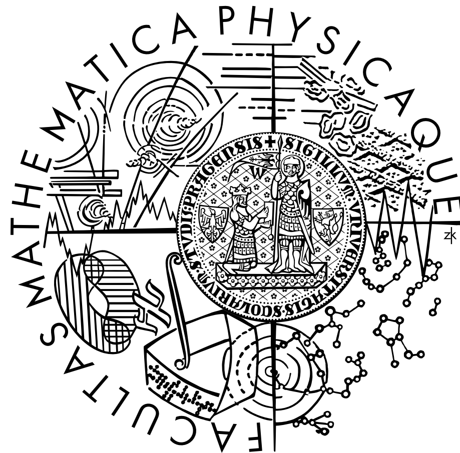
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Petr Bělohlávek

OpenMP for Java

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: JUDr. Antonín Steinhauser

Study programme: Computer Science

Specialization: General Computer Science

Prague 2015

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, May 21, 2015 signature of the author

I would like to express my gratitude to JUDr. Antonín Steinhauser, the supervisor of my thesis, for his guidance and invaluable advice. Furthermore, it would be impossible for me to finish the thesis without the support, patience and understanding of my family.

Název práce: OpenMP pro Javu
Autor: Petr Bělohlávek
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí bakalářské práce: JUDr. Antonín Steinhauser, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Vzhledem k zpětné kompatibilitě je vícevláknové programování v Javě často implementováno neefektivně a odrazuje nezkušené programátory od vývoje paralelních aplikací. Tato práce představuje omp4j - preprocesor pro Javu 6, 7 a 8, který je nezávislý na běhových knihovnách a podporuje kompaktní podmnožinu nejdůležitějších OpenMP direktiv. Vyvinutý preprocesor podporuje všechny běžně užívané JVM a jím přeložené programy jsou navíc dobře škálovatelné. Pro zajištění paralelního běhu zdrojových kódů je použita kombinace syntaktické a bytecode analýzy. Vyvinutý program lze používat dvěma způsoby - přímo z příkazové řádky nebo jako knihovnu pro Java a Scala projekty. Druhý zmíněný způsob použití je demonstrován ve formě online dema v rámci stránky projektu - `www.omp4j.org`. Práce obsahuje vypracované vyhodnocení výkonnosti, ve kterém jsou diskutovány jednotlivé aspekty preprocesoru a srovnání s podobnými projekty v C++ a Javě. Ve srovnáním s těmito projekty byla na některých měřeních u omp4j prokázána signifikantně lepší škálovatelnost na hladině významnosti $\alpha = 0.01$.

Klíčová slova: OpenMP Java Scala Preprocesor

Title: OpenMP for Java
Author: Petr Bělohlávek
Department: Department of Distributed and Dependable Systems
Supervisor: JUDr. Antonín Steinhauser, Department of Distributed and Dependable Systems

Abstract: Due to its backward compatibility, multi-threaded programming in Java is often performed ineffectively and repels inexperienced programmers from development of parallel applications. This thesis introduces omp4j - an OpenMP-like preprocessor that supports Java language standards 6, 7 and 8 without any runtime dependencies. Furthermore, the thesis develops a scalable and portable solution for all commonly used JDKs. The combination of the syntax and bytecode analyses is employed by the preprocessor in order to parallelize the source code. The developed project may be used either as a CLI utility or as a third-party library for Java and Scala projects. The latter possibility is demonstrated in an online demo which was developed together with the project website - `www.omp4j.org`. Additionally, the performance evaluation, which discusses various aspects of the implemented preprocessor, is presented and the comparisons to the related C++ and Java solutions are elaborated. In comparison to the similar projects, significantly better scalability of omp4j is proven at significance level $\alpha = 0.01$ for some of the benchmarks.

Keywords: OpenMP Java Scala Preprocessor

Contents

1 Introduction **5**

1.1 Aim of the Thesis 5

1.2 Thesis Organization 7

1.3 Related Work 8

1.3.1 JOMP 8

1.3.2 JaMP 9

1.3.3 Pyjama 10

1.3.4 JPPF 10

1.3.5 AKKA 11

2 Theory of Parallel Computation **12**

2.1 Moore's Law Limitation 12

2.2 Amdahl's Law 13

2.3 Parallel Architectures Classification 15

2.4 OpenMP 15

2.4.1 History 16

2.4.2 Fork-Join Model 16

2.4.3 Modern development 17

3 Preprocessor Architecture **19**

3.1 Level of Processing 19

3.2 Processing Work-flow 20

3.3 Code Analysis 21

3.3.1 Bytecode Analysis 22

3.3.2 Syntax Analysis 22

3.3.3 Class Hierarchy Model 23

3.4 Directive Recognition 24

3.4.1 Directive Grammar 24

3.4.2 Directive Hierarchy Model 26

3.5 Directive Translation 26

3.5.1 Phase 1: validate 27

3.5.2 Phase 2: preTranslate . . . 28
3.5.3 Phase 3: postTranslate . . . 28
3.5.4 Rewriting the Source Code . . . 29
3.6 Runtime Library . . . 29
3.6.1 Executor Interface . . . 30
3.6.2 Dynamic Executor . . . 31
3.7 Supported Directives . . . 31
3.7.1 omp parallel . . . 32
3.7.2 omp [parallel] for . . . 32
3.7.3 omp section(s) . . . 33
3.7.4 omp master . . . 34
3.7.5 omp single . . . 35
3.7.6 omp critical . . . 36
3.7.7 omp atomic . . . 37
3.7.8 omp barrier . . . 38
3.7.9 public, private, firstprivate . . . 38
3.7.10 threadNum . . . 39
3.7.11 Thread-ID Macros . . . 39
3.7.12 schedule . . . 40
3.8 Language and Coding Style . . . 41

4 Implementation 42
4.1 Version Control System . . . 42
4.2 License . . . 43
4.3 Behavioral-Driven Development . . . 43
4.4 Continuous Integration . . . 44

5 Performance Evaluation 45
5.1 Methodology . . . 45
5.1.1 Benchmark Framework . . . 45
5.1.2 Data Processing . . . 46
5.1.3 Used Hardware . . . 47
5.1.4 Thread Limitation Techniques . . . 47

5.1.5 Statistical Notations 47
5.2 Fibonacci Numbers 48
5.2.1 CPUs Dependence 49
5.2.2 Workload Dependence 50
5.2.3 Speedup Distribution 51
5.2.4 Linear Model 52
5.3 Matrix Multiplication 55
5.4 Function Maximum 56
5.5 Single & Master Overhead 57
5.6 Related Projects Comparison 58
5.6.1 GCC Comparison 59
5.6.2 JOMP Comparison 60

6 User Documentation **62**
6.1 Prerequisites 62
6.2 Preprocessor Installation 62
6.2.1 Compiled Preprocessor 63
6.2.2 Source Code 63
6.3 Installation and Invocation 65
6.4 Example Input 65

7 Project Website **67**
7.1 Website Architecture 67
7.1.1 Front-end 67
7.1.2 Back-end 68
7.2 Responsive Appearance 69
7.3 Hosting 69

Conclusion **71**

Bibliography **76**

List of Figures **77**

List of Tables **78**

Attachments **79**

Appendix A Preprocessor Work-flow **81**

Appendix B OMP Directive **82**

Appendix C IOMPExecutor **84**

Appendix D hitBarrier implementation **85**

Appendix E BSD License **86**

Appendix F Fibonacci Distribution **87**

Appendix G Simple Linear Regression **88**

Appendix H Multiple Linear Regression **88**

Appendix I Matrix Multiplication **89**

Appendix J Demo Controller **90**

Chapter 1.

Introduction

Parallel programming has increased in importance during the last the ten years more than ever before. Apart from common multicore desktop computers and servers, mobile devices such as smartphones and tablets feature multicore CPUs as well [1]. Much attention has been paid to multicore solutions since it seems to be a reasonable way of performance boosting while using as little energy as possible.

The distributed systems and cloud computing are becoming more popular and widely used as well. The current situation encourages developers to use the concurrent programming techniques in order to increase the system performance.

Therefore, a lot of technologies that implement the concurrent approach have been developed, e.g. OpenMP for C++ and Fortran, Threading Building Blocks (TBB) for C++ and AKKA for Java and Scala.

1.1 Aim of the Thesis

The initial aim of this thesis is to implement Java source code preprocessor which we denote omp4j (OpenMP for Java). It allows developers to use OpenMP-like directives for simple parallelization and enhanced scalability of their algorithms.

Secondly, an overview of related projects for Java is provided. The range of different technologies is presented (see chapter 1.3) as well as multiple employed design patterns. These solutions are compared with the original OpenMP API (described in chapter 2.4).

Finally, the thesis intends to evaluate the preprocessed code performance in terms of the total speedup and scalability (see chapter 5). The comparison with related projects is also covered.

The original thesis requirements are good portability of the preprocessor and no runtime dependencies of the translated code. The more detailed requirements are listed below.

- The preprocessor should support a basic subset of OpenMP directives including:
  - omp parallel
  - omp parallel for
  - omp for
  - omp section[s]
  - omp barrier
  - omp critical

  Furthermore, thread-id entities should be supported as well as compatible attributes including:
  - public
  - private
- The preprocessor should run on all common Java Virtual Machines (JVMs) including:
  - OpenJDK 6
  - OpenJDK 7
  - Oracle JDK 7
  - Oracle JDK 8
- The preprocessor should be able to modify all commonly used Java standards (the official notation as described by OpenMP API [2]) including:
  - Java SE 6
  - Java SE 7
  - Java SE 8
- The preprocessor should not have any runtime dependencies except for proper JDK.
- The preprocessed code should not have any runtime dependencies, i.e. no installed classes in the classpath are required.

During the development, we extended the list of requirements in order to maximize the real-life user experience. The added requirements are listed below.

- The preprocessor may be used as a Java compiler instead of regular javac. That enables easy integration into various IDEs such as IntelliJIdea, Eclipse and NetBeans.
- Apart from being used as a standalone program, the preprocessor can be used as a library that can be employed by third-party software.
- The translated code should be easily modified, thus it should differ as little as possible from the original code.

1.2 Thesis Organization

In the following section 1.3, an overview of similar Java projects is provided. Both OpenMP-like and library-based solutions, that allow a simple parallelization, are covered.

Chapter 2 consists of concurrent programming theory information that is used later on. The OpenMP project itself is described in detail and classified in terms of Flynn's taxonomy (see chapter 2.3). In addition, two important laws - Moore's (2.1) and Amdahl's (2.2) - are explained. Further, the motivation for parallel programming and its limitations are discussed.

In chapter 3, the omp4j preprocessor architecture is described. The language and used technologies are discussed as well as all supported directives. The whole preprocessing work-flow is explained including the used methods of code analysis.

The process of project development is presented in chapter 4. All important tools that were employed are listed in detail. The information about future extension of the project in terms of technical knowledge is provided.

In chapter 5, a complex performance evaluation of the output code is introduced. For this purpose, a set of benchmarks is provided. As a result of the performance evaluation, two linear models of speedup dependence (see chapter 5.2.4) are created. Finally, comparisons with similar projects are made, e.g. gcc OpenMP implementation (see chapter 5.6.1) and JOMP (see chapter 5.6.2).

Chapter 6 describes the user documentation. The ways of obtaining source code are listed together with the compilation instructions. All command line options and their effects are described. Finally, an example of an input code is attached.

In the final chapter, 7, a library-like preprocessor usage is presented. A modern website, whose back-end is written entirely in Scala, is developed in order to feature a live demo of code transformation via omp4j.

The thesis is accompanied with a digital attachment which includes the whole project, benchmarks and website. Refer to Attachment chapter for detail information about the structure of the provided data.

1.3 Related Work

OpenMP is specification of a set of directives that are used in order to provide the parallelization of the following block of code. It was originally implemented for C/C++ and Fortran. OpenMP is covered in detail in a separate chapter (see chapter 2.4) since it is only API specification. Besides, this section examines concurrent preprocessors and libraries for Java.

Since 1995, when Java was introduced by Oracle Corporation [3], various parallel frameworks and preprocessors have been developed. Because of the Java backward compatibility maintenance throughout the decades, in recent times the language itself has become rather old-fashioned and not flexible enough for modern applications. Hence, the number of libraries and other parallelizing tools, that provide easy concurrency approach, has been increasing.

1.3.1 JOMP

JOMP[1] is an academic research project that implements the OpenMP API for Java [4]. It implements a wide range of OpenMP directives including reduction. The directive comments must start with //omp without any whitespaces allowed

[1] `https://www2.epcc.ed.ac.uk/computing/research_activities/jomp`

before omp which is uncomfortable for the user. The project is divided in two separate parts - the preprocessor itself and the runtime library.

The preprocessor takes a Java source file that must be appended with a .jomp extension [5]. The output is a regular Java source file which may be compiled with any Java compiler. The main disadvantage of JOMP is that the processed code is dependent on the runtime library that must be installed in the classpath.

The library itself defines the most important class - jomp.runtime.BusyTask [6]. The source processing works as follows. Initially, a new context class, that extends previously mentioned BusyTask, is created in the file scope. The class name is prefixed with underscore in order to prevent name conflicts, however the code readability decreases.

The class contains public fields for all variables that may be used in the parallel code. These fields are initiated immediately after the class instance creation.

The parallel invocation is executed in the overridden go method of BusyTask. Hence, the code following a directive is moved from its original position into go method without being modified. [6].

The advantage of the adopted approach provides a simple source procession, however the output differs seriously from the input as the blocks of code are transferred to fundamentally distinct locations.

JOMP does not support Java SE 8. Thus, the modern language constructs such as lambdas are unable to be used. In addition, the JOMP parser often fails when parsing Java source code decorated with the annotations such as @Override, as follows from the experiments that we made.

1.3.2 JaMP

JaMP is an OpenMP implementation that is fitted into Jackal DSM that distributes Java parallel processes onto a cluster [7]. It is implemented in a similar manner to JOMP (described in chapter 1.3.1).

The main advantage of JaMP is its integration to Jackal DSN, thus potentially enhanced scalability is provided. However, the benchmarks have not been executed on more than eight nodes [7].

Apart from requirement of the runtime library, the main disadvantage is unsupported parallel for directive which leads to an inelegant work-sharing directive notation.

1.3.3 Pyjama

Pyjama intends to implement OpenMP for Java with a direct graphical user interface (GUI) support [8], i.e. in the main, the responsiveness maintenance and "allowing correct execution of GUI-related code within those regions" - [8].

Pyjama preprocessor requires .javamp source files which are processed into regular .java output files [8].

The project, as well as previously listed OpenMP implementations, requires a runtime library for correct compilation and execution of the translated code [8]. In addition to this portability limitation, nested directives are not supported and some fields, such as those marked static, remain unsupported [9]. Furthermore, the directive attributes must follow a particular order, e.g. num_threads must come as the first attribute [9].

1.3.4 JPPF

JPPF[2] is a powerful solution for distributed computing on the grid. In contrast to previously introduced projects, JPPF doesn't provide anything like OpenMP in terms of directives. Since JPPF is a library, the user must be directly aware of implementing a massively distributed algorithm and divide the workload into tasks i.e. classes implementing JPPFTask interface.

This is a fundamentally different approach to OpenMP, however, it is a commonly used alternative for concurrent computation especially when the design

[2] http://www.jppf.org

pattern supported by directives is not suitable.

1.3.5 AKKA

The project AKKA[3] is the "state-of-the-art" library for concurrent and distributed JVM computing. It supports both Java and Scala languages and is used for general purposes. A C# implementation of AKKA - AKKA.NET - was also developed for .NET framework [10].

Similarly to JPPF, the user must be aware of the application concurrency. In contrast to Fork-Join model, that is brought by OpenMP (see chapter 2.4.2), AKKA implements the Actor model which is both a major advantage and disadvantage at the same time.

Actor model extends object oriented programming, replacing "everything is an object" policy with "everything is a concurrent actor" model. Actor itself features the asynchronous message-passing with other actors (see figure 1.1 for Actor pattern demonstration).

The main advantage of this approach is possible data immutability maintenance which encourages the functional design. It leads to greater stability and scalability of the applications. On the contrary, some trivial problems become less effective when actor pattern is used instead of Fork-Join model.

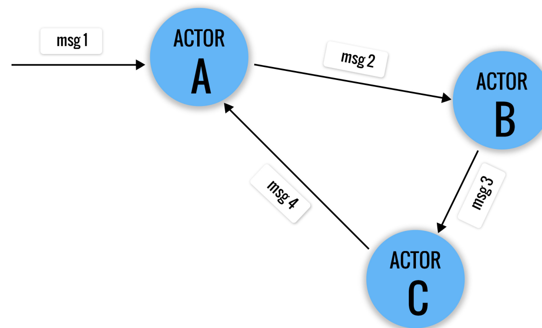


Figure 1.1: Three actors passing messages to each other

[3]http://akka.io

Chapter 2.

Theory of Parallel Computation

In this chapter presents two important theses which are highly relevant for the motivation for writing parallel programs and for their performance evaluation. A basic overview of multiprocessing architectures is provided and the one which is relevant for this thesis is highlighted. Finally, OpenMP API standard is briefly introduced together with Fork-Join model, history and usage in modern development.

2.1 Moore's Law Limitation

Throughout the last fifty years, a fundamental observation called Moore's law has held. It states that the number of transistors in integrated circuits doubles every 18 months (respectively 24) [11]. Therefore, it implies the exponential growth of the transistor number as well as exponential decrease of the physical size of the transistor.

However, due to physical limitations such as the fact that a transistor cannot be manufactured smaller that an atom, it's obvious that Moore's law is not going to hold forever. Moore himself stated in 2010 that:

> " ... In terms of size [of transistor] you can see that we're approaching the size of atoms which is a fundamental barrier ... "[12]

In 2014, Intel Co. introduced a generation of CPUs called Haswell featuring $22nm$ transistor technology. Early this year, in January 2015, a new generation of CPUs named Broadwell featuring $12nm$ technology was released. In comparison to atom size, which is c. $0.1nm$, it is only $100\times$ bigger. In April 2015, the three-atoms thick layer that may be employed as a transistor was introduced [13]. Even though the whole research project is rather experimental, it is clear that the atom-units transistor are manufacturable.

Since the limitation obviously exists, the CPUs manufacturers are motivated to seek different approaches in order to increase the performance. Frequency boosting increases both power consumption and heat production during the computation. For this reason, CPU producers almost stopped frequency development of CPUs in 2004 [14].

The previously described context suggests that a new approach to increasing performance should be used. The majority of CPU and GPU manufacturers are focusing on parallel architectures that may increase computation power.

2.2 Amdahl's Law

Parallel programming provides only limited computation power boost. Obviously, a N-core computer might theoretically provide at most N-times greater computation power than identical single-core machine. This leads to the linear upper bound of the possible speedup. However, there exists a non-trivial limitation of maximal possible speedup that is called Amdahl's law.

As Amdahl states, his rule models the expected speedup of parallelized algorithm of fixed size [15]. The following notations, definitions and equations are taken from Yaghob [16].

Definition. $T(1)$ *is the total time of the execution of the sequential algorithm implementation.*

Definition. $T(P)$ *is the total time of the execution of the parallel algorithm implementation using* P *processing elements.*

Definition. $S(P)$ *is the speedup using* P *processing elements. The formal definition is as follows:*

$$S(P) = \frac{T(1)}{T(P)}$$

In every parallel implementation there is a fraction of the code that cannot run concurrently. Usually it contains I/O operations such as reading a file, scanning the network or writing the output.

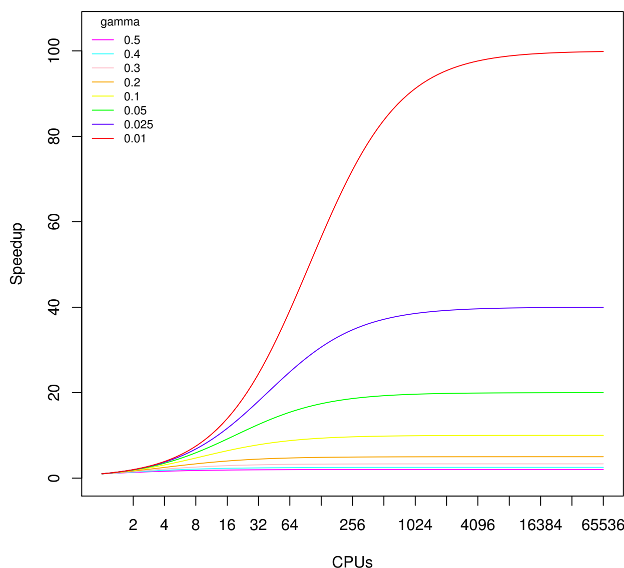


Figure 2.1: Amdahl's law visualization with different values of γ. Logarithmic transformation is applied to x-axis in order to demonstrate the difference properly.

Definition. *γ is the serial fraction of the parallel implementation and it is naturally defined as follows:*

$$\gamma = \frac{T_{serial}(1)}{T(1)}$$

Finally, Amdahl's law may be formulated by using prior definitions. The theorem asserts that given any positive γ, the maximal speedup is never unbounded. Thus the scalability of the algorithms, which have at least a single instruction executed serially, is bounded.

Theorem 1.

$$\lim_{P \to \infty} S(P) = \frac{1}{\gamma}$$

Proof. Proof follows simply from previous definitions [16]. □

Figure 2.1 demonstrates the influence of γ. As it might be observed, the speedup difference of $\gamma = 0.01$ and $\gamma = 0.025$ is greater then the speedup difference of $\gamma = 0.025$ and $\gamma = 0.5$

Amdahl's law is an important rule that is referred in performance evaluation which is covered in chapter 5. In reality, Amdahl's law is not the expected speedup since the concurrency approach inevitably leads to overhead. Additionally, it is

almost impossible to determine γ precisely. Thus, Amdahl's law may be assumed to be an upper bound of the possible speedup.

2.3 Parallel Architectures Classification

Multi-processing may be achieved via different architectures. There are four major architectures, that form Flynn's taxonomy, based on data and instruction load that are processed at once [16]. Table 2.1 presents Flynn's taxonomy in tabular form.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's taxonomy

In this thesis, only the MIMD architecture is used as the preprocessor provides exclusively thread-level parallelization. Thus, different threads may execute different instructions concurrently. Even though the combination of SIMD and MIMD is often used in low-level programming, this thesis is limited to development of a high-level preprocessor compatible with all JVMs. Therefore it cannot deal with SIMD instructions because Java provides no interface for instruction-level programming. [17].

2.4 OpenMP

OpenMP has been standard for shared memory multiprocessing since 1997. The parallelization itself is achieved via compiler directive usage. Therefore programming using OpenMP directives is rather straightforward for the programmer as it is possible to write serial codes decorated with simple directives. Nevertheless, the developer ought to be aware of future parallelization in order to maximize the final speedup. Example in listing 2.1 demonstrates a simple usage of directive programming.

```cpp
#pragma omp parallel for
for(int x=0; x < 10000; x++) {
  runTask(x);
}
```

Listing 2.1: OpenMP (C++) parallel for

2.4.1 History

Throughout the years, the OpenMP project has been developed by OpenMP Architecture Review Board (OpenMP ARB). It consists of both hardware and software related companies such as IBM, AMD, ARM, HP, Intel, NVidia, Oracle co., RedHat and many more [18]. ARB is responsible for all released OpenMP specifications.

Initially implemented only for Fortran, it became popular across many fields especially mathematical computing and physical simulations [19]. In 1998 the C/C++ support was provided.

Even though OpenMP 4.0 was released in 2013 [20], only few C/C++ compilers support all new directives. However, the majority of commonly used C/C++ compilers (in the latest versions) such as gcc (GNU), icc (Intel) and clang (LLVM) supports OpenMP 3.0 with some directives introduced in 3.1 [21]

2.4.2 Fork-Join Model

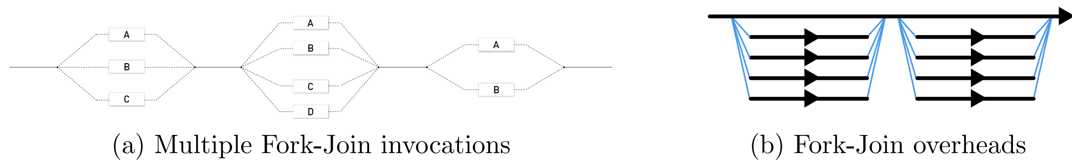
OpenMP uses Fork-Join design pattern for majority of supported directives. In Fork-Join model, the developer determines points where the program branches and runs in parallel (fork) and also the points where branches meet again (join).

Fork-Join model is illustrated in figure 2.2a where tasks invoked in parallel are denoted with letters. As the example demonstrates, each fork may branch into a different number of tasks independent of actual CPUs number.

In contrast to other concurrent pattern, e.g. Actor pattern that is implemented e.g. by AKKA (see chapter 1.3.5), Fork-Join pattern is older and less object-oriented. Additionally, it usually requires the imperative style of pro-

gramming. However, for some problems it scales better and it is generally easier to implement.

Some overhead is naturally connected with fork operation. For example, new thread creation causes a delay. Figure 2.2b demonstrates tasks invoked in parallel (black bold horizontal lines) and overhead created by parallelization (blue oblique lines).



(a) Multiple Fork-Join invocations

(b) Fork-Join overheads

Figure 2.2: (a) In Fork-Join model scheme multiple forks and joins are employed. (b) The fork/join (blue) overhead caused by thread creations and joining.

2.4.3 Modern development

As modern programming languages, such as Scala, Ruby, Groovy and Rust, were developed, programming concurrent application became easier to maintain. However, for the majority of more traditional languages a lot of libraries and tools have been developed (see chapter 1.3 for more information about Java tools).

In addition, many modern programming languages directly support multiprocessing. In contrast to C++ OpenMP example (see listing 2.1), listing 2.2 provides the same algorithm implemented in Scala (inspired by [22]). No third-party library is used, even though the whole execution is a single command.

```
(1 to 10000).toList.par.map(getTask(_).run())
```

Listing 2.2: Scala parallel execution

Nevertheless, for high performance systems, low-level programming languages, such as Fortran and C/C++, are still often used for their great performance and advanced optimizations. Due to the backward compatibility, even the modern versions of these languages cannot offer a comfortable and easy-to-use API for multiprocessing. The following code (listing 2.3), that is taken from [23], demonstrates the complexity of pthread library in C/UNIX.

Clearly, Scala approach is much easier-to-use than C pthread approach. In contrast to that design, OpenMP provides very comfortable design pattern for multi-threading as it is demonstrated in the prior example (see listing 2.1). For that reason, OpenMP is still frequently used as a good compromise between performance and readability [19].

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <pthread.h>
4  using namespace std;
5
6  #define NUM_THREADS     5
7
8  int main () {
9     pthread_t threads[NUM_THREADS];
10    int rc;
11    int i;
12    for( i=0; i < NUM_THREADS; i++ ){
13       rc = pthread_create(&threads[i], NULL, runTask(i), (void *)i);
14       if (rc){
15          cout << "Error:unable to create thread," << rc << endl;
16          exit(-1);
17       }
18    }
19    pthread_exit(NULL);
20 }
```

Listing 2.3: C++ parallel execution

Chapter 3.
Preprocessor Architecture

This chapter describes the architecture of omp4j - the preprocessor that is introduced by this thesis. Used language, coding style and the variety of employed technologies are explained. Furthermore, the application design is clarified. The set of supported directives is presented and the directive implementations are discussed. The following definitions introduce the notation that is employed.

Definition. *A Directive is a single line comment starting with // omp.*

Since Java does not support direct compiler directives such as #pragma for C++, [8] the OpenMP directives must be supported via a different information channel. The one-line comments were employed in order to provide valid Java source code if it is not processed by omp4j.

Definition. *A Directive body is a statement that immediately follows a directive.*

Definition. *A Task is a block (of code) that may run independently on other tasks. It is implemented as an instance of the class which implements java.lang.Runnable interface. The overridden method run is not expected to throw any runtime exception. For tasks throwing exceptions the behavior is unspecified.*

Definition. *An Executor is a scheduling entity that accepts tasks and manages their execution regardless of their particular order. It is implemented as an instance of the class which implements the org.omp4j.runtime.IOMPExecutor interface.*

3.1 Level of Processing

There are four basic types of input processing into the output that are classified according to the types of input and output. For purposes of the preprocessor classification, the input and output types are distinguished as either the source code or JVM bytecode. For future references, these options are denoted in table 3.1.

		Parallel output	
		Source	**Bytecode**
Serial input	**Source**	SISO	SIBO
	Bytecode	BISO	BIBO

Table 3.1: Preprocess level notation

Neither BISO nor BIBO meets the original requirements (described in chapter 1.1) since they accept only bytecode as input. It would not be suitable to expect programmers to pass serial bytecode from their IDEs in order to parallelize it.

On the contrary, both SISO and SIBO are valid and reasonable alternatives that were implemented in order to provide the user with freedom of choice. Both alternatives are illustrated in figure 3.1. The user may use either SISO approach in order to obtain the parallel source code for advanced optimization and version control purposes or SIBO approach which replaces javac compiler. In the latter case, the user provides source code(s) with OpenMP-like directives and the preprocessor returns compiled parallelized output in the form of the Java bytecode (.class files).

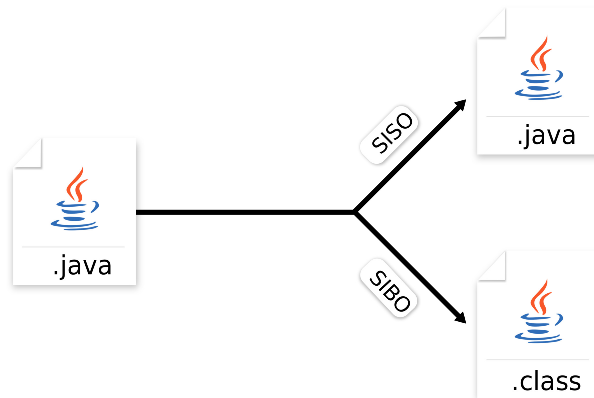


Figure 3.1: Supported translation types. omp4j either behaves like standard Java compiler or only preprocesses the source code without compilation.

3.2 Processing Work-flow

The code transformation is divided into six logical phases. Each phase employs one or several classes which communicate with each other. The phases are as follows from the list below.

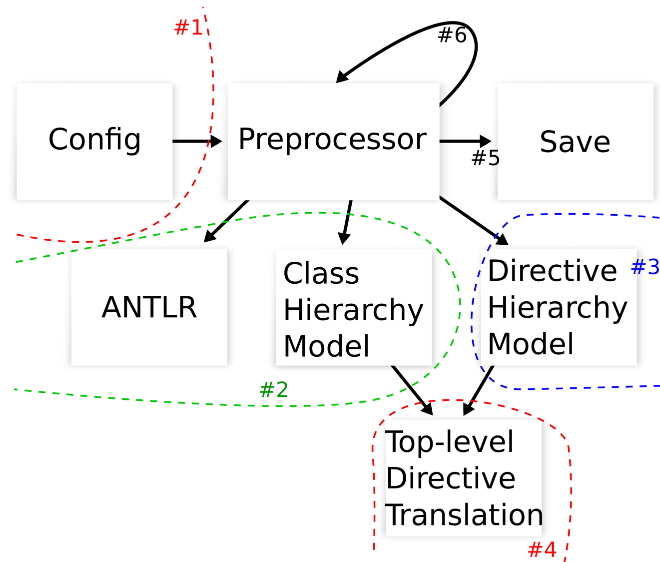


Figure 3.2: Major preprocessor actions divided into phases.

1. Configuration creation
2. Code analysis
3. Directive recognition
4. Top level directives translation
5. Output writing
6. If any directive exists, goto 2.

Figure 3.2 illustrates the important preprocessor actions in phases context. Appendix A provides the list of all major classes that are used in each phase. The classes descriptions are provided in the API reference that is attached to this thesis. While phases 1, 5 and 6 are quite straightforward, the remaining phases are described in detail in the following chapters 3.3, 3.4 and 3.5.

3.3 Code Analysis

Two approaches of source code analysis are used for omp4j preprocessor - bytecode and syntax analyses. The former approach is applied for classes whose source code is not accessible during the process while the latter analysis is employed when Java reflection API is unable to reflect some object such as an anonymous class.

3.3.1 Bytecode Analysis

The bytecode analysis is essential since the user may provide input source code which is dependent on some classes that are installed in the classpath. The classes may be already compiled into bytecode and the preprocessor might not access their source code in order to perform syntax analysis instead.

The process of the bytecode analysis is as follows. Initially, thread-id macros are replaced with the number 1. Then, the sources are compiled, hence their syntax is verified using Java compiler. A special class - org.omp4j.system.Compiler - is provided in order to simplify Java compiler API and JAR creations.

Additionally, org.omp4j.system.Loader class is declared, which employs the Java reflection API in order to load the JAR and to provide the required information. By using the loader, the hierarchy model (described later in chapter 3.3.3) is built.

3.3.2 Syntax Analysis

In contrast to the bytecode analysis, the syntax analysis is essential for two different reasons. Given an input source code, the preprocessor assumes that anonymous classes may be present which is the fundamental problem since those classes are impossible to be reflected via Java reflection API. Furthermore, the preprocessor needs local variable information that cannot be obtained via the bytecode analysis.

Since the preprocessor finally modifies the source code in order to provide its concurrent version, simple navigation through the source code is required. For that purpose an abstract syntax tree (AST) is employed.

For syntax analysis, we decided to use commonly used Another Tool for Language Recognition version 4 (ANTLRv4) for its speed, great documentation and the possibility of writing additional grammars. However, other Java parser generators are available such as BISON[1] or JavaParser[2]. The former generator is

[1]https://www.gnu.org/software/bison
[2]http://javaparser.github.io/javaparser

unusable for this project the generated parser cannot be employed from Java, even though it may recognize Java grammar. The latter recognizer is Java compatible but has very poor unit tests and documentation.

ANTLR allows the user to provide .g4 grammars in order to get generated lexer and parser for that language. omp4j benefits from this approach as an OpenMP-like grammar was implemented as a part of the whole project. The grammar is employed during the directive recognition which is described in chapter 3.4.1.

For Java recognition, the original ANTLR grammar for Java 8 was used. Since the grammar is maintained by community, it varies a lot. Even though it was tested, there is no proof of correct recognition. Despite this fact, we decided to use it anyway in order to provide Java 8 support at maximum possible level.

The parser, that is generated by ANTLR, provides AST of the source code given. The AST might be iterated through the visitor pattern [24]. We implement the extractors located in org.omp4j.extractor. The visitors implement the ANTLR visitor pattern design. They are employed for two main purposes - the directive validation (e.g. break detection in omp parallel for directive) and the creation of class hierarchy model (see chapter 3.3.3).

3.3.3 Class Hierarchy Model

The package org.omp4j.tree provides a complex model of provided source code. By using both bytecode and syntax analyses, all useful information about classes may be obtained from the input source code. That includes inherited methods and fields even for the anonymous classes.

In addition, the model itself forms a tree, i.e. it preserves relations between classes such as one class is declared in another and vice versa. This model is employed during the directive translation which is described later in chapter 3.5.

3.4 Directive Recognition

After the syntax analysis is finished (see chapter 3.3.2), the AST is visited one more time via org.omp4j.preprocessor.DirectiveVisitor that seeks all one-line comments beginning with // omp. As Parr suggests [24], a hidden channel must be defined for proper comment passing (see listing 3.1 for hidden channel definition in Java 8 grammar).

```
1783 //
1784 // Whitespace and comments
1785 //
1786
1787 WS  :  [ \t\r\n\u000C]+ -> channel(OTHER)
1788     ;
1789
1790 COMMENT
1791     :   '/*' .*? '*/' -> channel(OTHER)
1792     ;
1793
1794 LINE_COMMENT
1795     :   '//' ~[\r\n]* -> channel(COMMENTS)
1796     ;
```

Listing 3.1: Java8.g4; Comments passed via hidden channel

After the proper comments are fetched, the ANTLR is used one more time in order to parse the line into AST according to OMP grammar, which is described in chapter 3.4.1.

3.4.1 Directive Grammar

A grammar was employed in order to obtain information from the comment line. In comparison to regular expressions, grammars provide higher flexibility. In addition, they are more readable and extendable. The whole grammar may be found in appendix B.

The grammar, that was developed as a part of this project, has rather straightforward structure. The start rule [24] - ompUnit - branches into several directive rules such as ompParallel and ompBarrier according to the directive type.

Each rule consists of a lexer rule (e.g. omp or parallel) and optionally some other rules. For example, the critical section omp critical(lock) is parsed into AST as ompUnit, ompCritical and ompVar which is directly mapped to a lexer rule VAR

that matches lock variable. Therefore, the preprocessor may employ the visitor pattern [24] in order to obtain information from the AST such as “Was lock variable specified?”.

Even though a context-free grammar was initially intended to be developed for its simplicity and parsability [25], some advanced techniques must been employed in order to provide permutation of parameters [24]. As the result, user is allowed to write all the directives as are listed in listing 3.2.

```
omp parallel public(a) private(b,c,d) public(e)
omp parallel private(b,c,d) public(a,e)
omp parallel public(a) schedule(static) private(b,c,d) public(e)
```

Listing 3.2: OMP grammar possible attribute permutations

This behavior can be achieved via two different approaches. First, each permutation of attributes is represented as a single rule in the grammar. Since the grammar may contain at most a finite number of rules [25], this approach limits the total number of attributes used and makes the grammar unreadable and exponentially large.

The second approach, which was actually employed, implements a parser that uses random access memory in the form of a HashMap in order to store parser attributes. The comparative advantage of this method is the linear number of rules, better grammar readability and maintenance. Nevertheless, the grammar itself cannot be longer classified as context-free because of the additional memory use. Thus the matched language remains recursively enumerable.

We conclude that the adopted approach is much more beneficial for both the users and the developers. Compared with Pyjama (see chapter 1.3.3) which requires the thread-limiting attribute to be before the other attributes [9], omp4j provides more comfortable API. Furthermore, the grammar may be easily extended with additional directives (e.g reduction) since only one new rule addition is sufficient.

3.4.2 Directive Hierarchy Model

After ANLRT parses the comment, a Directive is constructed using the comment information. Similarly to the class hierarchy model (see chapter 3.3.3), all directives are stored in the form of a tree, preserving mutual relations.

Directives are partially validated during the hierarchy model construction, however, the logical structure of the model is tested during the first translation phase (see chapter 3.5).

All directives and directive-related classes are stored inside the org.omp4j.directive package which forms an object oriented class hierarchy.

3.5 Directive Translation

In comparison with related projects (see chapter 1.3), omp4j employs a fundamentally different approach to code transformation. For example, JOMP creates a translation class, that has public fields initialized with real references, and transfers the directive body into the class. Since the class is defined at the end of the source file, which may be long, the developer loses track of the translated code. On the contrary, omp4j does not transfer code anywhere. Instead, the code is slightly modified and wrapped with a clause that manages the parallel invocation.

Even though this approach is much more difficult to implement correctly and depends on ANTLR Java 8 grammar, it leads to the code that an ordinary programmer would produce. Additionally, the processed code remains more readable and maintainable. Therefore, omp4j may be used for educational purposes as it easily demonstrates the straightforwardness of parallel solutions.

The following definitions are employed later in this chapter. The definitions from the beginning of chapter 3 are often employed.

Definition. *The Capturing is a process of AST analysis that obtains all non-local variables used in the AST.*

Definition. *A Context class for the given directive is a class extending* java.lang.Object *that defines appropriate fields according to the captured variables of the directive body.*

Definition. *A Context is an instance of a context class with all fields properly set to captured variables references.*

Given the directive hierarchy model (described in 3.4.2), the main preprocessing class org.omp4j.preprocessor.Preprocessor starts translating top directives, i.e. those that are at the top of the model tree. Therefore, the directives are translated by layers.

Since the directive bodies in the same hierarchy level cannot overlap, the selected directives are mutually independent, hence they may be translated in any particular order.

When the directive is translated, all direct subdirectives that do not create a new executor are also translated at once i.e. nested for, atomic, critical, master, single and section. This approach ensures that once the layer translation is finished, a valid output is provided. Hence, the functional design may be employed and directive translation might be processed recursively.

The directive translation itself is divided into three separate phases implemented via three directive methods - validate, preTranslate and postTranslate (see figure 3.3). All three phases are described below in chapters 3.5.1, 3.5.2 and 3.5.3.

Figure 3.3: Three processing phases of the translation process.

3.5.1 Phase 1: validate

Phase 1 is implemented via the overridden validate method. This method is invoked while processing the directive. The method throws a SyntaxErrorException

if and only if the location of the directive in the hierarchy model is not supported. For example, nested directive such as master etc. must not appear in the first level of the hierarchy.

3.5.2 Phase 2: preTranslate

Phase 2 is implemented via preTranslate method. When invoked, it creates a new instance of org.omp4j.preprocessor.TranslationVisitor that iterates through the directive body by using the visitor pattern.

During the iteration, the process of variable capturing is realized. By using the class hierarchy model and both bytecode and syntax analyses, TranslationVisitor builds a set of accessed variables that are defined as non-local or defined before directive body starts. The variables are stored with all important information such as type and meaning (i.e. local variable, parameters, fields, etc.).

Additionally, preTranslate assures that all captured variables are prepended with context name (see chapter 3.5.4 for details). Even though the context is not constructed yet, since not all variables are captured, its name was generated before and hence known.

Thus, once the phase 2 is finished, the proper variables in the directive body are ready to be used with the proper context. Furthermore, these variables are captured and returned.

3.5.3 Phase 3: postTranslate

Phase 3 is implemented via commonly overridden preTranslate method of Directive class that manages the translation . When invoked, it manages the parallelization itself. This method is abstract and each directive must override it differently in order to provide a sensible parallelization. Invariably, the core concept remains the same.

Initially, the context class is created according to data provided by variable capturing. Secondly, the context is instanced and its variables are set to the

captured variables references. In case of primitive types, the captured values are duplicated.

At this point, a valid code is obtained since directive body uses the context fields instead of regular variables. Nevertheless, the program still runs serially.

Finally, the executor is created based on schedule attribute (see chapter 3.6 for runtime library description). Then the parallelization itself may happen, which means that the directive body is split into tasks that may run concurrently. These tasks are scheduled and executed by the executor. Each directive implements the task decomposition differently (refer to chapter 3.7 for implementation details).

3.5.4 Rewriting the Source Code

While executing phases of the translation, the source code is modified. Since the AST provided by ANTLR is immutable [24], it is impossible to modify nodes or insert and delete branches. For this purpose ANTLR provides Rewriter API [24].

The Rewriter API is implemented via org.antlr.v4.runtime.TokenStreamRewriter that supports string insertion before and after a token. Also token sequence replacements are supported, however multiple changes of the same token sequence throw exceptions. Hence, the the usage of this tool is very limited and a lot of discussions have been made for future modification. This approach was chosen as it is the only available option provided by ANTLR version 4.

3.6 Runtime Library

In order to produce output that is independent of installed runtime libraries, there are two basic approaches to the code modification - either implementing the whole changes directly into preprocessed code or implementing the parallelization routines in a separate class.

The latter approach was chosen for two main reasons. First, it makes the application design more elegant as the same routines are not implemented mul-

tiple times when more than one directive is preprocessed. Second, the enhanced modularity is available. For example, the user may reimplement some of the classes and use them instead of the original ones. For this purpose, a separate git repository was created, as it is described in chapter 4.1

Even though the runtime library is provided, the preprocessed code has not any runtime dependencies since all required class-files are provided during the compilation. Consequently, the users might simply compile theirs sources using omp4j and run it on a different machine. Thus, the user may employ omp4j instead of javac as the compiled bytecode is accompanied with all dependencies.

In order to allow the users to define their own executor, they may let the preprocessor omit the compilation and preprocess only the sources. In that case sources of the runtime library must be provided by the user as they are not copied into the output directory. Additionally, the user may compile the whole application with some different compiler.

3.6.1 Executor Interface

Each executor implements org.omp4j.runtime.IOMPExecutor interface which extends commonly used java.util.concurrent.Executor. The whole interface may be found in appendix C.

The life-cycle of the executor is as follows. Initially, the tasks are scheduled by overridden execute method. When the first task is scheduled, the executor may invoke the already scheduled tasks in any order. The method itself is non-blocking and returns immediately. Finally, waitForExecution method is invoked. It prevents future task scheduling via method execute and remains blocked until all tasks are finished.

The supplementary thread-id methods are specified - getThreadNum and getNumThreads. The former method returns a positive integer in range $[1; NumThreads]$ depending on which thread the method was invoked from. The latter method returns the number of used threads. These methods are as well non-blocking and operates in constant time complexity.

Definition. *A barrier hit is the operation performed by a thread in which a barrier counter is increased.*

Furthermore, an executor must support the barrier-like behavior. For that purpose, hitBarrier method is specified as follows. The method accepts a single string which denotes the barrier that should be hit. Until all threads have hit the barrier, the already hit threads block and wait on the barrier. The barriers are registered on demand, i.e. the first use of hitBarrier with parameter P atomically creates all resources that may be needed for the proper barrier behavior. When P barrier is hit again the executor atomically allocates the resources. The example of the method implementation may be found in appendix D.

3.6.2 Dynamic Executor

As a possible executor implementation, org.omp4j.runtime.DynamicExecutor is provided. It uses the Executors.newFixedThreadPool method for the executor service creation [26]. Dynamic executor is employed for the performance evaluation in chapter 5.

We suggest using DynamicExecutor for the majority of applications since it is well tested by Java API developers. However, for specific purposes it may not be the optimal solution.

3.7 Supported Directives

This chapter describes all implemented directives and usages. omp4j preprocessor supports a subset of original OpenMP directives [27] - majority of commonly used directives are provided though. Furthermore, the directive behavior and translation policy is explained and the additional attributes are listed.

The directives are translated if and only if located inside a class function. Directives that are inside enum methods are ignored.

3.7.1 omp parallel

The most basic supported directive is omp parallel which invokes the directive body in parallel. That means that the body will be executed as many times as is the number of available threads (see chapter 3.7.10 for detailed behavior). The example (listing 3.3) demonstrates a simple usage and the directive's behavior.

```
// omp parallel
{
  System.out.println("hello")
}

/* output */
hello
hello
hello
hello
```

Listing 3.3: omp parallel example

The implementation of the code translation is as follows. For each thread a single task, containing the whole directive body, is created and executed by the executor.

3.7.2 omp [parallel] for

Directive omp parallel for is a shortcut for omp for nested in omp parallel. It must be used directly only before a for-loop. Only regular for-loops are allowed - e.g. for (int i = 0; i < 10; i++) is a valid example. On the contrary, for-each loops are not supported (similarly to JOMP [28]) - e.g. for (T x : collection). Even though a task might be added identically as in basic for-loop, a lot of unexpected behavior might happen. For example, the collection may read from network and/or from file. In that case, the order is crucial.

In contrast to omp parallel, this directive invokes the for-loop only once regardless of the specified number of threads. However, each iteration is represented as a separate task that might run concurrently - i.e. each task may run in different thread and their execution order is unspecified (see example in listing 3.4)

The use of keywords break, continue and return is not allowed in the directive body since the order of execution is broken up. Thus, the use of these keywords is

unnecessary. Additionally, the iterator is considered as final. Thus, if the for-loop header is as follows: for (int i = 0; i < 10; i++), variable i cannot be changed in the loop body.

```
// omp parallel for
for (int i = 0; i < 10; i++) {
  System.out.print(i)
}

/* output */
2 0 6 7 1 9 5 3 4 8
```

Listing 3.4: omp parallel for example

The implementation of the code transformation is as follows. Initially, the for-loop is iterated whilst a new task is created in each iteration. Meanwhile, tasks created in prior iterations may have been already invoked in parallel. Since each iteration is represented as a single task, it ought to run long enough in order to benefit from parallel invocation as the overhead with task scheduling and thread creation inevitably occurs.

3.7.3 omp section(s)

Wrapper directive omp sections may contain only omp section directives - other directives and/or raw code are not supported. Each omp section is treated as an independent task i.e. it might be invoked in parallel. omp section should be followed by an empty statement, i.e. {...} .

This directive is commonly used for heterogeneous tasks for which omp parallel [for] is not suitable. The number of sections is fixed and it's determined by the number of sections, that are actually present in the source code.

The example in listing 3.5 demonstrates the common usage of the input reading from multiple independent resources at once.

The implementation of omp sections is similar to omp parallel for. A for-loop is created and iterated through in parallel. Each section is numbered by their mutual order in the source code and invoked if and only if the for-loop iterates through the proper index.

```
int a,b,c;
// omp sections
{
  // omp section
  { a = readFromDrive(); }

  // omp section
  { b = readFromStdIn(); }

  // omp section
  { c = readFromNetwork(); }
}
```

Listing 3.5: omp section(s) example

The major possible pitfall of omp sections is the unused CPU time created when differently long sections are declared. As an example (illustrated in figure 3.4) we assume four independent tasks (each implemented as a section). Assuming four CPUs, the sections may run in parallel all at once. Due to the different lengths of the sections, however, the final speedup is not 4 as it might have appeared at the beginning. As the scheme demonstrates, early finished threads must wait for the join operation (see chapter 2.4.2 for Fork-Join model details).

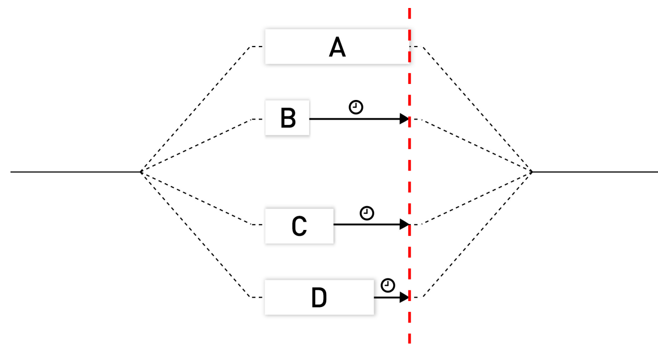
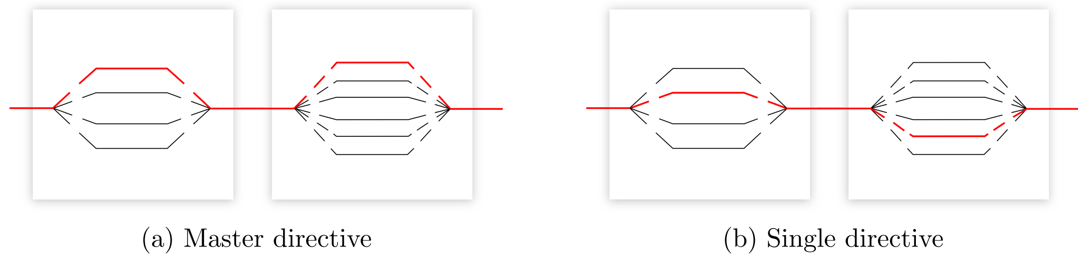


Figure 3.4: Bold lines, accompanied with clock icons, demonstrate the time when the CPU waits until the other sections are completed.

3.7.4 omp master

omp master is a directive assuring that its body is invoked only from the master thread which is the thread denoted with $ID = 0$. omp master is not a top level directive which means that it must be nested in some other parallel directive. Usually, omp master is used inside omp parallel where a block of code should run only once.



(a) Master directive (b) Single directive

Figure 3.5: Red color represents the thread that executed the directive body. (a) Thread 0 only i.e. the top one. (b) Any thread

Even though the use of this directive enables enhanced flow control, the use of the master directive should not be redundant as it provides an overhead due to thread creations. As an example (see listing 3.6) we provide parallel directive with master body. The example leads to only one invocation that might have been achieved by omitting both directives. The benchmarks of this behavior may be found in chapter 5.5.

Figure 3.5a demonstrates the master thread (red color). Three blocks of code decorated with omp master are shown. In each, only the master thread invokes the specified directive body.

```
// omp parallel
{
   // omp master
   { compute(); }
}
```

Listing 3.6: omp master wrong usage example

The master directive is implemented by a simple condition. The directive body is wrapped into the if statement where executor getThreadNum method is invoked and compared to number 1.

3.7.5 omp single

omp single works similarly to omp master - the directive body is invoked only once, however, by any particular thread. This leads to possibly better scheduling then the master directive approach. Nevertheless, due to the omp single implementation, some overhead is naturally present.

omp single is implemented using atomic boolean variable. Unless some thread

assigns itself to invoke the body, all threads intend to do so. Finally, when the atomic boolean is set, other threads passes the body without its invocation.

Figure 3.5b demonstrates the different threads (red color) that executes singles in each of blocks of code. It may be a different thread in each block as well as the same one in all.

3.7.6 omp critical

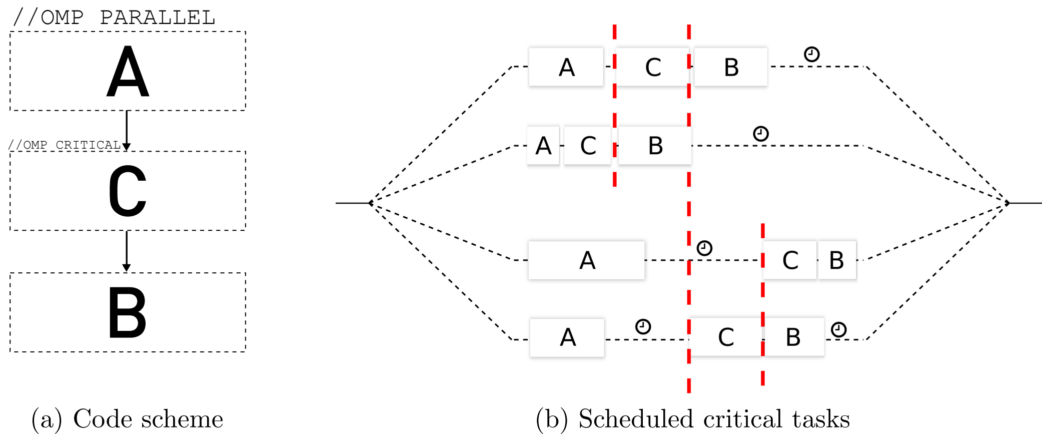
omp critical also must be nested in some parallel directive. Compared to omp master and omp single, its body is invoked by all threads. Additionally, mutual exclusion is provided, meaning at most single thread will access the body at any particular time.

The critical sections are implemented using Java embedded synchronized keyword, providing huge overhead (see benchmark in chapter 5.4). In general, the user should avoid any synchronization primitives, especially the critical sections.

omp critical optionally accepts a final variable as a parameter that is used as synchronized parameter. This technique is suggested in order to minimize race conditions as much as possible.

Figure 3.6a represents basic critical section usage. omp parallel is invoked in four threads which each sequentially execute tasks A, C, B in this order. Furthermore, task C is marked as the critical section.

Figure 3.6b illustrates one possible schedule, where at any point task C is being executed at most by one thread. Consequently, some threads wait until other threads run C which inevitably wastes the CPU time. Even though the waiting is implemented as passive, time is being lost. When a thread leaves the critical section, all waiting threads intend to lock the critical section variable (or the class itself if no variable is specified), hence they cause some additional overhead that increases with the number of threads.



(a) Code scheme

(b) Scheduled critical tasks

Figure 3.6: Overhead visualization of scheduled tasks. Task C may be executed by at most one thread at any time.

3.7.7 omp atomic

omp atomic is implemented equally to omp critical. Given the fact that not all operations have their atomic alternative, is not possible to provide a general solution.

In addition, the user might use already compiled classes. These classes are impossible to be translated on the source code level. Instead, we decided to discourage the user from the use of omp atomic directive.

However, in order to develop as compatible set of OpenMP-like directives as possible, the atomic operations are implemented in the form of critical sections. This has a huge impact on performance and omp atomic should be avoided. The author of the technical report of JOMP suggests the following:

> " ... In any case, the kind of optimisations which the directive is designed to facilitate are unlikely to be possible in Java.
>
> Should one wish to include support, it would be quite simple and efficient to implement where the atomically updated entity is an object or array, using Java's synchronized statement. However, in practice, the value is much more likely to be of a primitive type, in which case there is no obvious way to implement it short of using a single lock for all atomic statements. ... " [28]

Due to Java memory model and the JVM instruction set design, that differs from C/C++ and Fortran, it is impossible to support atomic operations in general:

> " ... The kind of optimisations which the directive is designed to facilitate (for example, atomic updates of array elements) require access to atomic test-and-set instructions which are not readily available in Java ... " [29]

3.7.8 omp barrier

The last implemented directive, omp barrier, simulates barrier-like synchronization primitive. The directive must be applied to a blank statement. Until all threads hit the barrier, they remain blocked at this directive.

The implementation uses java.util.concurrent.CyclicBarriers, stored in the executor. For implementation details see chapter 3.6.

3.7.9 public, private, firstprivate

The first set of attributes we describe - public, private and firstprivate - may be used with parallel directives. They accept either a single variable as a parameter or a comma-separated list of variables. All attributes may repeat, however only last variable occurrence is applied.

public is a default setting for all variables. The public variables are considered as shared variables among all threads. No synchronization primitives are implicitly provided, thus these variables should be read-only because any mutable operation may collide with other threads.

On the contrary, private creates an array of variables in which each cell is accessed by one thread only. Hence, mutable operations are possible as they do not interact with other thread instances. When the array is created, parameterless constructors are used. A compilation error is raised if the variable class does not provide this constructor.

firstprivate is the same as private, except it uses a copy-constructor. It means that the variable itself is passed to the constructor. A compilation error is raised if the type of the variable does not provide this constructor.

3.7.10 threadNum

threadNum is the thread number restriction attribute the parallel directives. The common usecase is as demonstrated in listing 3.7.

The attribute may be used at most once in a directive. The passed parameter is a single integer representing the number of threads that will participate in the directive body execution. If threadNum attribute is missing the total number of CPUs, that JVM is aware of, is used implicitly.

```
/* Running on 48 CPU server */
// omp parallel threadNum(2)
{
  System.out.println("hello");
}

/* Output */
hello
hello
```

Listing 3.7: threadNum example

3.7.11 Thread-ID Macros

omp4j supports macros (or constants) through which the information about the current thread is provided. Nevertheless, the source code with these macros will not be compiled properly using the standard Java compiler. This is the only difference from standard Java source code. For portability maintenance, the usage of directives without Thread-ID macros is suggested. All macros

The example in listing 3.8 demonstrates proper usage of OMP4J_THREAD_NUM and OMP4J_NUM_THREADS that are translated into current thread id (numbered from 1) and total number of threads invoking this directive body respectively.

We decided to employ macros rather than specific methods like related projects do (e.g. JOMP). The use of macros brings two main advantages - *first*, the us-

er do not need to be aware of the preprocessor package structure since they do not invoke any methods (such as org.omp4j.runtime.Config.getThreadNum()); *second*, the user may easily redefine the constants in the class that should be parallelized, thus the code may be compiled with other compilers as well. Therefore, enhanced portability is provided in comparison with similar projects.

The macros are translated into constant integers, hence all common operation such as addition, multiplication and array-indexing are provided. On the contrary, the macros cannot be assigned with a new value as they are constants. The macros follow Java coding style of constants, hence they appear like regular constants.

```
// omp parallel threadNum(5)
{
  System.out.println(OMP4J_THREAD_NUM + "/" + OMP4J_NUM_THREADS);
}

/* Output */
2/5
3/5
0/5
1/5
4/5
```

Listing 3.8: Thread-ID example

3.7.12 schedule

The last supported attribute is schedule. It accepts a fully qualified name of a class that implements org.omp4j.runtime.IOMPExecutor interface. As an abbreviation, parameters - dynamic and static are supported. The former parameter is translated into org.omp4j.runtime.DynamicExecutor which is described in chapter 3.6.2. The latter parameter is analogously translated into StaticExecutor. Each executor modifies the scheduling policy and the final speedup.

schedule may be used at most once in each parallel directive. The default value is dynamic as it better suits majority of cases. The dynamic policy registers tasks into a single central queue whence the tasks are being assigned to the workers singularly on demand. See chapter 3.6 for detail description.

On the contrary, the static policy determine each task to one thread before the execution. We advise the user to use the default dynamic approach for its

better performance.

3.8 Language and Coding Style

Due to the original requirements (described in chapter 1.1), Java bytecode-based language is required. From all JVM-based languages we decided to employ Scala for several reasons.

Initially, Scala applications may be assembled[3] into JARs that do not require Scala installed for their execution [30]. Additionally, Scala features full Java interoperability which enables native Java libraries usage [31]. That is an essential property for the syntax analysis that is described in chapter 3.3.

Furthermore, Scala is strongly typed, thus makes development easier as some syntax errors are discovered during compilation. In addition, according to Odersky, Scala features both object-oriented programming (OOP) with multiple inheritance via traits and the functional design approach [31]. Functional programming means programming without side-effects [32], hence this application design makes the debugging easier as well as the whole development more straightforward. Furthermore, it provides a deeper reasoning about the behavior of the algorithm.

The language construct, that has been used the most, is implicit keyword. The variables defined implicit are automatically passed to the invoked functions. This concept is the welcomed alternative to static configuration context.

As well as Haskell [33], Scala provides simple pattern matching that we often use during exception handling and subclass classification.

[3]`https://github.com/sbt/sbt-assembly`

Chapter 4. Implementation

This chapters briefly describes used techniques and environment during the development of the thesis. The various links to project-related websites are presented as the preprocessor consists of several modules.

4.1 Version Control System

A version control system (VCS, revision control) was employed in order to increase efficiency of the development process even though only one programmer has been committing the changes. Additionally, VCS supports further development when more developers are involved.

Currently the most popular version system - git - introduced by Linus Torvalds in 2005, was chosen because of its great simplicity of command line interface and general speed. Since git is distributed, the current backup and the whole history of code changes have been saved on multiple machines [34].

All commits have been uploaded to GitHub[1] which is currently the most popular git repository hosting for both open-source and private projects. The commits and the code changes may be easily accessed using GitHub visualization tools. GitHub will maintain further repository support in terms of bug-listing and the community will be able to fork omp4j in order to develop new features.

Table 4.1 presents the set of URLs that leads to the appropriate git repositories that are hosted on GitHub.

[1] `https://github.com`

URL	Target description
https://github.com/omp4j	omp4j group
https://github.com/omp4j/omp4j	preprocessor repository
https://github.com/omp4j/runtime	runtime library repository
https://github.com/omp4j/www	webpage repository
https://github.com/omp4j/grammar	ANTLR grammar repository
https://github.com/omp4j/benchmark	performance evaluation repository
https://github.com/omp4j/doc	API reference

Table 4.1: GitHub repository overview

4.2 License

Due to the public presentation on GitHub, BSD license (refer to appendix E) was chosen for its general freedom. We believe that choosing benevolent license encourages the community to maintain the project integrity and keep the preprocessor up-to-date with modern trends.

4.3 Behavioral-Driven Development

The unit tests were executed frequently in order to maintain the stability and the professional quality of the software. The whole project was developed in modern agile Behavioral-Driven Development (BDD) style that implements test-driven development [35]. Even though BDD is relatively new approach to software development, it has quickly become popular for its robust description of the tests that are usually written from the users point of view [35].

ScalaTest - an unit testing library for Scala and Java projects [36] - was used in order to fulfill the BDD requirements. It enables an extremely straightforward test composition via natural-like language (see example 4.1). In addition, the tests may be invoked directly from both SBT (via test) and IntelliJ Idea IDE. In order to provide faster testing, a subset of tests (e.g. only previously failed tests) might be run. Hence, duplicate testing is eliminated which internally accelerates the development life-cycle. By employing the parallel test invocation, even large number of tests may be run within minutes.

```
describe("'omp sections' children in file:") {
  it("01.java should not contain other statements") {
    an [SyntaxErrorException] should be thrownBy new
    Preprocessor()(...).run()
  }
}
```

Listing 4.1: ScalaTest test definition

4.4 Continuous Integration

In order to increase the preprocessor stability and to ensure that it supports various JVMs, Travis-CI[2] was employed. This continuous integration service allows the developer to specify various JVMs, Scala versions and other properties.

The whole continuous integration approach is possible because of git hooks [34] that are triggered after every GitHub push. When the hook is triggered, a build matrix (BM) is created which is a cartesian product [37] of all possible settings that are described above (i.e. JVMs, Scala versions and other options). Each member of BM represents a unique setting of the environment in which the tests are independently invoked. That leads to a complex result of the unit testing on different machines[3].

[2] `https://travis-ci.org`

[3] All tests, that were ever executed, may be found at `https://travis-ci.org/omp4j/omp4j`

Chapter 5.

Performance Evaluation

This chapter describes several benchmarks that were run in order to evaluate the performance of the translated sources. Several different types of benchmarks are implemented and their differences explained. In addition, two linear models are derived from the data of the most independent benchmark in order to provide deeper insight into measured data dependence. Furthermore, a comparison with related projects is presented.

5.1 Methodology

The following benchmarks avoid measuring the absolute values since a tool for parallel invocation is being evaluated. Instead, the relative quantities are measured such as total speedup.

5.1.1 Benchmark Framework

In order to run the benchmarks automatically, org.omp4j.benchmark.AbstractBenchmark class is introduced. Each benchmark class extends AbstractBenchmark and implements methods runBenchmark and runReference. The former method should contain the body of the parallel version of the algorithm whilst the latter method implements the original serial algorithm.

If warmup method is overridden (default blank), it is invoked as a warm-up before any measurements are taken. Even though using the proper warm-up stabilizes the results [38], we employ warmup only for short-time benchmarks since the long running benchmarks stabilize themselves if they run long enough.

In addition, org.omp4j.benchmark.Benchmark is provided which invokes the specified benchmark itself according to the passed parameters. For this purpose, Java reflection API is employed. The process of benchmark invocation is as follows.

Initially, the benchmark class is loaded, followed by an instance creation. After this process is completed, the benchmark is invoked. This approach is used in order to eliminate the class-loading overhead. Loading and instancing the benchmark class before the time measurement forces JVM to run all class-loading routines before the measurement. As a result, the measured data is more coherent [38].

5.1.2 Data Processing

As described above, each benchmark consists of two parts - the serial reference algorithm and its the parallel implementation. Two values, t_{serial} and $t_{parallel}$, are obtained by measuring the durations of the execution ot these methods. These quantities are always measured immediately one after another. The speedup is consequently calculated as a fraction $\frac{t_{serial}}{t_{parallel}}$.

Each benchmark returns the measured speedup rather than raw times of the executions. Even though the latter approach provides greater mean speedups, the former method reflects the reality in more detail. By using the speedup approach, the effects of the environment (e.g. background processes) are minimized since the speedup is calculated pairwise.

Each measured speedup is considered to be a random variable. We do not presume any specific distribution, especially not the normal distribution (referring to appendix F for various plots that demonstrate the speedup distribution). For this reason, each benchmark was executed multiple times (at least ten times, usually more) in order to compute their arithmetic mean. This number is denoted as the final speedup which is considered for the following computations. According to Anděl, this technique ensures normal distribution of the final speedups as it follows from the central limit theorem (CLT) and the law of large numbers (LLN) [39].

The use of CLT is proper since the obtained results are both identically distributed (one benchmark executed multiple-times) and independent of each other (only one being executed at a particular time) [39].

5.1.3 Used Hardware

All benchmarks were executed on the same (linux) NUMA server (named navarin) that has four sockets with 12-core CPUs in each socket. Hence, benchmarks using up to 48 cores are supported which is important for the creation of the linear models. In contrast to similar projects (such as Pyjama with 16 cores[8]), benchmarks that employ higher number of cores provide the valuable information.

Even though no other user was logged into the described machine during the benchmarking, some system processes may have been running. These effects are filtered out by using multiple executions of each benchmark as described in the prior paragraphs.

5.1.4 Thread Limitation Techniques

In order to obtain results that depend on the number of used cores while running the benchmarks on the same hardware, the following thread limitation techniques are employed.

For omp4j benchmarks, the number of threads is directly specified in each benchmark (see chapter 3.7.10) in order to minimize any runtime overhead which could be caused by communication with the operating system in order to obtain CPUs count.

However, the number of threads for the original C++ OpenMP was limited by using the environmental variable OMP_NUM_THREADS. Finally, we limited the number of threads used by JOMP via −Djomp.threads=N option since JOMP does not support other thread limitation techniques.

The used techniques are not supposed to have any impact on measured results.

5.1.5 Statistical Notations

All hypotheses testing are executed at the level of significance $\alpha = 0.01$ if not stated otherwise.

The correlation coefficient classification [40] is used according to the following table 5.1 .

Coefficient	Label
0.00 - 0.35	Weak
0.36 - 0.67	Moderate
0.68 - 0.90	High
0.91 - 1.00	Very high

Table 5.1: Correlation coefficient labeling

The majority of benchmarks, that are provided in the following chapters, are accompanied with various plots. When the blue color is used in the form of a non-described line, it represents the theoretical optimum which is a rough upper bound of the real values. Furthermore, Amdahl's law limits possible speedup results even more (see chapter 2.2 for more details about the law).

In the following chapters, the letter γ, as defined in chapter 2.2, is employed to represent the serial fraction of the algorithm.

5.2 Fibonacci Numbers

In the first benchmark, γ minimization is intended in order to achieve the highest possible speedup. For that purpose, the most straightforward Fibonacci numbers computation is implemented (see listing 5.1).

```
int fibonacci(int n) {
  if (n <= 1) return 1;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

Listing 5.1: Fibonnaci recursive computation

omp parallel for directive was used for its simple and straightforward iteration through the workload that represents the computation of 30th to 35th Fibonacci numbers. The results are not stored anywhere in the memory in order to isolate each computation absolutely in terms of the memory sharing and the cache synchronization.

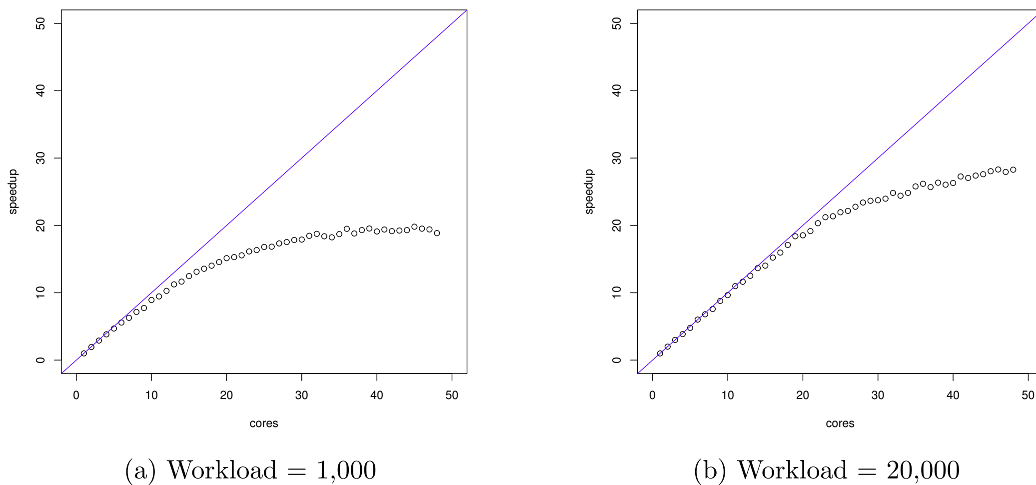


Figure 5.1: Fibonacci benchmark - the dependence of the speedup on the number of cores while the workload is fixed and dynamic scheduling

5.2.1 CPUs Dependence

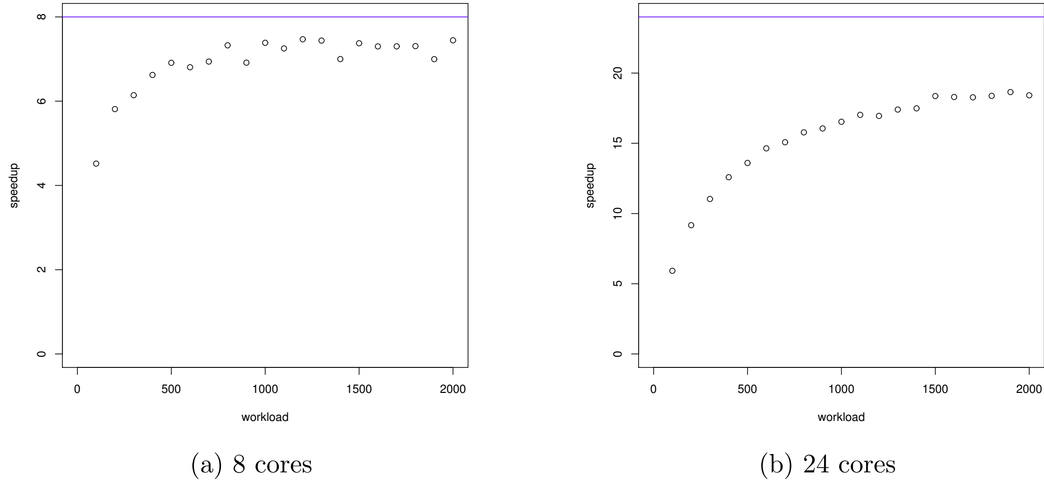
Initially, figure 5.1 is provided which demonstrates the dependence of the speedup on the CPUs number (denoted as cores). Two different workloads are displayed - a thousand (see figure 5.1a) and twenty thousand iterations (see figure 5.1b).

The observed dependence is as follows. Up to twenty used cores (especially with greater workload) the speedup almost reaches the theoretical optimum. Table 5.2 shows some configurations of workload and number of cores with computed correlation coefficients. We conclude that the scalability is almost linear even for higher number of threads (up to 20) since the correlation is classified as "very high" (according to table 5.1).

With more than twenty employed cores, the growth of the speedup declines as follows from figure 5.1. This phenomenon can be explained by two independent reasons. First, Amdahl's law limits the maximal speedup (see chapter 2.2 which is observable in greater a number of cores. Second, the lock contention among the threads rise whilst locking the task queue.

Cores	Workload	Coefficient
10	1,000	0.9987635
20	1,000	0.9960283
10	20,000	0.9994774
20	20,000	0.9992499

Table 5.2: Correlation coefficient labeling



(a) 8 cores

(b) 24 cores

Figure 5.2: Fibonacci benchmark with fixed CPUs, dynamic scheduling

5.2.2 Workload Dependence

Secondly, the dependence of the speedup on the workload size is presented, for which purpose the number of used cores was fixed. The 8 cores result, that simulates a regular desktop computer, is provided (see figure 5.2a) as well as the 24 cores result, that simulates a powerful server (see figure 5.2b). We observe that the increasing workload causes the growth of the speedup. In addition, the speedup stabilizes quickly.

Given the fixed number of CPUs, we test the difference between the mean of speedup on $1,000$ and $20,000$ workload. the null-hypothesis H_0 is tested against the one-sided alternative H_1 as follows.

H_0: "The means of the measured speedups are equal"
H_1: "The mean speedup measured on 1,000 workload is less than the mean speedup measured on 20,000 workload."

Anděl suggests using non-pair t-test [41]. The table 5.3 presents the computed p-values of this test for different fixed core numbers. In both cases, H_0 is rejected at previously set significance level since the computed p-values are less than α. Hence, we conclude that the workload *significantly* influences the speedup when the number of CPUs is fixed.

Cores	**1,000 mean**	**20,000 mean**	**p-value**
8	7.150499	7.588703	0.001656
48	18.85290	28.27256	$< 2.2 \times 10^{-16}$

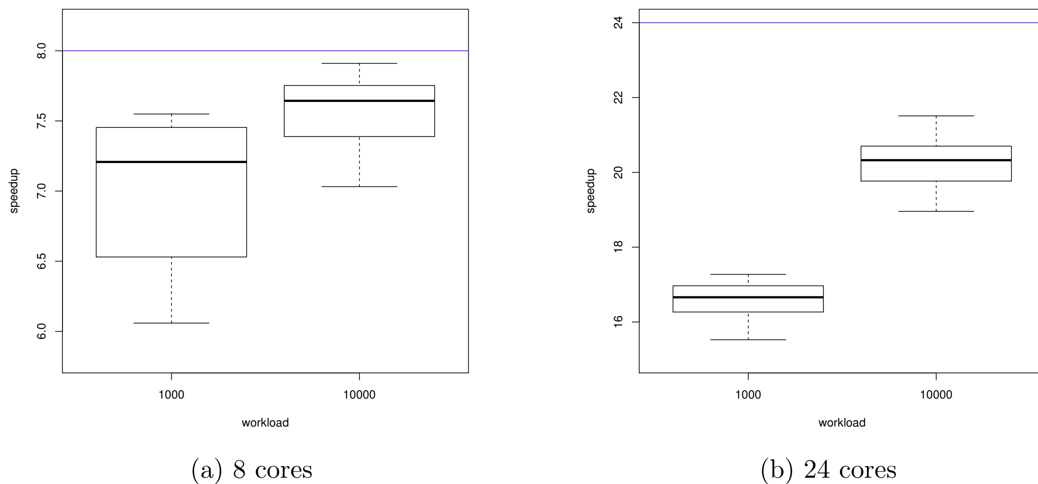
Table 5.3: Speedup dependence on the workload

Additionally, figure 5.2 illustrates speed of the convergence to the theoretical optimum, which suggests quick convergence. We interpret this phenomenon as that omp4j scales sufficiently even for small amounts of workload.

5.2.3 Speedup Distribution

Finally, the box-and-whisker diagrams are provided in order to illustrate the speedup distribution. Analogously to the prior charts, we provide both 8 CPUs (see figure 5.3a) and 24 CPUs (see figure 5.3b) results.

In addition, each plot contains both $1,000$ and $10,000$ iterations. The histograms together with the normality discussion may be found in appendix F.



(a) 8 cores (b) 24 cores

Figure 5.3: Fibonacci speedup distribution

5.2.4 Linear Model

Both simple and multiple regression is elaborated for Fibonacci benchmark in order to obtain two linear models representing different dependences [41].

The initial model aims to determine how much variation in the speedup is explained by the number of the cores. In order to provide maximal simplicity of the model and not to over-fit it, we used the quadratic dependence (see equation 5.1).

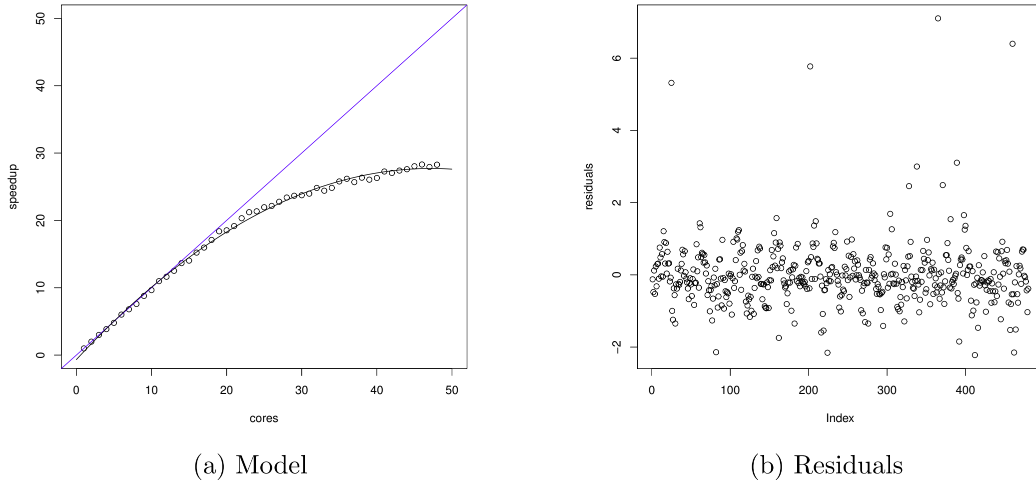
$$Speedup_i = \beta_0 + \beta_1 Cores_i + \beta_2 Cores_i^2 + \varepsilon_i \tag{5.1}$$

The significance level remains the same as defined in chapter 5.1.5. By using the ordinary least squares method [42] the $\hat{\beta}_i$ coefficients are estimated as follows from equation 5.2.

$$\hat{\beta}_0 = -0.66988 \quad \hat{\beta}_1 = 1.20196 \quad \hat{\beta}_2 = -0.01273 \tag{5.2}$$

Since the p-values of both $\hat{\beta}_1$ and $\hat{\beta}_2$ significances are less then 2×10^{-16}, all three coefficients remain in the model [43]. Due to Amdahl's law (see chapter 2.2), the speedup cannot be linear. Hence, $\hat{\beta}_2$ is non-zero and therefore *significant*. Additionally, the speedup cannot exceed the optimal speedup, hence $\hat{\beta}_2$ must be negative. In order to compensate for the quadratic decrease of the speedup growth, $\hat{\beta}_1$ is slightly greater than one [43]. Thus, he speedup is almost optimal for smaller numbers of cores (see table 5.3).

The visualization of the regression line that was estimated is shown in figure 5.4a. The computed residuals are demonstrated in (see figure 5.4b). Their unbiased appearance suggests that the model fits well. The coefficient of determination is calculated: $R^2 = 0.9889$ which suggests very good variance coverage. The p-value of the calculated F-statistics is less than 2.2×10^{-16}. According to the residuals plot and the attributes above, we conclude that the model fits well [44, 45]. For details of linear model refer to appendix G



(a) Model

(b) Residuals

Figure 5.4: Fibonacci linear model

Secondly, we create a linear model which depends on both the number of CPUs and the workload, hence a multiple regression is used. In order to maximize the simplicity of the model and provide a reasonable coefficient discussion, only polynomial variables bounded by degree of four are employed. The model follows equation 5.3.

$$Speedup_i = \beta_0 + \beta_1 Cores_i + \beta_2 Cores_i^2 + \beta_3 Cores_i^4 + \\ + \beta_4 Workload_i + \beta_5 Workload_i^2 + \varepsilon_i \quad (5.3)$$

Analogously to the prior linear model, the ordinary least square method [42] was used in order to estimate $\hat{\beta}_i$. The estimated coefficients are as follows from equations 5.4.

All the p-values of the variable significances are less than the set *significance level* (see chapter 5.1.5) [43]. Considering a fixed workload, $\hat{\beta}_1$ and $\hat{\beta}_2$ may be explained analogously to the proper coefficients of the simple model. Nevertheless, the quadratic dependence is stronger since it must balance the $\hat{\beta}_3$ that smooths the curve. On the contrary, considering the fixed number of cores, the influence of the workload is similar. However, since $\hat{\beta}_5$ is stronger than $\hat{\beta}_2$ in the prior model, we conclude that workload influences the total speedup less than the number of

cores. It implies, that the scalability of the benchmark is sufficient.

$$\hat{\beta}_0 = -3.905 \tag{5.4a}$$

$$\hat{\beta}_1 = 1.369 \tag{5.4b}$$

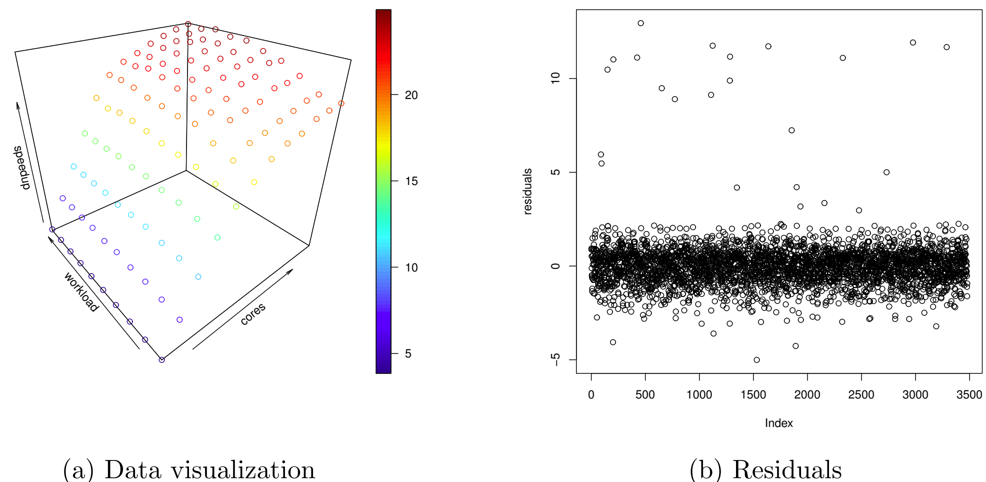
$$\hat{\beta}_2 = -2.198 \times 10^{-02} \tag{5.4c}$$

$$\hat{\beta}_3 = 1.873 \times 10^{-06} \tag{5.4d}$$

$$\hat{\beta}_4 = 6.726 \times 10^{-04} \tag{5.4e}$$

$$\hat{\beta}_5 = -3.660 \times 10^{-08} \tag{5.4f}$$

In order to maintain the clarity of the data, the model itself is not presented. Instead, a three-dimensional visualization of raw measured speedups is illustrated in figure 5.5a. As it may be observed, the core dependence is dominant. Furthermore, the residuals are plotted as proof of a good model (see figure 5.5b). Analogously to the prior model, the unbiased appearance suggests that the model fits well [45]. The coefficient of determination is calculated: $R^2 = 0.9688$ which suggests sufficient variance coverage. The p-value of the calculated F-statistics is less than 2.2×10^{-16}. According to the residuals and the attributes above, we analogously conclude that the model fits well [44]. For details of linear model we refer to appendix H



(a) Data visualization (b) Residuals

Figure 5.5: Fibonacci full linear model

According to residual plots of both previous models, we conclude that models do not appear to feature heteroscedasticity at all since we observe strong homoscedasticity instead [46]. We conclude that both models fit well the measured data and may be employed for speedup estimations. Nevertheless, the models are not appropriate for extrapolation.

5.3 Matrix Multiplication

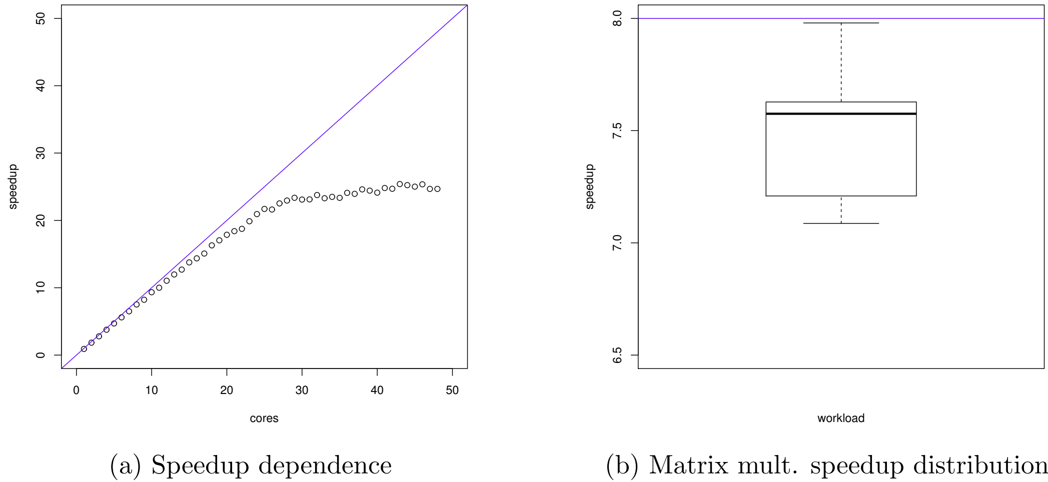
In the second benchmark, a more practical example - the matrix multiplication - is introduced. Initially two pseudo-random square matrices of shape $workload \times workload.$ are generated. The generator was seeded in order to behave deterministically for possible future benchmark repetition. The generated matrices are consequently multiplied according to the definition.

We decided to use this trivial approach in order not to use recursion, thus scalability of the problem is enhanced. Additionally, this approach is easily invoked in parallel since the result cells are computed independently of each other.

Even though `omp parallel for` was used analogously to the prior benchmark 5.2 and the amount of workload remains approximately the same, the measured speedups differ. This example stores the result in the memory which inevitably leads to the synchronization at some point.

Even though the workload is divided independently and no memory cell is accessed from multiple threads, the CPU cache must be synchronized among all CPUs. Hence, the observed speedups feature lower values then the Fibonacci benchmark.

The benchmark was run with the workload set to $2,000$. As figure 5.6a illustrates, the speedup of this benchmark is scalable similarly to the prior benchmark (see chapter 5.2) in terms of the observed shape of the dependence. The second figure (5.6b) indicates the speedup distribution.



(a) Speedup dependence

(b) Matrix mult. speedup distribution

Figure 5.6: Matrix multiplication speedup visualization

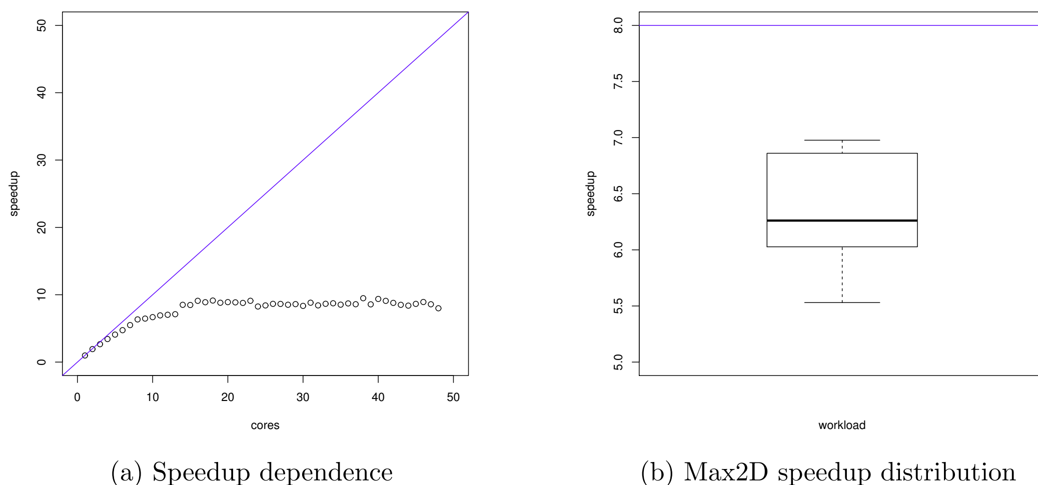
5.4 Function Maximum

Finally, as the last omp parallel for benchmark, we implemented the parallel version of the function maximum finding, that employs a critical section (omp critical). Function $f(x, y)$, that is defined in equation 5.5, is maximized over an integer matrix $[-3, 500; 3, 500)^2$.

$$f(x, y) = \log(x^2 + 6y) - \sin(x - y) + 4\sqrt{|x^3 - y^3|} - \cosh(3x) \tag{5.5}$$

The algorithm evaluates f in every integer point in the input interval by dividing total workload into chunks of 500 that are invoked in parallel. Nevertheless, the critical section, where the maximization happens, is an extreme bottleneck. Since all worker threads work approximately the same time, the contention on the critical section lock is enormous and leads to great overhead.

As a demonstration of the fact that the critical section lacks the scalability, the dependence of speedup on cores is presented in form of a graph (see figure 5.7a). We observe that the speedup steadily increases up to twelve CPUs where it reaches the value slightly less than ten. That value remains constant whilst the number of CPUs increases. Even though the scalability seems to be insufficient, the increasing number of CPUs does not cause any decrease of the speedup which is



(a) Speedup dependence

(b) Max2D speedup distribution

Figure 5.7: Function maximum speedup visualization

expected for increasing race conditions over the critical section mutex.

Altogether, the box-and-whisker diagram is provided as an example of speedup stability across various executions (see figure 5.7b).

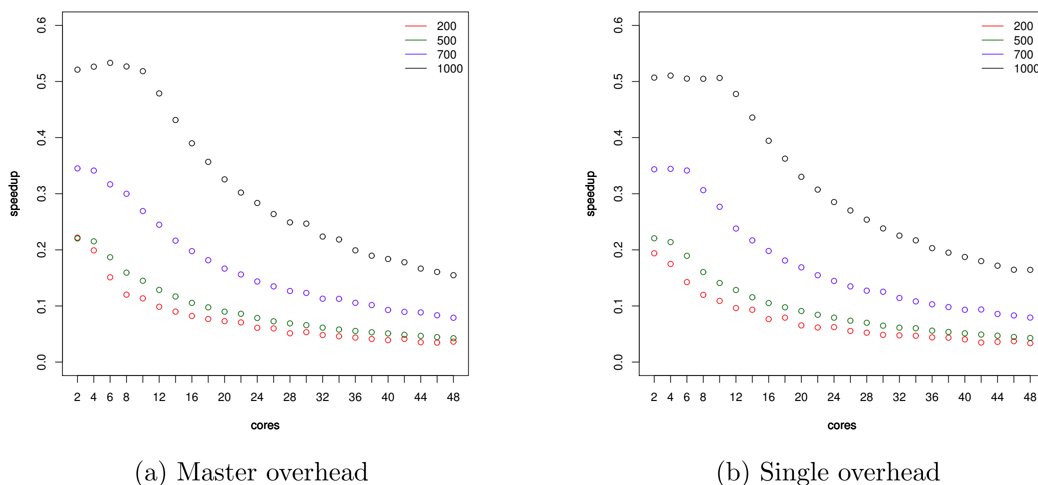
5.5 Single & Master Overhead

The inappropriate use of master or single directives causes an overhead. For the purpose of measuring its size, the following benchmark was developed. It measures the total speedup of completely useless directive usage, i.e. a master of single directive is located in a parallel region. Clearly, the directive body is executed only once, nevertheless the overhead caused by useless thread creations may be measured. Obviously, the total speedup is expected to be less than 1.

Levenshtein (edit) distance algorithm is executed given two pseudo-randomly generated strings of lengths equal to workload. The Java implementation of this algorithm from Wikipedia[1] was employed. The benchmark routines are implemented as follows from listing 5.2.

We observed that for non-trivial workloads the overheads are nearly unmeasurable as the mean speedup is not *significantly* lower than one. Hence, only

[1] `http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Java`



(a) Master overhead

(b) Single overhead

Figure 5.8: Overheads of single and master directives

smaller workloads are demonstrated in figure 5.8. According to the figure, even though the greater workload lowers the fraction of thread creation overhead, the growing number of CPUs implies the growth of the number of thread that are created. Consequently, a greater overhead is incurred.

```
/* reference algorithm */
runLevenstein(stringA, stringB);

/* parallel algorithm */
// omp parallel
{
  // omp master (single respectively)
  { runLevenstein(stringA, stringB); }
}
```

Listing 5.2: Master/Single benchmark design

According to the measured data, the single provides slightly higher speedup. This is caused by the fact, that the task may become executed during other threads creation. On the other hand, master requires only thread number 0 for its execution, possibly waiting for all other threads to be created. We recommend the use of single rather then master for that reason.

5.6 Related Projects Comparison

In order to provide an objective comparison among multiple projects that run on different platforms, a neutral benchmark across the languages - the Fibonacci

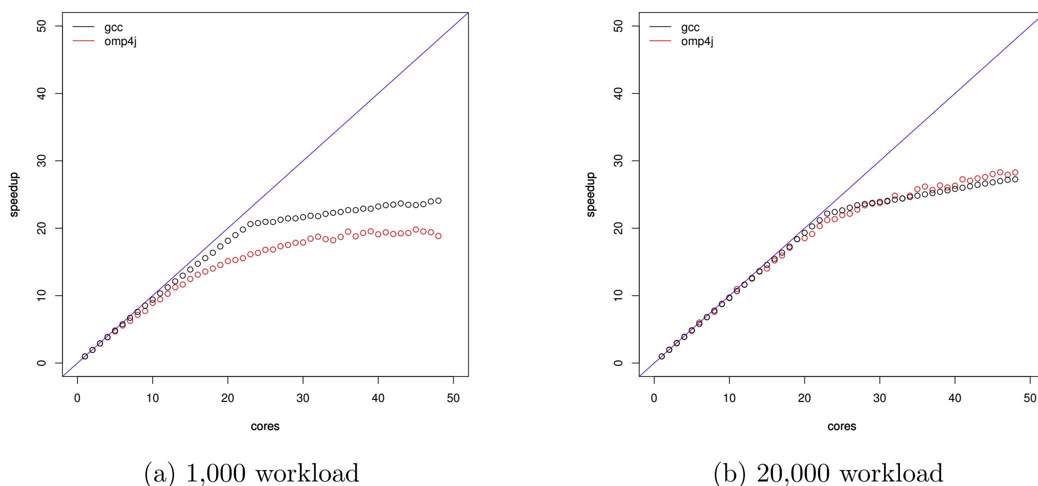
benchmark (described in chapter 5.2) - was chosen. By not storing the results into the shared memory, the data are cleared and the Java memory model effects are filtered out. Hence, the differences among different memory models are synchronization techniques are eliminated.

Three projects are compared - omp4j, gcc implementation of OpenMP and JOMP. All solutions are compared from the scalability point of view, i.e. the dependence of the speedup on the number of cores.

5.6.1 GCC Comparison

In order to provide practical results, the C++ source code was compiled only using g++ −Wall −O3 −fopenmp options. Figure 5.9 illustrates the comparison of gcc (black) and omp4j (red) on two fundamentally different workloads - 1,000 (see figure 5.9a) and 20,000 (see figure 5.9b).

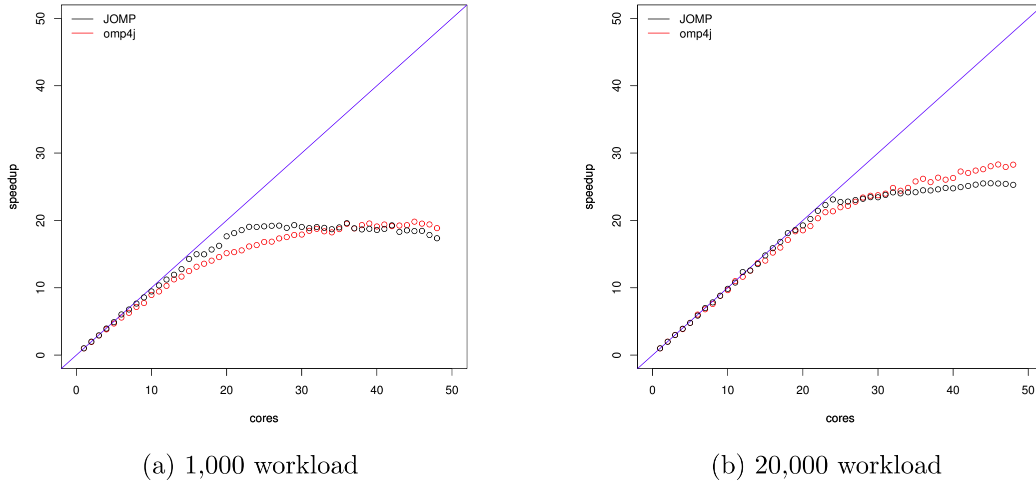
We observe that OpenMP implemented by gcc provides greater speedup in both shown workloads. This phenomenon is caused by the fact that OpenMP has built-in support directly in the gcc compiler which generates the final binary code. Hence, it might provide some low-level optimization for parallel code whilst omp4j preprocess only source codes.



(a) 1,000 workload

(b) 20,000 workload

Figure 5.9: GCC comparison on different workloads



(a) 1,000 workload

(b) 20,000 workload

Figure 5.10: JOMP comparison on different workloads

5.6.2 JOMP Comparison

Analogously to the prior C++ comparison, we aim to compare omp4j with JOMP that also provides source-level parallelization and runs directly on JVM (as described in chapter 1.3.1).

In order to create a valuable results, the same benchmark is used as well as the same workloads as the prior comparison. The results are shown in figure 5.10 that illustrates both measured workloads i.e. 1,000 and 20,000 iterations.

Considering the former workload (see figure 5.10a), we conclude that omp4j scales more fluently at the expense of the worse local performance around 24 CPUs. In order to determine whether omp4j performs *significantly* less satisfactory, unpaired t-test is used as Anděl suggests [41]. The significance level is set as follows from chapter 5.1.5. The null-hypothesis H_0 is tested against the one-sided alternative H_1 as follows.

H_0: "The mean of omp4j speedup is equal to the mean of JOMP speedup with fixed number of CPUs=24"
H_1: "The mean of omp4j speedup is *significantly* less than the mean of JOMP speedup with fixed number of CPUs=24"

The computed p-value is approximately equal to 4.968×10^{-10} which suggests the rejection of the hypothesis H_0. We conclude that when 24 cores are used,

JOMP provides *significantly* greater speedup then omp4j.

On the contrary, we might observe that with a higher number of CPUs, omp4j performs better. Using analogous hypotheses and t-test at the fixed number of cores equals 48 we reject H_0 with p-value equals to 3.709×10^{-05}. Hence omp4j scalability is *significantly* better.

Considering the latter figure (5.10b), we may observe that up to 30 used cores, both preprocessors perform equally. However, with increasing number of cores omp4j scales better. To prove this claim, the t-test with analogous hypotheses is employed. The null-hypothesis H_0 is rejected with p-value equal to 5.696×10^{-10}. Thus, we conclude that even for big workloads, omp4j scales better then JOMP.

Chapter 6.

User Documentation

In this chapter, a complex tutorial of downloading source codes, project compilation and installation is covered. Furthermore, the description of the supported program options is provided and a complex example of a real-world preprocessor usage is demonstrated.

6.1 Prerequisites

In order to run the preprocessor, having a proper Java Development Kit (JDK) installed is required since the Java compiler is frequently used during the process. omp4j is compatible with Java standards. It was explicitly tested on the following JDKs.

- OpenJDK 6
- OpenJDK 7
- Oracle JDK 7
- Oracle JDK 8 (recommended)

The preprocessor was initially developed for linux and UNIX systems. However, it may be used on MS Windows as well. The user must be aware of common pitfalls such as problematic environmental variable setting. For example, null must not be returned by ToolProvider.getSystemJavaCompiler(). We strongly discourage using the preprocessor on MS Windows since it was properly tested only on linux systems.

6.2 Preprocessor Installation

There are multiple ways of downloading and installing omp4j. This section provides a step-by-step tutorial regarding all important information. The preproces-

sor download may be achieved either by direct JAR download or by cloning the git repository and the following compilation.

6.2.1 Compiled Preprocessor

The compiled preprocessor including all dependencies may be downloaded from the project website[1] in the format of Java Archive (JAR) which may be executed directly from Java (e.g. $ java –jar omp4j.jar). Alternatively, the archive may be found in the digital attachment of this thesis.

6.2.2 Source Code

There are multiple ways of obtaining the project sources. The recommended way is using git CLI since the source code is developed using this system (hosted on GitHub). As the preprocessor repository depends on grammar and runtime submodules, the ––recursive option usage is suggested in order to download all dependent submodules [34].

```
# SSH
$ git clone --recursive git@github.com:omp4j/omp4j.git

# HTTPS
$ git clone --recursive https://github.com/omp4j/omp4j.git
```

Listing 6.1: Source download

The other possible way of accessing the source is to download directly a ZIP archive from GitHub [2]. However, this approach requires further submodule initialization since GitHub does not provide .git/ directory in the archive.

The compilation of downloaded sources is provided by Scala Build Tool (SBT), which is a popular building tool for Scala projects. For its proper usage, Scala and Scala compiler (scalac) must have been already installed. The tested version of Scala for SBT execution is v2.9.2+. However, SBT uses Scala v2.10.3 for compilation and execution of the preprocessor. This version is downloaded separately, hence even older installed Scala versions are valid. SBT may be used either as a

[1] http://www.omp4j.org/download
[2] https://github.com/omp4j/omp4j/archive/master.zip

command (e.g. $ sbt compile) or as an interactive CLI. We assume that the user prefers the interactive mode, thus the rest of the text follows this assumption. The preprocessor was developed and tested on version 0.13.6.

The compilation of source code is rather simple. Entering SBT CLI using $ sbt in project directory enables the user to command SBT in the interactive mode. The common commands may be as follows (see Table 6.1 on page 64).

Command	**Action**
compile	download all dependencies and compile all sources
run [args]	execute compiled program, passing [args] to the main function
test	run unit testing, printing (colorful) results
assembly	generate executable dependenceless .jar archive
doc	generate scaladoc API reference

Table 6.1: SBT interactive commands

Hence, in order to compile the project, run command must be selected. Before the compilation is started, SBT downloads all dependencies from the internet such as ANTLR and ScalaTest. If no internet connection is available, the user must provide all dependencies. This may by achieved either by the direct installation or by copying the proper JARs into lib/ directory. However, this approach is strongly recommended against. The ANTLR JAR may be found in the digital attachment of this thesis as well as the whole git repository.

The grammar repository[3] contains both Java 8 ANTLR grammar and the OMP grammar that was implemented as a part of this project. The former grammar is slightly modified in terms of different parser package. In addition, it allows the generated parser to access the single line comments. Apart from the grammars themselves (.g4 files), the repository also contains the generated parsers and lexers. Therefore, when omp4j is downloaded with −−recursive option, the grammars do not require further compilation. In order to recompile the grammars, update.sh may be executed as it downloads ANTLR complete JAR and compile the grammars.

[3]https://github.com/omp4j/grammar

6.3 Installation and Invocation

omp4j does not require any special installation since it is a regular JAR. The invocation of the preprocessor is as follows `$ java -jar omp4j.jar <params>`. We suggest that UNIX users create shell alias omp4j.

Alternatively, it may be run from SBT CLI via run command followed by parameters and flags. Nevertheless, the user must provide theirs own implementation of executors since the default executors are employed only while assembled JAR is used. We decided to design the behavior in this manner in order to simplify future development of executors. For regular purposes, the JAR assemblage is recommended for its higher performance.

Option		Behavior
Short	**Long**	
-d	--destdir	Directory where the output classes are stored.
-h	--help	Print help
-n	--no-compile	Do not compile preprocessed sources.
-v	--verbose	Provide progress information

Table 6.2: Additional CLI options

omp4j supports all javac CLI options. Additionally, it provides a few new options as it follows from table 6.2.

Finally, two examples of common omp4j usage are provided in listing 6.2. The former example behaves similarly to javac and thus may be used instead. The latter example, represents the bare preprocessor that does not compile final sources.

```
1 $ omp4j -d classes -v MyClass1.java MyClass2.java
2 $ omp4j -d sources --no-compile MyClass1.java MyClass2.java
```

Listing 6.2: omp example invocations

6.4 Example Input

Finally, a complex source code is provided as a demonstration of a problem from the real world. Parallel matrix multiplication is implemented by using OpenMP directive omp parallel for. For full code refer to appendix I.

More examples may be found in the digital attachment 7.3. The examples are also stored on GitHub[4]. The provided complex examples demonstrate all features of all directives. Some other examples may also be found on the project website[5].

[4]`https://github.com/omp4j/omp4j/tree/master/example`
[5]`http://www.omp4j.org/tutorial`

Chapter 7.
Project Website

This chapter describes the project website[1] which was developed in order to provide public information about installation and usage of omp4j preprocessor.

Furthermore, the website demonstrates the possible use of the preprocessor in terms of an external library. For this purpose, a live demo[2] is implemented which allows the user to submit a Java source code decorated with supported directives in order to obtain the processed result without the installation of the preprocessor.

7.1 Website Architecture

The whole website consists of two major parts - the front-end and the back-end. The former part provides a user interface and runs entirely on the client side (see chapter 7.1.1). The latter part handles requests and runs entirely on the server side (see chapter 7.1.2).

7.1.1 Front-end

AngularJS[3] - a powerful JavaScript library developed by Google Inc. - was employed in order to maximize the user experience.

The Angluar-based websites communicate with the back-end only via RESTful API. That provides complete freedom of back-end technology [47, 48].

The first page access requires the download of the whole page template, CSS styles and JavaScript libraries (c. 50kB for AngularJS scripts [49]). However, the following requests transfer the minimum amount of data as only the changing

[1] `http://www.omp4j.org`
[2] `http://www.omp4j.org/demo`
[3] `https://angularjs.org`

content is downloaded. Thus, once the page is loaded, the further navigation downloads only the changing content.

Therefore, the latency of navigation is minimized as well as the amount of transferred data which is the main advantage. Compared with regular websites, that must by fully downloaded with every request, AngularJS provides the modern approach of web development.

7.1.2 Back-end

As the back-end technology, Play![4] framework is employed as the modern and comfortable Scala-based solution for web development. Since the full Java interoperability is provided [50], the interaction with omp4j might be demonstrated since it is also a library.

Play! framework features Model-View-Controller design pattern (MVC) which is employed in order to provide distinct behavior of the sites. Two controllers are implemented - Application and Demo.

The former controller fetches html resources to the front-end. Thus, the time spent while proceeding a single request is minimized and the user may quickly navigate through the website.

The latter controller accepts a JSON object filled with the source code that the user requires to be processed via omp4j. Then the preprocessor is instanced and executed. Finally, the output is sent via HTTP response.

The preprocessor usage is rather simple. Initially, the configuration context must be created (org.ompj.Config) that is passed to the preprocessor constructor. The run method of the preprocessor stores processed source files into the directory defined by −d option.

Listing 7.1 demonstrates the described work-flow. Refer to appendix J for full Demo controller code.

[4] https://www.playframework.com

```scala
/* Create the configuration context */
val conf = new Config(Array(
  "-d", "path/to/output/directory",
  "--source-only",             // provide only processed sources
  "file1.java", "file2.java" // of file1.java and file2.java
))

/* Create preprocessor */
val prep = new Preprocessor()(conf)

/* Run the preprocessor */
prep.run()
```

Listing 7.1: omp4j as a library

7.2 Responsive Appearance

The website intends to be responsive to different screen sizes in order to maximize the user experience on mobile devices. For that purpose, Twitter Bootstrap[5] - a popular CSS framework - was employed.

By using proper CSS classes, the components behave properly according to size screen on which the website is presented [51, 52]. For example, the navigation menu collapses on mobile phones, thus it does not requires almost any space. Additionally, the margins are adjusted, hence the user does not mis-click when using touch-screens.

7.3 Hosting

Since the back-end is implemented in Scala, the most common web-hostings, that provide only Apache server and PHP, are not possible to use. Hence, a more modern service was chosen. Even though other services exist, such as Amazon AWS[6] and RedHat[7], Heroku provides overall support.

Thus, the website is hosted at Heroku[8] that offers a free-plan for small projects. Heroku was preferred for its great simplicity, git deployment [53, 34] and possible

[5] http://getbootstrap.com
[6] http://aws.amazon.com
[7] https://www.openshift.com
[8] https://heroku.com

scalability for future purposes.

Free plan at Heroku is limited as the dynos fall asleep after an hour of inactivity [54]. For that reason, the first request in the particular hour may take up dozens of second for processing.

Conclusion

The implemented OpenMP-like Java preprocessor - omp4j - fulfills the original requirements set out in the specification. The preprocessor is highly portable as it is compatible with all major JDKs. Additionally, it processes all modern versions of Java and the processed code is independent of any runtime libraries which is the main advantage of omp4j. In comparison with related projects, omp4j supports several qualities such as flexible grammar of the directives, which enhance the simplicity of the directive usage.

The discussed performance evaluation proves significantly better scalability of independent tasks execution than the similar JOMP project. With sufficiently large tasks provided, omp4j presents comparable results with gcc implementation of OpenMP.

Additionally to the original requirements, the following directives and attributes implemented

- omp master directive
- omp single directive
- firstprivate attribute
- threadNum attribute
- schedule attribute

schedule attribute is recognized which allows the user to employ either already provided executors - dynamic and static - or to implement a new executor that better suits the current situation.

The preprocessor may be employed in three different ways - a source-to-source preprocessor, a source-to-bytecode compiler and a third-party library. The last mentioned approach is demonstrated by a live demo that is presented on the project website[9]. The website was developed by using modern technologies such as AngularJS and Play! framework.

[9] http://www.omp4j.org

The major disadvantage of omp4j is the complex and complicated source code analysis which heavily depends on ANTLR and its Java 8 grammar. However, the analysis is essential as it leads to the output which a programmer would produce. In comparison with related projects, omp4j does not move the parallel region into a separate class. We conclude that this behavior is beneficial for educational purposes since it minimizes the amount of code transformation.

As a possible future improvement, Java 8 grammar might be reimplemened in order to develop a more effective, simpler and properly tested ANTLR resource, that is employed in the syntax analysis. Additionally, the project could be extended in order to implement the whole OpenMP 3.1 standard. For that purpose, the reduction attribute and omp task directive should be implemented.

Bibliography

[1] Yifan Zhang et al. *Towards Better CPU Power Management on Multicore Smartphones*. URL: http://research.microsoft.com/en-us/um/people/zhao/pubs/hotpower13gpu.pdf (visited on 04/16/2015).

[2] Oracle. *Java Platform, Standard Edition 8 Names and Versions*. URL: http://www.oracle.com/technetwork/java/javase/jdk8-naming-2157130.html (visited on 05/02/2015).

[3] Oracle. *The History of Java Technology*. URL: http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-19835.html (visited on 04/16/2015).

[4] Mark Bull. *JOMP*. URL: https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/index_1.html (visited on 05/03/2015).

[5] Mark Bull. *JOMP - download*. URL: https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/download.html (visited on 05/03/2015).

[6] J. M. Bull et al. *Towards OpenMP for Java*. EWOMP 2000. URL: https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/papers/ewomp00.pdf (visited on 05/03/2015).

[7] M. Klemm et al. *JaMP: An Implementation of OpenMP for a Java DSM*. URL: http://www.researchgate.net/profile/Michael_Klemm2/publication/220105283_JaMP_an_implementation_of_OpenMP_for_a_Java_DSM/links/02e7e52827539be4a9000000.pdf (visited on 05/03/2015).

[8] Vikas, Nasser Giacaman, and Oliver Sinnen. "Pyjama: OpenMP-like implementation for Java, with GUI extensions". In: *The 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*. Shenzhen, China: ACM, Feb. 2013. URL: http://homepages.engineering.auckland.ac.nz/~ngia003/files/vikas_pyjama_workshop_2013.pdf (visited on 05/05/2015).

[9] Nasser Giacaman and Oliver Sinnen. *Pyjama (PJ) help*. URL: http://homepages.engineering.auckland.ac.nz/~parallel/ParallelIT/downloads/Pyjama/PJHelp1.3.x.pdf (visited on 05/03/2015).

[10] James Conway. *Starting Akka.net*. Apr. 7, 2015. URL: http://blog.jaywayco.co.uk/starting-akka-net (visited on 05/03/2015).

[11] Martin Děcký. *Výkonnost počítače, Architektura počítačů*. URL: http://d3s.mff.cuni.cz/teaching/computer_architecture/docs/02_vykonnost.pdf (visited on 04/15/2015).

[12] Manek Dubash. "Moore's Law is dead, says Gordon Moore". In: *Techworld* (2010). URL: http://www.techworld.com/news/operating-systems/moores-law-is-dead-says-gordo-moore-3576581 (visited on 04/14/2015).

[13] Kibum Kang et al. "High-mobility three-atom-thick semiconducting films with wafer-scale homogeneity". In: *Nature* (Apr. 2015). URL: `http://www.nature.com/nature/journal/v520/n7549/abs/nature14417.html` (visited on 05/02/2015).

[14] Victoria Zislina. *Why has CPU frequency ceased to grow?* URL: `https://software.intel.com/en-us/blogs/2014/02/19/why-has-cpu-frequency-cease-to-grow` (visited on 04/15/2015).

[15] Gene Myron Amdahl. "AFIPS Spring Joint Computer Conference". In: American Federation of Information Processing Societies. 1967.

[16] Jakub Yaghob. *Programování v paralelním prostředí Teorie paralelního programování*. URL: `http://www.ksi.mff.cuni.cz/lectures/NPRG042/html/ppp-01-theory-en.ppt` (visited on 04/15/2015).

[17] Vladimir Kozlov and Sandhya Viswanathan. *Optimizing the future Java through collaboration*. Intel Corporation.

[18] OpenMP ARB. *About the OpenMP ARB and OpenMP.org*. URL: `http://openmp.org/wp/about-openmp` (visited on 05/01/2015).

[19] Barbara Chapman, Gabriele Jost, and Ruud van van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. One Rogers Street, Cambridge MA, USA: The MIT Press, Oct. 2007. ISBN: 978-0262533027.

[20] Matthijs van Waveren. *OpenMP 4.0 API Released*. URL: `http://openmp.org/wp/openmp-40-api-released` (visited on 04/13/2013).

[21] OpenMP ARB. *OpenMP Compilers*. URL: `http://openmp.org/wp/openmp-compilers` (visited on 05/03/2015).

[22] Aleksandar Prokopec and Heather Miller. *Parallel Collections, Scala Documentation*. URL: `http://docs.scala-lang.org/overviews/parallel-collections/overview.html` (visited on 04/16/2015).

[23] Tutorials Point (I) Pvt. Ltd. *C++ Multithreading, Tutorialspoint*. URL: `http://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm` (visited on 04/16/2015).

[24] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, Jan. 2013. ISBN: 978-1934356999.

[25] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Pro Git (Expert's Voice in Software Development)*. 2nd. Apress and Addison Wesley, Nov. 2000. ISBN: 978-0201441246.

[26] Oracle. *Executors (Java Platform SE 7)*. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool(int)` (visited on 05/05/2015).

[27] OpenMP ARB. *OpenMP Application Program Interface*. July 2013. URL: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[28] Mark Kambites. *Java OpenMP*. EPCC SSP 1999. URL: `https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/papers/ssp059.pdf` (visited on 04/24/2015).

[29] J. M. Bull and M. E. Kambites. “JOMP—an OpenMP-like Interface for Java”. In: *ACM Java Grande 2000 Conference*. San Francisco, California, USA: ACM, June 2000. URL: `https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/papers/acmJG0.pdf`.

[30] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. 2nd. Artima Inc, Jan. 2011. ISBN: 978-0981531649.

[31] Martin Odersky. *What is Scala?* URL: `http://www.scala-lang.org/what-is-scala.html` (visited on 04/17/2015).

[32] Paul Chiusano and Rúnar Bjarnason. *Functional Programming in Scala*. 1st. Manning Publications, Sept. 2014. ISBN: 978-1617290657.

[33] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. 1st. O’Reilly Media, Dec. 2008. ISBN: 978-0596514983.

[34] Scott Chacon. *Pro Git (Expert’s Voice in Software Development)*. 1st. Apress, Aug. 2009. ISBN: 978-1430218333.

[35] Scott Bellware. “Behavior-Driven Development”. In: *Code magazine* (June 2008). URL: `http://www.codemag.com/article/0805061` (visited on 04/16/2015).

[36] Daniel Hinojosa. *Testing in Scala*. 1st. O’Reilly Media, Dec. 2012. ISBN: 978-1449315115.

[37] Jiří Matoušek and Jaroslav Nešetřil. *Kapitoly z diskrétní matematiky*. 4th. Karolinum, 2009, pp. 34–36. ISBN: 978-80-246-1740-4.

[38] Antonín Steinhauser et al. “DOs and DON’Ts of Conducting Performance Measurements in Java”. In: *The 6th ACM/SPEC International Conference On Performance Engineering*. Ed. by Yolande Berbers and Willy Zwaenepoel. New York, USA: ACM, Feb. 2015, pp. 337–340. ISBN: 978-1-4503-3248-4. DOI: `10.1145/2668930.2688820`. URL: `http://d3s.mff.cuni.cz/~steinhauser/icpe2015tt.pdf`.

[39] Jiří Anděl. *Statistické metody*. Prague: MATFYZPRESS, 2007. ISBN: 8073780038.

[40] Richard Taylor. “Interpretation of the Correlation Coefficient: A Basic Review”. In: *Journal of Diagnostic Medical Sonography* (1 1990). URL: `http://www.sagepub.com/salkind2study/articles/05Article01.pdf` (visited on 04/21/2015).

[41] Jiří Anděl. *Základy matematické statistiky*. 3rd. Prague: MATFYZPRESS, 2011. ISBN: 978-80-7378-162-0.

[42] Milan Hladík. *Lineární algebra (nejen) pro informatiky*. URL: `http://kam.mff.cuni.cz/~hladik/LA/text_la.pdf` (visited on 04/21/2015).

[43] Jim Frost. *How to Interpret Regression Analysis Results: P-values and Coefficients*. URL: `http://blog.minitab.com/blog/adventures-in-statistics/how-to-interpret-regression-analysis-results-p-values-and-coefficients` (visited on 05/08/2015).

[44] Jim Frost. *Regression Analysis Tutorial and Examples*. URL: `http://blog.minitab.com/blog/adventures-in-statistics/regression-analysis-tutorial-and-examples` (visited on 05/08/2015).

[45] Jim Frost. *Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit?* URL: http://blog.minitab.com/blog/adventures-in-statistics/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit (visited on 05/08/2015).

[46] Karel Zvára. *Regrese*. Prague: MATFYZPRESS, 2008. ISBN: 978-80-7378-041-8.

[47] Ari Lerner. *ng-book - The Complete Book on AngularJS*. 1st. Fullstack io, Dec. 2013. ISBN: 978-0991344604.

[48] Leonard Richardson. *RESTful Web APIs*. 1st. O'Reilly Media, Sept. 2013. ISBN: 978-1449358068.

[49] Google. *AngularJS: Miscellaneous: FAQ*. URL: https://docs.angularjs.org/misc/faq (visited on 05/03/2015).

[50] Peter Hilton, Erik Bakker, and Francisco Canedo. *Play for Scala: Covers Play 2*. 1st. Manning Publications Co., Oct. 2013. ISBN: 978-1617290794.

[51] Alan Forbes. *The Joy of Bootstrap: A smarter way to learn the world's most popular web framework*. 1st. Plum Island Publishing, Aug. 2014.

[52] Core Team of Bootstrap. *Components - Bootstrap*. URL: http://getbootstrap.com/components/#navbar (visited on 05/03/2015).

[53] Neil Middleton and Richard Schneeman. *Heroku: Up and Running*. 1st. O'Reilly Media, Nov. 2013. ISBN: 978-1449341398.

[54] Inc. Heroku. *Dynos and the Dyno Manager*. URL: https://devcenter.heroku.com/articles/dynos#dyno-sleeping (visited on 05/03/2015).

[55] Typesafe Inc. *Getting Started with sbt: Directory structure*. URL: http://www.scala-sbt.org/0.13/tutorial/Directories.html (visited on 05/10/2015).

List of Figures

1.1 Actor pattern 11

2.1 Amdahl's law - γ influence 14

2.2 Fork-Join model 17

3.1 Level of preprocessing 20

3.2 Preprocessor work-flow 21

3.3 Processing Phases 27

3.4 omp sections overhead 34

3.5 Master and single directives work-flows 35

3.6 Critical section 37

5.1 Fibonacci CPU dependence 49

5.2 Fibonacci benchmark with fixed CPUs, dynamic scheduling 50

5.3 Fibonacci speedup distribution 51

5.4 Fibonacci linear model 53

5.5 Fibonacci full linear model 54

5.6 Matrix multiplication speedup visualization 56

5.7 Function maximum speedup visualization 57

5.8 Overheads of single and master directives 58

5.9 GCC comparison on different workloads 59

5.10 JOMP comparison on different workloads 60

List of Tables

2.1 Flynn's taxonomy 15

3.1 Preprocess level notation 20

4.1 GitHub repository overview 43

5.1 Correlation coefficient labeling 48

5.2 Correlation coefficient labeling 50

5.3 Speedup dependence on the workload 51

6.1 SBT interactive commands 64

6.2 Additional CLI options 65

1 Preprocessor work-flow 81

Attachments

This chapter describes the enclosed digital attachments. All attachments are compressed in a single ZIP archive which content is explained below.

omp4j/

The preprocessor repository. It contains all sources that are required for compilation (see chapter 6.2.2). The whole directory follow the SBT project structure [55].

omp4j/examples/ This directory contains advanced examples such as lambda function translation and various anonymous classes. The files are also employed as the unit tests.

omp4j/src/ Preprocessor source root following SBT project structure [55].

benchmark/

The benchmark repository which contains the whole benchmarking framework.

benchmark/resources/ This directory contains all resources that were measured.

benchmark/statistics/ The statistical analysis scripts and resources. The data/ subdirectory contains all measured data in .csv format.

www/

The website repository. In order to run the website locally, Activator platform must be installed. By using activator ui, the proper directory can be selected and the application run. However, the website runs also online[10] in order to provide a simple access without the local installation.

omp4j-1.2.jar

The assembled dependenceless preprocessor. The JAR was compiled with javac 1.6, thus, it should run on all supported JVMs.

antlr-runtime-4.4.jar

The packed ANTLRv4 grammar recognizer. It is automatically downloaded from the internet[11] during the compilation. However, it must be provided

[10] http://www.omp4j.org

[11] http://omp4j.petrbel.cz/antlr-runtime-4.4.jar

by the user if the internet connection is not available.

thesis.pdf

A digital copy of this text.

Appendix A Preprocessor Work-flow

Table 1 presents the preprocessor work-flow in detail. Used packages and classes are highlighted in the following columns.

<table>
<tr><th>#</th><th>Phase description</th><th colspan="3">Packages and classes used in the particular phase</th></tr>
<tr><td>1</td><td>Configuration creation</td><td>Config</td><td colspan="2"></td></tr>
<tr><td>2</td><td>Code analysis</td><td>Extractor, System and Tree packages</td><td></td><td rowspan="5">Preprocessor</td></tr>
<tr><td>3</td><td>Directive recognition</td><td>DirectiveVisitor</td><td rowspan="2">Directive</td></tr>
<tr><td>4</td><td>Top level directives translation</td><td>Translator, TranslatorVisitor, ANTLR-Rewriter</td></tr>
<tr><td>5</td><td>Output writing</td><td>Utils-package</td><td></td></tr>
<tr><td>6</td><td>GOTO (2) a directive exists</td><td></td><td></td></tr>
</table>

Table 1: Preprocessor work-flow

Appendix B OMP Directive

This appendix provides a fraction of implemented OMP grammar in .g4 format. The common lexer rules are omitted as well as similar rules such as omp parallel for which is basically the same as omp parallel. The latter directive demonstrates the random access memory usage. The complete grammar may be found in the digital attachment.

```
grammar OMP;

// PARSER RULES
ompUnit : OMP (
                  ompParallel      |
                  ompParallelFor   |
                  ompFor           |
                  ompSections      |
                  ompSection       |
                  ompSingle        |
                  ompMaster        |
                  ompBarrier       |
                  ompAtomic        |
                  ompCritical
                ) EOF ;

ompParallel
    locals [static java.util.HashSet<String> names]
    @init {
        OmpParallelContext.names = new java.util.HashSet<String>();
    }
    : PARALLEL ompParallelModifiers ;

    ompParallelModifiers
        : ompParallelModifier ompParallelModifiers
        |
        ;

    ompParallelModifier
        // Ensure that no duplicates have been provided
        // schedule, threadNum and accessModifiers
        : {!$ompParallel::names.contains("schedule")}?  ompSchedule
    {$ompParallel::names.add("schedule");}
        | {!$ompParallel::names.contains("threadNum")}? threadNum
    {$ompParallel::names.add("threadNum");}
        | ompAccessModifier
        ;

ompParallelFor ...
ompSections    ...

ompFor          : FOR ompAccessModifier*          ;
ompSection      : SECTION                         ;
ompSingle       : SINGLE                          ;
ompMaster       : MASTER                          ;
ompBarrier      : BARRIER                         ;
ompAtomic       : ATOMIC                          ;
ompCritical     : CRITICAL ( '(' ompVar ')' )? ;

ompPublic       : PUBLIC        ;
```

```
49 ompPrivate        : PRIVATE      ;
50 ompFirstPrivate   : FIRSTPRIVATE ;
51
52 ompSchedule         : SCHEDULE '(' ( VAR | '.' )*? ')' ;
53 threadNum           : THREAD_NUM '(' ompNumber ')' ;
54 ompAccessModifier : ( ompPublic | ompPrivate | ompFirstPrivate )
       '(' ompVars ')' ;
55
56 ompVars    : ( ompVar | ( ompVar ',' )+ ompVar ) ;
57 ompVar     : VAR                                 ;
58 ompNumber  : NUMBER                              ;
59
60
61 // LEXER RULES
62 OMP       : 'omp'       ;
63 PARALLEL  : 'parallel'  ;
64 FOR       : 'for'       ;
65 SECTIONS  : 'sections'  ;
66 SECTION   : 'section'   ;
67 SINGLE    : 'single'    ;
68 MASTER    : 'master'    ;
69 BARRIER   : 'barrier'   ;
70 ATOMIC    : 'atomic'    ;
71 CRITICAL  : 'critical'  ;
72
73 PUBLIC        : 'public'       ;
74 PRIVATE       : 'private'      ;
75 FIRSTPRIVATE  : 'firstprivate' ;
76 SCHEDULE      : 'schedule'     ;
77 THREAD_NUM    : 'threadNum'    ;
78
79 ...
```

Appendix C IOMPExecutor

This appendix provides the executor interface. This interface is final and may not be modified in order to maintain backward compatibility of the processed files.

```
1  package org.omp4j.runtime;
2
3  import java.util.concurrent.Executor;
4
5  /**
6   * Extension to the Executor class. Provides methods for omp4j
      preprocessor.
7   */
8  public interface IOMPExecutor extends Executor {
9
10   /** Block current thread until all tasks are finished. */
11   public void waitForExecution();
12
13   /** Get unique id of the thread from which the method is called.
       The range is [0, threadNum) */
14   public int getThreadNum();
15
16   /** Return total number of thread used. This number is usually
       the same as the one given to the constructor. */
17   public int getNumThreads();
18
19   /**
20    * Simulate barrier hit. Delay this thread until all threads have
       hit the barrier.
21    * @param barrierName name of the barrier hit.
22    */
23   public void hitBarrier(String barrierName);
24 }
```

Appendix D hitBarrier implementation

The following source code demonstrates simple hitBarrier implementation that is provided by org.omp4j.runtime.AbstractExecutor class.

```
1
2  protected ConcurrentHashMap<String, CyclicBarrier> barriers;
3
4  public void hitBarrier(String barrierName) {
5
6    CyclicBarrier barr = null;
7
8    // if barrier already exists, fetch it
9    if (barriers.containsKey(barrierName)) {
10     barr = barriers.get(barrierName);
11   } else {
12     // else block
13     synchronized(barriers) {
14       // check whether some other thread have created it while this
       thread was waiting before synchronized block
15       if (barriers.containsKey(barrierName)) {
16         barr = barriers.get(barrierName);
17       // if not, create it by itself
18       } else {
19         barr = new CyclicBarrier(numThreads);
20         barriers.put(barrierName, barr);
21       }
22     }
23   }
24
25   try {
26     barr.await();
27   } catch (BrokenBarrierException e) {
28     System.err.println("An BrokenBarrierException occurred while
       processing barrier '" + barrierName +"'. This is unexpected
       behavior probably caused by thread manipulation. Please do not
       access threads created by the executors.");
29     System.exit(1);
30   } catch (InterruptedException e) {
31     System.err.println("An InterruptedException occurred while
       processing barrier '" + barrierName +"'. This is unexpected
       behavior probably caused by thread manipulation. Please do not
       access threads created by the executors.");
32     System.exit(1);
33   }
34 }
```

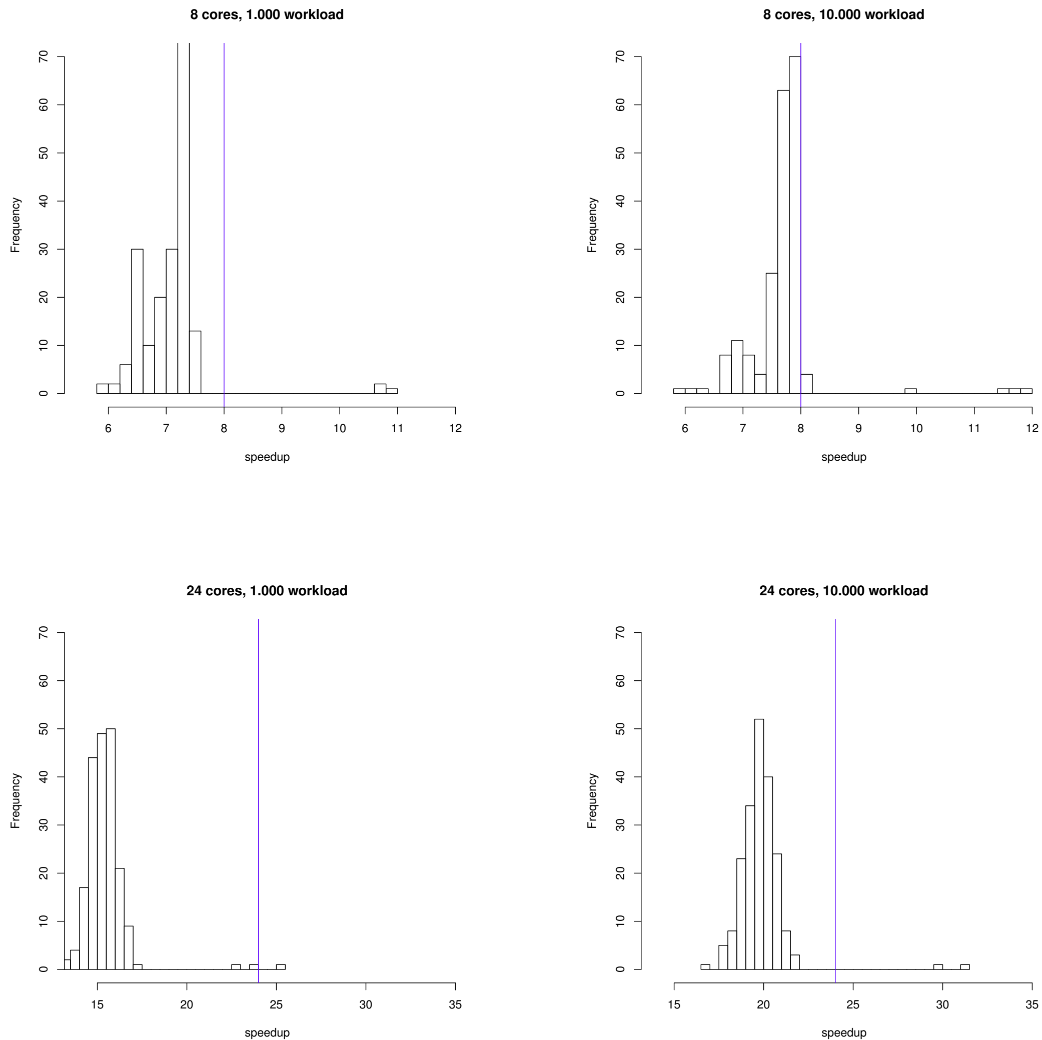
Appendix E BSD License

This appendix provides full BSD license. All project repositories are published under the following license including the text of this thesis.

```
[The "BSD license"]
Copyright (c) 2015 Petr Belohlavek
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote products
    derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Appendix F Fibonacci Distribution

This appendix provides several distribution charts. The Fibonacci benchmark (see chapter 5.2) was executed 200 times for each configuration. We decided to measure both 8 and 24 cores cases combined with workload of 1.000 and 10.000. The following histograms demonstrates the distribution.



The measured data contains values that exceed the optimum speedup. This phenomenon is caused by pairwise speedup computation and is eliminated by CLT [39]. The distributions appear normal, however, Shapiro-Wilk test rejects the normality hypothesis because of the described phenomenon. We decided not to filter out these results in order to provide clear data.

Appendix G Simple Linear Regression

This appendices G and H present the outputs of the R command summary applied to the first and second linear models which are described in chapter 5.2.4.

```
Call:
lm(formula = dataset$speedup ~ dataset$cores + I(dataset$cores^2))

Residuals:
    Min      1Q  Median      3Q     Max
-2.2220 -0.3854 -0.0917  0.3252  7.1028

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)        -0.669882   0.126153   -5.31 1.68e-07 ***
dataset$cores       1.201956   0.011876  101.20  < 2e-16 ***
I(dataset$cores^2) -0.012727   0.000235  -54.16  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8832 on 477 degrees of freedom
Multiple R-squared:  0.9889,  Adjusted R-squared:  0.9888
F-statistic: 2.122e+04 on 2 and 477 DF,  p-value: < 2.2e-16
```

Listing 2: Simple linear regression summary

Appendix H Multiple Linear Regression

```
Call:
lm(formula = rdataset$speedup ~ rdataset$cores +
    I(rdataset$cores^2) +
     I(rdataset$cores^4) + rdataset$workload +
    I(rdataset$workload^2))

Residuals:
    Min      1Q  Median      3Q     Max
-5.0056 -0.6009  0.0208  0.5325 12.9548

Coefficients:
                         Estimate Std. Error t value Pr(>|t|)
(Intercept)            -3.905e+00  1.201e-01  -32.53   <2e-16 ***
rdataset$cores          1.369e+00  1.355e-02  101.05   <2e-16 ***
I(rdataset$cores^2)    -2.198e-02  4.364e-04  -50.36   <2e-16 ***
I(rdataset$cores^4)     1.873e-06  9.286e-08   20.17   <2e-16 ***
rdataset$workload       6.726e-04  3.001e-05   22.41   <2e-16 ***
I(rdataset$workload^2) -3.660e-08  2.660e-09  -13.76   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.14 on 3472 degrees of freedom
Multiple R-squared:  0.9688,  Adjusted R-squared:  0.9688
F-statistic: 2.156e+04 on 5 and 3472 DF,  p-value: < 2.2e-16
```

Listing 3: Multiple linear regression summary

Appendix I Matrix Multiplication

The following source code was used in matrix multiplication benchmark (see chapter 5.3).

```
protected void runBenchmark(final int[][] A, final int[][] B) {

  int result[][] = new int[workload][workload];

  // omp parallel for schedule(dynamic)
  for (int i = 0; i < workload; i++) {
    for (int j = 0; j < workload; j++) {
      int d = 0;
      for (int k = 0; k < workload; k++) {
        d += A[i][k] * B[k][j];
      }
      result[i][j] += d;
    }
  }
}
```

Appendix J Demo Controller

This appendix introduces a controller that is employed in website online demo. As it may be seen, the use of omp4j as a library is extremely simple.

```
object Demo extends Controller {
  def translate = Action(parse.json) { implicit request =>
    val f_in = File.createTempFile("pre", ".java")
    var f_out: File = null
    var toDelete: List[File] = null
    var conf: Config = null

    val writer = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(f_in)))

    try {
      val source: play.api.libs.json.JsValue =
request.body.\("source")

      val code: String = source.toString
      writer.write(StringContext treatEscapes code.substring(1,
code.length-1))
      writer.flush()

      conf = new Config(Array("-d",
System.getProperty("java.io.tmpdir"), "--source-only",
f_in.getAbsolutePath))    // set up configuration based on
program arguments
      val prep = new Preprocessor()(conf)    // create preprocessor

      val (translatedFiles, dirs) = prep.run()
      val tmpDirs :+ ((tmpDir, prepDir)) :+ ((lastDir,
lastPrepDir)): List[(File, File)] = dirs
      toDelete = tmpDir :: prepDir :: lastDir :: lastPrepDir ::
tmpDirs.foldLeft[List[File]](List()){ case (z, (a,b)) => a :: b
:: z}

      f_out = translatedFiles.head
      val output = scala.io.Source.fromFile(f_out).mkString
      Ok(output)

    } catch {
      case e: Exception => Ok(e.getMessage)
    } finally {
      writer.close()

      try f_in.delete()
      catch {case e: NullPointerException => ; }

      try f_out.delete()
      catch {case e: NullPointerException => ; }

      try toDelete.foreach(FileTreeWalker.recursiveDelete)
      catch {case e: NullPointerException => ; }

      try FileTreeWalker.recursiveDelete(conf.workDir)
      catch {case e: NullPointerException => ; }
    }
  }
}
```