




# Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications

Hamid Arabnejad<sup>1</sup> · João Bispo<sup>1</sup> · João M. P. Cardoso<sup>1</sup> · Jorge G. Barbosa<sup>2</sup> 

Published online: 17 December 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

Directive-driven programming models, such as OpenMP, are one solution for exploring the potential parallelism when targeting multicore architectures. Although these approaches significantly help developers, code parallelization is still a non-trivial and time-consuming process, requiring parallel programming skills. Thus, many efforts have been made toward automatic parallelization of the existing sequential code. This article presents `AutoPar-Clava`, an OpenMP-based automatic parallelization compiler which: (1) statically detects parallelizable loops in C applications; (2) classifies variables used inside the target loop based on their access pattern; (3) supports *reduction* clauses on scalar and array variables whenever it is applicable; and (4) generates a C OpenMP parallel code from the input sequential version. The effectiveness of `AutoPar-Clava` is evaluated by using the NAS and Polyhedral Benchmark suites and targeting a x86-based computing platform. The achieved results are very promising and compare favorably with closely related auto-parallelization compilers, such as Intel C/C++ Compiler (`icc`), ROSE, TRACO and CETUS.

**Keywords** Static code analysis · Compiler framework · Parallel programming · Code transformations

---

✉ Jorge G. Barbosa  
jbarbosa@fe.up.pt

Hamid Arabnejad  
hamid.arabnejad@gmail.com

João Bispo  
jbispo@fe.up.pt

João M. P. Cardoso  
jmpc@fe.up.pt

<sup>1</sup> INESC TEC and Faculdade de Engenharia, Universidade do Porto, Porto, Portugal

<sup>2</sup> LIACC and Faculdade de Engenharia, Universidade do Porto, Porto, Portugal

## 1 Introduction

Multicore processors have the potential to improve the performance of applications on modern computing platforms. Parallel programming approaches, such as OpenMP [30] and OpenCL [29], have emerged to assist application developers to exploit more and more parallelism in multi- and many core processors. However, even these approaches require the users (e.g., programmers) to identify the parallelizable regions from their applications and apply the target parallel programming model on those regions. Thus, to take advantage of these programming models, programmers must parallelize their applications, a non-trivial task which requires in-depth knowledge of both the parallel paradigm and the target architecture to deal with fundamentals of parallel programming such as data dependencies, load balancing, synchronization and race conditions.

OpenMP [15, 30], one of the most popular directive-driven programming models to write shared-memory parallel programs, provides a simple and flexible interface for developing parallel applications for platforms ranging from embedded and standard desktop computers to supercomputers. For instance, programmers can change the execution pattern to exploit loop-level parallelism by simply adding a `#pragma omp parallel` directive before a target parallelizable for-type loop. Despite the usefulness of these directive-based parallel programming models, the main critical issue that needs to be addressed by programmers is which regions (e.g., loops) are parallelizable. To make this decision, programmers need to analyze the target regions to ensure that those regions can be safely executed in parallel. Also, to add the directive for the detected parallelizable loops, programmers need to classify each variable in the loop body into the proper OpenMP scoping (e.g., `private`, `first-private`, or `reduction` clauses).

Many efforts have been made to support developers [6, 23, 27] as well as to provide automatic parallelization [5, 32, 39, 40]. In this paper, we focus particularly on a flexible source-to-source compilation with C-code as an input, and the parallelized version, annotated by OpenMP directives, as the output. We target loop-level parallelism as the source of potential parallelism, and we provide parallelization strategies that improve performance by enhancing the quality of the parallelization, as well as the number of candidate parallel regions when compared with state-of-the-art compilers.

Arbitrary control flow and sequential program analysis, on our automatic OpenMP-based parallelization approach, named `AutoPar-Clava` (our first efforts were presented in [3, 4]), is based on a source-to-source C compiler, namely `Clava`, which provides an Abstract-Syntax Tree (AST) and a high-level programming environment for specifying source code analyses and transformations [14, 34]. Parsing in `Clava` is done using a frontend based on Clang [16]. `Clava` uses a custom C/C++ AST, which closely resembles the intermediate representation used by Clang, with extensions to support AST-based transformations, and the capability to generate code directly from the AST. To identify data dependencies inside a loop, we rely on the work of Pugh et al. [20, 37] known as the `Omega-test` and available by means of the `Petit` [19] tool.

The main contributions of this paper are:

- A versatile auto-parallelization engine integrated in a source to source compiler and exposing to users its parallelization strategies in a high-level and modifiable code;
- A sequential code analyzer that increases the number of candidate parallel loops, by providing techniques such as scalar/array variable privatization, induction variables and parallel scalar and array reductions;
- Validation and evaluation by using an extensive set of benchmarks showing the impact of the proposed approach in terms of execution time and providing experimental results and comparisons over other auto-parallelization compilers, such as ROSE [38], Cetus [17] and TRACO [32].

In relation to our first versions of the tool [3, 4], the current version improved: (a) the data dependency analysis and, consequently, is able to parallelize more loops; (b) identifies array reductions if the size of the array can be determined during loop analysis; and (c) performs array scoping. In this paper, we do not consider loop transformations, such as loop fusion, loop unrolling and loop tiling techniques [12]. For example, approaches, such as polyhedral-based [1], that allow to exploit data locality and parallelism, might be considered in future developments of `AutoPar-Clava` and may improve its current performance. Additionally, the current version does not support functions with multiple exit points, recursive functions and calls to functions whose source code is not available in the input source.

The rest of the paper is organized as follows. Section 2 motivates and introduces core concepts. The proposed approach, `AutoPar-Clava`, is introduced and discussed in detail in Section 3. The experimental methodology is presented in Sect. 4, and in Sect. 5, experimental results are presented and discussed. Section 6 presents related work and available automatic parallelization tools with a brief description of some well-known approaches. Finally, Sect. 7 draws some conclusions and briefly outlines future work.

## 2 Motivation

Today's high-performance computing (HPC) systems are composed of many multi-core CPUs [42]. To take advantage of the existent parallelism, one of the following two approaches is usually applied. Either the application code is almost fully rewritten or the code is parallelized by developers, possibly using tools to help them on this task. Parallelization of software code is of paramount importance in order to cope with the target system resources, to improve the system utilization rate and to significantly reduce execution time. However, it is a time-consuming process that requires parallel programming skills, which is an obstacle for many developers. Hence, there is a need to continue researching and developing parallelizing compilers.

Source-to-source compilers have shown to be an effective solution for automatic parallelization. The main advantage is that the generated output code can be easily

inspected, manually modified, if desired, and the binary code can be generated with any compiler chosen by the user.

There is a variety of source-to-source compilers, tools and libraries to make easier the auto-parallelization of code (see a recent study of autoparallelization frameworks in [40]), such as the Intel Compiler and PGI [35] as well-known commercial ones, and ROSE [25, 38, 39], Cetus [5, 17, 24], TRACO [32, 33] and PLUTO [13] as examples of academic approaches. Many of these tools present limitations on the set of OpenMP scoping, input code size, reduction for arrays and at the array element level, as expressed in Sect. 6. `AutoPar-Clava` is able to receive a complete program as input and produce a parallel version, ready to be compiled, which achieves better or equivalent performances as the state-of-the-art source-to-source auto-parallelization tools considered in this paper.

Additionally, the Clava framework, provides a higher level of abstraction, when compared with other approaches, and offers an accessible path that allows the reproduction or modification of parallelization strategies. A flexible compiler infrastructure that allows users to develop or extend parallelization strategies, without depending on low-level internal details of a specific compiler, allows to efficiently address the following:

- The evolution of the OpenMP standard needs adaptability and extensions to the parallelization strategies included in compilers providing OpenMP-based automatic parallelization. One such example is the introduction of task-based parallelization and of new clauses;
- No parallelization strategy fits all cases (e.g., applications and target machines), and it is important that advanced programmers, tuning experts, performance engineers and compiler specialists be able to extend or include new parallelization strategies whenever needed;
- Ideas regarding new parallelization strategies need experiments and evaluation considering benchmark repositories to decide about their inclusion in production compilers. Thus, a compiler framework that facilitates this process helps research and innovation.

Bearing in mind the previous aspects, this paper presents `AutoPar-Clava`, an OpenMP-based automatic parallelization library implemented on top of the Clava source-to-source compiler.

### 3 Automatic parallelization

This section provides an overview and technical details of `AutoPar-Clava` for automatic parallelization of the input C code via OpenMP directives. Loops are often the main code regions to parallelize due in part to their time-consuming nature. However, there are several reasons that may prevent loop parallelization, such as the existence of data dependencies [46]. Generally, auto-parallelization approaches try to detect any occurrence of data dependencies between loop iterations by analyzing scalar variables and array access patterns in a static or dynamic way (e.g.,

```

1 #pragma omp parallel for private(i) firstprivate(N, M, index, a, z, c)
  | reduction(+ : b[:M]) reduction(+ : r[index])
2 for (int j=0; j<N; j++) {
3   double d = 0.0;
4   #pragma omp parallel for private(i) firstprivate(M, a, z, c) reduction(+ : d)
5   for (int k=0; k < M; k++) {
6     i = k + 1;
7     d += a[k] * z[c[k]];
8     b[i] += a[k];
9   }
10  r[index] += d;
11 }

```

**Fig. 1** Example of OpenMP C code generated by AutoPar-Clava

using profiling). The auto-parallelization strategy proposed in this paper is focused on static data-dependence analysis.

AutoPar-Clava starts by analyzing the input source code and identifies all candidate loops for parallelization. Typically, a loop is a candidate to be parallelized if it follows a certain canonical form and avoids certain restrictions, e.g., not containing any `break`, `exit`, `return`, statements and system calls. Then, to decide if a candidate loop can be parallelized, AutoPar-Clava performs inter-iteration data-dependency analysis. In the end, AutoPar-Clava generates an OpenMP version of the input source code for each loop to be parallelized. The proposed approach contains four main phases: (1) preprocessing of the sequential code; (2) data-dependency analysis; (3) parallelization via OpenMP; and (4) code generation.

Figure 1 shows a C code excerpt annotated with OpenMP directives (as C pragmas) automatically generated by AutoPar-Clava, from the input C code without the annotations. When considering the parallelization of the outermost loop, the AutoPar-Clava generates the OpenMP pragma in line 1. When considering the parallelization of the innermost loop, the AutoPar-Clava generates the OpenMP pragma of line 4. At the moment, the AutoPar-Clava strategy parallelizes the outermost loop of Fig. 1 and thus generates only the pragma in line 1. Further strategies could be research schemes to decide the loop to be parallelized.

### 3.1 Preprocessing

The source-to-source Clava compiler processes LARA code [14, 34] that allows to perform code queries, modifications and source-code generation requests over the AST. With Clava, users can adapt, extend, or develop their own custom program analyses and transformations using a high-level programming model based on aspect-oriented concepts and JavaScript. Figure 2 shows an excerpt of LARA code that reports variable pattern accesses (i.e., `read`, `write`, or `readwrite`) within innermost loop bodies from a given input C code. For example, let us consider the C code example of Fig. 1 as input to Clava and the execution of this LARA code. In this LARA example, line 1 queries the code and selects all variable accesses (`varref` join points) inside the two available loop bodies (`loop.body`) from the C code. By

<pre> 1  select 2  loop.body.stmt.varref 3  end 4  apply 5  println( 6    \$varref.name + 7    \$varref.useExpr.use + 8    \$varref.useExpr.joinpointType 9    \$stmt.code 10 ); 11 end 12 condition 13 \$loop.nestedLevel === 1 14 end </pre>	<table border="1"> <thead> <tr> <th>Var</th> <th>useExpr</th> <th>Type</th> <th>used in</th> </tr> </thead> <tbody> <tr><td>i</td><td>write</td><td>varref</td><td>i = k + 1</td></tr> <tr><td>k</td><td>read</td><td>varref</td><td>i = k + 1</td></tr> <tr><td>d</td><td>readwrite</td><td>varref</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>a</td><td>read</td><td>arrayAccess</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>k</td><td>read</td><td>varref</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>z</td><td>read</td><td>arrayAccess</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>c</td><td>read</td><td>arrayAccess</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>k</td><td>read</td><td>varref</td><td>d += a[k] * z[c[k]]</td></tr> <tr><td>b</td><td>readwrite</td><td>arrayAccess</td><td>b[i] += a[k]</td></tr> <tr><td>i</td><td>read</td><td>varref</td><td>b[i] += a[k]</td></tr> <tr><td>a</td><td>read</td><td>arrayAccess</td><td>b[i] += a[k]</td></tr> <tr><td>k</td><td>read</td><td>varref</td><td>b[i] += a[k]</td></tr> </tbody> </table>	Var	useExpr	Type	used in	i	write	varref	i = k + 1	k	read	varref	i = k + 1	d	readwrite	varref	d += a[k] * z[c[k]]	a	read	arrayAccess	d += a[k] * z[c[k]]	k	read	varref	d += a[k] * z[c[k]]	z	read	arrayAccess	d += a[k] * z[c[k]]	c	read	arrayAccess	d += a[k] * z[c[k]]	k	read	varref	d += a[k] * z[c[k]]	b	readwrite	arrayAccess	b[i] += a[k]	i	read	varref	b[i] += a[k]	a	read	arrayAccess	b[i] += a[k]	k	read	varref	b[i] += a[k]
Var	useExpr	Type	used in																																																		
i	write	varref	i = k + 1																																																		
k	read	varref	i = k + 1																																																		
d	readwrite	varref	d += a[k] * z[c[k]]																																																		
a	read	arrayAccess	d += a[k] * z[c[k]]																																																		
k	read	varref	d += a[k] * z[c[k]]																																																		
z	read	arrayAccess	d += a[k] * z[c[k]]																																																		
c	read	arrayAccess	d += a[k] * z[c[k]]																																																		
k	read	varref	d += a[k] * z[c[k]]																																																		
b	readwrite	arrayAccess	b[i] += a[k]																																																		
i	read	varref	b[i] += a[k]																																																		
a	read	arrayAccess	b[i] += a[k]																																																		
k	read	varref	b[i] += a[k]																																																		

(a)
(b)

**Fig. 2** **a** An example of LARA code that *prints* variable access types inside innermost loops in input C code; and **b** the output when Clava applies the sample LARA code to the source code in Fig. 1

filtering the query, in line 13, LARA only considers the body of the innermost loop in the sample code (i.e., `$loop.nestedLevel === 1`). By querying the different attributes available in the join point `varref`, we can retrieve information such as variable name (i.e., `name`), variable access pattern (i.e., `use`) and variable type (i.e., `joinpointType`). This type of information is extensively used in the data-dependency analysis process.

### 3.2 Dependency analysis

Dependency analysis [28] involves finding the four types of data-dependencies, i.e., *Anti-dependency*, *Output-dependency*, *Flow(true)-dependency* and *Input-dependency*.

To determine if a loop can be parallelized, two types of data-dependencies are analyzed: (1) *loop-independent* that represents dependencies within a single loop iteration; and (2) *loop-carried* that represents dependencies among different iterations of a loop.

AutoPar-Clava uses separate dependency analysis strategies to process *scalar* and *array* dependencies. A loop is considered for parallelization if it is determined that: (1) it has no true dependencies; or (2) it has a true dependency, but it is a reduction operation; or (3) it has a false dependency so that it can be resolved by loop-private variables.

To perform dependency analysis on scalar and array variables, first AutoPar-Clava does dataflow analysis over all statements in the loop and extracts how each variable is used. From the AST representation of the input code, Clava compiler provides information, such as the access type of variables (i.e., *Read-R*, *Write-W*). With this information, an access pattern (e.g., `RRWRRR`) for each variable (scalar/array) is identified. The *usage pattern* is defined as a compressed version of the access pattern by removing consecutive repetitions from it (e.g., the *usage pattern* of `RRWRRR` is `RWR`) and is used to identify the data dependencies of the variables. In addition to determine access patterns within each target loop, the first access outside

the loop, to each variable, is registered in the *nextUse* attribute. For both scalar and array variables, *AutoPar-Clava* identifies reduction operations and categorize them into a reduction scope by using a pattern matching algorithm which follows the rules specified by OpenMP [31].<sup>1</sup>

One of the most common obstacles to loop parallelization is loop-carried dependencies over array elements. Array elements can be characterized by subscript expressions, which usually depend on loop index variables. As with most auto-parallelization tools, *AutoPar-Clava* only deals with array affine index functions when determining data-dependencies [28] and thus loops with non-affine index functions, e.g.,  $A[B[i]]$  are not considered for parallelization when there is a write in the access pattern.

There are several techniques to perform data-dependency analysis, and the most well-known tools use one or more of those techniques. For example, *AutoPar* [25], used in the ROSE compiler [38], uses the Gaussian elimination algorithm to solve a set of linear integer equations of loop induction variables, in the form of Banerjee–Wolfe inequalities [45]. Other approaches use tests such as GCD [7], Extended GCD [26] and Omega [37]. *AutoPar-Clava* uses the Omega library<sup>2</sup> for data dependency analysis. Loop dependencies are converted into dependency relations in the form of Presburger arithmetic that are directly analyzed using the Omega library. Then, the output dependency analyzed for scalar and array variables within a loop are used to classify each variable into the proper OpenMP scoping.

### 3.2.1 Induction variables

Induction variables [28] represent scalar variables that are updated in each iteration of a loop. Induction variable substitution is used for resolving certain classes of data dependencies, and it is useful to resolve iteration dependencies on array accesses when they appear in indexes of those arrays. Furthermore, the combination of induction variable substitution with array privatization provides a powerful technique for removing cross-iteration dependencies [43]. *AutoPar-Clava* uses this technique to handle induction statements, specifically when they appear in subscript expressions for array variable access. Considering the C code in Fig. 1, the occurrence of the scalar variable  $i$  in array access  $B[i]$  is replaced by its expression (i.e.,  $k+1$ ) and becomes  $B[k+1]$ . Note that, all these substitutions are applied during the analysis phases and are temporary in order to not change the original code.

### 3.2.2 Privatization

A private variable serves as temporary data by assigning a separate storage to each thread in the parallel execution. This resolves many data-dependencies for the target variable if all loop iterations use the same storage. Privatization has a strong impact on the performance obtained by loop parallelization, since it reduces the number of

<sup>1</sup> <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf#page=210>.

<sup>2</sup> <http://www.cs.umd.edu/projects/omega/>.

accesses to shared memory by each thread. For example, it is very common that at each iteration of a loop, a variable is initialized and then is used by other statements. By privatizing this variable, a local copy of the variable is provided for each thread and the loop becomes parallelizable.

---

**ALGORITHM 1:** Pseudo-code for determining scalar variable scoping

---

```

Input : forLoopStmt
Output: privateVarList, firstPrivateVarList, lastPrivateVarList
1 begin
2   privateVarList = {}; firstPrivateVarList = {}; lastPrivateVarList = {};
3   varUseList = forLoopStmt.varUses();
4   ▷ Add loop control variables to private list
5   privateVarList.push(forLoopStmt.getAllControlVars());
6   forall varUse in varUseList do
7     if varUse.declaration == "inside loop" then
8       | continue;
9     end
10    if varUse.Type == "pointer" then
11      | break;
12    end
13    if varUse.use == 'R' then
14      | firstPrivateVarList.push(varUse.name); continue;
15    end
16    if varUse.nextUse != 'R' && (varUse.use == 'WR' || varUse.use ==
17      'W') then
18      | privateVarList.push(varUse.name); continue;
19    end
20    if varUse.nextUse == 'R' && varUse.usecount('W') > 0 &&
21      varUse.usecount('RW') == 0 then
22      | lastPrivateVarList.push(varUse.name); continue;
23    end
24  end

```

---

AutoPar-Clava implements a simple but effective variable privatizer approach, shown in Algorithm 1 and Algorithm 2, for scalar variables and array variables, respectively.

For scalar variables, Algorithm 1 starts by reading, from the loop statement, all variables information, on line 3. Control loop variables are made private on line 4. The *usage pattern* of each variable is processed in order to decide which OpenMP scoping should be considered. If the *usage pattern* is only R, it can be set as a *firstprivate* variable (line 12), and if the *usage pattern* equals to WR or W and the usage after the loop (*nextUse*) is not a R, then it is a loop variable and it can be categorized as a *private* variable (line 15). Similarly, if a scalar variable *usage pattern* is W and the *nextUse* attribute is R, then it can be categorized as *lastprivate* variable (line 18).

For array variables, Algorithm 2 starts by reading, from the loop statement, the information of the arrays used inside the loop, on line 3. Then, for each array, the *usage pattern* and results of the dependency analysis are used to decide the OpenMP scoping. If there is no writes ( $\bar{W}$ ) to an array, it is added to the *firstprivate* list, on lines 5–8. Otherwise, from line 9 to 22, the dependency objects, returned from the dependency analysis, are processed in order to identify the arrays that can be made *firstprivate*. Line 10 identifies if the dependency statement refers the



array under analysis (`arrayUse.name`). If so, line 13 verifies if the dependency cannot be solved or ignored. If false, line 16 verifies if the array is simultaneously dependent of a inner loop, not dependent of the current loop and dependent of an outer loop. If false, on lines 19–20, the array is added to the `firstprivate` list if its *usage pattern* is WR and if it has no dependency with the current loop.

According to OpenMP 4.5 reference manual [31], an array variable cannot appear in a `private` clause, otherwise the allocated memory would not be accessible inside the threads and the private pointer would have an invalid address. Therefore, all array variables that do not have data-dependencies among different iterations of a loop are added to the `firstprivate` variable list, as it is the case for the array variables `a`, `z` and `c` for both loops in the code of Fig. 1. Additionally, if there are no access dependencies on a shared array, a `firstprivate` scoping makes a copy of the array address to each thread so that no overhead results when a set of threads access the same array in a different range of indexes.

---

#### ALGORITHM 2: Pseudo-code for determining array variable scoping

---

```

Input : forLoopStmt
Output: firstPrivateArrayList

1 begin
2   firstPrivateArrayList = {};
3   ArrayUseList = forLoopStmt.arrayUses();
4   forall arrayUse in ArrayUseList do
5     if arrayUse.count('W') == 0 then
6       firstPrivateArrayList.push(arrayUse.name);
7       continue;
8     end
9     forall depObj of forLoopStmt.PetitDependencies do
10      if depObj.arrayName != arrayUse.name then
11        continue;
12      end
13      if depObj.canBeIgnored || depObj.cannotBeSolved then
14        continue;
15      end
16      if (depObj.isDependentInnerLoop &&
17         !depObj.isDependentCurrentLoop &&
18         depObj.isDependentOuterLoop) then
19        continue;
20      end
21      if (arrayUse == 'WR' && !depObj.isDependentCurrentLoop) then
22        firstPrivateArrayList.push(arrayUse.name);
23      end
24 end

```

---

### 3.2.3 Scalar and array reductions

In some cases where privatization is not possible, a *reduction* operation (e.g., sum, product, or other commutative and associative operations over variables) may enable parallelization by computing a partial result locally by each thread and updating a global result only upon completion of the loop by a reduction operation. Algorithm 3 shows our approach for reductions over scalar and array variables.

Reduction variables are those with read and write access in different iterations, which causes a data-dependency to be reported by the dependency analyzer. Generally, they are in the form of `var op= expr` or `var= var op expr` where the *reduction-identifier* (i.e., `op`) can be one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&`, and `||` followed by assignment operator (i.e., `=`).

For scalar variables, our analysis detects reduction variables and its associated operator that satisfy the above criteria and excludes the detected reduction variables and its dependencies from the loop-carried dependencies. For example, the *usage pattern* of variable `d` within the inner loop, of the C code in Fig. 1, is `RW` and follows the OpenMP reduction form for scalar variables. Therefore, in the output generated code, variable `d` appears into a reduction clause of the annotated `#pragma` for the inner loop. The same classification for variable `d` cannot be used for the outer loop since its *usage pattern* is `WRW`.

Reductions on array variables are a potential source of significant improvements of parallelization performance. Although following the OpenMP reduction criteria for array variables would need a more complex analysis, as we focus on loops where all array subscript expressions are affine functions of the enclosed loop indices and loop-invariant variables, the analysis is simplified. In order to apply the above criteria for the array variables in the C code of Fig. 1, each access within the target loop body is considered as a scalar variable in the detection procedure.

---

**ALGORITHM 3:** Pseudo-code for detecting reduction operation for a given scalar or array variable

---

```

Input : candidate Variable, UsagePattern, Statement
Output: reduction operation for the Statement or null

1 begin
2   if (UsagePattern.count("RW") != 1) then
3     | return null
4   end
5   Check if all occurrences of candidate Variable have the same dependency for
   the current, inner and outer loops. Otherwise return null.
   ▷ Determine the reduction operation
6   candidateVarOp = {}
7   VariableRefs = Statement.references(Variable);
8   forall Variable_Ref over VariableRefs do
9     | Operation op = Variable_Ref.ancestor('operation');
   ▷ Rejects pattern 'x = expr - x'
10    | if op.kind == "subtraction" && op.right == Variable then
11      | candidateVarOp = {}
12      | break
13    | end
14    | candidateVarOp.push(op.kind);
15  end
   ▷ Accepts operation only if it is the only one associated with the
   variable or if the second operation is an assign (i.e., "=" or
   "op=")
16 if candidateVarOp.size == 1 then
17   | return candidateVarOp[0]
18 end
19 else if candidateVarOp.size == 2 && candidateVarOp[1] == 'assign' then
20   | return candidateVarOp[0]
21 end
22 else
23   | return null
24 end
25 end

```

---

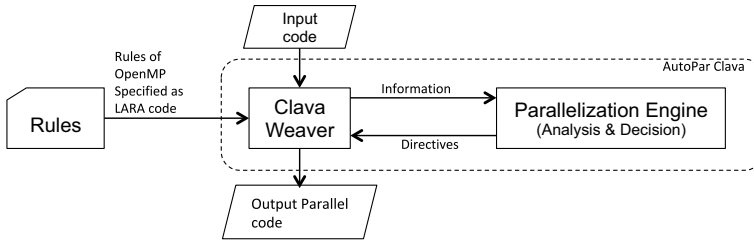
Therefore, having similar memory accesses by means of array subscripts for both source and destination variables, in the target dependency relation, is an extra initial condition for array reduction recognition. For illustrating the process of array variables detection within loops, consider the code in Fig. 1. Among all array variables accesses, only two array variables  $b$  and  $r$  have *write* memory reference in their own access pattern. For the outermost loop, the data-dependency analysis detects output dependency relations for the two array variables  $b$  and  $r$ , which cannot be solved by a preliminary privatization process (their subscript expression is not affine functions of the loop indices or loop-invariant variables). For the outermost loop at line 2, since the array access  $r[index]$  is not a function of the enclosed loop index (i.e., variable  $j$ ) or loop-invariant variables, it can be treated similarly to a scalar variable (i.e., all loop iterations update the same element  $r[index]$ ). In this case, as it meets the general reduction form "`var op= expr`" with acceptable OpenMP operator `+` for reduction clauses and does not appear elsewhere in the loop body, it is classified as a reduction clause for the outermost loop at line 2 (i.e., `reduction(+: r[index])`). For array access  $b[i]$ , by applying the induction variable procedure, the subscript expression  $b[i]$  is converted to  $b[k+1]$  which is a function of the inner loop iterator (i.e., variable  $k$ ). From the outer loop point of view, the array variable  $b$  within element range of  $[0, \dots, M]$  is updated at each individual iteration. Since the update statement at line 8 (i.e.,  $b[i] += a[k]$ ) satisfies the criteria of OpenMP reductions, it can be classified as an array reduction variable. However, unlike the reduction on scalar variables, for reduction on array variables, one must specify the lower and upper bound for each dimension of the target array (array  $b$  in this example). Therefore, if the array size is obtained by static analysis, a reduction operation is identified, otherwise, the loop is marked as non-parallelizable.

For the inner loop at line 5, in Fig. 1, the subscript expression of array access  $b[k+1]$ , converted by induction variable substitution from original statement  $b[i]$ , is a function of the inner loop, each loop iteration  $k$  updates individual array elements  $b$  and there is no data-dependency issue due to the access of array  $b$ .

Concerning the implementation shown in Algorithm 3, it receives as input, the candidate variable, its usage pattern and the loop statement. On line 2, if the pattern `RW` appears more than once, then a reduction is not applicable and the process finishes. Otherwise, on line 5, it is verified if the candidate variable has the same dependency in all references made inside the current loop. If true, the process continues and on lines 8 to 15, it is registered all operations where the candidate variable appears. From line 16 to 21, it is selected an operation only if it is the only one associated with the candidate variable, or if the second operation is an assignment, i.e., the pattern `"="` or `"op="`.

### 3.3 Code parallelization

Figure 3 shows the components of the `AutoPar-Clava` framework. The Clava weaver applies generic rules, written in LARA, over the input source code, call the parallelization engine and, finally, generates the output parallel code, where each for-type loop statement is annotated either with an OpenMP pragma or with a comment



**Fig. 3** AutoPar-Clava framework

explaining why the loop could not be parallelized. The parallelization engine module is the main core of *AutoPar-Clava* to analyze and decide which loops are parallelized and which pragmas to include in the output code.

One of the most important features to trace access patterns for each variable used inside of a candidate loop body is the capability to deal with function calls in the loop body. However, most static auto-parallelization frameworks do not include inter-procedural analysis and thus do not consider loops for parallelization when they contain user's function calls. In other cases, function inlining needs to be performed before parallelization, or the functions need to be identified as a non-side-effect functions. Although function inlining may enable loop parallelization, its use may not be adequate as it may decrease performance, e.g., due to register pressure.

Avoiding loops because of function calls or only considering them when function inlining is previously performed or when dealing with functions previously marked without data-dependencies that prevent parallelization may miss important performance improvement opportunities. *AutoPar-Clava* overcomes this problem by performing temporary function inlining, whenever possible during the analysis phase, to find a complete access pattern for each variable, even inside the called functions. This gives the compiler the opportunity to analyze the called function without inter-procedural analysis which is not yet supported. It also allows the compiler to identify statements in the functions (using the same analysis performed to the loop body to mark loops as non-parallelizable) that may also prevent the loop to be parallelized. Note that the function inlining is used by *AutoPar-Clava* at the analysis phase, and all changes in the code due to inlining are discarded in the generated output code unless a LARA strategy instructs the compiler to inline the function (we note however that for the code generated and evaluated in this paper we do not include function inlining in our code optimization strategies). The current function inlining implementation does not support functions with multiple exit points, recursive functions and calls to functions whose source code is not available in the input source files. Additionally, in order to not miss loop parallelization opportunities due to library function calls, *AutoPar-Clava* uses a simple reference list, that can be modified by users, which contains functions that are known to not modify their input variables or that do not have side effects on I/O functionality (e.g., `sqrt`, `sin`).

Algorithm 4 presents the pseudo-code of the overall strategy for the parallelization engine, which is implemented in the form of a LARA strategy. It receives a loop

to analyze and verifies if it satisfies the OpenMP canonical form. Then, from lines 5 to 9, it is verified if there are function calls and if they can be either in-lined or are known safe functions calls. At line 10, it is processed the induction variable procedure which replaces the induction variable if the following conditions are satisfied: (a) it is not in the loop header; (b) it is not in an `if` statement; (c) the right side of the "=" operator does not have any reference to arrays; (d) it has no function calls; (e) it has no unitary operators; and (f) it has no binary shift operations. From line 11 to 13, it is determined the usage pattern of each variable in the loop statement by performing data flow analysis. At line 14, it is invoked the dependency analyser, using Petit [19] tool. If there is no unsolved dependencies, at line 18 and 19, it is defined the scoping for each variable and added the OpenMP directive to the loop.

---

#### ALGORITHM 4: AutoPar-Clava parallelization engine

---

```

Input : LoopStmt (candidated loop), VariableList
Output: Parallelized loop annotated by OpenMP directives

1 begin
2   if loopStmt is NOT in OpenMP canonical loop form then
3     | return // skip parallelization process
4   end
5   forall callStmt(function call) within LoopStmt body do
6     | if (callStmt  $\notin$  SafeFuncCallStmts) & (Clava-inline Not successful) then
7       | return // skip parallelization process
8     | end
9   end
10  Apply induction variables procedure
11  foreach variable within VariableList do
12    | Find the usage pattern by performing dataflow analysis
13  end
14  Call dependency analyzer for all scalar and array variables
15  if unsolved dependencies  $\neq \emptyset$  then
16    | return // skip parallelization process
17  end
18  Categorize all variables into proper OpenMP scoping according to their usage
19  pattern
20  Add OpenMP directives to the LoopStmt
21  return
22 end

```

---

### 3.4 Generation of the output code

As the last step, after determining the loops that can be parallelized, the output code is generated. The Clava AST contains the necessary information to reconstruct the source-code, including text elements such as comments and pragmas determined by Algorithm 4. Clava separates implementation files (e.g., .c) from header files (e.g., .h) and is able to reproduce them from the AST.

Figure 4 shows the LARA script for adding OpenMP `#pragma` directives into the target input loop statements (i.e., `loopStmt`), represented as part of the Rules block in Fig. 3.

As part of the flexibility offered by the Clava source-to-source compiler, one can instruct the compiler to add OpenMP pragmas for all the loops that were detected as being parallelizable, customize which loops should be annotated with pragmas (e.g.,

```

1  aspectdef AddloopOpenMPDirectives
2  input
3      $loopStmt, privateVars, firstprivateVars, lastprivateVars, reduction
4  end
5  ...
6  var OpenMPstmt = '#pragma omp parallel for ';
7  if (privateVars.length > 0)
8      OpenMPstmt += 'private(' + privateVars + ') ';
9  if (firstprivateVars.length > 0)
10     OpenMPstmt += 'firstprivateVars(' + firstprivateVars + ') ';
11  if (lastprivateVars.length > 0)
12     OpenMPstmt += 'lastprivateVars(' + lastprivateVars + ') ';
13  if (reduction.length > 0)
14     OpenMPstmt += reduction;
15  ...
16  /* Insert OpenMP directive string before loopStmt */
17  $loopStmt.insert before OpenMPstmt
18  ...
19  end

```

**Fig. 4** An example of a LARA aspect to *insert* an OpenMP directive before the loop statement and considers the inclusion of different clauses according to code properties

if they are not inside a parallel loop, if they are detected as a hot-spot after a profiling step), and customize the generated pragmas (e.g., remove clauses related to reduction of arrays). LARA scripts provide enough flexibility in order to allow users to try and evaluate their own OpenMP-based parallelization strategies and/or develop application-/domain-specific parallelization strategies.

## 4 Experimental methodology

In this section, we provide details about the target platform, experimental methodology, comparison metrics and benchmarks used throughout the evaluation.

### 4.1 Platforms

The evaluation was performed on a Desktop with two Intel Xeon E5-2630 v3 CPUs running at 2.40 GHz and with 128 GB of RAM, using Ubuntu 16.04 x64-bits as operating system. To reduce variability in the results, the Turbo mode and the NUMA feature are disabled. Also, `OMP_PLACES` and `OMP_PROC_BIND` are set to `cores` and `close`, respectively.

### 4.2 Benchmarks

The Polyhedral/C 4.2<sup>3</sup> Benchmark Suite [36] and the NAS Parallel Benchmarks (NPB)<sup>4</sup> are used for performance evaluation in this work. The Polyhedral/C 4.2 contains many patterns commonly targeted by parallelizing compilers, with three different dataset sizes, namely MEDIUM, LARGE and EXTRALARGE. The NAS Parallel

<sup>3</sup> <https://sourceforge.net/projects/polybench/>.

<sup>4</sup> <https://www.nas.nasa.gov/publications/npb.html>.

Benchmarks have available both sequential and manually parallelized OpenMP versions. Three input classes are used, namely *W*, *A* and *B*, being class *W* the smaller input, class *B* the largest one, and class *A* the medium size input for a single machine.

### 4.3 Compared compilers and configurations

Taking into account that AutoPar-Clava performs automatic static parallelization over unmodified source-code, among all automatic parallelization approaches presented on Sect. 6, the ones closest to our objective are ROSE [38], Cetus [17] and TRACO [32]. As part of the ROSE compiler, autoPar [25] can automatically insert OpenMP pragmas in C/C++ code. The tools used in our experiments are: (a) autoPar version 0.9.9.199; (b) TRACO<sup>5</sup>; and (c) Cetus version 1.4.4.<sup>6</sup> The Intel Compiler `icc`, a well-known commercial solution, is used in the version 18.0.0 free academic license. Polyhedral compilers such as PLUTO [13], which applies loop transformations (e.g., tiling and loop fusion), are not considered in this section as our tool does not support such transformation in the current version. Our plans for future work include code transformations in parallelization strategies. However, to show the potential of AutoPar-Clava, a comparison with Pluto is shown for the Polybench benchmarks in Sect. 5.1.4. Also, other well-known approaches such as SUIF [44] and POLARIS [10, 41], address the parallelizing Fortran77 code and are therefore not considered here.

For both the original serial code and the parallel OpenMP versions (i.e., parallelized with AutoPar-Clava, `icc`, Cetus, TRACO and ROSE), we use `icc` to compile the target C code, using `-O2`. The optimization flag `-O2` is used instead of `-O3` because: (i) `-O2` is the generally recommended optimization level by Intel<sup>7</sup>; and (ii) to be able to do a fair comparison between serial code and parallel code annotated with OpenMP pragmas, since we detected that in some cases, using the flag `-qopenmp` in serial code without OpenMP pragmas slows down the performance of code compiled with `-O3` to the same level as `-O2`.<sup>8</sup>

For each benchmark, each experiment was repeated 30 times and the average of execution time was used. For each data size, we have run the programs with 4, 8 and 16 threads. In the case of the Polybench Benchmarks, we verified the output of all generated parallelized versions from each tool, by using the flag `-POLYBENCH_DUMP_ARRAYS` that dumps all live-out arrays to the `stderr` and comparing it with the equivalent output of the sequential versions, and only parallel implementations with similar results are reported.

Note that, only `icc`, AutoPar-Clava and ROSE compilers are able to compile the original source files without imposing any limitations on the source code. Therefore, to provide a compatible input file for TRACO and Cetus, we modified the input code according to the restrictions of each of these compilers.

<sup>5</sup> <https://sourceforge.net/projects/traco>.

<sup>6</sup> <https://engineering.purdue.edu/Cetus>.

<sup>7</sup> <https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler>.

<sup>8</sup> <https://software.intel.com/en-us/forums/intel-c-compiler/topic/755677>.

To generate the parallelized versions by `icc`, we use the flag `-parallel`. `icc` uses a cost model with a threshold parameter to decide whether to parallelize a loop. The `threshold[n]` compiler option adjusts this parameter.<sup>9</sup> The value of `n` ranges from 0 to 100, where 0 means to always parallelize a safe loop, irrespective of the cost model, and 100 tells the compiler to only parallelize those loops for which a performance gain is highly probable. The default value of `n` is conservatively set to 100. In this study, the values 0 and 100 are considered.

#### 4.4 Evaluation metrics

The evaluation is focused on the speedup obtained with the generated parallel code, which is defined as the ratio between the execution time of the sequential code and the execution time of the parallelized version. Additionally, in order to evaluate the ability of each compiler to detect parallelism, the number and type (i.e., inner or outer) of parallelized loops in the generated output code are reported.

## 5 Experimental evaluation

This section presents and discusses results obtained for Polybench and NAS benchmarks. For simplicity, we refer to `autoPar` tool in ROSE compiler as `ROSE` in all figures and tables. Also, `icc-par-threshold[0]` and `icc-par-threshold[100]` represent the results of `icc` compiler when the `-par-threshold` parameter is set to 0 and 100, respectively.

### 5.1 Polybench benchmarks

Before presenting results for the Polybench benchmarks, we discuss the performance impact of the proper OpenMP scoping, in particular concerning array variables, which leads to significant performance improvements.

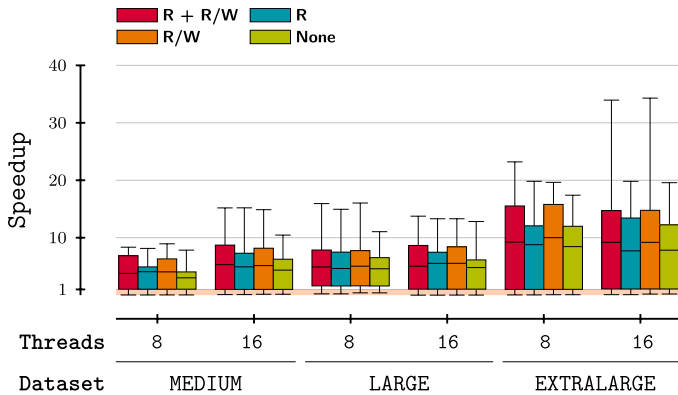
#### 5.1.1 OpenMP scoping

In order to demonstrate the effectiveness of proper OpenMP scoping for array variables (see Sect. 3.2), Fig. 5 compares four different possible types of array scoping, tested in `AutoPar-Clava`. Based on the usage pattern, an array access can be classified into: (i) *read* only pattern (R); and (ii) *write* or *readwrite* pattern (R/w). In both classes (R) and (R/W) for accessing arrays, they are categorized with the `firstprivate` clause instead of `shared` (see Sect. 3.2.2). Note that

---

<sup>9</sup> <https://software.intel.com/en-us/node/522957>.





**Fig. 5** The effectiveness of proper OpenMP scoping for array variables

AutoPar-Clava applies the OpenMP scoping *if and only if* no data dependencies, data conflicts and race conditions within the target loop are found.

Figure 5 presents the SpeedUp geometric mean obtained among all polybench benchmarks when array variables are made `firstprivate`, for different access patterns. For both access patterns `R + R/W` and `R/W`, AutoPar-Clava achieved higher performances than making `firstprivate` only read array variables (i.e., `R`) or without any privatization of array variables (i.e., `None`). Hence, in our proposed approach, we analyze access patterns of arrays to select the scoping that maximizes performance. As mentioned before, to have a fair comparison with the parallelized code generated by `icc`, we use `icc` to compile the annotated OpenMP output code generated by each tool in our experiments. Some algorithms presented super-linear speedups due to the limitation of one thread in managing efficiently a huge amount of data. However, the geometric mean is below 10, for 16 threads, for all cases. Based on our observations, using `gcc`, the performance of different strategies for array variable scoping does not present relevant performance differences among them, which indicates that `icc` performs some additional memory management for array variables according to their OpenMP scoping.

### 5.1.2 Polybench results

Table 1 shows the number of total loops parallelized by each auto-parallelization tool, as well as the type of the loops detected, i.e., inner or outer loop. At the moment, when a nested loop is identified to be parallelized, only the outermost one is marked to be parallelized. It is, however, easy to include a decision rule based on the number of iterations of the loop.

Figures 6 and 7 show the speedups of the parallelized versions of the PolyBench benchmarks, relative to the sequential versions, for each compared tool. Due to space limitations, we only present benchmarks with significant difference in terms of speedup, whereas for other benchmarks the improvements are similar for all compared tools. However, full details of the number of detected loops for all benchmarks

**Table 1** The number and type of loop parallelization generated by each compared tools

Benchmark name	TRACO		Cetus		ROSE		AutoPar- Clava	
	Outer	Inner	Outer	Inner	Outer	Inner	Outer	Inner
2mm	2	–	2	–	2	–	2	–
3mm	3	–	3	–	3	–	3	–
adi	–	2	–	2	–	2	–	2
atax	1	1	▲		1	1	2	–
bicg	1	–	▲		1	–	2	–
cholesky	✘		–		–		–	
correlation	4	–	4	–	2	–	4	–
covariance	3	–	3	–	3	–	3	–
deriche	6	–	6	–	6	–	6	–
doitgen	–	2	▲		–	2	1	–
durbin	–	2	–	3	–	3	–	3
gemm	1	–	1	–	1	–	1	–
gemver	4	–	4	–	4	–	4	–
gesummv	1	–	1	–	1	–	1	–
gramschmidt	✘		–	3	–	3	–	3
heat-3d	–	2	–	2	–	2	–	2
jacobi-1d	–	2	–	2	–	2	–	2
jacobi-2d	–	2	–	2	–	2	–	2
lu	–	1	–	1	–	–	–	–
ludcmp	–		–	4	–	4	–	4
mvt	2	–	2	–	2	–	2	–
seidel-2d	–		–		–		–	
symm	☞		–	1	–	1	–	1
syr2k	1	–	1	–	1	–	1	–
syrk	1	–	1	–	1	–	1	–
trisolv	–		–		–		–	
trmm	–	1	–	1	–	1	–	1
Total	30	15	28	21	28	23	33	20

▲: The functionality of the parallelized version is NOT similar to the sequential one.

✘: No output file generated by the tool

☞: the code is changed and modified by the tool

are presented in Table 1. Below, we discuss the results achieved for each individual benchmark.

2mm and 3mm: both benchmarks perform matrix multiplications with 2 and 3 individual nested loops, respectively. The outermost one at each nested loops is marked and parallelized by all auto-parallelization tools. However, as shown in Fig. 6a, b the performance improvements are not similar. The reason can be explained by the variable classification into the proper OpenMP scoping per-

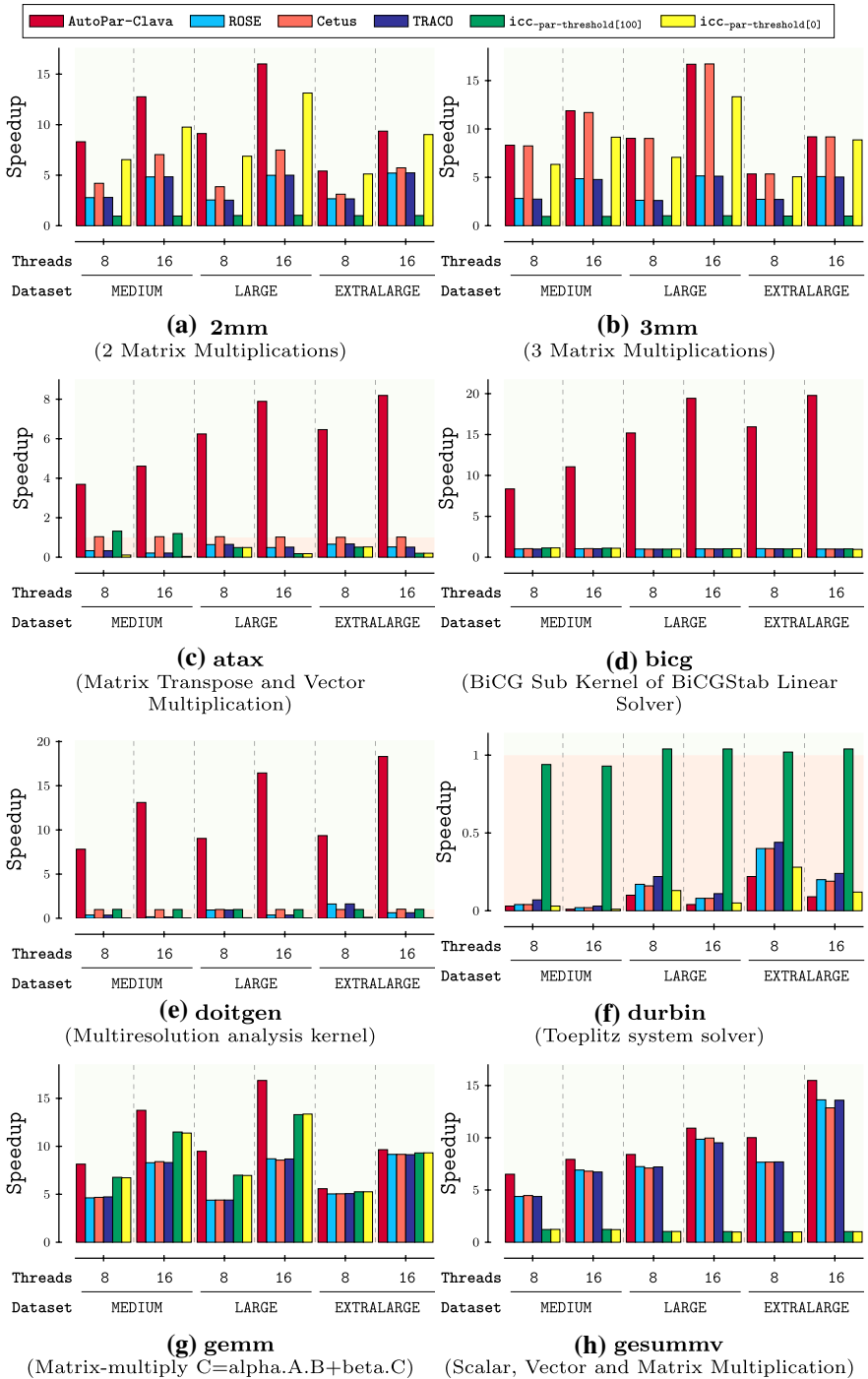


Fig. 6 Geomean speedups for PolyBench benchmarks (part 1)

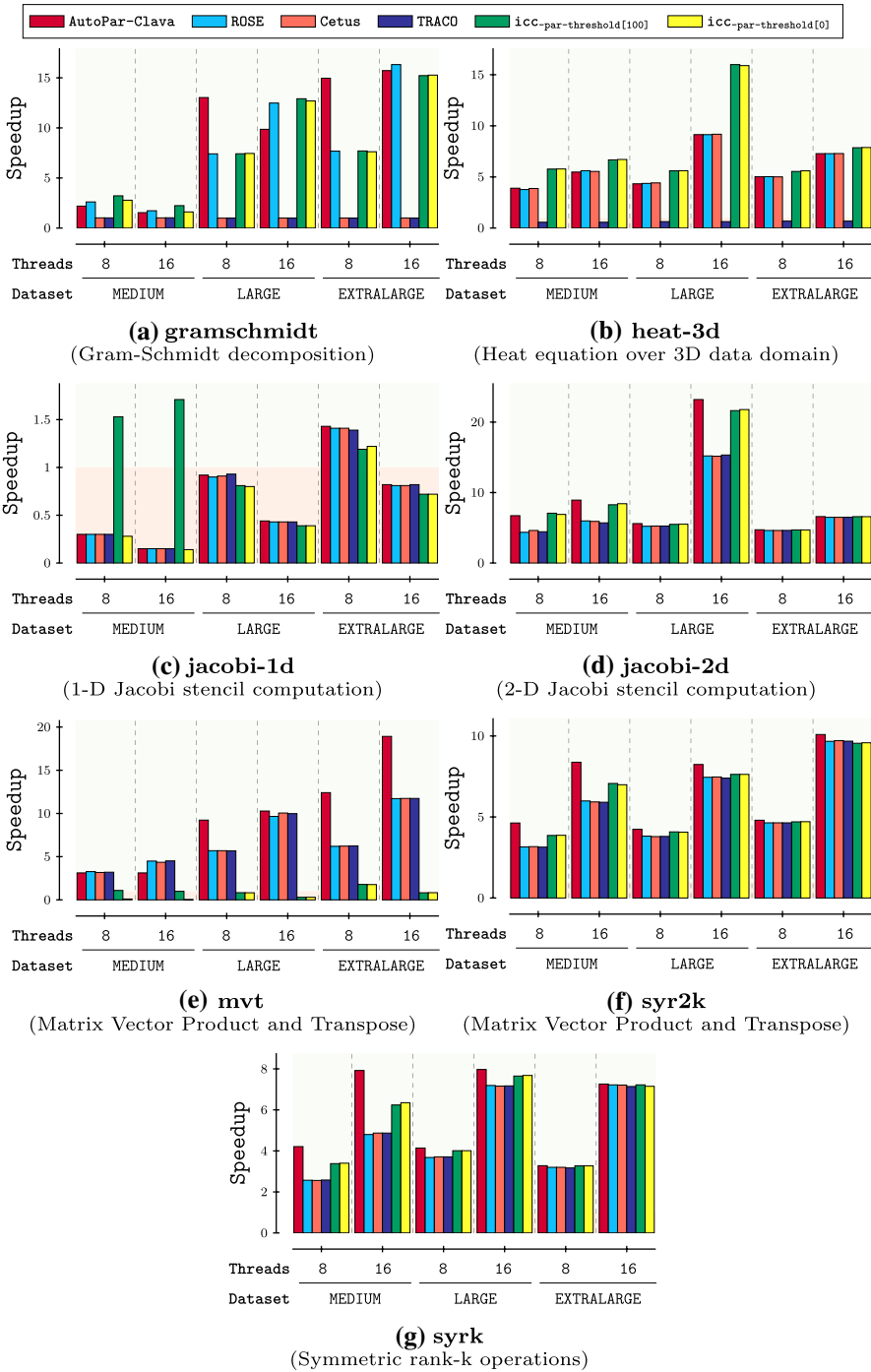


Fig. 7 Geomean speedups for PolyBench benchmarks (part 2)

formed by each tool. Among all compared tools, TRACO just finds parallelizable loops and inserts the OpenMP `parallel for` pragma without any variable scoping. In the case of the ROSE autoPar tool, it only supports variable scoping for scalar variables. Cetus compiler applies more analysis on used variables, for both scalar and array variable types. By comparing the generated OpenMP directives provided by each tool, our proposed tool has a wide range of variable scoping. To illustrate this feature, Fig. 8 shows the parallelized output code for the first nested loop of the kernel function in *3mm*, for all compilers.

As shown in Fig. 8, AutoPar-Clava presents more variable scoping for scalar variables than the other compilers which only focus on privatization of loop indexes. In addition to scalar variable scoping, our approach categorizes each array variable based on its usage pattern. For instance, since both arrays A and B have *read* only pattern access inside of the outermost loop, they are categorized as `firstprivate` in the clause variable list (see Sect. 3.2.2). In terms of performance, AutoPar-Clava and Cetus show the best improvement for both *2mm* and *3mm*. However, in the case of *3mm*, based on our observations, Cetus achieved that improvement by classifying modified variable array E (see code in Fig. 8) as a `firstprivate` and `lastprivate` variable. `icc` achieved a SpeedUp of 1 for both *2mm* and *3mm*, as no loop was parallelized by this compiler, when the cost model is applied. Without cost model, the performance of `icc` is higher than 1, but lower than the performance achieved by AutoPar-Clava.

*atax* and *bicg*: both *atax* and *bicg* contain 2 individual nested loops. In both cases, the first outermost loop does a simple initialization of an array, and the second loop, which consumes more time, computes the mathematical opera-

```

1 void kernel_3mm(...)
2 {
3   ...
4   // ROSE autoPar:
5   #pragma omp parallel for private(i, j, k)
6   //TRACO :
7   #pragma omp parallel for private(i)
8   //Cetus :
9   #pragma omp parallel for private(i, j, k) firstprivate(E)
   ↪ lastprivate(E)
10  // AutoPar-Clava
11  #pragma omp parallel for private(i, j, k) firstprivate(A,
   ↪ B, E, ni, nj, nk)
12  for (i = 0; i < ni; i++)
13    for (j = 0; j < nj; j++)
14    {
15      E[i][j] = 0.0;
16      for (k = 0; k < nk; ++k)
17        E[i][j] += A[i][k] * B[k][j];
18    }
19    ...
20 }

```

**Fig. 8** Annotated OpenMP C code for *kernel\_3mm* considering ROSE, TRACO, Cetus and AutoPar-Clava

tions. Both ROSE and TRACO compilers parallelized the same loop. Since both approaches only parallelized the first nested loop in *bicg* and could not parallelized the second loop, which has higher influence on execution time improvement, the obtained SpeedUp is near to 1 (i.e., similar execution time to sequential code), as shown in Fig. 6d. For *atax* in Fig. 6c, both ROSE and TRACO approaches show performance slowdowns. This can be explained as parallelizing the inner loop from the second nested loop causes a higher time overhead of threads starting and releasing at each iteration. In the case of Cetus, the input source code is modified by transforming of the loop structure, but the parallelized version of the code does not have correct functionality.

Since our proposed approach supports the OpenMP array reduction feature, for both benchmarks, AutoPar-Clava could obtain significant SpeedUp of 10× and 25× for *atax* and *bicg*, respectively. The parallelized output of our approach for *atax* is presented in Fig. 9. AutoPar-Clava could parallelized the second outer loop (line 12) due to solving data dependency for the array access *y* at line 19 by classifying it as an array reduction variable (i.e., `reduction(+:y[:2100])`). However, other approaches could not support this feature and only parallelize the inner loop at line 18 which causes the degradation of performance as shown in Fig. 6c.

Doitgen and durbin: for *Doitgen*, the best SpeedUp performance is obtained by AutoPar-Clava with a maximum improvement of 19. Both ROSE and TRACO parallelized the exact similar inner loops in their generated output code and did not obtain any performance improvement. The only exception is the parallelized output generated by Cetus, which did not pass the verification step as it is noted in Table 1. In the case of *durbin*, all compared auto-parallelization tools parallelized inner loops, resulting in a degradation of performance as shown in

```

1 void kernel_atax(int m, int n, double A[1900][2100], double x[2100],
  ↪ double y[2100], double tmp[1900])
2 {
3   ...
4   // ROSE : #pragma omp parallel for private (i)
5   // TRACO : #pragma omp parallel for private(i)
6   // Cetus : #pragma omp parallel for private(i)
7   #pragma omp parallel for private(i) firstprivate(y, n)
8   for(i = 0; i < n; i++) {
9     y[i] = 0;
10  }
11  #pragma omp parallel for private(i, j) firstprivate(A, tmp, x, m, n)
  ↪ reduction(+:y[:2100])
12  for (i = 0; i < m; i++) {
13    tmp[i] = 0.0;
14    for (j = 0; j < n; j++)
15      tmp[i] += A[i][j] * x[j];
16    // ROSE : #pragma omp parallel for private (j)
17    // TRACO : #pragma omp parallel for private (j)
18    for (j = 0; j < n; j++)
19      y[j] += A[i][j] * tmp[i];
20  }
21  ...
22 }

```

**Fig. 9** Annotated OpenMP C code for *kernel\_atax* considering ROSE, TRACO, Cetus and AutoPar-Clava

Fig. 6f. The SpeedUp obtained by `icc` with the `-par-threshold` parameter equal to 100, for both *Doitgen* and *durbin*, indicates no loop parallelization for both benchmarks. However, by setting `-par-threshold` to 0, i.e., loops get auto-parallelized always, regardless of computation work volume, similar to other compared approach, `icc-par-threshold[0]` indicates a degradation of performance as well.

*Gemm*, *gesummv* and *gramschmidt* (Fig. 7a: for all three benchmarks, `AutoPar-Clava`, `ROSE`, `Cetus` and `TRACO` marked the same loops for parallelization. However, due to the wide range of variable scoping (see Fig. 5), `AutoPar-Clava` obtains, in general, better performance when compared to the other tools. In the case of `icc`, the SpeedUp ranks the second place for *gemm* and *gramschmidt*, but the last place without performing any parallelization for *gesummv*.

*Heat-3d*, *jacobi-1d* and *jacobi-2d*: all tools, except `icc`, annotated the similar inner loop in their generated output code. There is a degradation of performance for *jacobi-1d* which can be explained by the amount of work (i.e., dataset size) performed by the inner loop. Since the parallelized inner loop has low computing work, the overhead of allocating and releasing threads has a significant influence on the execution time, as shown in Fig. 7c. In contrast, for *jacobi-2d* with higher amount of computation work, even by parallelizing an inner loop, all tools achieved better execution time, as shown in Fig. 7d, being `AutoPar-Clava` and `icc` the best ones with equivalent performances. For *heat-3d*, `ROSE` and `Cetus` show similar performance improvements as `AutoPar-Clava` being `icc` the tool with higher speedup.

*Mvt*, *syr2k* and *syrk*: for all these benchmarks, the outermost loop is detected and parallelized by all auto-parallelization tools. However, due to the wide range of variable scoping, our proposed approach shows better performance for different datasets and number of threads among all compared tools.

### 5.1.3 Average performance

Figure 10 presents a boxplot chart of SpeedUp as a function of the number of threads and dataset parameters among all PolyBench benchmarks. In addition to the geomean Speedup, the average Speedup is also indicated by an individual diamond symbol in each boxplot. We can see that `AutoPar-Clava` has the highest geomean SpeedUp with a wider dispersion in the distribution of the results. This confirms the results presented in Fig. 6 and shows the higher performances that can be achieved with `AutoPar-Clava` for automatic parallelization.

### 5.1.4 Comparison to the polyhedral approach

The polyhedral approach is used to solve data dependencies, to produce parallel versions of the sequential code, possibly including loop transformations. Pluto [13] is a fully automatic source-to-source compiler which uses the polyhedral

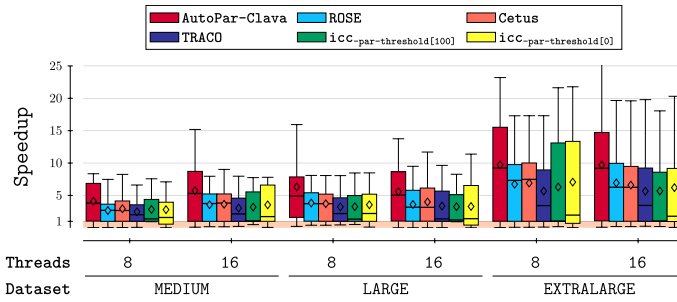


Fig. 10 Geomean speedups aggregated by dataset size and number of threads for all PolyBench benchmarks

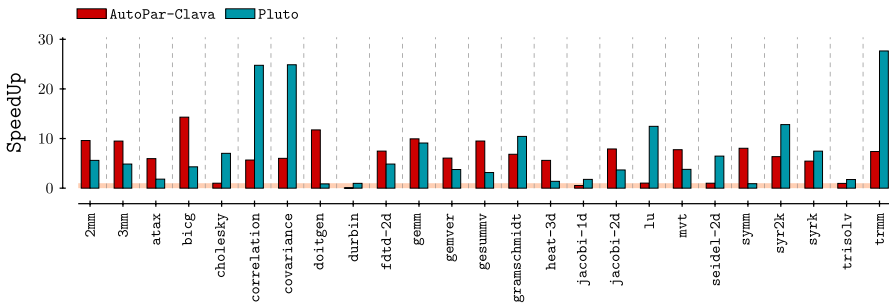


Fig. 11 Geomean speedups for PolyBench benchmarks achieved by AutoPar-Clava and Pluto

model for loop transformations. To evaluate the impact in performance of using loop transformations in the parallelization strategy, the proposed AutoPar-Clava approach is compared with Pluto. Figure 11 presents the obtained geomean speedups by these two approaches for all datasets and considering 8 and 16 threads. Among all benchmarks available, the parallelized output code generated by Pluto does not pass the verification step for *adi*, *deriche*, *ludcmp* and *nussinov*, which are excluded from Fig. 11.

As shown in Fig. 11, among all 150 combinations of the 25 benchmarks, 3 dataset sizes and 2 thread configurations, the proposed AutoPar-Clava approach could achieve higher performance for 70 cases with a geometric mean speedup of  $9.01\times$  against  $2.59\times$  of Pluto. In the same way, Pluto shows better improvements for 80 cases with geometric mean speedup of  $7.74\times$  against  $2.03\times$  of AutoPar-Clava.

The total average AutoPar-Clava improvements achieved for *2mm*, *3mm*, *gemm*, *gemver* and *gesummv* result from *fisrprivate* array scoping, 4, 6, 2, 9 and 3 arrays, respectively. For *atax*, *bicg* and *doitgen*, additional improvements are due to the identification of an array reduction, which allows to parallelize the outermost loop, resulting less overhead. In the case of *durbin*, the speedup is below 1 due to the characteristics of this benchmark. AutoPar-Clava parallelizes 3 inner loops, at the same level, that operate over 1-d arrays. The overhead of successive fork/joins



and the reduced work in each parallel loop result in low performance. At the current version, we do not have conditional parallelism in order to avoid this result. Note that in this benchmark, Pluto did not achieve any performance improvement.

The benchmarks *gramschmidt*, *heat-3d*, *jacobi-1d* and *jacobi-2d* have in common the fact that `AutoPar-Clava` parallelizes inner loops, being the worse results for the first and the third cases due to work over 1-d arrays.

For the benchmarks *cholesky*, *lu*, *seidel-2d* and *trisolv*, no parallelization is performed due to unsolved data dependencies, therefore achieving a speedup of 1.

The improvements for *symm* result from 2 *fisrprivate* arrays with the particularity of having 3 nested loops being the second loop parallelized. Pluto did not perform any parallelization for this case. The benchmark *syr2k* has more work than *syrk*, and the speedup is slightly higher as well. In these cases, the results of `AutoPar-Clava` are due to *fisrprivate* arrays, 2 and 1, respectively.

The cases where Pluto outperforms substantially `AutoPar-Clava`, namely *correlation*, *covariance* and *trmm*, are due to the fact that `AutoPar-Clava` performance results from *fisrprivate* arrays, while Pluto performs loop transformations and introduces also vectorization pragmas.

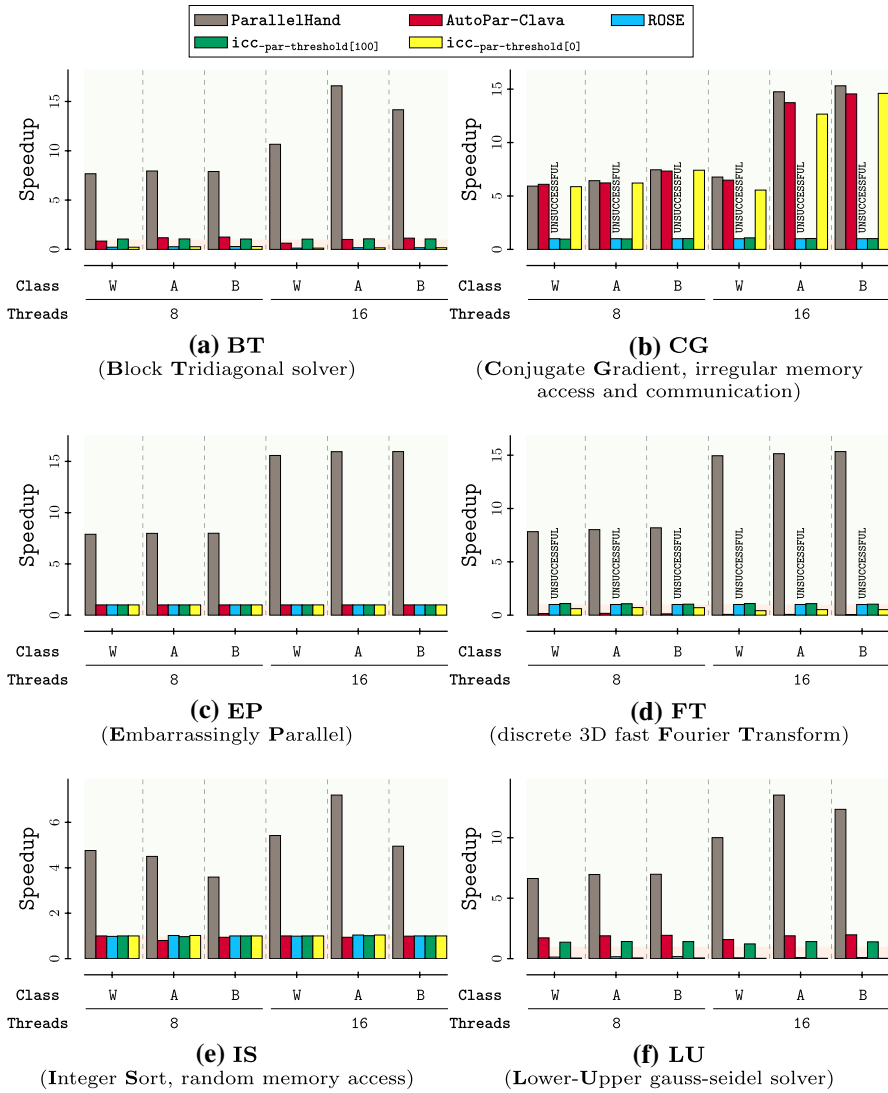
These results show that `AutoPar-Clava` can achieve good performances compared to Pluto and potential for improvement if polyhedral transformations are added.

## 5.2 NAS benchmarks

Figures 12 and 13 show the speedup of the parallelized versions over the sequential version for NAS benchmarks achieved by each approach. Since Cetus and TRACO could not parallelize any of the NAS examples (i.e., they failed during execution), they were excluded from the results.

The manually parallelized OpenMP versions achieve speedups between 2× and 15×, considering all programs and input sizes. The highest speedups were achieved with the *BT*, *CG*, *EP* and *FT* benchmark programs. This is not surprising, since these hand-parallelized versions have several source-code modifications besides the OpenMP pragmas, such as the usage of thread-local arrays (i.e., `threadprivate` clause) to save and collect temporary results.

The `icc` compiler, with both values 0 and 100 for the `-par-threshold` parameter, generally has very consistent performance for the programs *EP* and *IS*. However, by parallelizing loops when performance gains are predicted based on the `icc` compiler analysis data, i.e., `-par-threshold` equal to 100, we achieved higher performance, in the case of *BT*, *FT*, *MG*, *SP* and *UA*, compared to threshold level 0 where `icc` parallelizes all loops that it identifies as parallelizable. Particularly, only for *SP*, `icc-par-threshold[100]` demonstrates slightly speedUp improvement (maximum up to 2.14×), and for other benchmark programs (i.e., *BT*, *FT*, *MG* and *UA*) no parallelization is considered as shown in Figs. 12 and 13. The only two exceptions are *CG* and *UA* programs obtained by `icc-par-threshold[0]` that parallelizes all loops identified as parallelizable, regardless of computation work volume. For *CG* program, `icc-par-threshold[0]` obtained a significant improvement, as shown in Fig. 12b;



**Fig. 12** Geomean speedups of the NAS benchmarks for 8 and 16 threads, and W, A, B problem sizes (part 1)

the generated binary output file for *UA* program did not pass the benchmark validation phase, as shown in Fig. 13c.

The *autoPar* tool in ROSE compiler does not show significant performance improvements in most cases. Among nine programs, the parallelized code generated by *autoPar* tool did not pass the validation phase for *CG* and *FT*, and they are labeled as *UNSUCCESSFUL* in Figure 12b, d, respectively. In the case of *EP*, *IS* and *MG*, it does not demonstrate any performance improvement. Of the remaining four,

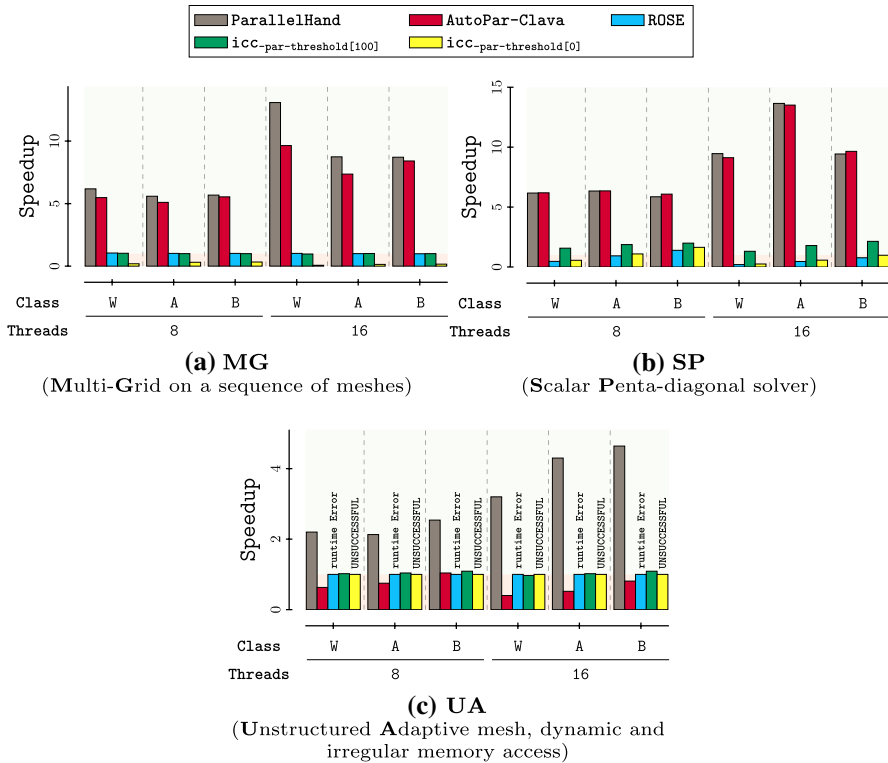
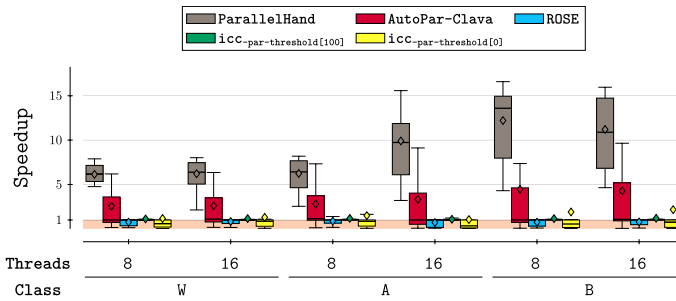


Fig. 13 Geomean speedups of the NAS benchmarks for 8 and 16 threads, and W, A, B problem sizes (part 2)

three programs, *BT*, *LU* and *SP*, show a degradation of performance. In the case of *UA* program, the generated code is compilable, but a *runtime Error* occurs during the program execution.

AutoPar-Clava was able to parallelize all the nine programs in NAS benchmark and generated a compilable parallelized version that *pass* the validation step. For two of the programs, *BT* and *EP*, it had performances close to *icc* and slightly better in the case of the *LU* program. However, there is a significant degradation of performance for *FT* program which is caused by two factors: (a) inner loop parallelization; and (b) lower amount of work performed by the parallelized loop in the generated output code. In contrast, for three programs, *CG*, *MG* and *SP*, AutoPar-Clava performed significantly better than *icc* and close to the parallel hand version which is mainly caused by inline functionality provided by our proposed tool that improves the accuracy of the analysis phase and increases loop parallelization opportunities for outermost for-type loops. By examining the output code generated by ROSE, and other approaches, we noticed that they do not consider loops for parallelization when the target loop body contains function calls.



**Fig. 14** Geomean speedups for NAS benchmarks, aggregated by 8 and 16 threads, and W, A, B problem sizes

One of the main shortcomings of the existing approach is the dependency analysis for array variables to which there is a high number of array accesses within the target loop, specially after function calls are inlined. As mentioned, in Sect. 3.2, loop dependencies are converted into dependency relations in the form of Presburger arithmetic and then analyzed by the Omega library. Therefore, a high number of dependency relations could not be handled due to Omega memory limitations to handle them.

To illustrate the results statistically, Fig. 14 presents a boxplot chart of the geomean speedups, obtained for NAS benchmark by each approach, as a function of the number of threads and Class size parameters. As it can be seen, AutoPar-Clava obtained a significant better performance compared to other auto-parallelization approaches. In conclusion, we improved the SpeedUp and also achieved higher values of performance with a wider dispersion in the distribution of the results in comparison to other auto-parallelization tools. Figure 14 also shows evidence of the large room for improvements, especially considering code transformations in parallelization strategies.

### 5.3 Statistical analysis

To determine the effect of AutoPar-Clava when compared to the other approaches, we conducted a two-tailed, paired samples,  $t$ -test (with  $\alpha = 0.05$ ) on the obtained speedups over the original benchmarks. We found a significant difference between the results obtained with AutoPar-Clava and the other tested approaches, meaning that the improvements observed are highly likely to be due to AutoPar-Clava, instead of happening by chance. The test was performed comparing our approach against all others, for each combination of number of threads and dataset size. The full results of the test can be found in our repository.<sup>10</sup>

<sup>10</sup> <https://github.com/specs-feup/specs-lara/tree/master/Publications/2019-SC>.

## 6 Related work

The efficient utilization of modern computer systems requires appropriate use of the computing cores available in current processors. Many efforts have been made in order to transform sequential code into scalable parallel versions, either automatically or by giving support to the programmer with the required transformations. From older efforts (e.g., [22]) toward automatic parallelization, to more recent ones (e.g., [27] which presented a system with cognitive properties in order to assist the programmers to avoid common OpenMP mistakes), there are still opportunities to improve the tools to generated parallel code automatically.

Automatic parallelization frameworks receive the source code as input and generate the parallelized version annotated by `#pragma` directives based on the target programming model (e.g., OpenMP) to express the potential parallelism of the parallelizable regions (e.g., loops) in the code or generate thread-based implementations using libraries to extend programs with threads (e.g., pthreads). As we mainly focus on a source-to-source compiler for automatic parallelization using a directive-driven programming model, i.e., OpenMP, we select and have a brief discussion on the automatic parallelization frameworks targeting OpenMP annotations. In addition, as our approach is fully based on static analysis, we mainly focus on parallelization frameworks which are not guided by runtime information acquired from program execution or by additional guidance provided by users. Thus, next we briefly discuss compilers that perform automatic loop parallelization by static analysis of the input source code.

The recent study in [40] provides important and interesting insights regarding the most well-known auto-parallelization frameworks. In the following paragraphs, we briefly summarize the ones more related to our approach and briefly compare them to `AutoPar-Clava`.

ROSE [38, 39] is an open source compiler, which provides source-to-source program transformations and analysis for C, C++ and Fortran applications. ROSE provides several optimizations, including auto-parallelization, loop unrolling, loop blocking, loop fusion and loop fission. As part of the ROSE compiler, `autoPar` [25] is the automatic parallelization tool used to generate OpenMP parallelized code versions from sequential code. For array accesses within loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables.

Pluto [13] is a fully automatic polyhedral source-to-source compiler. It translates C loop nests into an intermediate polyhedral representation called `CLooG` [8] (Chunky Loop Generator). With the `ClooG` format [9], the loop structure and its data-dependency and memory access pattern are kept, without its symbolic information. By using this model, Pluto is able to explicitly model loop tiling and to extract coarse-grained parallelism and locality, and finally to transform loops. However, it only works on well-behaved specific perfect nested loops that have to be marked in the source code using pragmas.

`icc` [18] supports a number of optimizations, such as variable privatization, loop distribution and permutation and includes an auto-parallelization engine that

automatically detects loops that can be safely and efficiently executed in parallel and generates a multi-threaded version of the input program.

Par4All [2] is an automatic parallelizing and optimizing compiler for C and Fortran and has backends for OpenMP, OpenCL and CUDA. The automatic transformation process is based on PIPS (Parallelization Infrastructure for Parallel Systems) [21], which is a framework for source-to-source program analysis, optimization and parallelization. Par4all does array privatization, reduction variable recognition and induction variable substitution. As far as we know, Par4all is not available online and that prevented us to include results using Par4all in our analysis.

Cetus [5, 17, 24] is a source-to-source compiler for ANSI C programs. Cetus uses static analyses such as scalar and array privatization, reduction variables recognition, symbolic data dependency testing and induction variable substitution. It uses the Banerjee–Wolfe inequalities [45] as a data dependency test framework and also contains the range test [11] as an alternative dependency test. Cetus provides auto-parallelization of loops through private and shared variable analysis and automatic insertion of OpenMP directives.

TRACO [32, 33] is a loop parallelization compiler, based on the iteration space slicing framework (ISSF) and on the Omega library. Loop dependency analysis is performed by means of the Petit [19] tool. TRACO accepts C/C++ code and performs coarse and fine-grained parallelism extraction using loop iteration space slicing, variable privatization and parallel reduction on scalar variables. The output code is completed with OpenMP directives.

Our parallelization strategy distinguishes from these approaches in the following aspects: (i) it produced functionally correct parallelized output code for all the evaluated benchmarks, similarly to the ROSE compiler (see Table 1); (ii) it considers a wider set of OpenMP scoping; (iii) it is not limited by the input code size, as occurred with TRACO and Cetus (see Table 1); (iv) besides reduction for scalar variables, it also supports reduction for arrays and at the array element level; and (v) it includes temporary function inlining in order to allow data-dependence analysis of loops with function calls. Additionally, as our framework provides users with libraries and a programming language (LARA) to easily query, analyze, extract source code information, such as variables, loop statements and function structures from the AST, and make actions to transform code, it can be easily modified and updated, even by non-compiler experts, in order to support more OpenMP features and extend/change its parallelization strategy.

## 7 Conclusion

This paper presented `AutoPar-Clava`, an automatic parallelization approach for the C source-to-source `Clava` compiler. The compiler is currently focused on parallelizing C programs by adding OpenMP directives (mainly `parallel-for`) and the necessary clauses.

The main contributions of our work include the data-dependency analysis on array reduction operations and on the support to induction variables, which increases the number of candidate parallel loops, leading to better performances than other

state-of-the-art compilers, as shown by the experimental results with a representative number of benchmarks. A second and highly relevant contribution, in comparison to other compilers, is its versatile mechanisms to evaluate and add new parallelization strategies, from the analysis of the programs being compiled to the selection and insertion of OpenMP directives and clauses.

The experiments provided show promising results and improvements regarding other auto-parallelization compilers when targeting a multicore x86-based platform. However, in some cases, `AutoPar-Clava` achieves a degradation of performance which is caused by two factors: (a) inner loop parallelization; and (b) lower amount of work performed by the parallelized loop in the generated output code. In both cases, making a decision if the parallelization of a loop is beneficial can be determined by runtime feedback from program execution or by additional guidance provided from the user.

As future work, we plan to extend the framework with additional parallelization strategies (e.g., to deal with task parallelism) and techniques to orchestrate the parallelization with code transformations provided by our source-to-source compiler (such as loop tiling and loop interchange). One possibility is to extend the framework with polyhedral model approaches. In addition, we intend to research a cost-based analysis for guiding decisions regarding parallelization.

**Acknowledgements** This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under Grant No. 671623. João Bispo acknowledges the support provided by Fundação para a Ciência e a Tecnologia, Portugal, under Post-Doctoral Grant SFRH/BPD/118211/2016.

## References

1. Acharya A, Bondhugula U, Cohen A (2018) Polyhedral auto-transformation with no integer linear programming. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp 529–542
2. Amini M, Creusillet B, Even S, Keryell R, Goubier O, Guelton S, McMahon J, Pasquier F, Péan G, Villalon P (2012) Par4all: from convex array regions to heterogeneous computing. In: IMPACT 2012: International Workshop on Polyhedral Compilation Techniques
3. Arabnejad H, Bispo J, Barbosa JG, Cardoso JMP (2018) An openmp based parallelization compiler for c applications. In: 2018 IEEE International Conference on Parallel Distributed Processing with Applications (ISPA), pp 915–923
4. Arabnejad H, Bispo J, Barbosa JG, Cardoso JMP (2018) Autopar-clava: an automatic parallelization source-to-source tool for c code applications. In: Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms. ACM, pp 13–19
5. Bae H, Mustafa D, Lee J-W, Lin H, Dave C, Eigenmann R, Midkiff SP (2013) The cetus source-to-source compiler infrastructure: overview and evaluation. International Journal of Parallel Programming, pp 1–15
6. Bagnères L, Zinenko O, Huot S, Bastoul C (2016) Opening polyhedral compiler's black box. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. ACM, pp 128–138
7. Banerjee U (2007) Loop transformations for restructuring compilers: the foundations. Springer, Berlin

8. Bastoul C, Cohen A, Girbal S, Sharma S, Temam O (2003) Putting polyhedral loop transformations to work. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer, pp 209–225
9. Bastoul C (2003) Efficient code generation for automatic parallelization and optimization. In: *Proceedings of the Second International Conference on Parallel and Distributed Computing*. IEEE Computer Society, pp 23–30
10. Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T (1996) Parallel programming with polaris. *Computer* 29(12):78–82
11. Blume W, Eigenmann R (1994) The range test: a dependence test for symbolic, non-linear expressions. In: *Supercomputing'94*, Proceedings. IEEE, pp 528–537
12. Bondhugula U, Bandishti V, Pananilath I (2017) Diamond tiling: tiling techniques to maximize parallelism for stencil computations. *IEEE Trans Parallel Distrib Syst* 28(5):1285–1298
13. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, pp 101–113
14. Cardoso JMP, Coutinho JGF, Carvalho T, Diniz PC, Petrov Z, Luk W, Gonçalves F (2016) Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Softw Pract Exp* 46(2):251–287
15. Chandra R (2001) *Parallel programming in OpenMP*. Morgan kaufmann, Burlington
16. Clang Clang: a C language family frontend for LLVM. <http://clang.llvm.org/>
17. Dave C, Bae H, Min S, Lee S, Eigenmann R, Midkiff S (2009) Cetus: a source-to-source compiler infrastructure for multicores. *Computer* 42(12):36–42
18. Intel. Intel C++ Compiler. <https://software.intel.com/en-us/c-compilers/>
19. Kelly W, Maslov V, Pugh W, Rosser E, Shpeisman T, Wonnacott D (1996) New user interface for petit and other extensions. *User Guide*. University of Maryland, pp 1–20
20. Kelly W, Pugh W, Rosser E, Shpeisman T (1996) Transitive closure of infinite graphs and its applications. *Int J Parallel Program* 24(6):579–598
21. Keryell R, Ancourt C, Coelho F, Eatrice B, Frann C, Irigoien F, Jouvelot P (1996) Pips: a workbench for building interprocedural parallelizers, compilers and optimizers. Technical report, École Nationale Supérieure des Mines de Paris, France., 04
22. Kremer U, Bast H-J, Gerndt M, Zima HP (1988) Advanced tools and techniques for automatic parallelization. *Parallel Comput* 7(3):387–393
23. Larsen P, Ladelsky R, Lidman J, McKee SA, Karlsson S, Zaks A (2012) Parallelizing more loops with compiler guided refactoring. In: *41st International Conference on Parallel Processing (ICPP)*. IEEE, pp 410–419
24. Lee S-I, Johnson T, Eigenmann R (2003) Cetus—an extensible compiler infrastructure for source-to-source transformation. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer, pp 539–553
25. Liao C, Quinlan DJ, Willcock JJ, Panas T (2008) Automatic parallelization using openmp based on stl semantics. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States)
26. Maydan DE, Hennessy JL, Lam MS (1991) Efficient and exact data dependence analysis. In: *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, pp 1–14
27. Memeti S, Pllana S (2018) Papa: a parallel programming assistant powered by IBM Watson cognitive computing technology. *J Comput Sci* 26:275–284
28. Muchnick SS (1997) *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco
29. OpenCL. OpenCL. <https://opencl.org>
30. OpenMP. OpenMP. <http://www.openmp.org>
31. OpenMP Application Programming Interface, version 4.5. <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>
32. Palkowski M, Bielecki W (2015) Traco parallelizing compiler. In: *Soft Computing in Computer and Information Science*. Springer, pp 409–421
33. Palkowski M, Bielecki W (2017) Traco: source-to-source parallelizing compiler. *Comput Inform* 35(6):1277–1306
34. Pinto P, Carvalho T, Bispo J, Ramalho MA, Cardoso JMP (2018) Aspect composition for multiple target languages using LARA. *Comput Lang Syst Struct* 53:1–26
35. The Portland Group. PGI Fortran & C. <http://www.pgroup.com>



36. Pouchet L-N Polybench: the polyhedral benchmark suite. <http://www.cs.ucla.edu/pouchet/software/polybench>. Accessed June 2019
37. Pugh W (1991) The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. ACM, pp 4–13
38. Quinlan D (2000) Rose: compiler support for object-oriented frameworks. *Parallel Process Lett* 10(02n03):215–226
39. Quinlan D, Liao C, Too J, Matzke RP, Schordan M Rose compiler infrastructure. <http://rosecompiler.org/>
40. Soundrarajan P, Nasre R, Jehadeesan R, Panigrahi BK A study on popular auto-parallelization frameworks. In: *Concurrency and Computation: Practice and Experience*, pp 1–28 (**in press**)
41. Stolte C, Tang D, Hanrahan P (2002) Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans Vis Comput Graph* 8(1):52–65
42. Top500. TOP500 supercomputer sites. <http://www.top500.org>
43. Tu P, Padua D (2001) Automatic array privatization. In: *Compiler Optimizations for Scalable Parallel Systems*, pp 247–281. Springer
44. Wilson RP, French RS, Wilson CS, Amarasinghe SP, Anderson JM, Tjiang SWK, Liao S-W, Tseng C-W, Hall MW, Lam MS, Hennessy JL (1994) SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices* 29(12):31–37
45. Wolfe M (1989) *Optimizing supercompilers for supercomputers*. The MIT Press, Cambridge
46. Wolfe MJ (1995) *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc., Boston

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.