–228.

[8] Pugh, W.—Wonnacott, D.: An Exact Method for Analysis of Value-Based Array Data Dependences. Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, Springer-Verlag, 1993.

[9] Kelly, W. et al.: The Omega Library Interface Guide. Technical Report. College Park, MD, USA, 1995.

[10] Kelly, W.—Pugh, W.—Rosser, E.—Shpeisman, T.: Transitive Closure of Infinite Graphs and Its Applications. International Journal of Parallel Programming, Vol. 24, 1996, No. 6, pp. 579–598.

[11] Beletska, A.—Bielecki, W.—Cohen, A.—Palkowski, M.: Synchronization-Free Automatic Parallelization: Beyond Affine Iteration-Space Slicing. Languages and Compilers for Parallel Computing (LCPC'09). SpringerVerlag, Lecture Notes in Computer Science, Vol. 5898, 2010, pp. 233–246.

[12] Bielecki, W.—Beletska, A.—Palkowski, M.—San Pietro, P.: Finding Synchronization – Free Parallelism Represented with Trees of Dependent Operations. 8th International Conference on Algorithms Architectures for Parallel Processing (ICA3PP'08), 2008, pp. 185–195.

[13] Bielecki, W.—Palkowski, M.: Extracting Both Affine and Non-Linear Synchronization – Free Slices in Program Loops. 8th International Conference on Parallel Processing and Applied Mathematics (PPAM'09). Springer Verlag, Lecture Notes in Computer Science, Vol. 6067, 2010, pp. 196–205.

[14] Bielecki, W.—Palkowski, M.: Using Message Passing for Developing Coarse-Grained Applications in OpenMP. 3rd International Conference on Software and Data Technologies (ICSOFT), 2008, pp. 145–152.

[15] Beletska, A.—Bielecki, W.—Cohen, A.—Palkowski, M.—Siedlecki, K.: Coarse-Grained Parallelization: Beyond Affine Iteration Space Slicing. Parallel Computing, Vol. 37, No. 8, pp. 479–497.

[16] Bastoul, C.: Code Generation in the Polyhedral Model is Easier Than you Think. Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04), Juan-les-Pins, 2004, pp. 7–16.

[17] Wonnacott, D. G.: A Retrospective of the Omega Project. Technical Report 2010-01, Haverford College, 2010.

[18] Kelly, W. et al.: New User Interface for Petit and Other Extensions. 1996, Available on: `www.cs.umd.edu/projects/omega/petit.ps.Z`.

[19] Verdoolaege, S.: Integer Set Library – Manual. Technical Report, 2011. Available on: `www.kotnet.org/~skimo/isl/manual.pdf`.

[20] Bielecki, W.—Klimek, T.—Palkowski, M.—Beletska, A.: An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations. In: Wu, W., Daescu, O. (Eds.): Combinatorial Optimization and Applications. Fourth International Conference on Combinatorial Optimization and Applications (COCOA 2010). Lecture Notes in Computer Science, Vol. 6508, 2010, pp. 104–113.

[21] Verdoolaege, S.—Cohen, A.—Beletska, A: Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. In: Yahav, E. (Ed.): Static Analysis. Proceedings of the 18th International Conference on Static Analysis (SAS '11). Springer, Lecture Notes in Computer Science, Vol. 6887, 2011, pp. 216–232.

[22] Kennedy, K.—Allen, J. R.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[23] Padua, D. A. (Ed.): Encyclopaedia of Parallel Computing. Springer, 2011.

[24] Darte, A.—Robert, Y.—Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, 2000.

[25] Darte, A.—Khachiyan, L.—Robert, Y.: Linear Scheduling is Nearly Optimal. Parallel Processing Letters, Vol. 1, 1991, No. 2, pp. 73–81.

[26] Beletska, A.—Barthou, D.—Bielecki, W.—Cohen, A.: Computing the Transitive Closure of a Union of Affine Integer Tuple Relations. In: Du, D. Z., Hu, X., Pardalos, P. M. (Eds.): Combinatorial Optimization and Applications. Third International Conference on Combinatorial Optimization and Applications (COCOA 2009), Springer, Lecture Notes in Computer Science, Vol. 5573, 2009, pp. 98–109.

[27] Bielecki, W.—Klimek, T.—Trifunovic, K.: Calculating Exact Transitive Closure for a Normalized Affine Integer Tuple Relation. Electronic Notes in Discrete Mathematics, Vol. 33, 2009, pp. 7–14.

[28] Chen, C.: Omega+ Library. School of Computing University of Utah, 2011. Available on: `http://www.cs.utah.edu/~chunchen/omega`.

[29] Lim, A. W.—Cheong, G. I.—Lam M. S.: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, ACM Press, 1999, pp. 228–237.

[30] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: I. One-Dimensional Time. International Journal of Parallel Programming, Vol. 21, 1992, No. 5, pp. 313–348.

[31] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: II. Multi-Dimensional Time. International Journal of Parallel Programming, Vol. 21, 1992, No. 5, pp. 389–420.

[32] Nugteren, C.—Corporaal, H.: Introducing 'Bones': A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5), ACM, New York, NY, USA, 2012, pp. 1–10.

[33] Garcia, S.—Jeon, D.—Louie, C.—Taylor, M.: The Kremlin Oracle for Sequential Code Parallelization. IEEE Micro 32, Vol. 32, 2012, No. 4, pp. 42–53.

[34] Intel® Compilers, 2013. Available on: `http://software.intel.com/en-us/intel-compilers`.

[35] Bondhugula, U.—Hartono, A.—Ramanujam, N.—Sadayappan, N.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. SIGPLAN Notices – PLDI 2008, Vol. 43, 2008, No. 6, pp. 101–113. Available on: `http://pluto-compiler.sourceforge.net`.

[36] Bondhugula, U.—Baskaran, M.—Krishnamoorthy, S.—Ramanujam, J.—Rountev, A.—Sadayappan, P.: Automatic Transformations for Communication – Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: Hendren, L. (Ed.): Compiler Construction (CC 2008). Springer, Lecture Notes in Computer Science, Vol. 4959, 2008, pp. 132–146.

[37] Bondhugula, U.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model. Ph.D. Thesis. Columbus, OH, USA, 2008.

[38] Mehdi, A.: Par4All User Guide. 2012, Available on: `http://www.par4all.org`.

[39] Amini, M. et al.: PIPS Is Not (Just) Polyhedral Software. First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011), Chamonix, France, 2011.

[40] Polylib – A Library of Polyhedral Functions. 2014. Available on: `http://icps.u-strasbg.fr/polylib`.

[41] Dave, C.—Bae, H.—Min, S.-J.—Lee, S.—Eigenmann, R.—Midkiff, S.: Cetus: A Source-to-Source Compiler Infrastructure for Multicores. Computer, Vol. 42, 2009, No. 12, pp. 36–42.

[42] Garland, M.—Kirk, D. B.: Understanding Throughput-Oriented Architectures. Communication of the ACM, Vol. 53, 2010, No. 11, pp. 58–66.

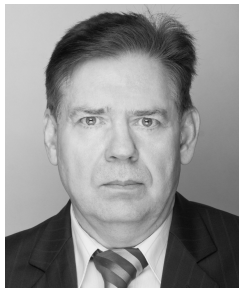
[43] Shajulin, B.—Gerndt, M.: Scalablity and Perfomance Analysis of OpenMP Codes Using the Periscope Toolkit. Computing and Informatics, Vol. 33, 2014, pp. 921–942.

[44] Verdoolaege, S.—Juega, J. C.—Cohen, A.—Gómez, J. I.—Tenllado, C.—Catthor, F.: Polyhedral Parallel Code Generation for CUDA. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 9, 2013, No. 4, Article No. 54.

[45] Lu, P.—Li, B.—Che, Y.—Wang, Z.: PIT: A Framework for Effectively Composing High-Level Loop Transformations. Computing and Informatics, Vol. 30, 2010, No. 5, pp. 943–963.

[46] Lu, P.—Che, Y.—Wang, Z.: UMDA/S: An Effective Iterative Compilation Algorithm for Parameter Search. Computing and Informatics, Vol. 29, 2010, No. 6+, pp. 1159–1179.

Marek Palkowski has graduated and received his Ph.D. degree in computer science from the Technical University of Szczecin, Poland. The main goal of his research is extracting parallelism and tiling from program loops, and developing the TRACO compiler.



Wlodzimierz Bielecki is Head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest includes parallel and distributed computing, optimizing compilers, extracting both fine- and coarse-grained parallelism available in program loops.