Chapter 3. The OpenMP-based Auto Parallelization AutoPar-Clava Approach

Hamid Arabnejad[1], João Bispo[1], Jorge G. Barbosa[1,2], and João M.P. Cardoso[1]

[1]*Faculty of Engineering, University of Porto (FEUP),* [2]*LIACC*
`{hamid.arabnejad, jbispo, jbarbosa, jmpc}@fe.up.pt`

Abstract

Modern processors are composed by several processing elements, known as multicore architectures, which brings to the common user the possibility to use parallel computing techniques in order to fully exploit the computational power available in modern machines. Directive-driven programming models, such as OpenMP, are a common solution for exploring the potential of multicore architectures, and enable users (i.e., developers) to accelerate software applications by adding annotations on for-type loops and code regions to change the execution pattern of their code. However, to transform a sequential code into a parallel version requires advanced programming knowledge and it is also a time consuming task to achieve performance. To overcome this burden, we present in this chapter a compilation tool, `AutoPar-Clava`, that is able to automatically detect parallelizable loops in a C application without any user intervention or profiling information; that classifies variables used inside the target loop based on their access pattern; and that generates a C OpenMP parallel code from the input sequential version. The tool is also able to implement automatically *reduction* operations either for scalar and array data. The implementation details of `AutoPar-Clava`, its usage and application examples are reported in this chapter. The tool showed to be of practical use and achieved good performance for several benchmarks, such as the NAS and Polyhedral Benchmark suites, targeting a 16-cores x86-based computing platform.

1 Introduction

The evolution of the microprocessors to the multicore architecture has brought the parallel computing paradigm to the general computer programmer. New tools have emerged, such as OpenMP, OpenCL and CUDA, as a complement to the programming languages like C/C++ and Fortran. For transforming a sequential code into a parallel efficient code, a programmer needs to understand the data dependencies existing in the code, load balancing issues, synchronization, race conditions, and to identify the appropriate parallel model to apply, namely, data parallelism, functional parallelism or pipelining. To assist the programmer in the code transformations to obtain a parallel version, we have developed a source-to-source compilation tool, named `AutoPar-Clava`, which is presented in this chapter.

Source-to-source compilation shown to be an effective solution for the automatic parallelization, due to the fact that it produces transformed source code from the original one instead of binaries. The main advantage is that the generated output code can be analyzed and, possibly, further tuned by the user, and the binary code is then obtained with the compiler chosen by the user. Source-to-source compilation can be classified as *fully* or *semi* automatic parallelization approach. The first category is based on a static analysis which may causes to miss some parallelizable regions due to unsolved data dependencies, which may be further improved by an expert user. To overcome this situation, *semi*-automatic approaches use some additional guidance information from the user as a code annotation and/or program execution. `AutoPar-Clava` uses a *fully* automatic parallelization and, therefore, a wider range of users may be able to use it, as it provides a solution without any user intervention.

There are a variety of source-to-source compilation tools and libraries to make auto-parallelization code easier, such as Intel Compiler and PGI[1] as well-known commercial ones, and ROSE[2, 3, 4], Cetus[5, 6, 7], TRACO[8, 9], and PLUTO[10] as examples of academic approaches. However, many of these approaches generate a binary output, or are often written with complex programming which makes it hard to reproduce or modify the parallelizing strategies for a non-expert user. Therefore, the approach of `AutoPar-Clava` allows a higher level of abstraction and also provides a simple, easy to read, open-source tool to reproduce or modify the parallelizing strategies for non-expert users.

`AutoPar-Clava` produces code annotated with OpenMP [11, 12] pragmas. OpenMP is one of the most popular directive-driven programming models to implement shared-memory parallel programming, and provides a simple and flexible interface for developing parallel applications for platforms ranging from standard desktop computers to supercomputers.

The novelty of `AutoPar-Clava` consists in detecting automatically parallel regions in the source code; in detecting proper scalar and array variable scoping, such as `private` and `firstprivate`; induction variables; and in detecting parallel reduction opportunities for scalar and array reductions. The following sections present the architecture of `AutoPar-Clava` and explain its usage.

2 AutoPar-Clava Approach

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. Loops are often one of the main sources of parallelism due to their time-consuming nature in each application. However, there are several reasons that may prevent loop parallelization, such as the existence of dependent variables. Generally, auto-parallelization approaches try to detect any occurrence of data dependencies between individual iterations by analyzing the access pattern for each variable (scalar/array) in a static or dynamic approach by using execution profiling. We mainly focus here on static strategies for analyzing access patterns for variables. 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.

`AutoPar-Clava` starts by analysing 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 dependencies between iterations. At a final step, it generates an annotated OpenMP version of the input source code. Figure 1 shows the annotated OpenMP output generated by the proposed compiler for a sample input file.

The proposed approach contains four main phases:

(i) **Preprocessing:** involves collecting pattern access (i.e., *read*, *write*, or *readwrite*) for each variable within the target loop. With Clava, users can create custom program analyses using a high-level programming model based on aspect-oriented concepts, and

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

Figure 1: OpenMP C code generated by `AutoPar-Clava`

retrieve information such as variable name, pattern access, and variable access type. This kind of information is extensively used in the dependency analysis process. In addition, for resolving certain classes of data dependencies, the induction variable technique is applied before the preprocessing phases. In particular, `AutoPar-Clava` uses this technique to handle induction statements, specifically when they appear as a/part of subscript expressions on array variable accesses, to the resolution of iteration dependencies on array accesses.

```
1 select function{'kernel_atax'}.loop.body.varref end
2 apply
3  println(
4     'VarName: '    + $varref.name +
5     'useExpr: '    + $varref.useExpr.use +
6     'Accesstype: ' + $varref.useExpr.joinpointType
7   );
8 end
9 condition $loop.rank.toString() === '1,2' end
```

```
VarName: y    useExpr: readwrite  Type: arrayAccess
VarName: j    useExpr: read       Type: varref
VarName: A    useExpr: read       Type: arrayAccess
VarName: i    useExpr: read       Type: varref
VarName: j    useExpr: read       Type: varref
VarName: tmp  useExpr: read       Type: arrayAccess
VarName: i    useExpr: read       Type: varref
```

Figure 2: An example of a LARA aspect that *print* variable access inside of the target loop (Top), and the AutoPar-Clava output when applying the aspect to the source code in Figure1 (Bottom)

Figure 2 shows the LARA aspect that extracts the pattern access (i.e., read, write, or readwrite) for each variable within the second inner loop body (line 12-13) of the C code in Figure.1. In the LARA aspect example, Figure 2(top), line 1 queries the code and selects all variable accesses (`varref` joinpoints) inside the 3 available loop bodies (`loop.body`) from the specific function `function{kernel_atax}` of the input C code. By filtering the query, in line 9, LARA only returns the loop body of the second innermost loop in the target function (i.e., `$loop.rank.toString()===’1,2’`). By querying the different attributes available in the joinpoint `varref`, we can retrieve information such as variable name (i.e., `name`), variable pattern access (i.e., `use`), and variable access type (i.e., `joinPointType`). This kind of information is extensively used in the dependency analysis process.

(ii) **Dependency Analysis:** involves finding occurrences of overlapping accesses in memory and, therefore, it plays a major role in any auto parallelization tool. 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 an higher level, `AutoPar-Clava` uses separate dependency analysis strategies to process variable access types. Variables 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 liveness 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 [13], the `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 dependencies of the variables. In addition to collecting pattern access within the target loop, the first pattern access outside of the loop is identified and saved as *nextUse* attribute for each variable.

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 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 accesses are 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. In `AutoPar-Clava`, we use the Omega library[1] [14] for data dependency analysis of array variables. Loop dependencies are converted into dependency relations in the form of Presburger arithmetic which are directly analyzed using the Omega library.

Additionally, 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 [15][2].

In the next step, the output dependency analysis for scalar and array variables within a loop are used to classify each variable into the proper OpenMP scoping. Generally, `AutoPar-Clava` contains two steps for variable OpenMP scoping, namely *privatization* and *variable reduction*.

[1]http://www.cs.umd.edu/projects/omega/
[2]http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf#page=210

(a) *Privatization*: has a strong impact on the performance obtained by loop parallelization, since it reduces the accesses to shared memory by each thread. A private variable serves as temporary data by assigning a separate storage to each thread in the parallel execution. This solves many data dependencies for the target variable if all loop iterations use the same storage. 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. For example, in Figure 1, since that `i` and `j` are loop control variables, they are classified as `private`. For scalar variables `m`, `n`, `A`, `x`, and `tmp`, the *usage pattern* is only `R`, therefore, they can be categorized as a `firstprivate` variables. Similarly, if a scalar variable has the potential to be classified as a `private` variable, alongside a *usage pattern* equal to `R` for *nextUse* attribute, then it is categorized as `lastprivate`.

(b) *Variable 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 to perform a *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.

(iii) **Parallelization engine:** is the main core of the `AutoPar-Clava` compiler. It accepts, as an input, the application source code, and generates a parallelized C OpenMP output code. It controls other modules such as preprocessing, dependency analysis, and code generation.

Function calls have a great impact since they can prevent the parallelization of the target loop, due to lack of information of access patterns for the variables within it. As one possible solution, `AutoPar-Clava` performs function call inlining, whenever possible during the analysis phase, to find a complete access pattern for each variable, even inside the body of the called function. This gives the compiler the opportunity to analyze the called function for side effects. Note that inline functionality of the `AutoPar-Clava` is used during the analysis phase, and all changes in the code due to inlining are discarded in the generated output code. The current implementation does not support functions with multiple exit points, recursive functions, or calls to functions whose implementation code is not available in the input source files. Additionally, to prevent missing parallelization opportunities due to system function calls, `AutoPar-Clava` uses a simple reference list which contains functions that do not modify their input variables or that do not have side effects on I/O functionality (e.g., `sqrt()`, `sin()`).

(iv) **Parallel code generation:** in the last step, the output code is generated by adding OpenMP directives for the detected parallelizable loops. The `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 reproduce them from the AST.

2.1 `AutoPar-Clava` usage

The main focus of `AutoPar-Clava` is loop parallelization. Therefore, to apply the auto-parallelization, the user needs only to send the for-loop joinpoint to `AutoPar-Clava` library. This library analysis the input for-loop statement and decides if the input candidate loop should be parallelized. It identifies the existence of data dependencies between iterations and generates an annotated OpenMP version of the input for-loop statement. An example of the usage of this library can be seen in Figure 3.

```
import clava.autopar.Parallelize;

aspectdef main
  var $loops = [];

  select function{"test"}.loop end
  apply
    $loops.push($loop);
  end

  Parallelize.forLoops($loops);
end
```

Figure 3: An example of how to use the `AutoPar-Clava` library for auto-parallelization of input source code.

Figure 3 shows a LARA strategy that uses the `AutoPar-Clava` library. First, imports the utility class Parallelize (line 1). Then, selects all the loops inside the function with name `test` and stores them in the array `$loops` (lines 6-9). Finally, calls the function `Parallelize.forLoops()` (line 11), which attempts to parallelize all the given loops. Loops that could be parallelized will have an OpenMP pragma. Loops that could not be parallelized remain unchanged.

3 Experimental Results

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

Platform: The evaluation 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. 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.

Benchmarks: Two benchmark suits, namely Polybench Benchmark Suite and NAS Parallel Benchmarks are used for performance evaluation in this work. The Polyhedral/C 4.2[3] Benchmark Suite [16] contains many patterns commonly targeted by parallelizing compilers. Three different dataset sizes, namely `MEDIUM`, `LARGE` and `EXTRALARGE`, were considered. Additionally, we use the NAS Parallel Benchmarks (NPB)[4] where both sequential and manually parallelized OpenMP version are available. For the NPB benchmarks, for input classes, namely W, A, and B, are used. Class w is the smaller input, class B is the largest one, and classes A is the medium size inputs for a single machine.

Compared tools and configurations: Taking into account that `AutoPar-Clava` performs automatic static parallelization over unmodified source-code, among all automatic parallelization approaches, the ones closest to our objective are ROSE [2], Cetus [5], and TRACO [8]. As part of the ROSE compiler, autoPar [4] can automatically insert OpenMP pragmas in

[3]https://sourceforge.net/projects/polybench/
[4]https://www.nas.nasa.gov/publications/npb.html

C/C++ code. The autoPar version used is 0.9.9.199. The TRACO [5] and Cetus version 1.4.4 [6] are used in our experiments. The Intel Compiler `icc`, 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 annotations, polyhedral compilers such as PLUTO [10], by applying loop transformations (e.g., tiling and loop fusion), change the loop structure in the generated output code, therefore, they are not considered in our study.

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 `-g -O2`. The optimization flag `-O2` is used instead of `-O3` because: 1) `-O2` is the generally recommended optimization level by Intel[7]; and 2) to be able to do a fair comparison between serial code and parallel code annotated with OpenMP pragmas, since we have observed 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`[8].

For each benchmark, each experiment was repeated 30 times and the geometric mean of execution time was used. For each data size, we have run the programs with 4, 8, and 16 threads. For 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 tools.

To generate the parallelized versions by `icc` we use the flag `-parallel`. The `icc` uses a cost model with a threshold parameter to decide whether to parallelize a loop. The `-Qpar-threshold[n]` (Windows) and `-par-threshold[n]` (Linux) compiler options adjust this parameter[9]. 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, we used two values of the Intel `icc -par-threshold` parameter, equal to 0 and 100, in order to determine how aggressively the compiler parallelizes loops.

For simplicity, we refer to `autoPar` tool in `ROSE` compiler as `ROSE` in all figures and tables. Also, $\texttt{icc}_{\text{-par-threshold[0]}}$ and $\texttt{icc}_{\text{-par-threshold[100]}}$ are represent the results of `icc` compiler when the `-par-threshold` parameter is set to 0 and 100, respectively.

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.

[5]https://sourceforge.net/projects/traco

[6]https://engineering.purdue.edu/Cetus

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

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

[9]https://software.intel.com/en-us/node/522957

3.1 Polybench benchmarks results

To illustrate the results statistically, Figure 4 presents a boxplot chart of SpeedUp as a function of the number of threads and dataset parameters among all PolyBench benchmarks. In addition to median value, the average 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 performances, in most cases.

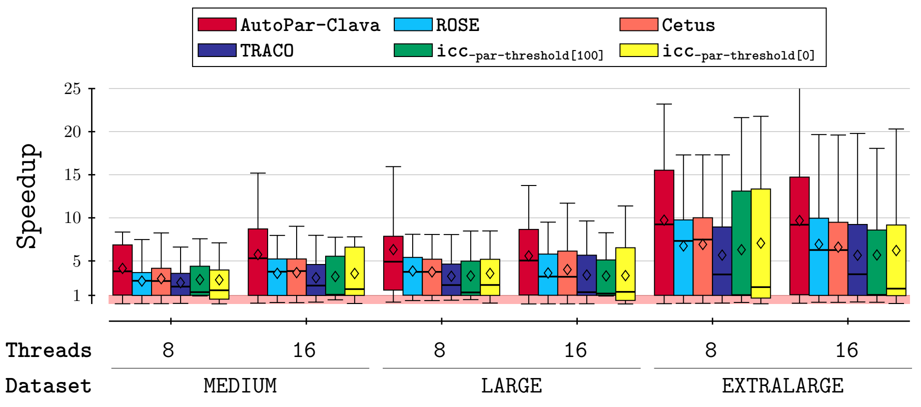

Figure 4: The average SpeedUps obtained by each compiler for Polybench benchmarks

3.2 NAS benchmarks results

To illustrate the results statistically, Figure 5 presents a boxplot chart of SpeedUp, obtained for NAS benchmark by each compared approach, as a function of the number of threads and Class size parameters. As it can be seen from Figure 5, `AutoPar-Clava` obtained a significant better performance compared to other parallelization approaches, except ParallelHand which was parallelized by an expert. In conclusion, we improved the SpeedUp and also achieved high values of performances with a wider dispersion in the distribution of the results.

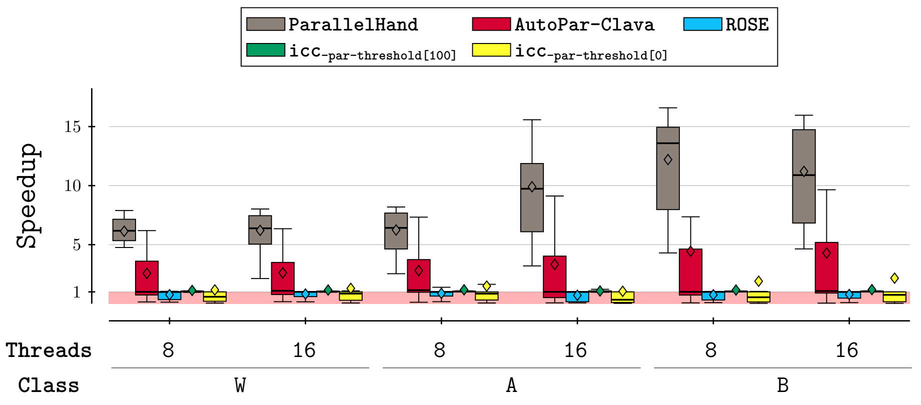

Figure 5: The average SpeedUps obtained by each compiler for NAS benchmarks

4 Conclusion

This work presents 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 work, 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 (Figures 4 and 5).

References

[1] The Portland Group. PGI Fortran & C. `http://www.pgroup.com`.

[2] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[3] Dan Quinlan, Chunhua Liao, Justin Too, Robb P Matzke, and Markus Schordan. Rose compiler infrastructure, 2012.

[4] Chunhua Liao, Daniel J Quinlan, Jeremiah J Willcock, and Thomas Panas. Automatic parallelization using openmp based on stl semantics. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2008.

[5] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12), 2009.

[6] S.-I. Lee, T. Johnson, and R. Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Int. Workshop on Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2003.

[7] H. Bae, D. Mustafa, J.-W. Lee, H. Lin, C. Dave, R. Eigenmann, and S. P Midkiff. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming*, pages 1–15, 2013.

[8] Marek Palkowski and Wlodzimierz Bielecki. Traco parallelizing compiler. In *Soft Computing in Computer and Information Science*, pages 409–421. Springer, 2015.

[9] Marek Palkowski and Wlodzimierz Bielecki. Traco: Source-to-source parallelizing compiler. *Computing and Informatics*, 35(6):1277–1306, 2017.

[10] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[11] OpenMP. OpenMP 4.5 specification. `http://www.openmp.org`.

[12] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] Clang. Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[14] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

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

[16] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.