An OpenMP based Parallelization Compiler for C Applications

Hamid Arabnejad*, João Bispo*, Jorge G. Barbosa†, João M.P. Cardoso*
*Faculty of Engineering, University of Porto
† LIACC, Faculty of Engineering, University of Porto
{hamid.arabnejad, jbispo, jbarbosa, jmpc}@fe.up.pt

***Abstract*—Directive-drive programming models, such as OpenMP, are one solution for exploiting the potential of multi-core architectures, and enable developers to accelerate software applications by adding annotations on for-type loops and other code regions. However, manual parallelization of applications is known to be a non trivial and time consuming process, requiring parallel programming skills. Automatic parallelization approaches can reduce the burden on the application development side. This paper presents an OpenMP based automatic parallelization compiler, named `AutoPar-Clava`, for automatic identification and annotation of loops in C code. By using static analysis, parallelizable regions are detected, and a compilable OpenMP parallel code from the sequential version is produced. In order to reduce the accesses to shared memory by each thread, each variable is categorized into the proper OpenMP scoping. Also, `AutoPar-Clava` is able to support *reduction* on arrays, which is available since OpenMP 4.5. The effectiveness of `AutoPar-Clava` is evaluated by means of the Polyhedral Benchmark suite, and targeting a N-cores 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 (i.e., icc), ROSE, TRACO, and Cetus.**

***Index Terms*—Automatic Parallelization, Source-to-source Compilation, Parallel Programming, Static Analysis**

I. Introduction

Parallel computing is no longer limited to supercomputers or mainframes. Personal desktop computers or even mobile phones and electronic portable devices can benefit from parallel computing capabilities. However, parallel programming is never an easy task for users. Generally, dealing with parallel programming issues, such as data dependencies, load balancing, synchronization, and race conditions, requires some level of knowledge about parallel paradigms and the target architecture. Therefore, auto-parallelizing compilers and tools, which (i) accept the sequential source code of an application, (ii) automatically detect and recognize parallelizable sections, and (iii) return the parallelized version, are becoming of increasingly importance for application developers. Based on that, many efforts have been made to support developers [1, 2] as well as to provide automatic parallelization [3, 4, 5, 6].

This paper presents the parallelization module of the source-to-source compiler Clava [7, 8], named `AutoPar-Clava`. A preliminary version of `AutoPar-Clava` has been presented in [9]. The present version extends the primary version [9] and includes additional variable analysis that provides wider range of variable scoping alongside providing support for new features of OpenMP 4.5, such as array reduction. `AutoPar-Clava` is a source-to-source compilers that accepts C-code as input, and returns the parallelized version, annotated by OpenMP directives, as the output. The compiler provides a Abstract-Syntax Tree (AST) and a high-level programming environment for specifying source code analysis and transformations. `AutoPar-Clava` analyses the input source code and parallelizes for-type loop regions where no dependencies or race conditions were found. Currently, we consider a static approach where all loops are considered for parallelization and it is up to the user to select which ones to keep parallel. However, the user can indicate if inner loops should be considered for parallelization, in order to avoid nested parallelism. This version does not include loop transformations (e.g, polyhedral model) or locality optimizations (e.g., temporal and spatial locality [10, 11]). Pointers are also not considered in the current version of the compiler.

Unlike other approaches, which are not integrated with their own compiler and often written with high complex programming, `AutoPar-Clava` provides a simple and general-purpose language to reproduce or modify the parallelizing strategies for non compiler expertise.

The main contributions of this paper are the following:

- automatic proper scoping of shared array variables that allows performance improvements;
- provides a widely range of variable scoping for scalar variables;
- provides the OpenMP reduction feature for array variables;
- an extensive evaluation study with four state-of-the-art compilers using the Polyhedral Benchmark suite, and improvements in 38% and similar results on 52% of the benchmarks.

The rest of the paper is organized as follows. The proposed approach, `AutoPar-Clava`, is presented in detail in Section II. The experimental methodology is presented in Section III, and in Section IV, experimental results are presented and discussed. Section V discusses related automatic parallelization tools with a brief description of some well-known approaches. Finally, Section VI draws conclusions and briefly outlines future work.

II. Automatic Parallelization

In this section, we provide an overview and technical details of AutoPar-Clava for automatic parallelization of the input C code with annotated OpenMP directives. Since loops are often the most time-consuming sections in applications, we mainly focus here on loop parallelization.

Typically, a loop *can* be a candidate to be parallelized by using OpenMP directives if it follows a certain canonical form, and avoids certain restrictions, e.g., not containing any `break`, `exit` and `return` statements. Therefore, AutoPar-Clava starts by analyzing the input source code and marks all loops that *can* be a candidate for parallelization. Then, to decide if a candidate loop should be parallelized, it identifies the existence of data dependency between iterations. At the final step, it generates an annotated OpenMP version of the input source code.

The proposed approach contains four main phases: (i) pre-processing of the sequential code; (ii) dependency analysis; (iii) parallelization engine; and (iv) code generation.

Figure 1 shows the annotated OpenMP output generated by AutoPar-Clava for an input example. In this example, we have chosen to insert all OpenMP `pragmas` found by our tool in the output code, just to illustrate what kind of information is added to OpenMP `pragmas` by AutoPar-Clava.

```c
void kernel_atax(int m, int n, double A[1900][2100],
↪  double x[2100], double y[2100], double tmp[1900])
{
  ...
  #pragma omp parallel for private(i, j) firstprivate(A,
  ↪  tmp, x, m, n) reduction(+:y[:2100])
  for (int i = 0; i < m; i++)
  {
    tmp[i] = 0.0;
    #pragma omp parallel for private(j) firstprivate(A,
    ↪  x, n, i) reduction(+:tmp[i])
    for (j = 0; j < n; j++)
      tmp[i] += A[i][j] * x[j];
    #pragma omp parallel for private(j) firstprivate(A,
    ↪  tmp, y, n, i)
    for (j = 0; j < n; j++)
      y[j] += A[i][j] * tmp[i];
  }
  ...
}
```

Fig. 1. Annotated OpenMP C code generated by AutoPar-Clava

AutoPar-Clava provides information about what loops can be parallelized and the corresponding OpenMP `pragma`. One of the contributions of this work is to provide an accessible way for a user to specify which loops should be parallelized. Although we provide some predefined strategies (e.g., parallelize only outermost loops), users can very easily develop their own strategies.

A. Preprocessing and variable access pattern

AutoPar-Clava allows to perform queries, modifications and source-code generation requests over the presenting the source-code. Users can develop custom program analyses and transformations using a high-level programming model based on aspect-oriented concepts and JavaScript, i.e., the LARA framework [7, 8].

B. Dependency analysis

Dependency analysis involves finding occurrences of overlapping accesses in memory and, therefore, it plays a major role in any auto parallelization compiler.

Generally, considering two statements S_1 and S_2, there are three types of data dependencies from source statement S_1 to destination statement S_2 (i.e., $S_1 \rightarrow S_2$): a) *Anti-dependence*, where S_1 reads from a memory location that is overwritten later by S_2; b) *Output-dependence*, where both S_1 and S_2 write to the same memory location; and c) *Flow(true)-dependence*, where S_1 writes into a memory location that is read by S_2.

To determine if a loop can be parallelized, two types of dependencies are analysed: (1) *loop-independent* that represents dependencies within a loop iteration; and (2) *loop-carried* that represents dependencies among different iterations of a loop. In both cases, a precise analyzer which detects a dependence, if and only if it actually exists, is needed. To describe and implement our approach at a higher level, AutoPar-Clava uses separate dependency analysis strategies to process the variable access type. Variables access in loops, are commonly of *scalar* and *array* access types. By performing dependency analysis tests, a loop is considered for parallelization if it is determined that: (i) it has no true dependencies; or (ii) it has a true dependency, but it is a reduction operation; or (iii) 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 variable pattern access (i.e., read, write, or readwrite) analysis over all statements in the loop, in order to find how each reference to a variable is used. By taking advantage of the AST generated by Clang [12], the AutoPar-Clava compiler can provide information, such as the list of variables that were referenced and how they were used (i.e., *Read*, *Write*, or *ReadWrite*). With this information, a pattern access (e.g. `RRWRRR`) for each variable (scalar/array) is generated. The *usage pattern* is defined as a compressed version of pattern access by removing consecutive repetitions from it (e.g., `RWR`), and is used to identify the data dependency of the variables. In addition to collecting pattern accesses within the target loop, the first pattern access outside of the loop is identified and saved as *nextUse* attribute for each variable. For both scalar and array variables, AutoPar-Clava can identify reduction operations and categorize them into `reduction` scope, by using a pattern matching algorithm which follows the rules specified by OpenMP [13][1].

The most common obstacle to loop parallelization are loop-carried dependencies over array elements. Array elements can be characterized by subscript expressions, which usually depend on loop index variables. Since our approach is based

[1]http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf#page=210

on static analysis, if the subscript expression is in the form of non-affine indices, i.e., A[B[i]], the bounds of the subscript cannot be estimated at compile time, therefore, loops with this type of array access not considered by the parallelization process. The main goal of an array dependency analysis is to find the cross-iteration distance vector for each array reference.

There are several tools to perform data dependency analysis, which use different strategies to decide whether a loop is parallelizable. AutoPar [14], used in the ROSE compiler [3], uses the Gaussian elimination algorithm to solve a set of linear integer equations of loop induction variables, in the form of Banerjee-Wolfe inequalities [15], to identify data dependencies. Other approaches to determine data dependencies between array accesses use tests such as GCD [16], Extended GCD [17] and Omega [18]. In AutoPar-Clava, we use the Omega library [18] for data dependency analysis. Loop dependencies are converted into dependency relations in the form of Presburger arithmetic which are directly analyzed using the Omega library.

Next, the output dependency analysis for scalar and array variables within a loop is used to classify each variable into the proper OpenMP scoping.

1) Privatization: The private clause creates a private variable 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 accesses to shared memory by each thread. We implemented a simple but effective variable privatizer in our compiler. The variable privatizer processes *usage patterns* to decide which OpenMP scoping should be considered for the variable. For scalar variables, if the *usage pattern* is only R, it can be set as a firstprivate variable, and if the *usage pattern* equals to WR or it is a loop index variable, it can be categorized as a private variable.

Similarly, if a scalar variable has the potential to be classified as a private variable, alongside the *usage pattern* equals to R for *nextUse* attribute, it is categorized into lastprivate clause.

For array variables, according to OpenMP 4.5 reference manual [13], a variable that is part of another variable (as an array or structure element) 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 which do not have data dependencies among different iterations of a loop, are set into firstprivate variable list.

In the example of Figure 1 the array variable y appears in the reduction clause at line 4, to resolve a data dependency in the input code. Local arrays are created for each thread to accumulate partial results. At line 11, the inner loop creates a new set of nested threads that will update the private copy of the array y. For this group of threads the array variable y is classified as firstprivate so that the address of the array is correctly copied. The OpenMP loop scheduling guaranties that each thread has a different range of the iterator variable, so that there in no writing conflicts.

2) Scalar and array reduction: In cases where privatization does not resolve dependencies, a *reduction* operation may enable parallelization of code (e.g., computation of sum over a variable) by computing a partial result locally by each thread, and updating a global result only upon completion of the loop. 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, the candidate variables for *reduction* operation 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 ||.

For scalar variables, our recognition analysis detects reduction variables and its associated operator that satisfies the above criteria, and excludes the detected reduction variable and its dependencies from the loop-carried dependencies.

Reductions on array variables are a potential source of significant improvements of parallelization performance. However, following the OpenMP reduction criteria for array variables is a complex analysis. Our assumption is that all array subscript expressions are affine functions of the enclosed loop indices and loop-invariant variables. In order to apply above criteria for the array variables in Figure 1, each access within the target loop body is considered as a scalar variable in the detection procedure. 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 Figure 1. Among all array variables accesses, only two array variables tmp and y have *write* memory references in their own pattern access. For the first inner loop at line 9, since the array access tmp[i] is not a function of the enclosed loop index (i.e., variable j) or loop-invariant variables, it can be considered as a scalar variable (i.e., all loop iterations will update the same element tmp[i]). In this case, as it meets the general reduction form var op = expr with acceptable OpenMP operator + for reduction clauses, and does not appears elsewhere in the loop body, it is classified as a reduction clause for the first inner loop at line 9 (i.e. reduction(+ : tmp[i])).

For the outermost loop at line 5, the dependency analysis detects output dependence relations for two array variables tmp and y, which cannot be solved by a privatization process. For array variable tmp, as the subscript expressions (i.e., tmp[i]) is a function of the outermost loop iterator (i.e., variable i), all related dependencies can be removed from analysis for the outermost loop. From the outer loop point of view, array variable y within element range of $[0, \cdots, n]$ is updated at each individual iteration. Since the update statements at line 13 (i.e., y[j] += expr) 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, we must specify the lower and upper bound for each dimension of the target array, here array `y`. Therefore, if the array size can be obtained by static analysis, it is classified to perform a reduction operation, otherwise, the loop is marked as non-parallelizable due to the lack of static information.

The implemented dependency analyzer has more complex checking conditions, and for the sake of simplicity, only some conditions that are applied for the example presented here, are mentioned.

C. Parallelization engine

This module is the main core of the `AutoPar-Clava` compiler. It accepts, as input, an input application code (e.g., avoids requiring as input only the loops, previously annotated code or imposing limitations on the source code, such as, not allowing the usage of macros), and generates a parallelized C + OpenMP code version as the output file. It controls other modules such as preprocessing, dependency analysis, and code generation. Most static auto parallelization tools do not consider loops for parallelization when they contain user's function calls. The current version of `AutoPar-Clava` does not include interprocedural data dependency analysis. It performs function call inlining, whenever possible, during the analysis phase, which allows a significant improvement on the ability to detect parallelism. The current implementation does not support functions with multiple exit points, recursive functions, or calls to functions whose implementation code is not available. Function inlining is *only* used during the analysis phase, and all changes in the code due to inlining are discarded before generating the code with OpenMP `pragmas`. Additionally, in order to not miss loop parallelization opportunities due to system function calls, `AutoPar-Clava` uses a simple reference list which contains functions that are known to not modify their input variables or that do not have any I/O functionality (e.g., `sqrt`, `sin`).

D. Parallel code generation

As the last step, after determining if the loop can be parallelized, the output code is generated by adding OpenMP directives for the detected parallelizable loops. The `AutoPar-Clava` AST represents all the information necessary to reconstruct the original source-code, including text elements such as comments and pragmas. `AutoPar-Clava` separates implementation files (e.g., .c) from header files (e.g., .h) and is able to regenerate them from the AST.

III. Experimental Methodology

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

A. Platforms

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

B. Benchmarks

The Polyhedral/C 4.2[2] Benchmark Suite [19] is used for performance evaluation. It contains many patterns commonly targeted by parallelizing compilers. The largest dataset sizes were considered, namely the `MEDIUM`, `LARGE` and `EXTRALARGE` datasets.

C. Compared tools and configurations

Taking into account that `AutoPar-Clava` performs automatic static parallelization over unmodified source-code, among all automatic parallelization approaches presented in Section V, the ones closest to our target are ROSE [3], Cetus [6], and TRACO [4, 5]. As part of the ROSE compiler, autoPar [14] can automatically insert OpenMP pragmas in C/C++ code. The autoPar version used is 0.9.9.199. The TRACO and Cetus version 1.4.4 are used in our experiments. The Intel C/C++ Compiler (i.e., `icc`), as a well-known commercial approach, is used, in the version 18.0.0 free academic license. Additionally, since our target in this study is to reconstruct the original source-code, including OpenMP annotation, polyhedral compilers such as PLUTO [20], by applying loop transformation (e.g., tiling and loop fusion), change the loop structure in the generated output code, therefore, they are not considered in our study and can be seen as compilation.

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: 1) `-O2` is the generally recommended optimization level by Intel[3]; and 2) 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`[4]. To generate the parallelized versions by `icc` we use the flag `-parallel`.

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. Additionally, 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, 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 were able to compile the original source files without imposing any limitations on the source code. Therefore, to

[2]https://sourceforge.net/projects/polybench/

[3]https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[4]https://software.intel.com/en-us/forums/intel-c-compiler/topic/755677

provide a compatible input file for TRACO and Cetus, we modified the input code according to the restrictions of each of these compilers.

D. Evaluation metrics

The evaluation is focused on the speedup obtained with the parallel generated code, which is defined as the ratio of the execution time of the sequential code to that 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 is reported.

IV. Experimental results

This section starts by showing detailed performance results for all benchmarks, then average results are presented and it ends with a comparison to Pluto, as polyhedral approach for code parallelization.

A. Performance results

TABLE I

NUMBER AND TYPE OF THE LOOPS PARALLELIZED BY EACH COMPILER

	TRACO		**Cetus**		**ROSE**		**AutoPar-Clava**	
Benchmark Name	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	–	2
durbin	–	2	–	3	–	3	–	3
fdtd-2d	–	4	–	4	–	4	–	4
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	–	–	–	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**	**19**	**28**	**25**	**28**	**27**	**32**	**27**

⚠ : The input source code is modified by transforming the loop structure
✖ : No output file generated by the tool
✎ : the code is changed and modified by the tool

Table I shows the loops parallelized by each auto-parallelization compiler, as well as the type of the loops detected, i.e, inner or outer. When a nested loop is identified to be parallelized, only the outermost one is marked to be parallelized.

Based on the parallelization report, `icc` also applies loop transformations such as loop fission. For simplicity, we refer to `autoPar` tool in ROSE compiler as `ROSE` in Figure 2 and Table I.

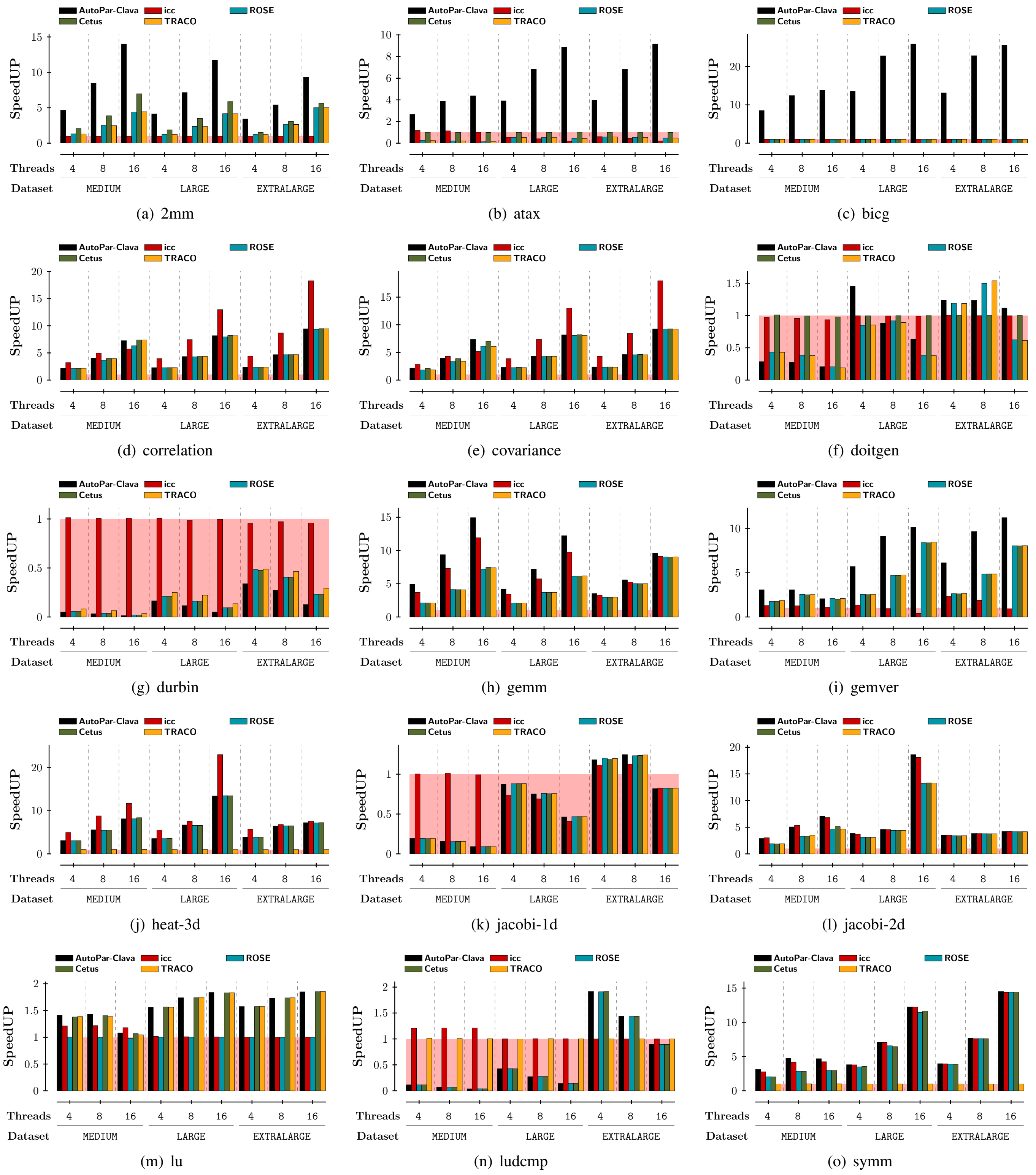
In terms of the compilation time, `AutoPar-Clava` was able to parallelize each input source files with an average of 2.5 seconds for all tested benchmarks, while `icc` required 0.5 seconds.

Figure 2 shows the speedups of the parallelized versions relative to the sequential versions, for each compared approach. Due to space limitations, we only present benchmarks with significant difference in terms of obtained speedup, whereas for other benchmarks the ratio improvements are similar for all compared tools. In Figure 2, each chart shows a PolyBench benchmark and contains a red zone that represents slowdowns. Values above the red zone represent performance improvements over the sequential version. Bellow we discuss the results achieved for each individual benchmark.

2mm and 3mm : they perform Matrix Multiplications with 2 individual nested loops, respectively. The outermost loop at each nested loop is parallelized by all auto-parallelization tools. However, as shown in Figure 2(a), for *2mm*, the performance improvements are not similar. The reason can be explained by the variable classification into the proper OpenMP scoping performed by each tool. Among all compared tools, TRACO has the fewest variable scoping, i.e., it just finds parallelizable loops and inserts the OpenMP `parallel for` pragma without any variable scoping. The ROSE compiler only supports variable scoping for scalar variables. Cetus 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 widely range of variable scoping. As it is shown in Figure 1, our approach categorizes each variable of type array based on its usage pattern. For instance, since array `A` has read only pattern accesses inside of the outermost loop, it is categorized as `firstprivate` in the clause variable list. This avoids accessing the variable by dereferencing a pointer[5]. In terms of performance, `AutoPar-Clava` and Cetus show the best improvements for *2mm*. For *3mm*, since all three outermost loops are parallelized by each tool as shown in Table I, the relative speedup improvement is similar to *2mm*.

atax and bicg : both 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 operations. Both ROSE and TRACO compilers parallelize the same loop. Since both approaches only parallelize the first nested loop in *bicg*, and

[5]https://docs.oracle.com/cd/E19059-01/stud.10/819-0501/7_tuning.html



(a) 2mm (b) atax (c) bicg (d) correlation (e) covariance (f) doitgen (g) durbin (h) gemm (i) gemver (j) heat-3d (k) jacobi-1d (l) jacobi-2d (m) lu (n) ludcmp (o) symm

Fig. 2. Speedups for PolyBench benchmarks obtained by each compared tool

could not parallelize 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 Figure 2(c). For *atax* in Figure 2(b), both ROSE and TRACO approaches show performance slowdowns. This can be explained by parallelizing the inner loop from the second nested loop, which causes a higher time overhead of threads starting and releasing at each iteration. Since our proposed approach supports the OpenMP array reduction feature, for both benchmarks, the `AutoPar-Clava` could obtain a significant speedup of 10× and 25× for *atax* and *bicg*, respectively. The parallelized output of our approach for *atax* is presented in Figure 1.

Correlation and covariance : the best speedup is obtained by Intel *icc* which achieved speedup of 18× and 17× for *correlation* and *covariance*, respectively. Since that other auto-parallelization tools marked the similar loops as parallelization target in their output, they demonstrate a similar performance for different input datasets and number of threads. However, AutoPar-Clava and Cetus show slightly better improvements for lower dataset sizes, as shown in Figure 2(d) and (e).

Doitgen and durbin : the obtained speedup by Intel icc in Figure 2(f) and (g) indicates no loop parallelization for both benchmarks. The other auto-parallelization tools parallelized the same loops. The only exception is the parallelized output generated by Cetus, which did not pass the verification step as it is noted in Table I.

Gemm and gemver : for both benchmarks, AutoPar-Clava, ROSE, Cetus, and TRACO parallelized the same loops. However, due to the wide range of variable scoping, our approach obtains better performance compared to other tools. In the case of Intel icc, the obtained speedup ranks the second place for *gemm*, and the last place without any parallelization for *gemver*.

Heat-3d, jacobi-1d, and jacobi-2d : all compared auto parallelization tools, except icc, annotated the similar inner loop in their generated output code. There is a degradation of performance for *jacobi-1d* benchmark which can be explained by the amount of work (i.e., dataset size) performed by the inner loop. As 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 Figure 2(k). In contrast, for *jacobi-2d* with higher amount of computation work, even by parallelizing an inner loop, we could achieve better execution time, as shown in Figure 2(l). For *heat-3d*, AutoPar-Clava, ROSE, Cetus, TRACO show similar performance improvements.

Lu, ludcmp, and symm : for all these benchmarks, the parallelized loop is the inner one, and similar in the output code generated by each tool. The only exception is the ROSE compiler, which did not parallelize any loop in *lu*. The differences between parallelized inner loops among these benchmarks are the rank of the selected loop for parallelization. For instance, in the case of *ludcmp*, Figure 2(n), the same innermost loop is parallelized by each tool, which causes a significant overhead in the execution time. For *lu*, the parallelized inner loop has a nested loop structure with high amount of computational work, which causes improvements on execution time, as shown in Figure 2(m). Also, in case of *symm*, the output code is changed and modified by TRACO, and since it did not pass the verification step, no parallelization is considered as it is shown in Figure 2(o).

B. Average performance

To illustrate the overal results, Figure 3 presents a boxplot chart of speedup as a function of the number of threads and dataset size among all PolyBench benchmarks.

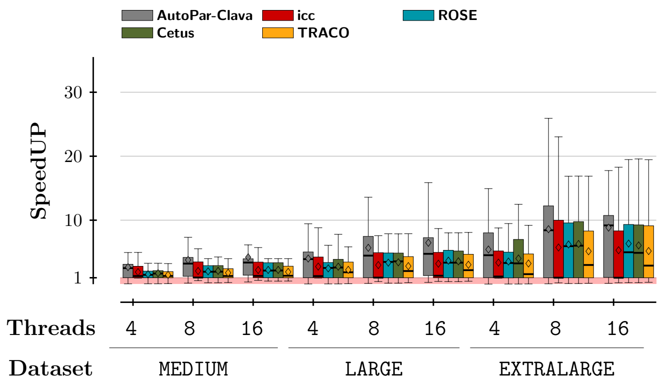


Fig. 3. Average speedup obtained by each tool

In addition to geomean value, the geomean value is also indicated by an individual diamond symbol in each boxplot. We can see that AutoPar-Clava has the highest average speedup with a wider dispersion in the distribution of the results. This is an important finding because with the proposed approach, we improved the speedup and also achieved high values of performance, in most cases as it is shown in Figure 2.

C. Comparison to Polyhedral approach

The polyhedral approach is used to solve data dependencies, to produce parallel versions of the sequential code, possibly including loop transformations. Pluto [20] is a fully automatic source-to-source compiler which uses the polyhedral 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 4 presents the obtained geomean speedup by these two approaches for all datasets and considering 4, 8 and 16 threads.

Among all benchmarks available the Polyhedral Suite [19], generated parallelized output code by Pluto did not passed verification step[6] for *adi*, *deriche*, *ludcmp*, and *nussinov*, which are excluded from Figure 4.

As it is shown in Figure 4, among all 225 combinations of the 25 benchmarks, 3 dataset sizes and 3 thread configurations, the proposed AutoPar-Clava approach could obtained better performance for 101 cases with geometric mean speedup of 6.73× against 2.35× speedup of Pluto. In same way, Pluto shows better improvements for 123 cases with geometric mean speedup of 5.76× against 1.8× speedup of AutoPar-Clava.

[6]dumping all live-out arrays by using the flag -POLYBENCH_DUMP_ARRAYS, and comparing it with the equivalent output of the sequential versions)

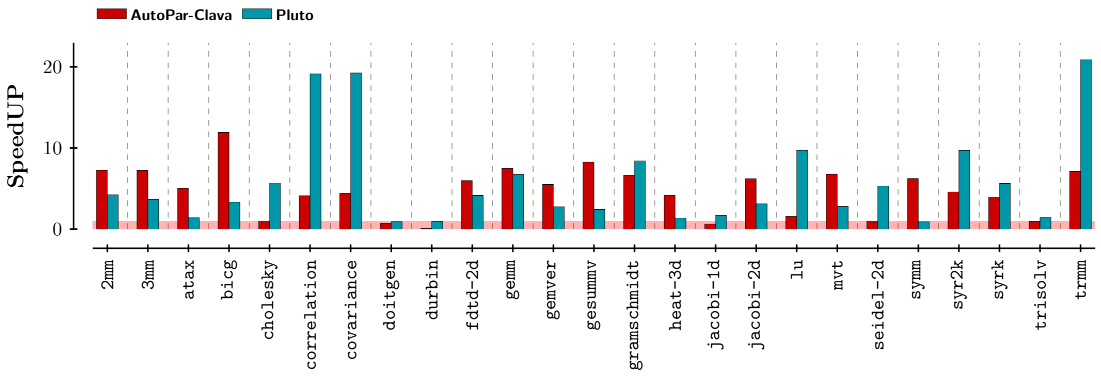


Fig. 4. Speedups for PolyBench benchmarks achieved by `AutoPar-Clava` and Pluto

V. Related Work

The efficient utilization of modern computing systems requires the appropriate usage 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. For example, Larzen et al. [1] propose an interactive compilation feedback system that guides programmers to iteratively modifying application source code, in order to obtain better results with auto-parallelization tools. Bagn et al. [2] present a framework that allows users to accept or modify polyhedral transformations suggestions to be applied to the application. In this case, the loops in the output code could be restructured from the initial version. Recently, Memeti et al. [21] developed a system with cognitive properties in order to assist the programmers to avoid common OpenMP mistakes.

As the objective of this paper is to propose a automatic source-to-source automatic parallelization compiler, we mainly focus on parallelization tools which are not guided by runtime information from program execution or by additional guidances provided from the user. Next, we briefly discuss tools which perform automatic loop parallelization by static analysis of the input source code.

Cetus [6] 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 [15] as a data dependency test framework, also contains the range test [22] as an alternative dependency test. Cetus provides auto-parallelization of loops through private and shared variable analysis, and automatic insertion of OpenMP directives.

Pluto [20] is a fully automatic polyhedral source-to-source program optimizer tool. It translates C loop nests into an intermediate polyhedral representation called CLooG [23] (Chunky Loop Generator). With the ClooG format, 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 tiling and to extract coarse grained parallelism and locality, and finally to transform loops. However, it only works on individual loops, which have to be marked in the source code using pragmas.

ROSE [3] is an open source compiler, which provides source-to-source program transformations and analysis for C, C++ and Fortan applications. ROSE provides several optimizations, including auto-parallelization, loop unrolling, loop blocking, loop fusion, and loop fission. As a part of ROSE compiler, autoPar [14] is the automatic parallelization tool used to generate OpenMP code versions of 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.

The auto-parallelization feature of the Intel Compiler icc [24] automatically detects loops that can be safely and efficiently executed in parallel and generates multi-threaded code of the input program. To detect loops that are candidates for parallel execution, it performs data-flow analysis to verify correct parallel execution, and internally inserts OpenMP directives. icc support variable privatization, loop distribution, and permutation.

TRACO [4, 5] is a loop parallelization compiler, based on the iteration space slicing framework (ISSF) and the Omega library, while loop dependence analysis is performed by means of the Petit [25] tool. Output code contains OpenMP directives.

Our parallelization strategy distinguishes to these approaches in the following aspects: (i) produced functionally correct parallelized output code for all the evaluated benchmarks, similarly to the ROSE compiler (see Table I); (ii) 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 I); and (iv) it supports reduction for arrays and at the array element level.

VI. Conclusion

This paper presented the `AutoPar-Clava` compiler, which provides a versatile automatic parallelization approach for Clava, a C source-to-source compiler. The compiler is currently focused on parallelizing C programs by adding OpenMP directives. The proposed source-to-source compiler deals with the original code, and inserts OpenMP directives (mainly parallel-for and atomic directives) and the necessary clauses. The main contribution of our approach, 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 parallelization strategy presented in this paper, and fully integrated in the `AutoPar-Clava` compiler, considers array reduction to significantly improve the execution time.

The experiments provided show promising results and improvements regarding other auto-parallelization compilers when targeting a multicore x86-based platform. With the Polyhedral Benchmark suite, `AutoPar-Clava` achieved better performance for 11 benchmarks, equal for 15 and worse in 3 cases. In average, `AutoPar-Clava` obtains higher speedups among all benchmarks compared to other tools (3).

Our ongoing work is focused on the evaluation of the compiler with other benchmarks. As future work, we plan to include 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 and by using, e.g., polyhedral model approaches. In addition, we intend to research a cost-based analysis for guiding decisions.

VII. Acknowledgments

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] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks, “Parallelizing more loops with compiler guided refactoring,” in *41st International Conference on Parallel Processing (ICPP)*, pp. 410–419, IEEE, 2012.

[2] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, “Opening polyhedral compiler’s black box,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pp. 128–138, ACM, 2016.

[3] D. Quinlan, “ROSE: Compiler support for object-oriented frameworks,” *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[4] M. Palkowski and W. Bielecki, “TRACO parallelizing compiler,” in *Soft Computing in Computer and Information Science*, pp. 409–421, Springer, 2015.

[5] M. Palkowski and W. Bielecki, “Traco: Source-to-source parallelizing compiler,” *Computing and Informatics*, vol. 35, no. 6, pp. 1277–1306, 2017.

[6] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, “Cetus: A source-to-source compiler infrastructure for multicores,” *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[7] J. M. Cardoso, J. G. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonçalves, “Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach,” *Software: Practice and Experience*, vol. 46, no. 2, pp. 251–287, 2016.

[8] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. Cardoso, “Aspect composition for multiple target languages using LARA,” *Computer Languages, Systems & Structures*, vol. 53, pp. 1–26, 2018.

[9] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. Cardoso, “Autopar-clava: An automatic parallelization source-to-source tool for c code applications,” in *9th and 7th Workshops on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM ’18, pp. 13–19, ACM, 2018.

[10] A. Susungi, A. Cohen, and C. Tadonki, “More data locality for static control programs on numa architectures,” in *7th International Workshop on Polyhedral Compilation Techniques IMPACT 2017*, p. 11, 2017.

[11] J. Carabaño, J. Westerholm, and T. Sarjakoski, “A compiler approach to map algebra: automatic parallelization, locality optimization, and gpu acceleration of raster spatial analysis,” *GeoInformatica*, vol. 22, no. 2, pp. 211–235, 2018.

[12] Clang, “Clang: a C language family frontend for LLVM.” http://clang.llvm.org/.

[13] “OpenMP Application Programming Interface, version 4.5.” https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[14] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas, “Automatic parallelization using OpenMP based on STL semantics,” tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2008.

[15] M. Wolfe, *Optimizing supercompilers for supercomputers*. Cambridge, MA; The MIT Press, 1989.

[16] U. Banerjee, *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media, 2007.

[17] D. E. Maydan, J. L. Hennessy, and M. S. Lam, “Efficient and exact data dependence analysis,” in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 1–14, 1991.

[18] W. Pugh, “The Omega test: a fast and practical integer programming algorithm for dependence analysis,” in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 4–13, ACM, 1991.

[19] L.-N. Pouchet, “Polybench: The polyhedral benchmark suite,” *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[20] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, 2008.

[21] S. Memeti and S. Pllana, “PAPA: A parallel programming assistant powered by ibm watson cognitive computing technology,” *Journal of Computational Science*, vol. 26, pp. 275–284, 2018.

[22] W. Blume and R. Eigenmann, “The range test: a dependence test for symbolic, non-linear expressions,” in *Supercomputing’94., Proceedings*, pp. 528–537, IEEE, 1994.

[23] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 101–113, 2008.

[24] “Intel C++ Compiler,” 2013. https://software.intel.com/en-us/c-compilers/.

[25] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, “New user interface for Petit and other extensions,” *User Guide*, 1996.