Hamid Arabnejad, João Bispo, Jorge G. Barbosa and João M. P. Cardoso. 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 (PARMA-DITAM '18). ACM, New York, NY, USA, pp. 13-19.

The published version is available online at: 10.1145/3183767.3183770

AutoPar-Clava: An Automatic Parallelization source-to-source tool for C code applications

Hamid Arabnejad
Faculdade de Engenharia (FEUP)
Universidade do Porto
hamid.arabnejad@fe.up.pt

João Bispo
Faculdade de Engenharia (FEUP)
Universidade do Porto, INESC-TEC
jbispo@fe.up.pt

Jorge G. Barbosa
Faculdade de Engenharia (FEUP)
Universidade do Porto, LIACC
jbarbosa@fe.up.pt

João M.P. Cardoso
Faculdade de Engenharia (FEUP)
Universidade do Porto, INESC-TEC
jmpc@fe.up.pt

ABSTRACT

Automatic parallelization of sequential code has become increasingly relevant in multicore programming. In particular, loop parallelization continues to be a promising optimization technique for scientific applications, and can provide considerable speedups for program execution. Furthermore, if we can verify that there are no true data dependencies between loop iterations, they can be easily parallelized.

This paper describes Clava AutoPar, a library for the Clava weaver that performs automatic and symbolic parallelization of C code. The library is composed of two main parts, parallel loop detection and source-to-source code parallelization. The system is entirely automatic and attempts to statically detect parallel loops for a given input program, without any user intervention or profiling information. We obtained a geometric mean speedup of 1.5 for a set of programs from the C version of the NAS benchmark, and experimental results suggest that the performance obtained with Clava AutoPar is comparable or better than other similar research and commercial tools.

KEYWORDS

Automatic parallelization, source-to-source Compilation, Parallel Programming, OpenMP

1 INTRODUCTION

Multi and many-core programming is becoming a hot topic in the field of computer architectures. Parallel computing is no longer limited to supercomputers or mainframes, moreover, personal desktop computers or even mobile phones and electronic portable devices can have benefits from parallel computing capabilities. In order to utilize the full capabilities of processors, parallel computing platforms can be beneficial and helpful for application development cycle. But, it requires some level of knowledge about parallel paradigms and system architecture making it more difficult for programmers. Therefore, having a tool that accepts, as input, the sequential version source file, automatically detects and recognizes parallelizable segments without any prior knowledge provided by user or program execution, and return the parallelized version, would be of great usage.

Most of the execution time of CPU-intensive applications are often spent in nested loops. As a potential solution, parallelization can help to distribute the overall execution among available threads by sharing processing units and reduce the total execution time. OpenMP [11] is a simple and portable application programming interface (API), that supports many functionalities required for parallel programming. OpenMP provides several environment variables for controlling the execution of parallel code at run-time. However, dealing with issues, such as data dependency, synchronization and race conditions, make it not an easy task for users. Therefore, automatic code parallelization is very attractive and a challenging area since the process of parallelization and optimization should be done without any effort from the programmer.

In this paper, we focus particularly on source-to-source compilation which uses C-code as an input, and returns the parallelized version, annotated by OpenMP directives, as the output without any user interaction. The proposed `AutoPar-Clava` tool acts as an engine that analyses the input source code and parallelizes the code segments which have no dependencies or race conditions. To improve the `AutoPar-Clava` tool capability of detecting and increasing the number of parallel loops, some techniques such as variable privatization and parallel reduction are used. The principal phases of the proposed framework include: (i) preprocessing of the sequential code, (ii) dependency analysis, (iii) parallelization engine, and (iv) code generation.

The rest of paper is organized as follows. Section 2 discusses about the background of automatic parallelization tools with a brief description of some well-known approaches. The proposed approach, `AutoPar-Clava`, is introduced and discussed in details in Section 3. Section 4 presents results of an experimental study. Finally, section 5 draws conclusions and briefly outlines `AutoPar-Clava` future path as a live and ongoing project.

2 OVERVIEW OF AUTOMATIC PARALLELIZATION TOOLS

Source-based automatic parallelization tools accept the code of a program as input, and create a parallelized version. If the analysis is done based only on the source code, without information about the program execution, it is considered that the tool does *static* analysis (as opposed to *dynamic* analysis). A lot of effort has been put into dynamic analysis tools [15, 26], which extend the information that is possible to obtain from the source code of a program by executing the application and gather runtime information. However, such tools are usually harder to use (e.g., require the user to prepare the program in order to be explored), or can take longer time to execute (up to several orders of magnitude), when compared with static analysis tools. Orthogonal to both approaches, it is also possible to improve parallelization analysis if the tools allow the user to provide additional information (e.g., which variables can be ignored from dependency analysis).

In this paper, we mainly focus on parallelization tools which are not guided by runtime execution or by user information. Therefore, we will briefly discuss tools which perform automatic static analysis for loop parallelization.

Cetus [4, 12, 16] is a source-to-source compiler for ANSI C programs developed by Purdue University. 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 [28] as a data dependency test framework, also contains the range test [8] as an alternative dependency test. Cetus provides auto-parallelization of loops through private and shared variables analysis, and automatic insertion of OpenMP directives.

ROSE [23, 24] is an open source compiler, and provides source-to-source program transformation and analysis tools for C, C++ and Fortan applications. ROSE provides several optimizations including autoparallelization, loop unrolling, loop blocking, loop fusion, and loop fission. As a part of ROSE source-to-source compiler infrastructure, `Auto-Par` is the automatic parallelization tool used to generated OpenMP code versions of sequential code.

Pluto [9] is a fully automatic polyhedral source-to-source program optimizer tool. It translates C loop nests into an intermediate polyhedral representation called CLooG [7] (Chunky Loop Generator). With ClooG[6] 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, transforms the loop structure while maintain semantics. However, it only works on individual loops, which have to be marked in the source code using pragmas.

Par4All [3, 27] is an automatic parallelizing and optimizing compiler for C and Fortran, and has back-ends for OpenMP, OpenCL and CUDA. The automatic transformation process is based on PIPS (Parallelization Infrastructure for Parallel Systems) which is a framework for source-to-source for program analysis, optimization and parallelization. Par4all does array privatization, reduction variable recognition and induction variable substitution.

The auto-parallelization feature of the Intel Compiler [2] 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. The Intel Compilers support variable privatization, loop distribution, and permutation.

TRACO [18, 19] is a loop parallelization compiler. It is based on the iteration space slicing framework (ISSF) and the Omega Calculator library, while loop dependence analysis is calculated by means of the Petit [14] tool. Output code is compilable and contains OpenMP directive.

3 OVERVIEW OF AUTOPAR-CLAVA

This section describes the AutoPar library written in LARA [10] for the Clava source-to-source compiler. The library accepts complete, unmodified programs and is capable of fully automatic generation of C code annotated with OpenMP directives. When AutoPar is not able to parallelize a loop, it provides information about the causes.

Identifying parallelizable segments (i.e. loops) is a crucial and difficult step in auto-parallelization approaches. In the absence of additional information from program execution or the user, identifying the data dependency relations are the most challenging and complex stage. Generally, long-running applications have a set of loops (e.g., `for`) that are responsible for most of the execution time of the program. When consecutive iterations of a loop are not data-dependent, they can be executed in parallel (e.g., in different threads), potentially improving the execution time in multi-core architectures. However, there are cases where even when there are data-dependencies, parallelization of loop iterations is possible. For instance, reduction operations gather a result from several iterations that, absent that operation, could be independent. Current parallel frameworks, such as OpenMP, provide tools to handle cases like this, and an important goal of an automatic parallelization framework is to be able to identify these situations in the source code.

3.1 Clava

Clava[1] is a source-to-source compiler that is capable of analyzing and transforming C/C++/OpenCL code. It is based on the LARA framework, which uses the Domain-Specific Language (DSL) LARA [10] to describe source-code analysis and transformations. The language provides specific keywords and semantics that allow queries to points of interest (i.e. *join points*) in the source code (e.g., `file`, `function`, `loop`). Join points provide *attributes* for querying information about that point in the code (e.g., `$function.name`), and *actions*, which apply transformations to that point (e.g., `$vardecl.exec setType('float')`). LARA also provides general-purpose computation by accepting arbitrary JavaScript code in LARA files.

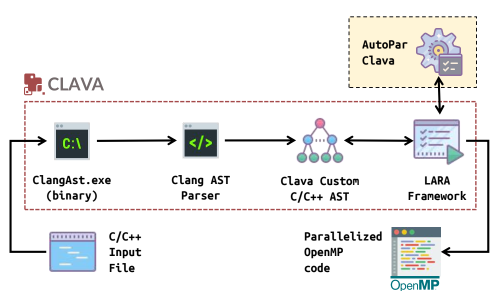


Figure 1: Clava block diagram

Figure 1 shows a block diagram of the Clava tool. Clava is mostly implemented in Java, and internally uses a binary based on Clang

[1]https://specs.fe.up.pt/tools/clava

[1] to dump information about C/C++ programs. This information is then parsed and used to build a custom AST in Java, which the LARA framework uses in the queries, modifications and source-code generation specified by LARA code. With Clava, users can create custom program analyses and transformations using a high-level programming model based on aspect-oriented concepts and JavaScript.

3.2 Loop Parallelization

Loop parallelism is a common form of parallelism that can be found in several types of programs (e.g., scientific models). Typically, a loop can be parallelized by using OpenMP directives if it follows a certain canonical form, and respects certain restrictions, such as not containing any `break`, `exit` and `return` statements. However, checking for situations such as data dependencies, data conflicts, race conditions or deadlocks are responsibility of the OpenMP user. In a first stage of our approach, AutoPar detects and marks all loops that can be parallelized. Then, in a second stage, this information is used to decide which loops should be parallelized. In this work we present a strategy that only parallelizes the outermost loop, in order to reduce the parallelization overhead. However, it is entirely possible to use the same information to create other strategies (e.g., to target other loops).

3.3 Dependence Analysis

An important element in any auto parallelizer tool is the data dependency analysis, which determines if a loop can be parallelized or not. Loop dependencies can be classified in two main categories: (1) dependencies within one loop iteration, i.e. *loop-independent* dependencies, and (2) dependencies between different iterations, i.e. *loop-carried* dependencies. Additionally, the results of the data dependency analysis will be used to determine the proper OpenMP scoping of the variables used in the loop (e.g., `private`, `firstprivate`, `lastprivate`), or if a variable is the target of a `reduction` or privatization.

The `AutoPar-Clava` tool uses separate dependency analysis steps for scalar and array variables. For arrays it uses Petit [14] and the Omega [22] library, which can be time consuming. In order to reduce the total execution time of analysis, if the scalar dependency analysis is not successful, the array dependency analysis will not be executed, and the loop will not be considered for parallelization. After we apply dependency analysis, a loop is considered for parallelization if for all scalar and array variables, it was determined that: (i) they have no true dependencies, or (ii) they have a true dependency, but are a `reduction` operation, or (iii) they have a false dependency so that it can be resolved by loop-private variables.

3.3.1 dependence analyzer for scalar variables. To perform dependency analysis on scalar variables, first we do liveness analysis over all statements in the loop, in order to find how each reference to a scalar variable is used. Clava allows, for each statement, to extract the list of variables that were referenced, and how they were used (i.e., Read, Write or ReadWrite). By applying this process to all statements, we can create an access pattern for each variable, e.g., `RWRRRRRRR`. This pattern can then be compressed by removing consecutive repetitions from it, e.g., `RWR`, and based on this usage pattern, we identify the data dependencies of the scalar variables. At a later step, we re-use these patterns to classify each variable into the proper OpenMP scoping, e.g, if a variable has the pattern `R`, it can be set as `firstprivate`. Variable `reduction` detection is handled by a pattern matching algorithm created to conform to the rules specified by OpenMP.

3.3.2 dependence analyzer for arrays. 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. The main goal of an array dependency analysis is to find the cross-iteration distance vectors for each array reference. There are already several works that use array subscripts to determine if different loop iterations are independent or not (e.g., GCD (Greatest Common Divisor)[5], Extended GCD [17], Banerjee [5], Omega [20, 21]). Our proposed approach, `AutoPar-Clava` uses the Petit [14], a free research tool for analyzing array data dependences, developed by the same team of Omega library project[2]. Petit accepts as input, code for a loop written in a language similar to Fortan 77. In turn, it outputs information about each dependency in the loop: (i) the type of dependency (i.e. flow, anti or output), (ii) the location (i.e. code line and corresponding variable name for the source and the destination of the dependency), (iii) the distance vector. Our two major challenges were 1) to provide the correct input to Petit, and 2) to interpret its output. For `AutoPar-Clava` we developed a translation layer that converts C loops into the code that Petit requires, and a parser that extracts the information we need from Petit output. However, there were challenges that we could not solve, mainly Petit's memory usage, which crashes for very large loops

3.4 `AutoPar-Clava` Library

`AutoPar-Clava` performs the steps shown in Algorithms 1 and 2.

Algorithm 1: AutoPar

Input : C code
Output : C code with OpenMP pragmas

Load input C program
Generate Clava AST
forall *loop* **in** C file **do**
  if *loop* has *OpenMP canonical loop form* **then**
    Mark *loop* as a candidate loop for parallelization
  end
end
foreach candidated *loop* **do**
  forall function *call* **within** *loop* body **do**
    Apply Clava `inline` action to function calls
  end
  Call *loop*-Parallelization function (Algorithm 2)
end
return parallelized version of C input program annotated by OpenMP directives

[2] http://www.cs.umd.edu/projects/omega/

Algorithm 2: *Loop*-Parallelization

Input : candidate *loop*
Output: parallelized *loop*

```
if loop contains unparallelizable function call then
    Skip parallelization process
    return loop without OpenMP directives
end
Perform liveness analysis for all statements in loop body
Build the usage pattern for scalar and array variables
Call dependency analyzer for all variables
Categorize variables into OpenMP variable classes according
  to their usage pattern
Insert the OpenMP directive and its corresponding variables if
  no data dependencies, data conflicts and race conditions are
  found
return parallelized version of loop, annotated by OpenMP
  directives
```

The main idea is to find the dependencies of all scalar and array variables inside the loop candidates, and minimize the scope of the variables inside the parallel loop as much as possible, based on their usage pattern. Finally, if no dependencies are detected, the parallelization process will add OpenMP directives to the loop.

4 EXPERIMENTAL EVALUATION

In this section we describe the evaluation of our proposed approach, regarding the effectiveness of the detection of parallel loops, e.g., how many loops are found.

4.1 Comparison with previous approaches

Taking into account that Clava AutoPar performs automatic static parallelization over unmodified source-code, we consider that among the tools presented on Section 2, the ones closest to AutoPar are ROSE, the Intel ICC compiler and TRACO. For the ROSE compiler, we used the newest available VM[3], which has Ubuntu 16.04 (Xenial Xerus) with ROSE using the EDG 4.12 frontend. As part of the ROSE compiler, autoPar tools can automatically insert OpenMP pragmas in C/C++ codes. The autoPar's version installed in the VM is v0.9.7.188. For Intel ICC compiler, we used the version 18.0.0, with the free student license. TRACO compiler was downloaded from the public svn repository[4] provided by its development team.

Since that our target in this paper is providing automatic parallelization tools as a source-to-source transformation without any changes in terms of loop structure such as loop tiling, among all auto parallelization tools presented on Section 2, we only chose ROSE, intel ICC compiler and TRACO.

4.2 Experimental setup

In this section we summarize our experimental setup, and provide details of the platform and benchmark used throughout the evaluation.

4.2.1 Platform. We evaluated the benchmarks on a desktop PC machine with an Intel Core i5-6260U processor running at 1.80GHz, 16 GB of RAM, under Ubuntu 17.10 64bits as operating system.

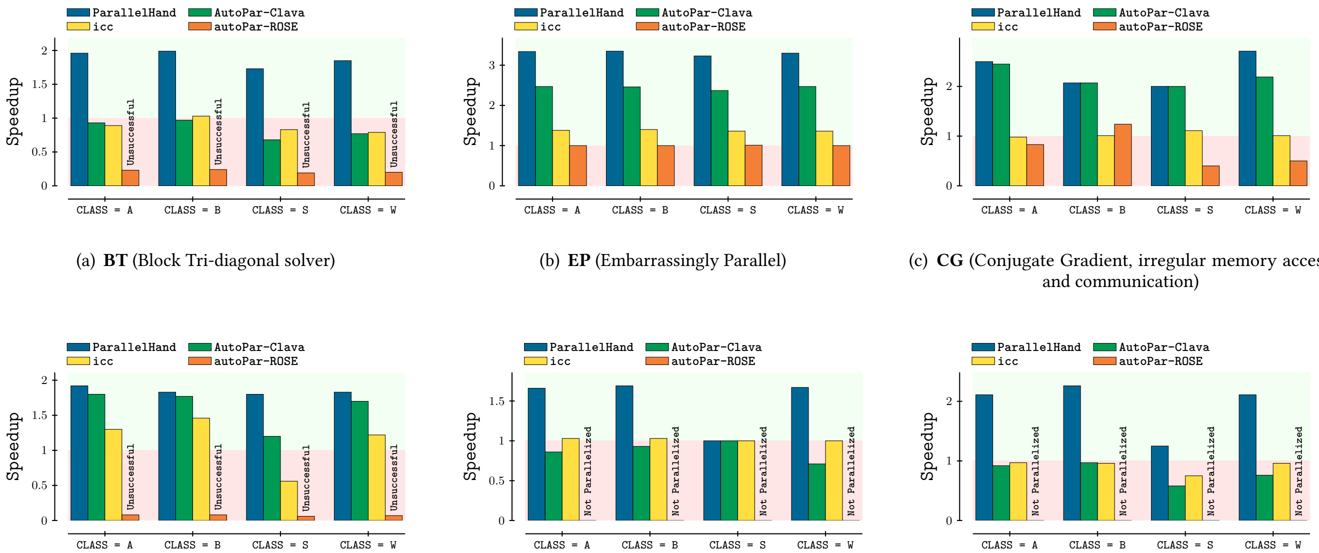
4.2.2 Benchmark. For our evaluation, we select the NAS Parallel Benchmarks (NPB)[5], which provides both serial and manually parallelized OpenMP version for each benchmark. More specifically, we have used the SNU NPB Suite[6] which is a C and OpenMP C[25] implementation of the original NPB v3.3, which is in FORTRAN. This version is provided by the Center for Manycore Programming, of Seoul National University. For the NPB benchmarks, we used four input classes, namely *S*, *W*, *A*, and *B*. Class *S* is the smallest input, class *B* is the largest one, and classes *W* and *A* are medium size inputs for a single machine. From the 8 programs available in NPB, for our evaluation we selected six: *BT* (Block Tri-diagonal solver), *CG* (Conjugate Gradient, irregular memory access and communication), *EP* (Embarrassingly Parallel), *SP* (Scalar Penta-diagonal solver), *IS* (Integer Sort, random memory access), and *LU* (Lower-Upper gauss-seidel solver).

Additionally, there are two other benchmark in NPB which we did not consider for our evaluation. For *UA* (Unstructured Adaptive mesh, dynamic and irregular memory access) benchmark, the parallel hand version did not show any improvement over the sequential code in our experiment. And, due to space limitation for this paper, we did not present the results of *MG* (Multi-Grid on a sequence of meshes, long-and short-distance communication, memory intensive) which are similar to `LU` benchmark, i.e., both parallelized code generated by `AutoPar-Clava` and `icc`, increase the execution time compared to sequential code.

4.2.3 Methodology. We evaluated four automatic parallelization approaches: (1) manual parallelization by an expert (*ParallelHand*), (2) our proposed approach (*AutoPar-Clava*), (3) auto parallelization using the Intel ICC compiler (*icc*) and (4) auto parallelization using the autoPar tool from the ROSE compiler framework (*autoPar-ROSE*). For both the sequential (i.e., original serial code) and the parallel OpenMP (i.e., parallelized with tools and manually) versions, we used the `gcc`[7] compiler v7.2.0 with flags `-g -O3 -mcmodel=medium`. The flag `-fopenmp` was also used with the OpenMP versions. For each NPB benchmark, each experiment was repeated 15 times and the average execution time was recorded. Also, for Intel `icc` compiler, we used the `-parallel` flag to enable the auto-parallelizer to generate multi-threaded code.

4.2.4 Comparison metric. To compare and discuss our experimental results, we have measured execution time, and then calculate the speedup of each approach. Speedup was 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 tool to detect parallelism, we compared the number of parallel loops in the target code obtained by each approach. However, the number of parallel loops can be a misleading metric, since some loops are more critical to the running time of a program than others [13].

[3]https://en.wikibooks.org/wiki/ROSE_Compiler_Framework/Virtual_Machine_Image
[4]https://sourceforge.net/projects/traco/
[5]https://www.nas.nasa.gov/publications/npb.html
[6]http://aces.snu.ac.kr/software/snu-npb/
[7]https://gcc.gnu.org



(a) **BT** (Block Tri-diagonal solver)

(b) **EP** (Embarrassingly Parallel)

(c) **CG** (Conjugate Gradient, irregular memory access and communication)

(d) **SP** (Scalar Penta-diagonal solver)

(e) **IS** (Integer Sort, random memory access)

(f) **LU** (Lower-Upper gauss-seidel solver)

Figure 2: Speedups of the NAS benchmarks obtained by `ParallelHand`, `icc`, `ROSE` and `AutoPar-Clava`

4.3 Results

Figure 2 demonstrates the speedup of the parallelized versions over the sequential version, for each approach. Each graph of the figure contains a red zone that represents slowdowns (i.e., speedups below 1). We consider that values above the red zone are considered as the *safe zone*. Among the selected tools, TRACO could not parallelize any of the NAS programs (it failed during execution), and was not included in the figure.

The manually parallelized OpenMP versions achieved a speedup between 1× and 3×, considering all programs and input sizes. The highest speedup was achieved by the program *EP*. This is not surprising, since this hand-parallelized version has several source-code modifications besides the OpenMP pragmas, such as using thread-local arrays to save and collect temporary results.

The Intel ICC compiler has the best performance of the tools we tested, and generally has very consistent performance. ICC uses heuristics that usually choose the correct loops to parallelize.

For the ROSE compiler, we had several problems. Among six programs, `autoPar` could not parallelize two, *IS* and *LU* (labeled in Figure 2 as *Not Parallelized*). Of the remaining four, two programs, *BT* and *SP*, did not pass the validation. This means that the tool inserted OpenMP pragmas in loops that should not have been parallelized, or that the inserted OpenMP pragmas have incorrect or incomplete clauses. For the programs that could be parallelized and that passed validation, it had consistently worse results than the other versions, except for *CG* with input class B.

`AutoPar-Clava` was able to parallelize the six programs and pass the validation step. For three of the programs, *BT*, *IS* and *LU*, it had performance that was either not far, or close to ICC. For the other three programs, `AutoPar-Clava` could achieve better performance than ICC, and in two cases, *EP* and *CG*, performed significantly better. Also, for the programs *EP*, *CG* and *SP*, it has performance that is close or very close to the parallel hand version.

From our experiences, we consider that the main reason for the `AutoPar-Clava` improvements are due to the use of Clava *inline* action. Inlining function calls provides a significant improvement on the ability to detect parallelism. By examining `ROSE` source code and the output of `icc`, we noticed that they do not consider loops for parallelization when they contain function calls.

Please note that 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. Also, the *inline* action is still in an initial phase, and may not be applicable to many types of function calls (e.g., it does not support functions with multiple exit points).

It should be mentioned that for the program *BT*, we performed manual tests that indicate that `AutoPar-Clava` should be able to obtain performance close to the `ParallelHand` version. Currently this is not possible due to memory issues in Petit, which is not able to handle certain loops after function calls are inlined. We expect to solve this limitation in future work.

Table 1 shows the number of loops that each tool parallelized. For `AutoPar-Clava` we also show, between parenthesis, the total number of loops that were detected as possible to be parallelized.

For `AutoPar-Clava`, `AutoPar-ROSE` and `ParallelHand` we counted the number of loops parallelized with OpenMP in the source code. Since `icc` produces an executable binary instead of source-code, we used the flag `-qopt-report=5` to generate a report that indicates which loops were parallelized (Level 5 produces the greatest level of detail).

NAS	Parallelized loop			
Benchmark	`AutoPar-Clava`	`icc`	`AutoPar-ROSE`	`ParallelHand`
BT	17/(28)	12	115	30
EP	2/(6)	1	3	1
CG	20/(21)	9	24	18
SP	20/(29)	33	185	34
IS	3/(4)	2	-	11
LU	29/(38)	17	-	33

Table 1: Number of parallelized loops

Intel `icc` uses an internal cost model to decide if a loop should be parallelized or not. In some cases (i.e., *BT*, *CG*, *IS* and *LU*), the number of parallelized loops is considerably lower than the number of loops that were parallelized by hand. They correspond to the cases where `icc` got a speedup close to 1× (see Figure 2).

The opposite happens in `autoPar-ROSE`. The tool tries to parallelize as many loops as possible, even when they are inside an already parallelized loop. This kind of nested parallelism can cause undesired overhead due to oversubscription, and it could explain the overall performance of ROSE.

In this work `AutoPar-Clava` uses a strategy that only parallelizes the outermost loops. The number of parallelized loops are higher than `icc` for all programs, except *SP*, but are mostly less than the number of loops parallelized by hand.

In summary, we consider that the AutoPar library for Clava shows high effectiveness regarding loop detection, when compared with other similar tools, and is capable of generating code that achieves performance that is generally comparable with other tools, and in some cases, close to versions parallelized by hand by an expert.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented AutoPar, an automatic parallelization library for the C/C++ source-to-source compiler Clava. This library is currently capable of parallelizing C programs with OpenMP pragmas without any user intervention or guidance from an profiling phase. The library tries to preserve the original code as much as possible, and inserts OpenMP `parallel-for` directives and the necessary clauses, such as `private`, `shared`, `firstprivate`, `lastprivate`, and `reduction`. It can also use the `atomic` directive in some cases, to solve array dependencies.

In the future, we will extend our approach to recognize and classify more OpenMP directives, such as `task` or `SIMD`, which can improve code parallelization. Further work also includes testing the performance of our approach on other benchmarks, such as PolyBench[8], and improve the memory limitations we currently have regarding Petit.

6 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.

[8]http://web.cs.ucla.edu/ pouchet/software/polybench/

REFERENCES

[1] clang: a C language family frontend for LLVM. (????). http://clang.llvm.org/.
[2] 2013. Intel C++ Compiler. (2013). https://software.intel.com/en-us/c-compilers/.
[3] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. McMahon, F. Pasquier, G. Péan, and P. Villalon. 2012. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012.*
[4] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P Midkiff. 2013. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming* (2013), 1–15.
[5] Utpal Banerjee. 2007. *Loop transformations for restructuring compilers: the foundations.* Springer Science & Business Media.
[6] Cédric Bastoul. 2003. Efficient code generation for automatic parallelization and optimization. In *Proceedings of the Second international conference on Parallel and distributed computing*. IEEE Computer Society, 23–30.
[7] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2003. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 209–225.
[8] William Blume and Rudolf Eigenmann. 1994. The range test: a dependence test for symbolic, non-linear expressions. In *Supercomputing'94., Proceedings.* IEEE, 528–537.
[9] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 101–113.
[10] João MP Cardoso, Tiago Carvalho, José GF Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. 2012. LARA: an aspect-oriented programming language for embedded systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development.* ACM, 179–190.
[11] Rohit Chandra. 2001. *Parallel programming in OpenMP.* Morgan kaufmann.
[12] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer* 42, 12 (2009).
[13] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. 1994. An empirical study of precise interprocedural array analysis. *Scientific Programming* 3, 3 (1994), 255–271.
[14] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. 1996. New User Interface for Petit and Other Extensions. *User Guide* 1 (1996), 996.
[15] Minjang Kim, Nagesh B Lakshminarayana, Hyesoon Kim, and Chi-Keung Luk. 2013. SD3: An Efficient Dynamic Data-Dependence Profiling Mechanism. *IEEE Trans. Comput.* 62, 12 (2013), 2516–2530.
[16] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. 2003. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 539–553.
[17] Dror E Maydan, John L Hennessy, and Monica S Lam. 1991. Efficient and exact data dependence analysis. In *ACM SIGPLAN Notices*, Vol. 26. ACM, 1–14.
[18] Marek Palkowski and Wlodzimierz Bielecki. 2015. TRACO Parallelizing Compiler. In *Soft Computing in Computer and Information Science.* Springer, 409–421.
[19] Marek Palkowski and Wlodzimierz Bielecki. 2017. TRACO: Source-to-Source Parallelizing Compiler. *Computing and Informatics* 35, 6 (2017), 1277–1306.
[20] William Pugh. 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, 4–13.
[21] William Pugh. 1992. A practical algorithm for exact array dependence analysis. *Commun. ACM* 35, 8 (1992), 102–114.
[22] William Pugh and David Wonnacott. 1993. An exact method for analysis of value-based array data dependences. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 546–566.
[23] Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters* 10, 02n03 (2000), 215–226.
[24] Dan Quinlan, Chunhua Liao, Justin Too, Robb P Matzke, and Markus Schordan. 2012. ROSE compiler infrastructure. (2012).
[25] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on.* IEEE, 137–148.
[26] Georgios Tournavitis and Björn Franke. 2010. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Int. Conf. on Parallel Architectures and Compilation Techniques.* IEEE, 377–388.
[27] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell. 2012. SESAM/Par4All: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation. In *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools.* ACM, 9–16.
[28] Michael Wolfe. 1989. Optimizing supercomputers for supercomputers. (1989).