Interactive Parallelization of C Programs in SAPFOR

Nikita Kataev[0000-0002-7603-4026]

Keldysh Institute of Applied Mathematic RAS Miusskaya sq., 4, 125047, Moscow, Russia
kaniandr@gmail.com

Abstract. SAPFOR (System For Automated Parallelization) is a software development suite that is focused on cost reduction of manual program parallelization. SAPFOR produces parallel programs according to the high-level DVMH parallel programming model. SAPFOR relies on an implicitly parallel programming model, so it includes an automatic parallelizing compiler. On the other hand, it allows the user to guide parallelization. The system provides the user with a set of source-to-source transformations which can be performed in an automatic way. The user may also assert some implicit program properties useful for parallelization. This paper presents the interactive subsystem of SAPFOR and discusses how it supports iterative parallelization. We advocate the use of the client-server model to organize the interaction with the user. We also present the data transfer optimization technique the automatic parallelizing compiler implements. This technique reduces communication overhead when GPU is used to execute a parallel program. We guide SAPFOR to perform semi-automatic parallelization of the NAS Parallel Benchmarks 3.3.1. The paper evaluates the performance of parallel versions that SAPFOR builds automatically and compares it with the performance of manually written versions.

Keywords: Semi-automatic Parallelization, Program Analysis, Program Transformation, Graphical User Interface, SAPFOR, DVM, LLVM

1 Introduction

Automation of parallel programming aims to simplify the time-consuming and error-prone manual development of programs for diverse parallel architectures. To achieve the best performance developers have to choose between different parallel programming models and even apply them simultaneously. Thus, the significant complication of hardware as well as the software makes the development of automation tools very much in demand. Existing research covers a wide range of approaches.

On the one hand, automatic parallelizing compilers [1–3] are the best way to parallelize a program. However, in general terms, parallel programs which are written in a manual way still outperform the results of automatic parallelization. Another approach is to impose restrictions on the use of natural constructs of sequential programming languages and to rely on additional user-defined assertions that reveal some implicit properties of a sequential program [4, 5]. Much attention is paid to tools

that examine a program being parallelized but is not able to make decisions on their own [6, 7]. These assistance tools only highlight the most important program properties but the developer is responsible for program transformation and insertion of parallel constructs.

SAPFOR (System FOR Automated Parallelization) [8] combines approaches mentioned above to reduce the cost of parallel programming and at the same time, it provides the performance gain requested by the user for the resulting programs. Essentially, SAPFOR relies on an automatic parallelizing compiler generating parallel programs in the directive-based DVMH model [9, 10]. So, the system is responsible for the exploitation of parallelism inherent in the program. At the same time, SAPFOR allows the user to guide program transformation and to provide analysis tools with hints that enable program optimization. In this semi-automatic way, a program can be prepared to further automatic parallelization. SAPFOR works on lower level LLVM IR [11] and higher level Clang AST (Abstract Syntax Tree) to analyze and transform C programs.

The paper makes the following contributions:

- The interactive subsystem of SAPFOR which gives the user control over the parallelization process and an approach to its application for semi-automatic program parallelization.
- An automatic heterogeneous compiler for well-formed sequential C programs which enables SAPFOR to generate parallel programs according to the DVMH model.
- Experimental evaluation of the implemented approaches on some C programs from the NAS Parallel Benchmarks [12, 13] and comparison of the obtained parallel programs with existing OpenMP and OpenCL versions.

The rest of the paper is organized as follows. Section 2 presents the capabilities of the interactive subsystem of SAPFOR and discusses its application to the program analysis, transformation, and parallelization. Section 3 is devoted to the automatic mapping of well-formed programs to systems with shared memory. We also consider the optimization of data transfer between the memory of CPU and memory of the accelerator. Section 4 briefly outlines the implementation details of the SAPFOR components. Section 5 focuses on the experimental evaluation of the implemented approaches and it compares the performance of generated parallel programs with existing ones. Finally, section 6 presents the conclusion and future work.

2 Interactive parallelization

SAPFOR allows the user to control the parallelization from the analysis of an original program to the construction of its parallel version. Interaction with the user in SAPFOR comprises some key aspects which determine the main capabilities of the interactive subsystem. This subsystem has to allow the user:

- to explore the information structure of the program being parallelized, and to use a suitable way to investigate its properties,
- to emphasize the program properties which cannot be determined in an automatic way but still which are essential for program parallelization,
- to guide the system through the available program transformations which are helpful to overcome parallelization issues,
- to manually parallelize some of code regions and to control decisions made by the system (for example, the choice of data distribution).

The interactive subsystem is a separate component of SAPFOR and it relies on the client-server model. The SAPFOR core subsystem, which includes analysis and transform passes, provides an interface to exchanged data with clients. It specifies a set of available requests, which the client produces, and corresponding responses, which the client has to understand, encoded in JSON. Thus, the client does not have to concern about how the core subsystem processes the request. So, the different interactive subsystems may exist at the same time. Some of them could be implemented as plugins for well-known development environments and others could be standalone tools.

At the same time, SAPFOR also provides the console application to access the main capabilities of the core subsystem. It simplifies the integration of SAPFOR with build automation tools, such as Make. For example, an existing Makefile can be modified to get instrumentation for the program before it is analyzed at runtime [14].

The interactive subsystem helps the user explore the parallelism available in a program and highlights the issues that prevent its parallelization. As SAPFOR is designed to reveal loop-level parallelism, the interactive subsystem shows the information about each loop in the program. This information summarizes properties of the program control flow and of the memory accesses inside the loop body. It also determines the form of each loop nest, because the parallelization is only possible for loops that have canonical loop form [15]. Moreover, parallelization of a whole perfect loop nest instead of a single loop may significantly increase program performance and even enables parallel execution of loops with regular loop carried data dependencies [16].

SAPFOR investigates whether any function call inside the loop has a side effect or performs I/O operations. The absence of indirect calls and the absence of multiple exits outside the loop body are also checked. To explore the mentioned control-flow issues in detail the user may request SAPFOR to draw the sequence of calls which produces them.

If spurious data dependence is found, the system shows a way to eliminate it. The system supports variable privatization, reduction, and induction variables recognition. DVMH model gives us an opportunity to implement hyperplane or pipeline parallel algorithms in a natural way. That means that SAPFOR exploits loop-level parallelism even if there are some kinds of loop carried data dependencies.

Source-level alias tree [17] is suitable to represent properties of memory accesses in detail. For instance, let us consider a source code in Table 1. The interactive subsystem draws an alias tree shown in Fig. 1.

Table 1. Example of a source code with regular data dependence

```
#pragma dvm parallel([I][J][K]) \
            tie(A[I][J][K]) across(A [1:1] [0:0] [0:0])
for (I = 1; I < NX - 1; I++)
  for (J = 1; J < NY - 1; J++)
    for (K = 1; K < NZ - 1; K++)
      A[I][J][K] = (A[I - 1][J][K] + A[I + 1][J][K]) / 2
```

If there is a loop carried data dependence between accesses to a variable, the corresponding node in the alias tree is colored in red (array A in Fig. 1). On the right we can observe dependence type and distance as well as the type of accesses which cause this dependence (read, write, function call), and the accuracy of the analysis (may or must). Spurious data dependencies and shared variables are colored in green. Induction variables and some types of privatizable variables (for instance, if there is a statement which reads a value of this variable after the loop) may require additional program transformation and are colored in white. The alias tree for a loop is a subtree for the alias tree for the entire function. It contains only variables which are accessed in the loop.

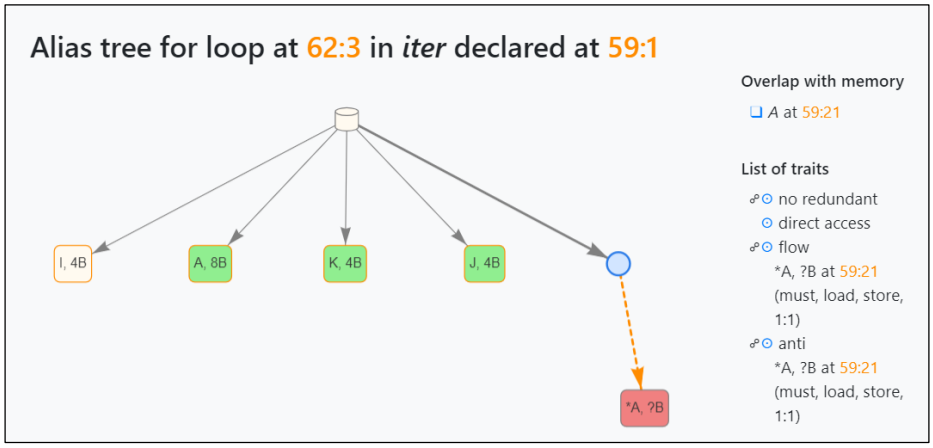

Fig. 1. Analysis results for the code in Table 1.

Data dependence in Fig. 1 has a constant size, so the loop nest in Table 1 can be parallelized in an automatic way. SAPFOR inserts corresponding annotations into a source code. The *tie* clause describes the correspondence between dimensions of the array and loops in the nest. It allows DVMH run-time system to optimize the placement of the array in memory of the accelerator.

In the presence of the dynamic analysis results, SAPFOR integrates static and dynamic data and shows the user an elaborate summary of the data-dependence analysis. Unfortunately, the lack of analysis results often affects parallel program performance.

That is why, the interactive subsystem helps the user manually describe the unknown properties of memory accesses and encode them, similarly to dynamic analysis results, in JSON.

In general, when SAPFOR is used, parallelization is an iterative process and it includes five main steps:

1. The user prepares the program to be analyzed and transformed. Firstly, some compiler options, which are necessary to parse source code, should be specified. Secondly, the user has to deal with some limitations of the current version of SAPFOR. The current implementation of the interactive subsystem is able to process a single source file at a time. Certainly, multiple header files and *include* directives can be used without restrictions. This limitation is primarily applied to the source-to-source transform passes that suffer from the inability to transform sources across different ASTs. It is not recommended placing macros inside the source code regions to be transformed. The Clang AST does not contain these constructs explicitly, so transform passes are not aware of macro definitions at an arbitrary point in a source code. We suggest replacing definitions of integer constants with enumerations and definitions of floating-point constants with const-qualified variables. Otherwise, SAPFOR would not be able to transform the program to ensure its correctness.
2. The user measures the original program performance to determine the most time-consuming program regions which should be parallelized in the first place. The performance analyzer, as a part of DVMH system, also allows the user to collect data for the original sequential program. The future version of the interactive subsystem will summarize gathered data and it will identify loops that need attention. Unfortunately, at this moment the user has to investigate the collected data himself. The second dynamic tool, which can be applied, is the dynamic analyzer, which is a part of SAPFOR. Collected data will be integrated with static analysis results in the next step.
3. The user runs the static analysis tool, which is a part of SAPFOR, and explores analysis results. The interactive subsystem allows the user to specify global analysis options and to assist SAPFOR with high-level hints.
4. The user selects regions of a source code to parallelize for the first place. If the parallelization of these regions is impossible without preliminary program transformation, the interactive subsystem highlights the main issues and offers the user to decide which transformation should be applied (inline expansion, dead code elimination, expression propagation, and others). If automatic transformation is successful, the parallelization process goes back to steps (2) or (3). Otherwise, the program should be transformed in a manual way.
5. Finally, if the previous steps have got a well-formed program suitable for automatic parallelization, the user can annotate the source-code to specify parts that should be executed in parallel. Then automatic parallelizing compiler is used to generate a parallel program. Directive-based DVMH model or OpenMP can be used. If the user does not annotate code regions, the entire program is parallelized.

3 Data Transfer Optimization

One of the key features of SAPFOR is an automatic parallelizing compiler which enables it to generate parallel code not only for multi-core CPU but also for an accelerator. This compiler inserts high-level specifications into an original program according to the directive based DVMH programming model.

SAPFOR identifies parallelizable loops and the outermost ones become the target for parallelization. As mentioned in the previous section the interactive subsystem highlights these loops and the user can see the way of further automatic compilation. However, the most complicated problem is to specify data transfer between a memory of CPU and a memory of accelerator.

A compute region is a region of a program which can be executed on accelerators. This region comprises one or more parallel loops. If the data transfer specifications were placed just after and before the compute region, communication overhead would drastically degrade the performance of the parallel program. Hence the compiler has to prepare data for the accelerator as early as possible and to request data from the accelerator as late as possible.

Data transfer optimization algorithm works as follows:

1. We analyze the data flow graph to determine input, output and local data for each compute region separately. A variable is said to be input if it has a value before the compute region and a statement in the region uses this value. A variable is said to be output if a statement defines its value inside the compute region and another statement uses this new value after the compute region. The value of a local variable is not important outside the region.
2. We use postorder traversal [18] and visit strongly connected components in a call graph.
   a. For each function we consider sibling compute regions and investigate the control flow between these regions. The compiler aims to remove data transfer specifications from any path between these regions and to place them before the first region and after the second region. This optimization is possible if the output data of the first region and the input data of the second region are not accessed on any path between these regions. If there are no computations between sibling regions the compiler also joins these regions to decrease initialization overhead at the region entry point.
   b. For each function we consider a sequential loop if its body contains compute regions. The compiler aims to move data transfer specifications outside the body of this loop.
   c. Finally, the control flow graph of the entire function is analyzed in order to move the data transfer specifications outside the function body. Necessary specifications should be placed just before and after each call to the processed function.

4 Implementation Details

The main component of SAPFOR is the core subsystem. For our work, we use LLVM to analyze the program and Clang to transform the source code. LLVM 11 is currently supported. We built a Visual Studio Code extension to implement the interactive subsystem. Fig. 2 shows a high level overview of SAPFOR.

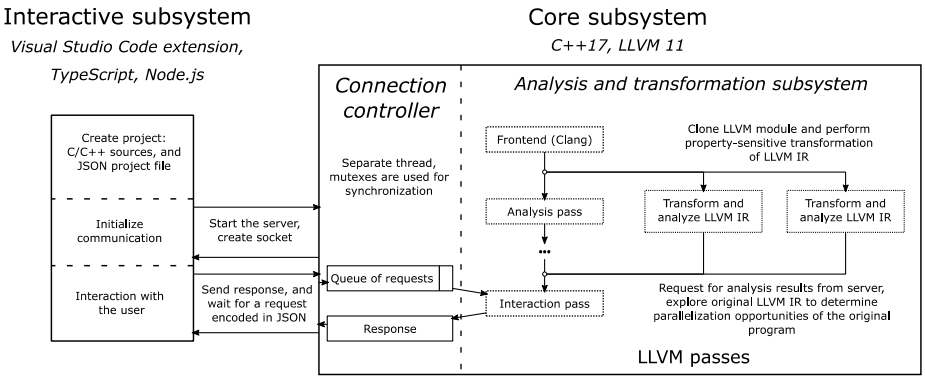

Fig. 2. A high level overview of SAPFOR

Firstly, the user creates a project which comprises program sources and a project file which includes JSON compilation database. This database specifies a way to compile translation units. Then the interactive subsystem initializes the connection with the core subsystem. We use Node.js and C++ socket libraries to send and receive messages encoded in JSON.

After the connection is established the core subsystem uses a compilation database to build Clang AST for the original program. Then it lowers Clang AST to LLVM IR. The program representation loses some language-specific information when lowering to LLVM IR level, so we maintain correspondence between Clang AST and LLVM IR. This allows us to propagate analysis results from lower level to higher level of program representation.

To improve analysis accuracy the core subsystem runs transform passes. Some of them are scalar replacement of aggregates, loop rotation, unreachable code elimination, removal of declarations of unused functions, elimination of unreachable internal globals, propagation of function attributes, instruction and control flow simplification and others. Moreover, SAPFOR makes property-sensitive transformations, i.e. it applies some sequence of transformation to analyze one kind of properties and another sequence to analyze another kind of properties. To maintain multiple transformation

sequences we clone LLVM IR and use source-level alias tree [17] to propagate analysis results from transformed LLVM IR to the original one.

5 Experimental Evaluation

We evaluate SAPFOR capabilities on three benchmarks EP (Embarrassingly Parallel), BT (Block Tri-diagonal solver), and CG (Conjugate Gradient) from the NAS Parallel Benchmarks. As hardware we use a workstation with a 6-core Intel Xeon CPU E5-1660 v2, 3.70 GHz (2 threads per core, 12 threads in total) and GeForce GTX 1660 Ti GPU. We compile programs with Intel Compiler 19.0.2.187 and CUDA tools V10.2. All programs are compiled with the -O3 optimization option.

After user-guided transformation of the code, the automatic parallelizing compiler is run. We also evaluate manually written OpenCL and OpenMP versions [12,13]. Fig. 3 demonstrates the execution time of the parallel versions and it compares results for three classes A, B, and C, which determine problem sizes and parameters.

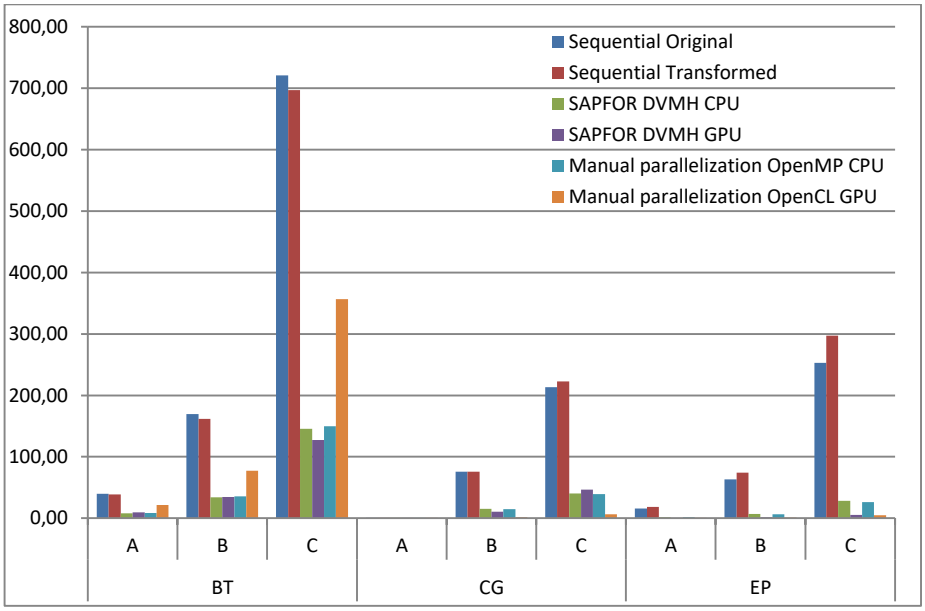

Fig. 3. The execution time(s) of the NAS Parallel Benchmarks (NPB)

The OpenCL version of CG benchmark has vectorized inner loops and uses shared memory on GPU. This explains the faster execution time on the GPU compared to the DVMH version. Despite this, a significant advantage of DVMH versions is the less complexity of maintenance. A normal compiler neglects specifications of parallelism and DVMH versions are still suitable for sequential execution. Hence, the user may use the normal C language to modify their programs.

The main transformation that is required to parallelize benchmarks is inline expansion. The current version of SAPFOR build a coarse summary to represent a memory the called function accesses, so it is not possible to disprove data dependencies in some cases. We also apply the dynamic analysis tool to reveal privatizable arrays in BT and EP benchmarks.

6 Conclusion

The paper presents the interactive subsystem of SAPFOR (System FOR Automated Parallelization) and it proposes an approach to optimizations of data transfer between CPU and accelerators. The automatic parallelizing compiler, as part of SAPFOR relies on this approach annotating source code with data transfer specifications. It uses DVMH programming model to exploit loop-level parallelism for multi-core processors and accelerators.

We implemented the interactive subsystem as an extension for the Visual Studio Code [19] editor which is available for many platforms. Moreover, it uses SSH to run extensions directly on the remote machine. Thus working directly on a parallel computing system it is possible to take all advantages of interactive parallelization.

The interactive subsystem helps the user realize parallelization issues and evaluate decisions made by SAPFOR. It does not only show analysis results, but it also allows the user to participate in parallelization and to guide SAPFOR through available program transformations. The main goal of interactive parallelization is to obtain a well-formed sequential program which can be parallelized in an automatic way.

Application of SAPFOR to the NAS Parallel Benchmarks 3.3.1 shows promising results. Automatically generated parallel versions have a similar performance to the manually parallelized ones. However, the conducted study demonstrates that SAPFOR still suffers from the inaccurate interprocedural analysis that sometimes unable to disprove data dependencies. In future work, we plan to improve analysis accuracy and extend the number of available transformations to increase the performance of parallel programs.

References

1. Grosser, T., Groesslinger, A., Lengauer, C.: Polly-performing polyhedral optimizations on a low-level intermediate representation. In: Parallel Processing Letters, 22(04), 1250010 (2012).
2. Grosser T., Hoefler, T.: Polly-ACC Transparent compilation to heterogeneous hardware In: ICS '16: Proceedings of the 2016 International Conference on Supercomputing June 2016, pp. 1–13 (2016), https://doi.org/10.1145/2925426.2926286.
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: SIGPLAN Notices, 43(6), 101–113 (2008).
4. Vandierendonck, H., Rul, S., Koen De Bosschere: The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE (2010).

5. Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A.F., Ketema, J., Absar, J., Haastregt, S., Kravets, A., Lokhmotov, A., David, R., Hajiyev, E.: Pencil: A platform-neutral compute intermediate language for accelerator programming. In: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), PACT'15, pp. 138–149, Washington, DC, USA, IEEE Computer Society (2015). https://doi.org/10.1109/PACT.2015.17.
6. Kim, M., Kim, H., Luk, C.-K.: Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming. In: 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10) (2010).
7. Intel Parallel Studio, https://software.intel.com/en-us/parallel-studio-xe, last accessed 2020/11/25.
8. Klinov, M.S., Krukov, V.A.: Avtomaticheskoe rasparallelivanie Fortran-programm. Otobrazhenie na klaster. Vestnik Nizhegorodskogo universiteta im. N.I. Lobachevskogo, (2), 128–134 (2009).
9. Konovalov, N.A., Krukov, V.A, Mikhajlov, S.N., Pogrebtsov, A.A.: Fortran DVM: a Language for Portable Parallel Program Development. In: Programming and Computer Software, 21 (1), 35–38 (1995).
10. Bakhtin, V.A., Klinov, M.S., Krukov, V.A., Podderiugina, N.V., Pritula, M.N., Sazanov, Iu.L.: Rasshirenie DVM-modeli parallelnogo programmirovaniia dlia klasterov s geterogennymi uzlami. Vestnik Iuzhno-Uralskogo gosudarstvennogo universiteta, seriia "Matematicheskoe modelirovanie i programmirovanie", 8 (277, (12), 82–92 (2012).
11. 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).
12. NAS Parallel Benchmarks, https://www.nas.nasa.gov/publications/npb.html, last accessed 2020/11/25.
13. 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).
14. Kataev, N., Smirnov, A., Zhukov A.: Dynamic data-dependence analysis in SAPFOR. In: CEUR Workshop Proceedings, 2543, 199–208 (2020), http://ceur-ws.org/Vol-2543/rpaper18.pdf.
15. OpenMP Application Programming Interface, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, last accessed 2020/11/25.
16. Kataev, N., Kolganov, A.: The experience of using DVM and SAPFOR systems in semi-automatic parallelization of an application for 3D modeling in geophysics, The Journal of Supercomputing 75, 7833–7843 (2019), https://doi.org/10.1007%2Fs11227-018-2551-y.
17. Kataev, N.: Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR. In: Voevodin V., Sobolev S. (eds) Supercomputing. RuSCDays 2018. Communications in Computer and Information Science, vol. 965, pp. 487–499. Springer, Cham (2018), https://doi.org/10.1007/978-3-030-05807-4_41.
18. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2nd edition (September 10, 2006). P. 1038, Chapter 9.
19. Visual Studio Code, https://code.visualstudio.com/, last accessed 2020/11/25.