

# TRACO Parallelizing Compiler

Marek Palkowski and Włodzimierz Bielecki

**Abstract** This 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. On its output, TRACO produces compilable parallel OpenMP C/C++ and/or OpenACC C/C++ code. The effectiveness of TRACO and efficiency of parallel code produced by TRACO are evaluated by means of the NAS Parallel Benchmark and Polyhedral Benchmark suites.

**Keywords** Source-to-source parallelizing compiler · Fine- and coarse-grained parallelism · Free scheduling · Transitive closure

## 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 decrease the productivity of programmers and increase 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 multicore systems.

---

M. Palkowski (✉) · W. Bielecki  
Faculty of Computer Science and Information Systems, West Pomeranian University  
of Technology, Żołnierska 49, 71210 Szczecin, Poland  
e-mail: mpalkowski@wi.zut.edu.pl  
URL: <http://www.wi.zut.edu.pl>

W. Bielecki  
e-mail: wbielecki@wi.zut.edu.pl

The goal of this paper is to present an open source parallelizing compiler, TRACO, implementing loop parallelization approaches based on transitive closure.

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 pre-processor of the C program, data dependence analyzer, parallelization engine, code generator, and postprocessor. To the best of our knowledge, there are no source-to-source compilers based on the transitive closure of dependence relation graphs.

Results of a comparative analysis of TRACO features and those demonstrated by Pluto, Par4all, Cetus, and ICC have been discussed.

## 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.

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 [18], 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 [18].

Standard operations on relations and sets are used, such as intersection ( $\cap$ ), union ( $\cup$ ), difference ( $-$ ), domain ( $\text{dom } R$ ), range ( $\text{ran } R$ ), and relation application ( $S' = R(S): e' \in S' \text{ iff exists } e \text{ s.t. } e \rightarrow e' \in R, e \in S$ ). In detail, the description of these operations is presented in [11, 18].

The positive transitive closure for a given relation  $R$ ,  $R^+$ , is defined as follows [11]:

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

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 [12]:  $R^* = R^+ \cup I$ , 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 [3].

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  $\text{domain}(R) \cup \text{range}(R)$ . 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 [2, 11].

### 3 Coarse-Grained Parallelism Extraction Using Iteration Space Slicing

Algorithms presented in paper [3] are based on transitive closure and allow us to generate parallel code representing synchronization-free slices or slices requiring occasional synchronization.

**Definition 1** Given a dependence graph defined by a set of dependence relations, a slice  $S$  is a weakly connected component of this graph.

**Definition 2** An ultimate dependence source is a source that is not the destination of another dependence. Ultimate dependence sources represented by relation  $R$  can be found by means of the following calculations:  $\text{domain}(R) - \text{range}(R)$ . 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 [3]. 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, a set of statement instances,  $S_{UDS}$ , are calculated. It describes all ultimate dependence sources of slices as

$$S_{UDS} = \text{domain}(R) - \text{range}(R). \quad (2)$$

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:

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

The condition  $(e < e')$  in the constraints of relation  $R_{USC}$  above means that  $e$  is lexicographically smaller than  $e'$ . 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  $S_{repr} = S_{UDS} - \text{range}(R_{USC})$ . 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:

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

Any tool to generate code for scanning polyhedra can be applied to produce parallel pseudocode, for example, the CLOOG library [2] or the *codegen* function of the Omega project [11].

## 4 Variable Privatization and Parallel Reduction

TRACO automatically recognizes 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 [13].

**Definition 5** Given  $n$  inputs  $x_1, x_2, \dots, x_n$  and an associative operation  $\otimes$ , a parallel reduction algorithm computes the output  $x_1 \otimes x_2 \otimes \dots \otimes x_n$  [16].

The idea of recognizing variables to be privatized and/or used for parallel reduction and being implemented in TRACO is the following. The first step 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 [13].

Next, we seek for variables that are involved in reduction dependences 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: (i) dependence relations not being eliminated by means of variable privatization and reduction and (ii) 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.

## 5 Finding (Free) Scheduling for Parameterized Loops

The algorithm, presented in our paper [4], 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** ([8]) The *free schedule* is the function that assigns discrete time of execution to each loop statement instance as soon as its operands are available, that is, it is mapping  $\sigma:LD \rightarrow \mathbb{Z}$  such that

$$\sigma(p) = \begin{cases} 0 & \text{if there is no } p_1 \in LD \text{ s.t. } p_1 \rightarrow p \\ 1 + \max(\sigma(p_1), \sigma(p_2), \dots, \sigma(p_n)); & p, p_1, p_2, \dots, p_n \in LD; \\ p_1 \rightarrow p, p_2 \rightarrow p, \dots, p_n \rightarrow p, & \end{cases} \quad (5)$$

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

The free schedule is the fastest legal schedule [8].

The idea of the algorithm to extract time partitions applying transitive closure is as follows [4]. Given preprocessed relations  $R_1, R_2, \dots, R_m$ , we first 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:

$$FS = \{[X] \rightarrow [k, Y] : X \in UDS(R) \wedge (k, Y) \in Range((R')^* \setminus \{[0, X]\}) \wedge \neg(\exists k' > k \text{ s.t. } (k', Y) \in Range(R')^+ \setminus \{[0, X]\})\}, \quad (6)$$

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 dependence 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  $\text{Range}(FS)$ , 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  $\text{Range}(FS)$  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 [2, 11].

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  $k$ .

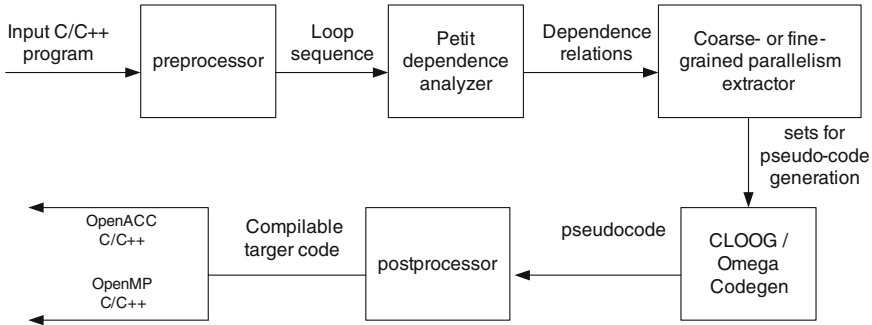
Finally, we expose independent statement instances, that is, those that do not belong to any dependence and generate code enumerating them. According to the free schedule, they are to be executed at time  $k = 0$ .

## 6 Implementation

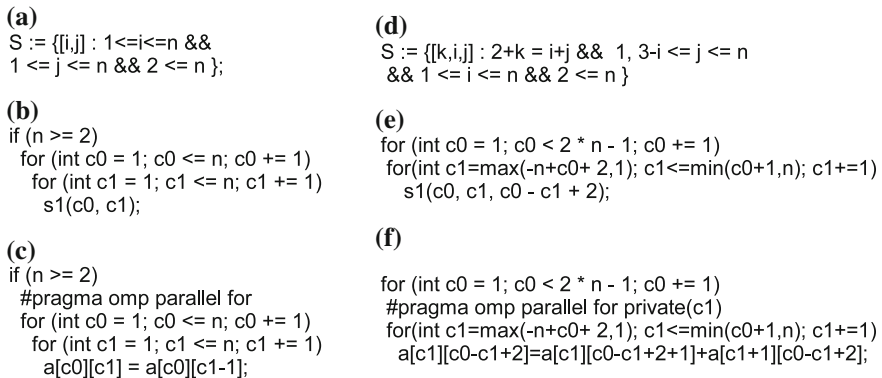
Figure 1 shows the details of the TRACO implementation. Currently, it supports C/C++ programs on its input. A preprocessor, written in the Python language, recognizes loops in a source program and converts them to the format acceptable by the Omega dependence analyzer, Petit, that returns a set of dependence relations representing all the 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 to be serial to produce coarse-grained code.

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 pseudocode representing slices are calculated and forwarded to a pseudocode generator. Otherwise, data necessary for extracting (free) scheduling are prepared and directed to the pseudocode generator. A postprocessor generates parallel code in OpenMP/OpenACC C/C++. Below, we present some details concerned code generation.

Input for the pseudocode 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



**Fig. 1** TRACO organization



**Fig. 2** Code generation details: **a–c** synchronization-free slices; **d–f** free scheduling

all dependent statement instances are originated from the same statement. An example set is illustrated in Fig. 2a. 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 pseudocode Fig. 2b, 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 for scanning set elements in the lexicographic order, can be applied in TRACO, for example, the codegen function of Omega or Omega+ [7].

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 parameterized time partition, and the last one represents a statement identifier, which may be skipped when all statement instances are originated from the same statement. An example set is given in Fig. 2d, 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 pseudocode Fig. 2e, 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.

Compilable OpenMP/OpenACC C/C++ code is produced by means of the post-processor written in Python. It inserts source loop statements with proper index expressions into pseudocode. 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. For example, provided that the set  $S$  on Fig. 2a is associated with the statement  $a[i][j] = a[i][j - 1]$ , in the pseudo statement  $s1(c0, c1)$  on Fig. 2b, variables  $c0, c1$  correspond to variables  $i, j$  which are substituted for  $c0, c1$  in the source statement Fig. 2c.

Given the set  $S$  in Fig. 2d is associated with the source statement  $a[i][j] = a[i][j + 1] + a[i + 1][j]$ , the code generator recognizes that in the pseudocode on Fig. 2e  $c0$  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 Fig. 2f  $a[c1][c0 - c1 + 2] = a[c0][c0 - c1 + 2 + 1] + a[c0 + 1][c0 - c1 + 2]$ ).

Depending on whether pseudocode 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

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 (i) be a source-to-source compiler; (ii) support the C language; (iii) produce compilable code in OpenMP/ACC C/C++. The following compilers were chosen to be compared with TRACO: ICC, Pluto, Cetus, and Par4All.

- ICC** [10]. 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.
- Pluto** [5]. An automatic parallelization tool is based on the polyhedral model [6]. 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 [6]. Pluto does not support variable privatization and reduction recognition.

**Par4All** [14]. A tool is composed of different components: the PIPS tool [1], the Polylib library [17], 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** [9]. It provides an infrastructure for research on multicore 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 is limited only to basic transformations: induction variable substitution, reduction recognition, array privatization, pointers, alias, and range analysis.

The compilers, mentioned above, do not based on the transitive closure of dependence graphs.

## 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), and efficiency of parallel loops produced. Another goal was to compare these features of TRACO with those demonstrated by the compilers classified for comparison (see Sect. 7). To evaluate the effectiveness of TRACO, we have experimented with NAS Parallel Benchmarks 3.3 (NPB) [15] and Polyhedral Benchmarks 3.2 (PolyBench) [19].

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 Sect. 4 is applied, this results in parallelization of 70 NAS loops. Finally, for the remaining benchmarks, techniques extracting (free) scheduling are applied that yield 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*.

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 [12, 20] are not able to produce transitive closure for these loops. There exists a strong need in improving existing algorithms for calculating transitive closure to enhance their effectiveness. 30 PolyBench loops

**Table 1** Techniques of loop parallelization

Technique	No. of NAS loops	No. of Polybench loops
Privatization only	39	0
Slicing with privatization and reduction	70	30
Free scheduling	22	15
<i>Loop parallelized</i>	131	45
<i>All loops</i>	134	48

were parallelized by applying algorithms of synchronization-free slices extraction [3]. For 15 PolyBench loops, fine-grained parallelism was found only (the outermost loop is serial).

To check the performance of coarse-grained parallel code, produced with TRACO, we have selected the following four computative heavy 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), linebreak *UA\_diffuse\_5* (Unstructured Adaptive Benchmark) and the three PolyBench loops: *fddt-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 speedup is calculated. Speedup is a ratio of sequential time and parallel time,  $S = T(1)/T(P)$ , where  $P$  is the number of processors. Experiments were carried out on an Intel Xeon Processor E5645, 12 Threads, 2.4 GHz, 12 MB Cache, and 16 GB RAM.

Figure 3 illustrates code execution times(in seconds) in a graphical way.

To check the performance of fine-grained parallel code, we have selected the two NBP loops: *CG\_cg\_4* (Conjugate Gradient Benchmark), *LU\_pintgr\_4* (Lower–Upper symmetric Gauss–Seidel Benchmark) and the three PolyBench loops: *adi* (Alternating Direction Implicit solver), *jacobi-2D* (2-D Jacobi stencil computation), and *reg-detect* (2-D Image processing).

Figure 4 presents parallel code speedups. There exist  $\log_2 N$ ,  $\log_2(N2 - N1)$ ,  $6*STEPS$ ,  $2*STEPS$ , 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 speedup is achieved ( $S > 1$ ).

Next, we present the comparison of TRACO features with those of the compilers classified for comparison (see Sect. 7). Table 2 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 2 shows also what kind of parallelism the compilers extract.

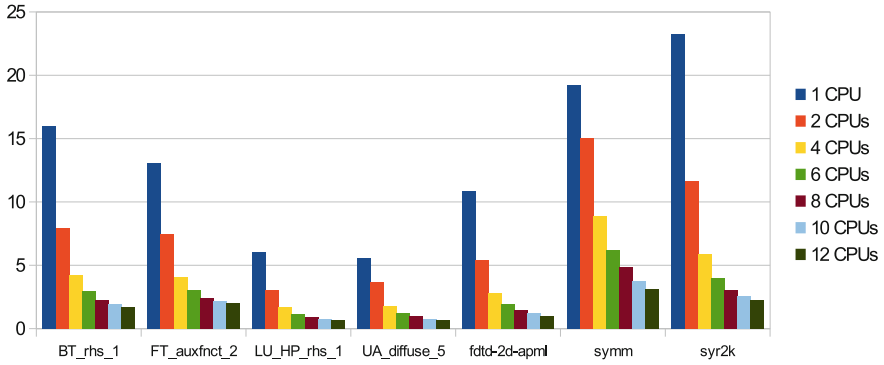


Fig. 3 Times (in seconds) of program loops execution for various numbers of CPUs

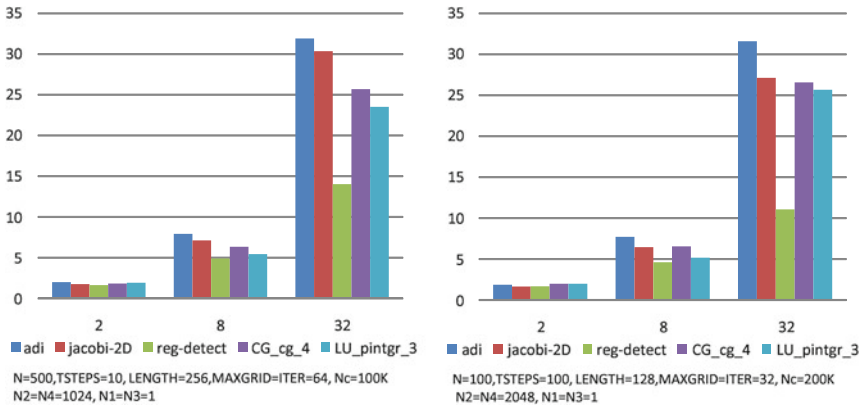


Fig. 4 Speedup of loops representing fine-grained parallelism for various numbers of CUDA cores and loop upper bounds

Table 2 Number of NPB and Polybench loops parallelized by various compilers

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

## 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 implements parallelization algorithms based on the transitive closure of a relation describing all the dependences in a loop and produces compilable parallel OpenMP C/C++ or OpenACC C/C++ code. It is the first compiler that applies the transitive closure of a dependence relation to extract loop parallelism, this enlarges the scope of loop nests which can be parallelized in comparison with existing compilers.

## References

1. Amini, M., et al.: PIPS Is not (just) Polyhedral Software. In: International Workshop on Polyhedral Compilation Techniques (IMPACT'11). Chamonix, France April 2011
2. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques. pp. 7–16. Juan-les-Pins September 2004
3. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. *Parallel Comput.* **37**, 479–497 (2011)
4. 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*. Lecture Notes in Computer Science, LNCS 7174, pp. 72–83. Springer, Berlin (2012)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: *Proceedings of SIGPLAN Not.* 43(6), 101–113 (June 2008), <http://pluto-compiler.sourceforge.net>
6. Bondhugula, U., et al.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) *Compiler Construction*. Lecture Notes in Computer Science, LNCS 4959, pp. 132–146. Springer, Heidelberg (2008)
7. Chen, C.: Omega+ library. School of Computing University of Utah, (February 2011), <http://www.cs.utah.edu/chunchen/omega/>
8. Darte, A., Robert, Y., Vivien, F.: *Scheduling and Automatic Parallelization*. Birkhauser, Boston (2000)
9. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A source-to-source compiler infrastructure for multicores. *Computer* **42**, 36–42 (2009)
10. Intel<sup>®</sup> Compilers (2013), <http://software.intel.com/en-us/intel-compilers>
11. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical report, College Park (1995)
12. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program.* **24**(6), 579–598 (1996)
13. Kennedy, K., Allen, J.R.: *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., CA (2002)
14. Mehdi, A.: Par4All User Guide (2012), <http://www.par4all.org>
15. NAS benchmarks suite. <http://www.nas.nasa.gov>
16. Padua, D.A. (ed.): *Encyclopedia of Parallel Computing*. Springer (2011)
17. Polylib - a library of polyhedral functions, <http://icps.u-strasbg.fr/polylib>
18. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*. Springer, Berlin (1993)

19. The Polyhedral Benchmark suite (2012), <http://www.cse.ohio-state.edu/pouchet/software/polybench/>
20. Verdoolaege, S., Cohen, A., Beletka, A.: Transitive closures of affine integer tuple relations and their overapproximations. In: Proceedings of the 18th international conference on Static analysis, SAS'11. pp. 216–232. Springer, Berlin (2011)