Dynamic Discovery of Parallelisable Code

Bruce Collie, Trinity Hall

***Abstract*—The automatic discovery of parallelism in programs has been a goal of compiler engineers for many years. However, many traditional static analysis techniques fail to do so effectively. In this report I describe a dynamic approach to discovering potentially parallelisable code in C programs using an iterative approach to compilation. I provide a framework for integrating iterative compilation into a project, and show how this approach can be used to refine the structure of a C program based on dynamic analysis of the program's execution. Further to this, I discuss the limitations of my approach and how future work could provide stronger analyses and a more complex evaluation framework for executions. Finally, I give demonstrations of the tool being used on both simple example programs written for this purpose and a larger corpus of open source code, finding that the dynamic analysis is able to effectively detect potential sources of parallelism.**

Keywords*—*auto-parallelisation, optimisation, dynamic analysis

ACKNOWLEDGEMENTS

I would like to thank Dr. Björn Franke at the University of Edinburgh for his suggestion to investigate loop commutativity dynamically. All implementation and evaluation of this technique in the report is my own.

I. INTRODUCTION

An increasingly prevalent trend in computer hardware is that of parallelism. More than ever, improvements in single-threaded performance are slow in comparison to increased parallelism (in many different forms, such as GPGPU[1] computation or hyper-threaded CPU cores). However, writing concurrent code that is both performant and correct remains an open problem in software engineering. The inherent difficulty in this task arises for a number of different reasons—cognitive overhead caused by increased complexity in the programming model, difficulty in debugging nondeterministic executions, and unintuitive performance characteristics. It is for these reasons that the goal of having parallelism in programs be automatically detected and exploited is attractive. Being able to abstract away as many of the problems of writing parallel code as possible means that programmers will be able to work more effectively using modern hardware and software features.

The automatic discovery of parallel structure in code is made far more difficult by the unstructured way in which programs are usually written. The intent of a piece of code may not be easily determined from its textual representation, especially in a language with little abstraction from the hardware such as C. Some previous work aims to deliberately write programs such that they are parallel by construction (for example, the language of "Parallel Skeletons" described by Gorlatch and Cole [6] composes larger programs out of parallel primitives). In some cases, these structured approaches are optimisable for significant performance gains (an example of this is the machine-learning approach taken by Collins et al. [2] to optimising parallel skeletons).

However, these approaches are not broadly applicable to existing code. Maleki et al. [9] give an evaluation of auto-vectorising implementations in mainstream C compilers, finding that almost all of the available opportunities are not capitalised upon by the static analysis techniques used.

In this report I describe an alternative method for discovering potentially parallelisable code in C programs. Fursin, O'Boyle, and Knijnenburg [5] introduce the idea of *iterative compilation*, where a program is compiled in multiple steps with feedback applied at each step to improve the program. I present a basic implementation of the iterative compilation idiom that uses the runtime behaviour of programs to provide feedback to successive compilations. This implementation is then used (together with user feedback) to discover and annotate possible source-level parallelism in a program. I evaluate the success of this approach compared to a static analysis with similar goals.

II. DYNAMIC ANALYSIS

In this section I describe loop commutativity analysis, a dynamic method for discovering parallelisable code. I also compare this analysis to a static counterpart that aims to discover similar patterns in code (at the AST level).

A. Specification

Darlington et al. [4] describe a number of the primitive "parallel skeletons" that can be composed to implement a larger program. Perhaps the simplest of these primitives is `map`, which simply performs a given operation for every element of a collection (this could be a computation and assignment to another array, or a call to a side-effecting function). In the language of parallel skeletons, each operation must be independent of all the others such that execution of the `map` is parallelisable (or more accurately, that the result of the map is independent of the way in which its subtasks are executed). The core of a simple `map` is given in Figure 1.

The simplicity of the example in Figure 1 is somewhat deceptive. There are many possible variations on the idea that may disguise the underlying intent of the code (e.g. intermediate computations and function calls or control flow). Figure 2 gives an example in which the map structure is less obvious.

The key property that both of these examples share is that (subject to `f` being atomic and free of side effects), the order of loop execution is not important to the behaviour of the loop as

[1] General Purpose GPU

```
int f(int);

void map(int *in, int *out, size_t n) {
  for(size_t i = 0; i < n; i++) {
    in[i] = f(out[i]);
  }
}
```

Fig. 1. Simple `map` operation written in C

```
int f(int);

void map_complex(int *in, int *out, size_t n) {
  for(size_t i = 0; i < n; i++) {
    int value = in[i];

    if(f(value) > 0) {
      out[i] = 0
    } else {
      out[i] = f(value) + f(0);
    }
  }
}
```

Fig. 2. More complex `map` operation that includes control flow

a whole. That is, the loop iterations could be reordered while preserving behaviour.

This property motivates the analysis implemented as part of this project—if the iterations in a loop can be reordered without changing the program's behaviour, then there is a good chance that the loop can be parallelised. It is worth noting that this analysis cannot identify parallelisable loops with complete certainty. For example, if iterations share state, then interleaving executions can cause behavioural changes not observable only by reordering iterations in a sequential execution. This is one of the fundamental trade-offs between static and dynamic analysis, and is discussed in more depth in section IV.

B. Implementation

The primary goal of the loop commutativity analysis implemented in this project is to identify loops that could potentially be parallelised, and subsequently to reorder the iterations of these loops in an attempt to trigger changes in the program's behaviour. If there are no such changes, then there is some evidence that the loop could be parallelised (though as discussed previously, this evidence is not conclusive). In this section I describe the implementation of this analysis:

1) Clang Tooling: Clang provides a rich set of libraries for analysing and modifying source code, at both the textual and AST levels. In particular, the AST matchers framework allows for predicates on the AST to be written that match nodes with a certain structure. Components of these nodes can have names bound to them and analysed as required. The implementation of the loop reordering component of the dynamic analysis uses this framework to detect loop structures in the code that can potentially be reordered, as well as the source rewriting support in Clang to perform the modification of source files.

```
unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(ignoringParenImpCasts(
    declRefExpr(
      to(varDecl(hasType(isInteger())
    )
  )
)
```

Fig. 3. AST matcher that matches an increment of an integer variable

Idiomatically, Clang tools that analyse source code can either make changes in place to the file or output a modified file to standard output. Because of this, the loop reordering component simply replaces loops in a source file with a reordered version. In section III, a description of the tooling developed to better integrate this into a traditional development workflow is given.

2) Reordering Strategies: There are many different ways in which a particular `for` loop could be reordered. For example, iteration could be reversed, or all even iterations could be executed first. Not every loop can be reordered in every possible way—to work around this problem, I implemented several different "strategies" for reordering a loop:

Reverse: the loop iterates in reverse order. This strategy can be applied to any conventionally structured `for` loop (that is, one that has an initializer, upper bound and loop increment).

Random: if the loop initializer and upper bound are compile-time constants, then loop iterations can be randomly ordered by unrolling the loop at the source level (with an unrolled instance of the body for each loop iteration).

Odd-Even: half of the iterations are executed first, then the other half. This requires that the loop increment is exactly constant (so that the two reordered copies can each have a doubled increment).

Each of these strategies is implemented as a method that takes a Clang `ForStmt` node together with some contextual information, then returns a string representing the reordered loop. This string is then used as a direct replacement for the detected loop.

The tool accepts a command line parameter specifying the reordering strategy to be used on a source file. Each strategy specifies an AST matcher that detects loops with the appropriate structure (for example, the odd-even matcher specifies a constant increment, and the random matcher specifies compile-time constant initial and bound values).

3) Experimentation: Because the reordered loops are intended to be run experimentally (so as to determine whether or not their reordering affects the behaviour of the program), the tool only reorders a single loop at a time when it is run on a source file. If more loops were reordered, it may become difficult to determine which reordering caused a behavioural change. By default, the first detected loop for a given strategy is reordered and all others are left unmodified.

To allow the tool to be used as part of an experimental process, the tool prints the file name and source line on which the reordered loop was detected to standard output.

4) Summary: To summarise, the initial component of the dynamic analysis method implemented by this project is a command line utility that when passed the name of a source file and a reordering strategy, will attempt to detect a reorderable loop in the source file. If one is found, the reordering strategy is applied, modifying the file in place. The source location of the reordered loop is printed to standard output.

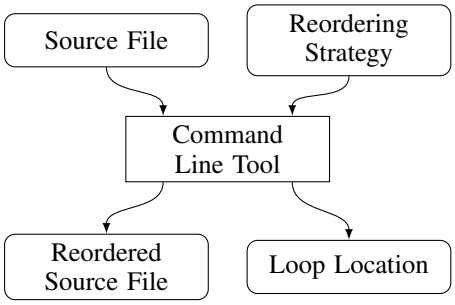

Fig. 4. Data flow in the loop reordering component

C. Static Analysis

As well as the dynamic analysis described in subsection II-B, I investigated the implementation of a static method of performing a similar analysis. A comparison of the two methods is given in subsection IV-C. As with the dynamic analysis described previously, the Clang AST matching framework was used to match structures within the program's AST.

In order to statically detect loops that are structured as a `map`, I wrote AST matchers that matched `for` loops with a particular structure. The basic structural criteria for loops that could possibly be parallelised were:

- Must initialize a variable to a constant value in the loop initializer (with or without declaring it).
- Must check whether that variable is less than a constant value in the loop condition.
- Must increment the loop variable (pre- or postfix).

These criteria are in fact overly strict, but they characterise those loops that iterate over a fixed range in increasing order. It is possible that a loop intended to act as a `map` operation iterates in a different way, but in the interests of keeping the AST matching code tractable, more complex matching strategies were not used.

Once a loop with the correct iteration structure has been identified, its body must be analysed separately to determine whether or not it performs a mapping operation. Unfortunately, characterising an arbitrary AST fragment as having a specific intent is not an easy problem—the AST matching idiom only allows for exact (modulo "implicit" components such as casts or parentheses) matches to be detected. As a result, each possible behaviour that a loop could have must be encoded as an AST matcher. In addition, further analysis is likely to be needed (for example, to ensure that the loop iteration structure is not altered inside the loop body).

For these reasons, and as it was not the primary focus of this project, only a limited set of loop body AST matchers and analyses were implemented:

- Assignment to an element of an array, where the value being assigned is an expression that includes a reference to the loop iteration variable. Additionally, a check that the array is not assigned to by other code in the body.
- Check to ensure that neither the loop bound (if it is a variable) nor the loop iteration variable are assigned to.

III. Iterative Compilation Framework

In this section I describe the framework I have implemented to integrate the dynamic analysis described in section II into a project written in C.

A. Background

A modern C compiler implements a multi-stage pipeline that transforms source code into an executable format. Figure 5 shows a simplified version of this pipeline—source code is parsed into a tree structure, which is then compiled into an intermediate representation, and finally lowered into target code. Each stage of the pipeline is independent of the results of future stages, and the data flow is unidirectional.

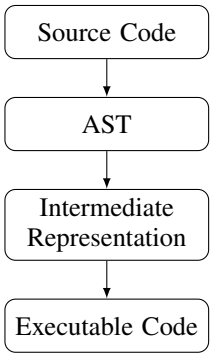

Fig. 5. Traditional compiler pipeline (simplified)

In contrast, iterative compilation allows data to flow "backwards" through the pipeline. Output from a stage can inform the action of a previous stage (on the next compilation of the program), as shown in Figure 6. In the context of this project, the feedback takes the form of loop reorderings as generated by the command line tool described previously.

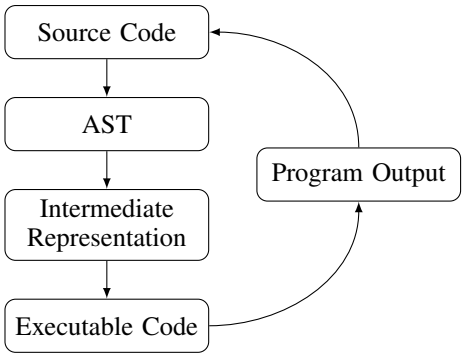

Fig. 6. Iterative compiler pipeline (simplified)

Compilation using an iterative compiler is likely to take a far greater time than compilation using a traditional compiler (as time to compile depends not only on the static complexity of the program, but on the execution time as well). Fursin, O'Boyle, and Knijnenburg [5] suggest that their iterative strategy is well suited to an embedded environment, where long compile times are a worthwhile tradeoff for increased program efficiency at runtime.

B. Program Behaviour

Implementing an experimental framework around the loop reordering tool described previously requires a notion of what it means for a program's behaviour to change. This is obviously a subtle idea that will differ greatly from program to program. However, within the scope of this program, only a comparatively simple definition of program is considered—I restrict the definition of program behaviour to be the text that is printed to *standard output* by the program.

Even when only considering command line programs, this definition is likely to miss some aspects of program behaviour. For example, return code, standard error and network requests are not considered. Another consequence of this restricted definition is that programs must be deterministic in their output for a given input. If they are not, then there is no general way to determine whether a change in behaviour is intended or caused by an incorrect loop reordering.

C. Implementation

In subsection II-B, I described the implementation of a command line tool that detected and reordered loops in a C source file. In this section I describe the implementation of the associated tooling that allows for that tool to be integrated into an iterative compilation strategy. The simplest possible usage of this tool is a manual one—a programmer would be able to use this tool to compile multiple versions of their software and evaluate the results themself. Indeed, in some cases this may be the most appropriate usage. However, in most cases it is likely to be more useful to have the iterative compilation integrated into a program's build process.

This integration takes the form of a collection of scripts and a Makefile that wrap the reordering tool and implement the iterative compilation process. A summary of the data flow through the iterative compiler is given in Figure 7.

1) Reordering: A wrapper script for the reordering tool was written to automatically generate reordered source files given a list of reordering strategies at the command line. This wrapper is also responsible for making copies of the original source file so that changes are not destructive. The source location output from the reordering tool is forwarded back onto standard output (if all strategies report the same location).

2) Makefile: A standard Unix Makefile was chosen as the build tool for this part of the project. The Makefile is broadly similar to a standard "generic" C Makefile (allowing for compilation of `.c` files into `.o` files which are then linked into an executable). However, it also specifies one source file in the project as an "experiment". This file is passed to the reordering script described above to generate several reordered versions. The application is then compiled several times, each time with a differently reordered version of the experimental source file.

3) Macros: The interface between the programmer and the iterative compilation process is three C macros: `MAYBE_PAR_FOR`, `NO_PAR_FOR` and `PAR_FOR`. Each of these replaces the keyword `for` in a C program, and indicate whether or not a loop can be parallelised. The default definition for all three macros is the same:

```
#define *_PAR_FOR(x) for(x,0)
```

This definition preserves the loop behaviour if the program is recompiled with annotations. The effect of `,0` is to change the AST of the loop so that it is not detected by the reordering tool after it has been annotated (such that another loop is detected next time round).

Eventually, the programmer should redefine `PAR_FOR` so that the loop is parallelised appropriately. For example, if OpenMP [3] was the parallelisation strategy chosen:

```
#define PAR_FOR(x) _Pragma("omp parallel for") for(x)
```

4) Annotation: The annotation script takes a command line parameter ('y', 'n' or 'm'), a file and a line number. It then replaces the keyword `for` with the appropriate macro on that line in the file.

5) Running: Finally, the iterative feedback to the compilation process is automatically generated by a runner script. Given a list of strategies and an executable name (as generated by the Makefile described above), this script executes the compiled executable for each strategy. The output of the unmodified executable is stored. If any of the reordered executables produce different output, then the experiment has shown that the loop in question cannot be parallelised. The annotation script can then be called to add a macro in the correct place in the source file.

IV. Evaluation

In this section I describe the methodology by which the project was evaluated. I give the results of this evaluation, as well as a summary of ways in which the project could be improved in future work.

A. Strategy

The framework was evaluated against both example code constructed solely for the purposes of evaluation and a larger body of existing code. This body of existing code was chosen to be the GNU Scientific Library (GSL) 2.2.1 [7] for the following reasons:

- There is a large amount of code available for analysis. The `cloc` utility [1] reports a total of 204,189 lines of C code (not including headers, blank lines or comments).
- The code is largely concerned with implementing numerical algorithms, and so is likely to contain loops that are potentially parallelisable.
- GSL is equipped with a comprehensive unit test suite that can be used to evaluate program correctness.

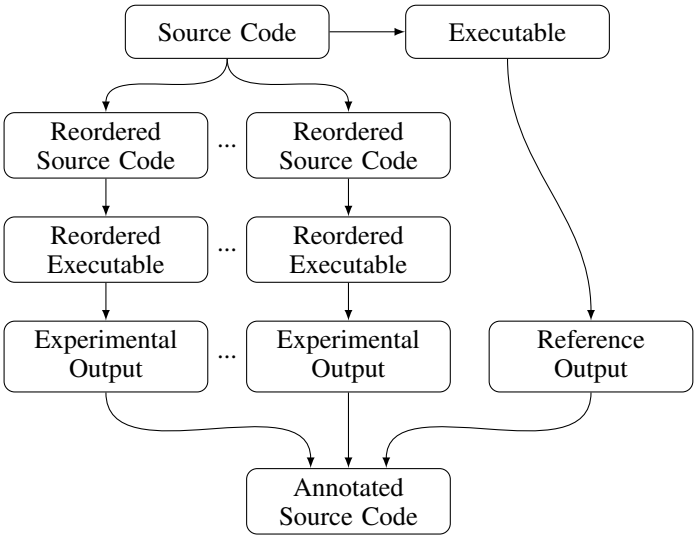

Fig. 7. Data flow through one step of the iterative compiler tooling. The annotated source code produced at the end of this process can be used as the input to the next step.

Because the GSL is a large project with its own complex build system, integrating the automated tools described in section III was not practical. Instead, my evaluation was performed by using the unit project's own unit tests to determine the correctness of a loop reordering.

B. Examples of Dynamic Analysis Results

In this subsection I provide some example results from the dynamic tool being run on code written specifically for this purpose. The test application is comprised of a small library that exposes three functions (randomly filling an array with integers, calculating the sum of an array, and copying cumulative sums over an array into another array), together with a small executable harness. The declarations of these functions are given in Figure 8.

```
void random_fill(long *arr, long n);
long sum(long *arr, long n);
void csum(long *in, long *out, long n);
```

Fig. 8. Function definitions from the test library

The implementation of each of these functions was then analysed using the iterative compilation framework described previously. The results are given below, and serve as a useful indication of the usefulness and shortcomings of the analysis.

```
for(long i = 0; i < n; i++) {
  arr[i] = rand() % FILL_RANGE;
}
```

Fig. 9. Implementation of random array filling

1) Random Fill: The implementation of the random fill function is given in Figure 9. The random number generator was seeded deterministically between runs so that each execution would generate the same sequence of random numbers. Running the loop reordering analysis on this loop reports it as *not* being parallelisable. This is expected—even though the calls to `rand()` generate the same sequence of random numbers, they will be placed at different locations when the loop is reordered.

```
long sum = 0;
for(long i = 0; i < n; i++) {
  sum += arr[i];
}
```

Fig. 10. Implementation of array summation function

2) Sum: The implementation of the sum function is given in Figure 10. When this loop was analysed, the tool reports it as possibly being parallelisable (as addition is commutative, modulo concerns such as integer overflow). However, this serves as a good example of the shortcomings of this analysis. The loop is not parallelisable because the addition and assignment is not an atomic operation, but non-atomicity cannot be triggered by any sequential reordering of the loop. The programmer must still be able to recognize the potential flaw in this program.

```
for(long i = 0; i < n; i++) {
  out[i] = sum(in, i);
}
```

Fig. 11. Implementation of array cumulative summation function

3) Cumulative Sum: The implementation of the cumulative sum function is given in Figure 11. As with the sum function, the tool reports this loop as being potentially parallelisable. However, in this case the loop is in fact parallelisable (as the input array is unmodified, each call to sum is independent, and each iteration of the loop only writes to a single location in the output array).

C. Comparison with Static Analysis

To compare the usefulness of static and dynamic analysis, I ran both tools on the source code of a GSL constituent library. In this section I give the results obtained, and a comparison between the two analyses.

The `integration` sub-library of the GSL provides code that performs numerical integration of various kinds. `cloc` reports it as containing a total of 7981 lines of code over 38 source files.

Running the static analysis tool on the source code found a total of 16 parallelisable loop structures within the library (based on the criteria described in subsection II-C). In contrast, the dynamic analysis tool finds 97 loops that could potentially be parallelised. A manual evaluation of 20 of these loops found 8 that were parallelisable, and 12 that were not (evaluated by the GSL unit tests as described previously).

These results emphasise the difference between the two methods—the dynamic analysis is able to find more possible

avenues for parallelisation at the expense of more developer time, while the static analysis has a narrower scope but requires less developer time and is more amenable to automation.

D. Possible Improvements and Further Work

There are several issues with the dynamic analysis and iterative compilation framework implemented in this project. Here I give a brief summary of each, along with an indication of how the the issue might be mitigated in future work.

1) Difficult to Integrate: Integrating the iterative feedback loop into a traditional compilation model is not easy. Traditional build tools do not support a "feedback loop", and so support for this project needs to be built separately, greatly increasing the complexity of a project's build system. A more streamlined set of tools and scripts would alleviate this problem to some extent.

2) Slow: Using the tool to iteratively compile a program is slow—compilation becomes an interactive process where the programmer must provide feedback, and the program in question must be executed multiple times at each step. Unfortunately, this is the nature of dynamic analysis, and as such it is currently impractical to include as part of a development workflow. If execution time is a bottleneck, then one mitigation would be to develop small test harnesses that execute only the code in question. However, this would ential extra development time to implement.

3) False Positives: The initial loop detection is very sensitive. Little filtering is applied to the loops that are discovered, and as such there is likely to be a large proportion of these that are not in fact parallelisable. Because each loop requires the programmer to manually inspect it, this false positive rate affects the usabililty of the tool negatively. A possible partial solution to this problem is static analysis of loop bodies for patterns that can conclusively rule them out of being parallelisable (for example, a call to a function that is known to not be thread safe).

4) Limited Applicability: The limited definition of program behaviour given in subsection III-B means that the tool cannot be applied usefully to software whose primary interactions are not textual in nature. Layton [8] suggests monitoring the system calls made by the executing software in order to characterise its IO behaviour. This is a natural extension to the standard output model used in this report, but is operating-system dependent in nature. It would also be possible to write an application-specific testing script (for example, a UI automation tool) to report behavioural regressions. This approach would be the most likely to identify differences, but is the least portable between programs.

V. Conclusion

In this report I have described the implementation of a system for performing iterative compilation of a C program, with feedback from a dynamic analysis of loop reorderings. I have shown that this strategy can be used to discover sources of potential parallelism within an existing codebase, and can be integrated into new or smaller projects with a greater degree of automation. Additionally, I have given the results of applying this model to a large corpus of numerical code. I have compared this dynamic system to a static analysis with similar goals, and shown the advantages and disadvantages of each method. Finally, I have given a summary of the future work that could be done on this system in order to improve its usability.

References

[1] *AlDanial/Cloc*. URL: https://github.com/AlDanial/cloc (visited on 01/05/2017).

[2] A. Collins et al. "MaSiF: Machine Learning Guided Auto-Tuning of Parallel Skeletons". In: *20th Annual International Conference on High Performance Computing*. 20th Annual International Conference on High Performance Computing. Dec. 2013, pp. 186–195. DOI: 10.1109/HiPC.2013.6799098.

[3] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: http://dx.doi.org/10.1109/99.660313 (visited on 11/06/2016).

[4] John Darlington et al. "Parallel Skeletons for Structured Composition". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 19–28. ISBN: 978-0-89791-700-1. DOI: 10.1145/209936.209940. URL: http://doi.acm.org/10.1145/209936.209940 (visited on 11/22/2016).

[5] G. G. Fursin, Michael O'Boyle, and P. M. W. Knijnenburg. "Evaluating Iterative Compilation". In: *Languages and Compilers for Parallel Computing*. Ed. by Bill Pugh and Chau-Wen Tseng. Lecture Notes in Computer Science 2481. Springer Berlin Heidelberg, July 25, 2002, pp. 362–376. ISBN: 978-3-540-30781-5 978-3-540-31612-1. DOI: 10.1007/11596110_24. URL: http://link.springer.com/chapter/10.1007/11596110_24 (visited on 11/06/2016).

[6] Sergei Gorlatch and Murray Cole. "Parallel Skeletons". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer US, 2011, pp. 1417–1422. ISBN: 978-0-387-09765-7 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_24.

[7] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. 3rd. Network Theory Ltd., 2009. ISBN: 978-0-9546120-7-8.

[8] Jeffrey Layton. "IO Pattern Characterization of HPC Applications". In: *Proceedings of the 23rd International Conference on High Performance Computing Systems and Applications*. HPCS'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 292–303. ISBN: 978-3-642-12658-1. DOI: 10.1007/978-3-642-12659-8_22. URL: http://dx.doi.org/10.1007/978-3-642-12659-8_22 (visited on 01/05/2017).

[9] Saeed Maleki et al. “An Evaluation of Vectorizing Compilers”. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.68. URL: http://dx.doi.org/10.1109/PACT.2011.68 (visited on 11/06/2016).