Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR

Nikita Kataev[0000−0002−7603−4026]

Keldysh Institute of Applied Mathematics RAS, Moscow, Russia
kaniandr@gmail.com

Abstract. The paper proposes an approach to implementation of program analysis in SAPFOR (System FOR Automated Parallelization). This is a software development suit that is focused on cost reduction of manual program parallelization. It was primarily designed to perform source-to-source transformation of a sequential program for execution on parallel architectures with distributed memory. LLVM (Low Level Virtual Machine) compiler infrastructure is used to examine a program. This paper focuses on establishing a correspondence between the properties of the program in the programming language and the properties of its low-level representation.

Keywords: Program analysis · Program parallelization · Source-to-source transformation · LLVM

1 Introduction

The main applications of program analysis are program optimization and program correctness. Program optimization may require a significant transformation of the source code of a program. In this case, it is rather difficult to estimate the quality of the generated code for a particular program in advance. This leads to the fact that compilers are forced to apply fixed sequence of optimizations to all programs, which does not always produce the desired results. Compiling for execution on parallel architectures (multiprocessors, accelerators and distributed memory systems) drastically complicates the situation. The auto-parallelization feature of modern compilers may have the opposite effect and lead to a significant slowdown of the program.

User-guided program transformation that relies on recommendation of some interactive tools is paramount to simplify the mapping of sequential programs to parallel architectures. This approach should be considered as one of the key factors in the development of SAPFOR (System FOR Automated Parallelization) [1]. Unfortunately, we did not managed to find a compiler infrastructure which supports the development of source-to-source transformation passes, provides detailed information about high-level program items (alias analysis, data dependency analysis, reduction and induction variables recognition, privatization) and allows us to compile large applications in C and Fortran. This paper is devoted to the use of the capabilities of the LLVM (Low Level Virtual Machine) [2] compiler infrastructure for program analysis in SAPFOR. The considered questions involve the interpretation of information derived from the LLVM intermediate representation (LLVM IR) and its relation to the items of the higher level language. In addition, the possibilities of analysis of the transformed LLVM IR are explored to improve the quality of the source program analysis.

The rest of the paper is organized as follows. Section 2 advocates the necessity of a new compiler architecture for the SAPFOR and determines the directions for the future enhancement of the system. Section 3 discusses open-source compiler infrastructures that most closely match our goals. Section 4 focuses on high-level representation of accessed memory locations based on programming language items. Section 5 presents sequence of LLVMs analysis and transform passes that SAPFOR's analysis uses. Section 6 discusses application of analysis techniques implemented in SAPFOR to explore the C version of the NAS Parallel Benchmarks [3]. Section 7 presents the conclusion and future work.

2 Motivation

SAPFOR (System For Automate Parallelization) [1] is a software development suit that is focused on cost reduction of manual program parallelization. It is developed in Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, with the active participation of graduate students and students of Faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. SAPFOR can be used to produce a parallel version of a program in a semi-automatic way according to DVMH [4, 5] model of the parallel programming for heterogeneous computational clusters.

SAPFOR was primarily designed to support Fortran. It was successfully applied to simplify parallelization of different applications including the NAS Parallel Benchmarks [6], programs designed for solving hydrodynamic and geophysics problems and applications from the field of laser material processing [1, 7–9].

The system is written in C/C++ and operates with a representation of the program based on abstract syntax tree used in Sage++ [10]. Sage++ is an object-oriented compiler preprocessor toolkit aimed to perform source-to-source program transformation. However, it has not been developed for a long time and the responsibility for the support of modern programming languages lies on the developers of SAPFOR. At the moment, Fortran 95 is only supported.

The system implements static and dynamic analysis techniques and relies on automatic parallelizing compiler. The system is helpful to explore the information structure of programs and to perform automatic parallelization of well-formed sequential programs. This implies that the user must prepare the program himself for parallelization, guided by the hints and results of the program analysis provided to him by SAPFOR. Thus, implicitly parallel programming is applied to the development of parallel programs.

It is important to note that a significant mutation of the program may be required. Sometimes a programmer needs to choose a more parallel but not necessarily completely equivalent algorithm. However, a lot of transformations are essential to reveal hidden parallelism in a potentially parallel code. The work [12] shows the program transformation impact on a parallelization of an application designed for laser material processing [11]. To obtain the implicitly parallel version of a program written in C99, it took about 35 simple transform passes such as variable propagation, loop-invariant code motion, loop unrolling, loop distribution and other. We should clarify that a one-to-one correspondence can be established between the operators of the source and the transformed programs and these two programs are completely equivalent.

These results were collected when we explored different approaches to automate program transformation in SAPFOR. The approach proposed in [12] involves an automatic execution of individual passes in the order specified by the user. Another approach considered in [7] supposes an automatic selection of transform passes depending on the hints identified by the SAPFOR compiler. These hints describe problems which hinder the parallelization.

The desire to reduce the cost of manual parallelization of large-scale computational applications written in C and Fortran encounters problems described in [13]. This dictated the necessity of major improvment of SAPFOR in the following directions: (a) incremental parallelization for heterogeneous clusters [14], (b) automation of program transformations, (c) improving the quality of static and dynamic analysis of programs, (d) the ability to utilize program profiling and code coverage information to determine hot-spots, (e) support for C language.

This, in turn, led to investigation of available compiler infrastructures that have ample opportunities to enable the development of SAPFOR in these areas.

3 Related Works

3.1 Cetus

The Cetus compiler infrastructure [15] is designed for the source-to-source transformation of programs and is being developed at Purdue University since 2004. The system is written in Java and at the moment Cetus can parse programs that follow ANSI C89/ISO C90 standard. ANSI/ISO C99 standard is not fully supported. For example, Cetus parser breaks when it sees a variable declaration within the for statement header.

Cetus is initially designed to support interprocedural analysis across multiple files. This gives it an advantage over standard compilers such as GCC, which compile one source file at a time. Cetus implements the basic types of analysis (data dependence analysis, induction variable recognition and substitution, reduction variable recognition, privatization, points-to analysis, alias analysis) which are necessary for program parallelization. In some situations, Cetus makes too conservative assumptions. For example, the presence of 'goto' statement in loops hinders their analysis. A significant drawback is that only the addition operation is supported for reduction variables.

Cetus does not support standard compiler options, which limits the use of build automation tools to organize code analysis and compilation. Therefore investigation of large software is not straightforward.

Cetus does not support Fortran which is one of the SAPFOR target languages. Further SAPFOR is developed in C/C ++ languages and the use of the component written in Java will cause additional difficulties in their integration. The Cetus system is being developed by a small team. Updates do not come out often. The latest version was released in February 2017.

3.2 ROSE

ROSE [16] compiler infrastructure is developed at Lawrence Livermore National Laboratory (LLNL). ROSE is an open source project to build source-to-source program transformation and analysis tools. The main supported languages are C(89/98), C++(98/11), Fortran (77/95/2003), Java, Python, Haskell. The main supported platform in Linux. The project implements a large number of analyses. A software engineering environment is provided for new tools developers.

We have used ROSE to implement a tool for semi-automatic transformation of programs written in C [12]. The transform request is specified in the form of directives placed in a source code. The tool checks preconditions and applies specified transformations.

The experience of using ROSE shows the presence of implementation issues in some libraries. A large size of the project (several million lines of code) makes it difficult to correct them by ourselves. At the same time, the authors of the system give the main preference to the binary analysis subsystem, therefore it is not known when errors may be corrected.

Testing of the system on the NAS Parallel Benchmarks (NPB) [6, 3] reveals that ROSE is not capable to process some programs (LU, FT). We attempt to take input source files, build AST (Abstract Syntax Tree), and then unparse the AST back to compilable source code. However, unhandled runtime errors occur.

3.3 OPS

OPS (Optimizing Parallelizing System) [17] is a program tool oriented for development of (a) parallelizing compilers, parallel language optimizing compilers, semi-automatic parallelizing systems; (b) electronic circuits computer-aided design systems; (c) systems of automatic design of hardware based on FPGA.

The main supported language is C, there is a limited support for Fortran. The main research direction of the OPS developers is the automatic mapping of sequential programs to systems with FPGA. Another direction is the user-guided program transformation and the search for new optimizing transformations.

OPS uses Clang [18] to parse programs and build its internal representation. The higher level of the internal representation distinguishes OPS from GCC and LLVM. High-level representation is more convenient to create interactive mode of program optimization into the compiler. Unlike Clang, it allows to perform transformations directly. The system is compatible with Clang 3.3, more recent versions are not supported yet.

Data dependence analysis, reduction variable recognition, privatization and alias analysis are implemented. Reduction variable recognition is only performed for fairly simple patterns. Calculations of maximum and minimum values are not allowed for reduction variables. Dependency analysis assumes that each array subscript must be a combination of surrounding loop indices. Variable substitution solves this problem, but the resulting program may differ significantly from the original version. However, the explicit execution of this transformation in many cases is not required for parallel execution of the program. A large number of different loop transformations are involved, but a source loop must be represented in a canonical form.

3.4 LLVM

LLVM is one of the most robust compiler infrastructures available to the research community. LLVM generates highly-optimized code for a variety of architectures. Its open-source distribution and continuous updates make it attractive. LLVM consists of a number of subprojects, many of them are being used in production. LLVM operates on its own low-level code representation known as the LLVM intermediate representation (LLVM IR). Unlike GCC, LLVM provides a friendly API for designing analysis and transform passes. LLVM is currently written using C++ 11 conforming code. The LLVM libraries are well documented.

One of the most important LLVM features is language-independent type system that can be used to implement data types and operations from high-level languages. LLVM IR does not represent high-level language features directly. Nevertheless, in general it includes enough meta information to utilize analysis results to evaluate a program in a higher level language.

The release in 2017 of the Fortran language front-end Flang [19] as well as the opportunity to obtain directly the LLVM IR for Fortran became an additional motivation for using LLVM to analyze programs in SAPFOR.

4 Representation of Analysis Results

The main interest for us is to describe the memory accesses in the program. It includes data dependence analysis, induction and reduction variable recognition, privatization. Moreover it is necessary to determine whether or not two pointers ever can point to the same object in the memory (alias analysis). All information provided by SAPFOR must be attached to the items of the source program. Lower level of LLVM IR does not directly allow LLVM to be applicable for the description of analysis results.

To achieve this goal, a novel structure is presented. We call it a source-level alias tree. It depicts the structure of accessed memory using source-level debug information. Set of memory locations forms a node of the tree. A memory location specifies a correspondence between the set of IR-level memory locations and some item in higher level programming language.

Each memory location is identified by the address of the start of the location and its size. Note, that sometimes its size can only be known at runtime. LLVM is a load/store architecture. It means that programs transfer values between registers and memory solely via load and store operations using typed pointers [2]. So that, there are no implicit accesses to the memory. Hence, an address can be specified with a sequence of LLVM instructions. At the source-level the distinct elements of arrays are collapsed into one object, if it can not be expressed as a base pointer plus a constant offset. However, members of a structure are distinguished. A special type of memory locations is introduced to summarize the unknown memory accesses when a function is called.

For example, consider directly accesses to a member `S.X` of a structure `S`. At the source level it will be represented as a single memory location. From the LLVM point of view the address of the start of the memory location will be represented as the result of the allocation instruction (for example, 'alloca') and as the instruction that calculates the address of a subelement of an aggregate data structure (for example, 'getelementptr'). Different accesses may produce different sequences of instructions. In case of `P->X`, where `P` points to `S` new memory location will be constructed. Regardless the pointer P can refer to different structures at different points in the program, all `P->X` will be represented by one memory location.

Two memory locations fall into a single node of a source-level alias tree, if they may alias. The pairwise alias analysis information provided by LLVM alias analysis passes is inspected to disambiguate memory references. Each memory location is established only once in the tree and can only refer to one node. A node may have a set of child nodes. The partitioning is done in such a way that the union of all the memory locations from the parent nodes covers the union of the memory locations from a child.

The ability to adjust its structure across the transformation of LLVM IR is a distinct advantage of the source-level alias tree. Fig. 1 (a) presents an alias tree fragment for the LLVM IR directly constructed by Clang for the function shown in Listing 1.1. Memory locations which are the dereference of different pointers fall into a common node, and thus may-alias relation is conservatively assumed. This relation is caused not only by mapping of separate low-level memory locations to the item of the source program. This confirmed by the investigation of the information produced by LLVM about alias sets that are active in the function `foo`.

Fig. 1 (b) contains the alias tree fragment constructed after memory references have been promoted to be register references. Accesses to structure members are now performed directly through IR-values associated with pointers A and B. Despite that source-level alias-tree also provides information about the items before mutation. The relation between P and IR-values associated with A and B is established so that it disambiguates different accesses to P in a source program. This information will be used to refine results of analyses. Transformed

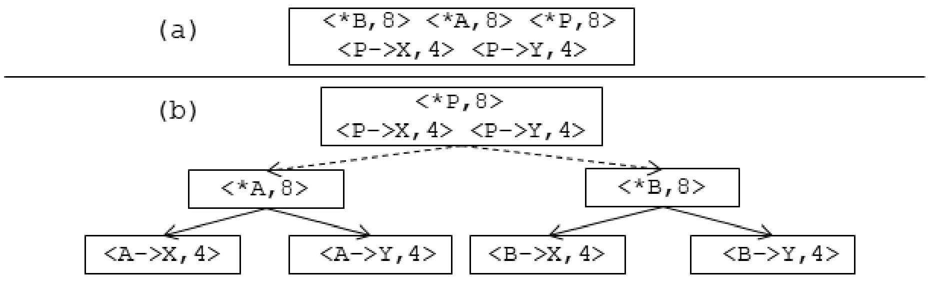

Fig. 1. Fragment of a source-level alias tree before (a) and after (b) transformation of LLVM IR for a function in Listing 1.1.

locations are stored in the alias tree node of a special kind. Dotted lines indicate that the union of memory locations from a given node is not required to cover the union of memory locations from descendant nodes. May-alias relation is assumed instead.

Listing 1.1. Source code for a source-level alias tree in Fig. 1.

```
struct S {float X; float Y;};
void foo(struct S * restrict A, struct S * restrict B) {
  struct S *P;
  P = A;
  P->X = 0;
  P->Y = 0;
  P = B;
  P->X = 0;
  P->Y = 0;
}
```

The capability to consider a memory location in conjunction with related high-level items is another important feature of the source-level alias tree. All analyses referred to at the beginning of this section are initially performed for memory locations which are explicitly mentioned in a source code. After that the obtained results are propagated to surrounded locations. Suppose `A->X` is accessed in a loop and is recognized as a private variable. If at the same time `A->Y` is recognized as a shared variable then the entire structure should be specified as a first private. Alias tree reflects relation between structure and its members and simplifies this investigation.

Another situation arises if several memory locations are explicitly mentioned in a loop and attached to a single node of the alias tree (or there is a path between them in the tree). If these locations correspond to different items in the original program, then a data dependence should be assumed. However, there is no dependency in another loop if only one of these locations is explicitly accessed.

5 Implementation Details

We have implemented the function passes to construct a source-level alias tree. The '-g' option should be specified when LLVM IR is emitted to generate source level debug information. LLVM 4.0 is currently supported. The debug intrinsics 'llvm.dbg.declarae' and 'llvm.dbg.value' are used to determine the relation between LLVM variables and source language variables. Value handles enable to track address of IR-level memory locations across RAUW (Replace All Uses With) operations. Similar handlers have been implemented to track memory locations across rebuilding of the alias tree after IR transformation.

LLVM implements reduction and induction variable recognition technics which are based on the scalar evolution pass. Dependence analysis pass is applied to detect dependences between memory accesses. As noted in the LLVM documentation, it implements the tests described in [20]. Actually these tests satisfy SAPFOR analysis goals. If necessary, the set of tests may be extended in the future. In addition, for scalar privatization a function pass has been written. This pass implements the approach described in [21]. We extended the mentioned approach to enable SAPFOR to analyze statements that do pointer accesses.

The static analysis in SAPFOR uses a sequence of passes discussed below. We divide this sequence into several steps. The first step consists of simple transform passes which simplify the analysis, but they do not require special efforts to maintain the correspondence between LLVM IR and higher level program items. These transformations available in LLVM are implemented by the following passes: unreachable code elimination, removal of declarations of unused functions, elimination of unreachable internal globals, propagation of function attributes and other passes. The next step builds the source-level alias tree and performs variable privatization. Subsequent transform passes may destroy some variables and debugging information. To avoid loss of source-level data dependencies, we perform the preliminary analysis as mentioned above.

After that, the SROA (Scalar Replacement of Aggregates) pass is executed. It breaks up 'alloca' instructions of aggregate type (structure or array) into individual alloca instructions for each member, if possible. Then, if possible, it transforms the individual 'alloca' instructions into SSA (Static Single Assignment) form. Debugging information is modified in accordance with the transformation performed and it allows SAPFOR to determine which variables in the source code correspond to registers.

At the following step a previously constructed source-level alias tree is updated. Then, induction and reduction variable recognition is performed, privatization pass is re-executed and appropriate information is updated. Promotion of memory references realized at the previous step simplifies expressions. Hence, array subscript becomes a combination of surrounding loop indices in many cases. So that data dependencies are discovered and classified.

The next step performs loop rotation transformation. Otherwise some reductions will not be recognized due to the features of the for-loop representation. Finally, reduction recognition is repeated, and previously obtained information is updated. Analysis of other types of dependencies is not repeated, since after the loop rotation, the results of the corresponding analysis may not correspond to the properties of the original program.

6 Evaluation

The implemented analysis techniques were examined on the C versions of the NAS Parallel Benchmarks (NPB) [3]. Each benchmark consists of several files and can be investigated in two modes: (a) file-by-file analysis (b) analysis of preliminary merged files. The last one uses the ability of Clang to merge together several ASTs in order to subsequently generate a single LLVM IR for all files. We have increased the applicability of the merge action to maintain large applications, we have improved the readability of diagnostic messages, and we also have eliminated some implementation errors.

Analysis of preliminary merged files is preferable to the file-by-file analysis for several reasons (a) the possibility of interprocedural analysis (b) the possibility of applying source code transformations affecting the entire project (for example, inline expansion) (c) more detailed debugging information is available. As mentioned above, it is necessary to use debugging information to present analysis results. However, Clang does not generate metadata in some cases, for example, external declarations are ignored. The merge action solves this problem.

The time of the analysis of the merged files, as well as the size of each benchmark in the number of files and lines of code are given in Table 1. The correctness of the merge action was verified by the compilation and execution of the emitted LLVM IR. The analyzer of SAPFOR enables us to use build automation tools, such as Make. Thus the original Makefiles have been easily updated. We have replaced the compilation command with the AST generation command and the linker command with the merge and analysis command.

Table 1. The analysis time(s) of the NAS Parallel Benchmarks (NPB))

Benchmark	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA
Files	20	7	13	6	12	5	23	6	22	16
Lines	4198	1331	3202	587	1333	977	4210	1640	3550	8015
Time(s)	6.59	0.12	N/A	0.05	N/A	0.03	5.98	0.58	4.21	N/A

In some cases, merging is not possible without modifying the original versions of the programs, mainly due to the conflicts between similar names with internal linkage. These benchmarks were not analyzed, and in the time row N/A is set.

Statistics of the analysis results are presented in the Table 2. It includes the total number of loops, the number of loops containing accesses to arrays, and function calls. Reduction, induction and privatizable variables are not considered as loop-carried dependencies in this statistic and they are presented separately.

Loops with data dependencies averaged 49% (Dep.) of the total number of loops. Among all loops with dependencies, approximately 45% contain function

Table 2. The analysis statistic for the NAS Parallel Benchmarks (NPB)

Benchmark	Number Of Loops										
	Total	Array	Call	Indep.	Dep.			Priv.		Ind.	Red.
					Total	Call	Only	Total	Indep.		
BT	181	171	51	101	80	50	16	109	65	179	0
CG	47	14	6	17	30	6	2	15	1	46	8
EP	9	6	5	3	6	5	2	3	0	9	2
IS	12	11	3	4	8	3	2	3	0	12	0
LU	187	156	40	96	91	39	27	84	39	171	3
MG	81	37	22	12	69	19	14	38	2	77	1
SP	250	243	48	158	92	47	16	145	87	248	0
Total	767	638	175	391	376	169	79	407	194	742	14

calls (Dep./Call), but only in 20% of cases the dependency is caused only by the presence of a call (Dep./Only). The analysis is hampered by a large number of global data and inability to determine that parameters of a function do not alias. In general, from the total number of loops only 10% of loops contain dependencies caused only by the function calls. This suggests that the limitation for interprocedural analysis did not have a significant impact on the accuracy of the results obtained. Sometimes, the presence of dependencies is caused by the use of indirect array accesses. But in general, a large number of dependencies is more likely to indicate the need to program transformation for their parallel execution rather than the limitations for analysis capabilities. This confirms that the use of advanced analysis techniques is necessary, but it is not sufficient for program parallelization.

It is important to note that variable privatization is necessary for the parallel execution of about the half of the loops. Moreover, this is also true for loops without data dependencies. To preserve the semantics of the original program, privatizable variables should be classified, first and last private variables should be identified. This advocates the implementation of a separate pass responsible for such kind of analysis. The built-in LLVM dependency analysis at best recognizes output dependency for the privatizable scalar variables. In fact, it can be successfully applied only after memory references have been promoted to be register references. Therefore, it is not applicable for the references that have been promoted. Although, for the compiler, this approach is acceptable, but in the case of source-to-source transformation and parallelization, information should be available on all variables presented in a source program.

7 Conclusion

This paper is devoted to the use of the capabilities of the LLVM compiler infrastructure for program analysis in SAPFOR. Low-level representation is used to obtain information about the original program. Investigation of transformed LLVM IR improves the quality of the source program analysis. We present a structure called the source-level alias tree to restore original program properties after transformation. It depicts the structure of accessed memory using source-level debug information.

Source-level alias tree (a) summarizes IR-level memory locations to higher level items, (b) corresponds to a hierarchical type system of a higher level language, (c) does not directly depend on a programming language and front-end, because it uses metadata, rather than abstract syntax tree, (d) adjust its structure across the transformation of LLVM IR which does not affect the structure of the memory used in the original program, (e) provides investigation of a memory location in conjunction with alias high-level items.

The implemented analysis techniques were examined on the C versions of the NAS Parallel Benchmarks.

Future works involve the usage of scalar evolution analysis in conjunction with source-level alias tree to implement array privatization techniques. We also focused on application of LLVM based analysis to check the correctness of source-level transformations. In our opinion, source-level transformations are vital to provide program parallelization in interaction with a programmer. Implementation of dynamic analysis in order to improve the alias analysis results and the accurateness of alias tree construction is also one of our future goals.

We plan to use the proposed approach in combination with Flang to analyze Fortran programs.

Acknowledgement. The reported study was funded by the Program of the Presidium of RAS 26 "Fundamental basis for creating algorithms and software for perspective ultrahigh-performance computing".

References

1. Klinov, M.S., Krukov, V.A.: Automatic parallelization of fortran programs. Mapping to cluster (in Russian). Vestnik of Lobachevsky University of Nizhni Novgorod, **2**, pp. 128–134. Nizhni Novgorod State University Press, Nizhni Novgorod (2009)
2. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (2004)
3. Seo, S., Jo, G., Lee, J.: Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In: 2011 IEEE International Symposium on. Workload Characterization (IISWC), pp. 137-148. (2011)
4. Konovalov, N.A., Krukov, V.A, Mikhajlov, S.N., Pogrebtsov, A.A.: Fortan DVM: a Language for Portable Parallel Program Development. In: Programming and Computer Software. vol. 21, no. 1, pp. 35–38 (1995)
5. Bakhtin, V.A, Klinov, M.S., Krukov, V.A., Podderugina, N.V., Pritula, M.N., Sazanov, Yu.L.: Extension of the DVM-model of parallel programming for clusters with heterogeneous nodes (in Russian). Bulletin of South Ural State University. Series: Mathematical Modeling, Programming & Computer Software, no. 18 (277), issue 12, pp. 82–92. Publishing of the South Ural State University, Chelyabinsk (2012)

6. NAS Parallel Benchmarks, https://www.nas.nasa.gov/publications/npb.html. Last accessed 14 Apr 2018
7. Kataev, N.A., Bulanov, A.A.: Automated transformation of Fortran programs essential for their efficient parallelization through SAPFOR system (in Russian). In: Parallel Computational Technologies (PCT'2015): Proceedings of the International Scientific Conference (Ekaterinburg, Russia, March 30th - April 3rd, 2015), pp. 172-177. Chelyabinsk, Publishing of the South Ural State University (2015)
8. Kataev, N., Kolganov, A., Titov, P.: Automated parallelization of a simulation method of elastic wave propagation in media with complex 3d geometry surface on high-performance heterogeneous clusters. In: Malyshkin V. (eds) Parallel Computing Technologies. PaCT 2017. Lecture Notes in Computer Science, vol 10421, pp. 32–41. Springer, Cham (2017)
9. Bakhtin, V.A., Kataev, N.A., Klinov, M.S., Krukov, V.A., Podderugina, N.V., Pritula, M.N.: Automatic parallelization of Fortran programs to a cluster with graphic accelerators (in Russian). In: Parallel Computational Technologies (PCT'2012). Proceedings of the International Scientific Conference (Novosibirsk, Russia, March 26-30, 2012), pp. 373–379. Publishing of the South Ural State University, Chelyabinsk (2012)
10. pC++/Sage++, http://www.extreme.indiana.edu/sage/. Last accessed 14 Apr 2018
11. Niziev, V.G., Koldoba, A.V., Mirzade, F.H., Panchenko, V.Y., Poveschenko, Y.A., Popov, M.V. Numerical modeling of melting process of two-component powders in laser agglomeration (in Russian). Mathematical modeling. Vol. 23, No. 4, pp. 90–102 (2011)
12. Baranov, M.S., Ivanov, D.I., Kataev, N.A., Smirnov, A.A.: Automated parallelization of sequential C-programs on the example of two applications from the field of laser material processing. In: CEUR Workshop Proceedings 1st Russian Conference on Supercomputing Days 2015, vol. 1482, pp. 536 (2015)
13. Armstrong, B., Eigenmann, R.: Challenges in the automatic parallelization of large-scale computational applications. In: Proc. SPIE 4528, Commercial Applications for High-Performance Computing, 50 (July 27, 2001) https://doi.org/10.1117/12.434876 (2001)
14. Bakhtin, V.A., Zhukova, O.V., Kataev, N.A., Kolganov, A.S., Krukov, V.A., Podderugina, N.V., Pritula, M.N., Savitskaya, O.A., Smirnov, A.A.: Automation of software packages parallelization. In: Scientific service on the Internet. Proceedings of the international scientific conference (September 19th - 24th 2016, Novorossiysk), pp. 76–85. Keldysh Institute of Applied Mathematics RAS, Moscow (2016)
15. Lee, S. I., Johnson, T. A., Eigenmann, R.: Cetus an extensible compiler infrastructure for source-to-source transformation. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 539–553. Springer, Berlin, Heidelberg (2003)
16. ROSE compiler infrastructure, http://rosecompiler.org/. Last accessed 14 Apr 2018
17. Optimizing parallelizing system, http://ops.rsu.ru/en/about.shtml. Last accessed 14 Apr 2018
18. Clang: a C language family frontend for LLVM, https://clang.llvm.org/. Last accessed 14 Apr 2018
19. GitHub - flang-compiler/flang, https://github.com/flang-compiler/flang. Last accessed 14 Apr 2018

20. Goff, Gina and Kennedy, Ken and Tseng, Chau-Wen: Practical Dependence Testing. In: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91), pp. 15–29. ACM, New York, NY, USA (1991)
21. Tu P., Padua, D.: Automatic array privatization. In: Compiler optimizations for scalable parallel systems, Santosh Pande and Dharma P. Agrawal (eds.), pp. 247–281. Springer-Verlag New York, Inc., New York, NY, USA (2001)