Metric-based Software Parallelisability Analyzer

Aleksandr Maramzin

Master of Science by Research

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2018

Abstract

Parallelism pervades the modern computing world. Almost all modern computing systems provide parallel computing resources to some degree or another. The major problem in the field is that these available resources are not always efficiently utilized. To take the most out of these parallel resources, applications running on them must be parallel as well.

Despite progress in parallel programming language design and increased availability of parallel programming frameworks, writing efficient parallel software from scratch is still a challenging task mastered by only a few expert programmers. While these experts combine domain knowledge, algorithmic insight and parallel programming skills, most average programmers are often lacking skills in at least one of these areas. In this project we investigate methods for providing programmers with real-time feedback on the quality of their code with respect to parallelisation opportunities and scalability to address short-comings before they manifest as bad and hard-to-parallelise code.

We draw on the experience of the software engineering community and software metrics originally developed to identify bad sequential code, typically prone to errors and hard to maintain. The ambition of this project is to develop novel software parallelisability metrics, which can be used as quality indicators for parallel code and guide the software development process towards better parallel code.

Acknowledgements

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Aleksandr Maramzin*)

Table of Contents

Chapter 1

Introduction

Parallelism pervades the modern computing world. In the past parallel computations used to be employed only in high performance scientific systems, but now the situation has changed. Parallel elements present in the design of almost all modern computers from small embedded processors to large-scale supercomputers and computing networks. Unfortunately, these immense parallel computing resources are not always fully utilized during computations due to several problems in the field:

1. Abundance of legacy applications from previous sequential computing era. That abundance is one source of problems. Legacy applications are not designed to run on parallel machines and, by default, do not take advantage of all underlying resources. Automatic parallelisation techniques have been developed to transform these sequential applications into parallel ones. However, these techniques cannot efficiently deal with some codes in the spectrum of existent applications. Pointer-based applications with irregular data structures, applications with loop carried dependencies and entangled control flow have proven to be challenging to automatic parallelisation. Very often such programs hide significant amounts of parallelism behind suboptimal implementation constructs and represent meaningful potential for further improvements.

2. Difficulty of manual parallel programming. Hidden potential can be realised by writing parallel programs (applications designed to run on parallel systems) manually. However, the task of manual parallel programming is rather challenging by itself. To create efficient and well-designed parallel software programmer must be aware of application's domain field, must have good algorithmic background as well as solid general programming skills and working knowledge of exact parallel programming framework they are using. Most average programmers lack some of the necessary skills out of that set, which hinders the potential of manual parallelisation. Sometimes

sloppy program parallelisation can even slow sequential programs down due to parallel synchronisation/communication overhead incurred. In our project we propose to research the question of software parallelisability metrics. This research idea draws on the existent work in the area of software quality, where numerous software metrics have been proposed. Section 3 of this proposal gives a brief overview of the major software metrics to date. In many cases they can be used to supplement software engineering expertise and common sound judgement when it comes to engineering and managerial decisions during software development. These metrics are designed to address the issues of source code complexity, testability, maintainability, etc. and usually show a good correlation between these properties of software and their values. Despite possible correlations between some of these metrics and application performance, these metrics are not designed for that task. Performance of many compute-intensive applications on modern computers is directly proportional to their parallelisibility. To our knowledge, there are no software metrics, which can be used for judging about source code parallelisability and that research area seems to be unexplored. Integration of such parallelisibility metrics into major Interactive Development Environments (IDEs) could alleviate parallel programming task by providing programmers with real-time feedback about their code. Moreover, new software parallelisibility metrics have a potential of paving the way into the new areas of parallel programming research.

Chapter 2

Background

This chapter of the thesis introduces a reader into the context of the work. First, sections 2.1 and 2.2 give a broad overview of various software source code metrics proposed in the general field of computer science and specifically in the subfield of parallel computing. None of these metrics can be directly used to judge about software source code parallelizability. For tackling that problem there is a need in development of new source code parallelizability metrics.

The field of parallel computing is a really old field, dating back to 1960s and there has been done a huge amount of work. Talking on the very high abstract level, the parallelization of a sequential program is ultimately constrained by different sorts of dependencies, which exist between program instructions. By that time, there has been established a theory on program dependence analysis. This theory has been well absorbed into many commercial compilers and is being ubiquitously used everywhere in practice. The book [1] conducts a broad compilation of all major results in the field of parallelizing compilers and program dependence theory. Section 2.8 briefly describes modern state-of-the-art parallelization tools with Intel's Parallel Studio [2] as an example. Section 2.3 takes the main theory results from [1] and describes those, which are used in the current project.

This abundance of previous work makes it clear, that there is only one way along which software source code parallelizability metrics study can be conducted. Proposed metrics must use the results of well-established dependence theory. In particular, the whole MSc project work is based on the intermediate dependence-based program representation, namely Program Dependence Graph (PDG). This program dependence data structure has been proposed by in their work [3] and is briefly described in section 2.6.

Once PDG of a program source code is built, some metrics can be computed straight away. Regular program consists out of modules, subroutines, blocks of code (such as if-statements, for-loops, etc.). As, according to many observations, loops represent 20% of code, which is executed 80% of time and are the most seducing piece of workload to parallelize by compiler, this MSc work computes parallelizability metrics on program loops.

The work on loop iterator recognition and loop decoupling [4] is another foundation this MSc project is based on. Section 2.7 describes the main results of this paper and introduces notions of loop iterator and payload. Metrics proposed and described in chapter 3 almost completely rely onto these concepts. The tool, developed for this MSc project (see chapter 4) implements algorithms described in [4] and also uses some graph theory results like strongly connected components (SCCs) finding algorithm and some results from compilers control flow analysis theory, which are briefly outlined in sections 2.4 and 2.5.

2.1 Software metrics in computer science

The idea of software source code metrics is definitely not a new one. Quantitative measurements lie as the essence of all exact sciences and there have been numerous efforts to introduce objective metrics in computer science as well. As of the moment, computer science quantitative metrics have found their application mostly in the fields of software quality assessment, software products complexity and software development as a process. These metrics measure properties of software products such as source code complexity, modularity, testability and ultimately maintainability. Combined with properties related to software development processes and projects, they are capable of delivering some estimates on the total amount of development efforts and associated monetary costs at the end.

The body of research in this field is very vast. There are a lot of publications on different types of metrics as well as on their evaluation criteria, axioms metrics must conform to, their validation, applicability, etc. There has been some efforts to conduct a survey of the field and present an overview of the most important and widespread software metrics to date ([5], [6], [7] to name a few). Work [6] distinguishes two major eras in the field: before 1991, where the main focus was on metrics based on the complexity of the code; and after 1992, where the main focus was on metrics based on the concepts of Object Oriented (OO) systems (design and implementation). Earlier

Fabrizio Riguzzi's work [5] dated as 1996 resembles [6], but also adds some critical insight. Jitender Kumar Chhabra and Varun Gupta in their paper [7] conduct an overview of dynamic software metrics. The later shows that software metrics have gone further from the field of static analysis and moved on to dynamic properties of the software.

2.1.1 Source lines of code (SLOC) / lines of code (LOC)

Source lines of code (SLOC) or lines of code (LOC) is one of the most widely used, well-known and probably one of the oldest software source code metrics to date. As its name implies, SLOC is measured by counting the number of source codelines in order to give approximate estimation to software size and the total amount of efforts (man-hours) required for development, maintenance, etc. Usually comparisons involve only the order of magnitude of lines of code in the projects. An apparent disadvantage of SLOC metric is that its magnitude on the piece of software does not necessarily correlate with the functionality provided by that piece. SLOC values differ from one language to another and heavily depend on the source code formatting and stylistic factors. Despite all of its disadvantages, SLOC is widely used in software projects size estimations and generally gives good correlations between its magnitude and programming efforts.

2.1.2 McCabe's cyclomatic complexity (CC)

Another well-known software metric is cyclomatic complexity (CC). The metric was first developed by Thomas J. McCabe in 1976 [8]. The metric is based on the control flow graph (CFG) of the section of the code and basically represents the number of linearly independent paths through that section. Mathematically cyclomatic complexity M of a section of the code is defined as M = E N + 2P, where E is the number of edges, N is the number of nodes, P is the number of connected components in the section's CFG. For example, the piece of code, which CFG is presented on the Figure 1, has cyclomatic complexity equal to 3. The same value 3 follows form it's mathematical equation M = 8 7 + 2 = 3. CC metric has been validated both empirically and theoretically and has a lot of applications.

2.1.3 Halstead's complexity measures

Maurice Halstead introduced his software science in 1977 [9]. In his work Halstead built an analogy between measurable properties of matter (such as volume, mass and pressure of a gas) and those of a source code. He introduced such notions as program length, program volume and program difficulty based on the number of distinct operands and operators in the program.

2.1.4 Software cohesion and coupling

Concepts of software coupling and cohesion were introduced into computer science by Larry Constantine in the late 1960s, when he was working on the field of structured design. The work [10], published in 1974 outlines the main results of Larry Constantine's research. Coupling is the degree of interdependence between software modules, while cohesion refers to the degree to which the elements inside the module belong together. These concepts are usually contrasted to each other and often establish inverse proportionality: high coupling often correlates with low cohesion and vice versa. Low coupling and high cohesion are usually a sign of a well-designed system. That system consists of the relatively independent modules. Changes in one part do not usually affect another parts. Degree of reusability is high and particular system parts (obsolete, malfunctioning, etc.) can be replaced without affecting the rest of the system.

2.1.5 Function points

Function point is a unit of measurement that is used in order to represent the amount of business functionality present in the piece of software. During functional requirements phase of software development, required functionality is identified. Every function is categorized into one of the following types: output, input, inquiry, internal files and external interfaces. Every function is given some amount of function points, which is based on the experience of the past projects. Function Points were proposed by Allan Albrecht in 1979 [11]. Albrecht observed in his research that Function Points were highly correlated to SLOC (3.1) metric.

2.1.6 Object-Oriented software metrics

In the work [12] Chidamber and Kemerer define a suite of metrics for object oriented designs. They define software metrics for several software properties like cohesion, coupling and complexity. Some examples are presented below:

Lack of Cohesion in Methods (LCOM):
LCOM = (P >Q) ? P − Q : 0, where P and Q are the numbers of pairs of class methods that do not use / use common class member variables correspondingly.

Coupling Between Object Classes (CBO):
For a class CBO equals to the number of other classes to which it is coupled. If methods of a class invoke methods or work with member variables of the other class, then classes are coupled.

2.1.7 Security metrics for source code structures

Software metrics have found their application in the field of source code security as well. Work [13] gives some examples.

2.1.8 Wrap-up

Described metrics can be used at different stages of software development. Function points 2.1.5 can be used at initial stages of functional requirements specification. Software cohesion and coupling concepts 2.1.4 can be considered during later stages of high-level design specification (particular object-oriented software metrics 2.1.6). Cyclomatic complexity 2.1.2, SLOC 2.1.1, Halstead's complexity measures 2.1.3 can be used during final and implementation stages for guiding coding efforts. All these metrics give assessments and predictions related to software quality, maintenance, testability, etc. Despite the possibility of correlations between some of these metrics and application parallelizability, these metrics are not designed to directly judge about it.

2.2 Metrics in the area of parallel computing

As in the whole computer science field, there have been proposals of numerous performance metrics in the area of parallel computing as well. Work [14] gives a critical overview of some of the existent parallel performance metrics. These metrics

assess software/architecture combinations and use a program running time as their basis. Subsection 2.2.1 takes some fragments of the work [14] to give a reader an impression of available parallel performance metrics. References to original authors of these metrics are available in the work [14].

2.2.1 Speedup variants

The basic question, which arises with program parallelization is ”How much faster are we running application on a parallel computer?”. The metrics described in this subsection are motivated by that question. While there is a general agreement that a speedup is the ratio:

$$\frac{serial\ execution\ time}{parallel\ execution\ time} \tag{2.1}$$

there is a diversity in definitions of parallel and serial execution times.

1. When we use the notion of *relative speedup*, parallel execution time is the time needed to execute a parallel version of a program on a single processor. The final resulting speedup depends on many factors: the number of processors in the system, the interconnection topology used for processor communication, the input dataset for the program, etc. Hence, the final speedup numbers might differ and we may futher introduce a number of such metrics as *maximum relative speedup*, *minimum relative speedup*, etc.

2. When we talk about *real speedup*, the role of the *serial execution time* is performed by the time, needed for the fastest known serial algorithm to solve the problem.

3. In yet another speedup definition the serial execution time is measured on the fastest serial computer, executing the best known algorithm. The term *absolute speedup* is used for this measure.

4. Let $t_{serial}(n)$ and $t_{parallel}(n)$ be asymptotic complexities of a serial and parallel algorithms used to solve a problem respectively. Then the ratio $\frac{t_{serial}(n)}{t_{parallel}(n)}$ is called an *asymptotic speedup*. Asymptotic speedup assumes unlimited number of available processors and is not a function of the number of processors in a system. As with regular speedup, this speedup can futher be classified into relative, real, etc.

5. If we introduce different parameters (such as hard drive read/write rate, memory latency L, number of processors P, processor cache sizes S, etc.) and write down a final equation for a speedup, then we get so-called *analytical speedup*.

All these enumerated speedup variants can be futher combined or modified to produce a numerous parallel performance metrics. While these metrics can be used to estimate the final speedup a parallel version of a program is going to have on the certain hardware system, they cannot be used for software source code parallelisability feedback and algorithmic parallelizability analysis. In other words, these metrics are not applicable to the problem being tackled in this MSc project.

2.3 Dependence theory

The ultimate task of compiler is to translate a program from a high-level human-oriented language to the code, designed to run on some certain machine. Apart from functional correctness, the code produced by compiler must be effective. In the modern computer science world effectiveness mostly determines, whether a language is going to be used. Hence, compilers spend a bulk of their work and time on optimization.

Despite the fact, that parallel programming research has been around for many decades, the primary focus of compiler optimizations used to be on scalar machines. But, relatively recent switch to multi-core architectures for general purpose CPUs, which happened around 2006 increased the need for compiler parallelizing optimizations.

The primary focus of modern day compiler research is program parallelization. Modern optimizing and parallelizing compilers use dependence-based approaches to the analyses and transformations they do. Data dependence has been explored since the early days of compilers, dating back to the 1960s, and by now there exist a vast body or research and theory in the domain. The main results and outlines can be found in the optimizing compilers for modern architectures book [1]. This section cites the main notions and principles in the field of program dependence analysis.

2.3.1 Dependence intuition

The primary requirement for optimizing and parallelizing compilers is that compiler analyses and sunsequent program transformations must not change the semantic (functional correctness) of the original sequential program. In other words, sequential

program specifies a set of constraints called dependencies.

Dependence is a binary relation on the set of program statements. The pair $\langle S_1, S_2 \rangle$ is in relation if S_2 must be executed after S_1. Generally speaking, a dependence is anything that introduces execution order constraints on statements or instructions of the sequential program.

Let's consider the following straight-line pseudocode fragment, which computes the area of a circle and is written in imperative programming style:

S_1 $\pi = 3.14159$

S_2 $R = 5$

S_3 $S = \pi \cdot R^2$

Here programmer specified the order, these statements should be executed in. And the correct functional result of this code fragment is obtained by executing these statements in the originally specified order S_1, S_2, S_3. While both statements S_1 and S_2 must precede statement S_3 for getting correct circle area, nothing prohibits execution of S_2 before S_1. In other words, sequential imperative programming languages introduce extra execution order constraints, which do not really need to be precisely followed.

Hence, the job of parallelizing compiler is to figure out the minimal set of critical constraints, which preserve the functional correctness of the program. There isn't really that much to gain in this little straight-line code fragment from its parallelization. Moreover, modern hardware does it implicitly to a programmer with pipelining [16] and out-of-order execution techniques [17]. The major source of parallelism is concentrated withing loops and other structures of repeated computation.

2.3.2 Types of dependencies

The topmost classification in the hierarchy of dependence types is data and control dependencies. Code fragment from the section 2.3.1 illustrates data dependencies between statement S_3 and statements S_1 and S_2. Statement S_3 is awaiting for the data from statements S_1 and S_2, and cannot proceed to its execution until it gets it.

Let's consider the other type of dependencies - control dependencies. In the code fragment presented in the listing 2.1 statement S_2 is control-dependent on the statement S_1 and cannot be evalueated until we check the condition in the if statement. If we reschedule S_2 computation to happen before S_1, then we are risking of getting a zero-division exception, which does not occur in the original sequential code.

Classification of data dependencies can be extended further. No matter how com-

```
S1 if (a!=0)
S2     c = b/a;
```

Listing 2.1: Code illustrating the notion of control dependence. Statement S_2 is control-dependent on the statement S_1.

plex programming language statement or structure is (loop, conditional statement, etc.), no matter what the exact source code form of the loop is (while-do, do-until, for-range loops), at the very end everything boils down to the order of memory acceses (reads and writes - loads and stores) to the same memory locations. That consideration leads to so-called *load-store classification*.

Read After Write (RAW) dependencies

This type of dependence (also frequently called *true dependence* or *flow dependence*) arises, when a first statement S_1 stores into location, that is later read by a second statement S_2:

$S_1 \quad X = ...$

$S_2 \quad ... = X$

This dependence ensures that S_2 receives the data, produced by S_1, and is conventionally depicted as an edge flowing from a first statement (*source*) to a second (*sink*).

Write After Read (WAR) dependencies]

This type of dependence (also frequently refered to as *anti dependence* or *false dependence*) arises, when a first statement S_1 reads from a location, which is being overwritten by a second statement S_2 later in a program:

$S_1 \quad ... = X$

$S_2 \quad X = ...$

This dependence is called false, since it is usually easily eliminated by renaming (whether by software, or by hardware means). The point of dependence is if we execute S_2 before S_1, then S_1 might load a wrong value at the end.

Write After Write (WAW) dependencies

This type of dependence (also frequently refered to as *output dependence*) arises, when a first S_1 and second S_2 statements write to exactly the same location:

$S_1 \quad X = ...$

$S_2 \quad X = ...$

If we change the order of statements, then after their execution location X is going to stay with the value computed by S_1, when it has to stay with the values of S_2.

Read After Read (RAR) dependencies

These are not usually classified as dependencies at all, and can be freely reodered and rescheduled.

Basically, these dependence types represent different scenarios, how a dependence can actually arise in a program.

2.3.3 Dependence in loop nests

2.4 Graph theory

The work uses some results from the graph theory. In particular, the depth-first search (DFS) graph traversal algorithm and its application to find strongly connected components (SCCs) of graphs. While there are a certain number of variations of these two basic algorithms, the work uses them in the exact form as described in the introduction to algorithms book [18].

2.5 Control flow analysis

Control flow analysis [19]

2.6 Program Dependence Graph (PDG)

A lot of work has been performed over the years in the area of dependence-based program representations and a lot of different

The Program Dependence Graph (PDG) is an intermediate dependence-based program representation that makes explicit both the data and control dependencies for each operation in a program. A control flow graph [l, 31 has been the usual representation for the control flow relationships of a program; the control conditions on which an operation depends can be derived from such a graph. An undesirable property of a control flow graph, however, is a fixed sequencing of operations that need not hold.

The program dependence graph explicitly represents both the essential data relationships, as present in the data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph. These dependence relationships determine the necessary sequencing between operations, exposing potential parallelism.

2.6.1 Data dependence graph (DDG)

2.6.2 Memory dependence graph (MDG)

2.6.3 Control dependence graph (CDG)

2.6.4 Program dependence graph (PDG)

2.7 Loop iterator recognition and loop decoupling

Logically the code, constituting a loop can be divided into two parts. The first part is an actual workload (payload) to be repeated multiple times. The other part is a loop iterator.

Different definitions have been proposed for loop iterators. For example, in the polyhedral model underpinning many loop transformations each loop iteration within loop nest is modelled as a lattice point inside a polytope describing the loop iteration space. A loop iterator is then an enumerator for point vector with integer valued components with bounds representing the faces of the loop limiting polytope.

In comparison, in object-oriented programming iterators are class objects, which enable a programmer to traverse a container data structure, e.g. a list. These objects provide a syntax, which abstracts away all container traversal details. Specifically, the next element of a container can be reached by *iterator*++ operator. Internal implementation of this operator can be arbitrarily complex.

The work, published in paper [4] introduces a new and more general loop iterator definition. Loop iterator is defined on the higher algorithmic graph theory level and works better on practice. As work [4] showed, this iterator definition enables compiler analyses to recognize even iterators of non-affine loops, which escape traditional compiler analyses.

2.7.1 Generalized loop iterator definition

As described in the paper [4], intuitively, a loop iterator is a variable (or a set of variables), which is updated in every iteration of a loop and is involved in controlling loop exits, e.g. as part of a conditional expression.

Definition 3.1. Generalized Loop Iterator. A generalized loop iterator is a minimal set of variables and operations manipulating these variables, which form a Strongly Connected Component (SCC) in the Program Dependence Graph (PDG) and exhibit a loop-carried dependence of distance 1. Furthermore, this SCC has no incoming edges from other SCCs in the PDG.

Conditional expressions controlling loop exits introduce control dependences to every operation contained in the loop body.

Since we are also interested in variables, which are updated in every loop iteration, we are looking for data dependences in the other direction, i.e. from update operations in the loop body towards read operations in loop termination expressions. Together these dependences will form a loop-carried dependence cycle or, more generally, a SCC in the PDG. Other operations may depend on this SCC, but it is the dominant SCC of operations, which does not depend on any other operations and variables that determines loop termination and thus constitutes the loop iterator.

2.7.2 Loop decoupling and iterator recognition analysis

Static Analysis Our analysis for determining loop iterators involves three stages closely following Definition 3.1: 1. PDG construction. We assume the IR provides us with a Control Flow Graph (CFG) in Static Single Assignment (SSA) form (Figure 5a). We apply the algorithm from [11] to construct the Control Dependence Graph (CDG) of the loop. Additionally, we combine the implicit def-use chain present in the intermediate representation with a static dependence analysis of memory accesses based on [19] to build the Data Dependence Graph (DDG) of the loop. We produce the PDG by combining the CDG and DDG (Figure 5b). 2. Determine SCCs. Once we have the program depen- dence graph of a loop, we determine its strongly connected components. We build a directed acyclic graph (DAG) con- necting the SCCs using Kosarajus algorithm [2]. 3. Dominant SCC and iterator recognition. Finally, we take the dominant SCC, i.e. the one that has no incoming edges in the SCC DAG. This dominant

SCC represents the loop iterator and we label instructions represented by the SCC as iterator instructions and variables involved as it- erator variables (Figure 5c). Inspection of the properties of these iterator instructions and variables reveals that together they satisfy Definition 3.1 by construction, showing that we have indeed recognized the loop iterator.

2.8 Modern parallelisability advisor tools

Probably, the state-of-the-art in the area of parallel programming support and assistance in different program performance optimizations and tunings is represented by Intel(R) Parallel Studio XE 2018 [2].

Intel Parallel Studio XE is a software development product developed by Intel. Parallel Studio is composed of several component parts, each of which is a collection of capabilities. These tools help developers boost application performance through superior optimizations and Single Instruction Multiple Data (SIMD) vectorization, integration with Intel Performance Libraries, and by leveraging the latest OpenMP* 5.0 parallel programming models. Enhanced optimization reports and integration with Intel VTune Amplifier and Intel Advisor give developers control over code profiles. For better performance, it is optimized to take advantage of advanced processor features like multiple cores and wider vector registers, including Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions.

To really get a program into a tuned shape, suitable for execution on a specific hardware platform, software developer has to be quite knowledgable and experienced not only in the field of parallel programming and computer architecture, but also with these particular complex tools as well.

Intel C/C++ compiler [15] out of Intel Parallel Studio XE tool suite has been used in this project to compile NAS benchmarks and provide some feedback about parallelizability of benchmark loops. ICC compiler was used instead of a parallelizability expert (withing the scope and timeframe of the MSc project), to classify (label) NAS benchmark loops as parallelizible or not for the machine learning part of metrics analysis 6.4.

```
    for (k = 0; k < n; k++) {
        for (j = 0; j < vlen; j++) {
S1        x[k][j] = scr[k][j];
        }
    }
```

Listing 2.2: *FT/src/fft3d.c:103* Algorithmically parallelizible loop. Intel compiler detects true and anti dependencies between S_1 statement executions on different loop iterations, despite the fact that these references address different arrays.

2.8.1 Automatic parallelisation with Intel(R) C/C++ compilers (ICC)

Parallelizing application for the sake of performance improvement can be a time-consuming and skill-requiring activity. For applications, containing relatively simple loops and targeting x86 platforms this task can be automated with the help of Intel C++ compiler [15]. With automatic parallelization ICC detects loops that can be safely and efficiently parallelized and generates multithreaded code. It relieves the programmer from searching for loops that are good candidates for parallel execution, performing dependence analysis and adding parallel compiler directives manually.

When it comes to automatic program parallelisation, Intel C/C++ compilers are apparently limited to certain types of loops.

Along with actual parallelization Intel C/C++ compilers provide developers with a parallelisation reports. Unfortunately, these reports have a binary nature and give ”yes”/”no” answers, when it comes to parallelizability questions. Moreover, since compiler has to be conservative, when it comes to optimizations in-order to preserve functional semantic of original program, sometimes these reports might be inaccurate or misleading. Listing 2.2 gives an example of a loop, which has not been parallelized by Intel’s compiler due to over-conservative dependence detection. Although, it is obvious from the loop, that loads and stores happen to different memory locations.

Section 6.3 of chapter 6 gives more examples of such loops and shows how parallelizability metrics can assist in such cases.

Chapter 3

Software Parallelisability Metrics

This chapter defines proposed software source code parallelizability metrics and gives the basic intuition behind them. It was decided, that all parallelizability metrics would be computed on the Program Dependence Graph (PDG) intermediate representation of programs (see 2.6) and would rely on dependence analysis theory (see 2.3) as their foundation.

The chapter is structured in the following way. Section 3.1 puts the metrics work into the context and gives the general perspective from which one has to look at parallelisability metrics. Section 3.2 gives a motivating example and presents the output, devised metrics should ideally provide. Section 3.3 introduces the actual metrics, along with the basic motivation for them. Metrics are introduced as a set of conceptual groups. Each group has roughly the same intuition and motivation for all its metrics.

3.1 General foundation and perspective of the work

3.1.1 Diversity in modern computer languages

There are thousands of different languages in the modern field of computer science. Computer programming languages have passed a long way from assembly languages operating at the level of native machine instructions to languages operating with concepts at a much higher abstraction levels. The reason behind such a change in the domain of computer languages is the ease, with which a human programmer can write a software.

Unfortunately, this move to a higher-level languages comes with drawbacks as well. With the gain in programmer's productivity, such change also brings losses in

software performance. It becomes increasingly difficult for the compiler to translate abstract languages into the sequence of machine instructions effectively.

If we are to use these easy for human comprehension high-level languages, we must have tools for their efficient transformation into the form, suitable for direct execution on different native machine platforms.

3.1.2 The modern role of compilers

As was outlined in the previous section 3.1.1, novadays compilers perform enabling role for the use of different sorts of modern computer languages.

In the modern state of the field, the principal role of compiler is to map high-level algorithms onto different sorts of high-performance architectures. The notion of high-performance architectures is really general and usually represents the combining term for all of the following: parallel cluster, multi-core and multi-processor architectures, vector processors, pipelined superscalar processors and all the possible combinations and co-designs of these (see [16] and [17]).

Before this mapping can be done, compilers must perform extensive analyses to determine what parts of program computations depend on one another and what parts can be scheduled for parallel execution on high-performance machines. These analyses are mostly dependence-based by their nature.

3.1.3 The famous 80/20 rule

As in many areas, computer science has its own 80/20 rule: 20% of the code executes 80% of the time. The hotspots of software programs are loops and other structures of repeted computation. Hence, loops and arrays present the most fertile ground for compiler optimizations. Metrics, being considered in this work operate on loops. For every single loop PPar tool (see chapter 4) computes a vector of metrics (features), characterizing it. Parallelizability metrics can be devised for the whole program functions or fragments of code, but for the start, loops seem to be the most interesting and promising program parts.

3.1.4 Dependence-based approach to metrics computation

Compiler parallelisation of program statements is basically hindered by the execution-order constraints imposed on those statements, which, in turn, are defined by different

sorts of program dependencies, which were described in the section 2.3 of the thesis. That consideration makes it clear, that in order to be credible software parallelizability metrics must be based on loop dependence properties and constraints. Program Dependence Graph (PDG) (see 2.6) represents an intermediate representation of the program, which reflects all types of dependencies present in the program.

3.2 Metrics use

The main goal of software parallelizability metrics is to provide a developer with additional feedback regarding loop parallelizability, which could extend binary ”yes”/”no” answers of modern parallelizing compilers (see section 2.8) by providing some quantitative measures of loop algorithmic parallelizability. Not only compiler answers have a binary nature, as it is shown in the sections 2.8 and 6.3, sometimes Intel compiler has to be over-conservative and assumes presence of dependencies, where they do not actually exist. Such feedback might be misleading for inexperienced parallel software developer.

Ideally, metrics should provide a quantitative measure of loop’s algorithmic parallelizability (say, this loop is 80% parallelizible), and where compiler has to be over-conservative due to functional semantic requirements, feedback metrics do not have any constraints.

Let’s consider several little hand-written examples.

```
for (unsigned int i = 0; i < size; i++) {
    c[i] = a[i] + b[i];
}
```

Listing 3.1: Example of parallelizible loop. Intel compiler successfully parallelizes it.

```
for (unsigned int i = 1; i < 100; i++) {
    a[i] = a[i-1];
}
```

Listing 3.2: Loop with true and anti cross iteration dependencies. Intel compiler detects dependency and refuses to parallelize the loop.

Listings 3.1 and 3.2 demonstrate two loops. While loop in listing 3.1 is parallelizible, loop in listing 3.2 is not. PPar tool (see 4) is going to compute the following

metric numbers for these loops.

Loop in listing 3.1: *loop-absolute-size*: 21, *loop-payload-fraction*: 0.5714, *loop-proper-sccs-number*: 0, *loop-critical-payload-fraction*: 0.0000, *iterator-payload-total-cohesion*: 0.3191, *iterator-payload-non-cf-cohesion*: 0.0638.

Loop in listing 3.2: *loop-absolute-size*: 17, *loop-payload-fraction*: 0.5294, *loop-proper-sccs-number*: 1, *loop-critical-payload-fraction*: 0.2222, *iterator-payload-total-cohesion*: 0.2821, *iterator-payload-non-cf-cohesion*: 0.0513.

As will be shown in the section 6.2, metrics *loop-payload-fraction*, *loop-critical-payload-fraction*, *iterator-payload-total-cohesion*, *iterator-payload-non-cf-cohesion* correlate with loop parallelizability property the most. Metrics *loop-proper-sccs-number* and *loop-critical-payload-fraction* represent the critical part of the loop payload. The bigger that part and correspondingly these metrics, the harder that loop is to parallelize. Figure 3.1 shows visualization of PDG for loop in listing 3.2. It is visible, that this loop has two SCCs. One for iterator (with no incoming dependencies) and the other one is for critical payload. Critical SCC contains back edge, which forms a cycle and thus SCC. This cycle represents actual dependencies between load *a[i-1]* and store *a[i]* in the payload, which hinder loop parallelization.

While loop iterator-payload cohesion metrics do not have that apparent intuition,

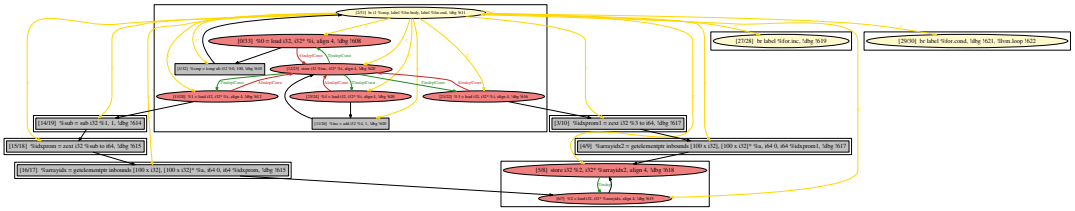

Figure 3.1: PDG of loop in listing 3.1. Graph contains two SCCs: one for iterator and one for critical payload part.

as section 6.2 shows, they also tend to correlate with parallelizability. The bigger their values, the better the chances of loop parallelization. There are different special cases, but these two loops prove such relations between metric values and loop parallelizability.

Let's consider another example. Loop in listing 3.3 represents a loop, which performs reduction. This algorithm is parallelizible by its nature, but a programmer hid all that parallelism behind unsuccessful data structure choice (linked list). While loop in listing 3.2 can be parallelized with additional memory buffer (probably cyclic shift instructions won't help because the memory area to shift might be a way larger than a

processor register), loop in listing 3.3 is chasing pointer and nothing can help to that code version. So, ideally, metric values for loop 3.3 should be worse that those for loop 3.2. While Intel compiler won't parallelize both, metric values should show a certain difference to a programmer (say 3.3 is 0% parallelizible, 3.2 is 60% parallelizible, while 3.1 is 100% parallelizible).

Loop in listing 3.3 can be rewritten like loop in listing 3.3. That transformation immedeately lets Intel compiler to parallelize that reduction. While, Intel compiler does not do that transformation automatically, software engineer (given metric values and feedback hints) can.

```
while (list_it != nullptr) {
    sum += list_it->value;
    list_it = list_it->next;
}
```

Listing 3.3: Algorithmic reduction, hidden behind unsuccessful data structure choice (linked-list). If linear array was used instead, the loop would be parallelized by Intel compiler. Intel compiler reports: "not a parallel candidate" for that version

```
while (i < size) {
    sum += c[i++];
}
```

Listing 3.4: Loop with optimal data structure choice for algorithmic reduction. Intel compiler successfully parallelizes it.

Metric values for these two 3.3 and 3.4 loops follow.

Loop in listing 3.3: *loop-absolute-size*: 14, *loop-payload-fraction*: 0.4286, *loop-proper-sccs-number*: 1, *loop-critical-payload-fraction*: 0.5, *iterator-payload-total-cohesion*: 0.2188, *iterator-payload-non-cf-cohesion*: 0.0312.

Loop in listing 3.4: *loop-absolute-size*: 13, *loop-payload-fraction*: 0.5385, *loop-proper-sccs-number*: 1, *loop-critical-payload-fraction*: 0.4286, *iterator-payload-total-cohesion*: 0.2759, *iterator-payload-non-cf-cohesion*: 0.0345.

Metric *loop-critical-payload-fraction* exibit such idealistically expected pattern for this particular case. Metrics *iterator-payload-total-cohesion*, *iterator-payload-non-cf-cohesion* do that as well, but in order to reach ideally expected feedback (if it is principally possible), some tuning and possibly even additional metrics and modifications

of these, are still required.

3.3 Metric Groups

The whole set of proposed metrics is divided into several concepual groups. To provide an illustrative description of different metrics, let's consider a loop 3.5 given below. This loop is taken form EP NAS benchmark (see 5).

```
for (i = 0; i < NQ; i++) {
    gc = gc + q[i];
}
```

Listing 3.5: Example loop, taken from EP NAS benchmark

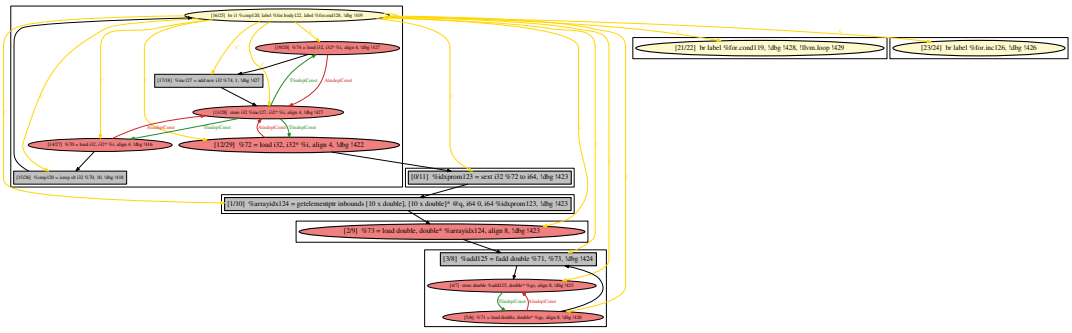

Figure 3.2: Program dependence graph (PDG) of the loop 3.5, as built and visualized by the PPar tool 4.

Figure 3.2 above shows program dependence graph of the loop, given in the example.

3.3.1 Loop Proportion Metrics

The first group of metrics computes proportions of the loop. Like Halsted's software science metrics (see 2.1), it draws an analogy with physical properties of objects (like size, volume, length, etc). This computation happens after PPar tool decouples a loop into iterator and payload code, as described in 2.7.

3.3.1.1 Loop Absolute Size

This metric represents the total amount of LLVM IR instructions in the loop. The intuition behind this metric is pretty staighforward: the bigger the loop, the harder it is to parallelize it. The metric has obvious drawbacks. The size of the loop does not, generally speaking, always correlates with loop parallelizability. However, it might be interesting to see, how loop absolute size correlates with loop parallelisability statistically. From figure 3.2 it is visible that the value of the metiric for the given loop 3.5 is 15.

3.3.1.2 Loop Payload Fraction

This metric is complementary to loop absolute size metric and reflects the proportion in which iterator and payload divide the whole loop. There are different considerations behind this metric. For example, if the payload is too small relative to the size of the loop and does not perform significant amount of computations, then parallelization of this loop might not worth the effort.

3.3.1.3 Loop Proper SCCs number

Once we decoupled a loop into iterator and payload parts, we can split payload part even further. Both iterator and payload are represented by subgraphs in the PDG of the loop. As was described in 2.7, iterator instructions form a strongly connected component (SCC), which has no incoming dependencies. Payload consists of a set of SCCs. These payload components have different sizes, starting from just 1 instruction and, principally, do not have any upper size limit. If SCC of PDG belongs to the payload of a loop and consists of more than 1 instruction, we call such SCC a ***proper SCC***. Usually, such components represent a true dependency in the body of a loop, preventing a loop from parallelization. Thus, the task of parallelizing compilers is to break the edges of such ***proper (critical)*** SCCs, and transform these SCCs into smaller ones (possibly just 1 instruction).

The example loop from the listing 3.5 contains one such proper (critical) SCC. There is a cross-iteration dependency in the body of this loop. The partial sum is being accumulated in the *gc* variable. We can see 2 edges (corresponding to true and anti dependencies) between variable *gc* load and store instructions in the PDG shown on figure 3.2. Despite the presense of cross-iteration dependency in the loop, Intel C/C++ compiler is capable of its parallelization with reduction techniques.

```
for (i = 0; i < 2 * NK; i++) {
    x[i] = -1.0e99;
}
```

Listing 3.6: Parallelizible loop, with no cross-iteration dependencies. Taken from EP NAS benchmark.

Simplier loops, like the one shown on the listing 3.6, do not contain any SCCs of size greater than 1 (besides iterator SCC). Once we split such loops into iterator and payload parts, all SCCs in the payload are of 1 instruction size. Figure 3.3 provides illustration.

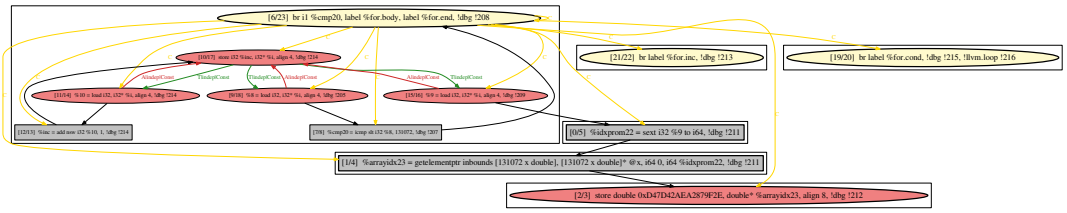

Figure 3.3: Program dependence graph (PDG) of the loop 3.6, as built and visualized by the PPar tool 4.

3.3.1.4 Loop Critical Payload Fraction

In the light of considerations given in the previous section, the fraction between critical and non-critical payload parts might have some correlation with loop parallelizability.

3.3.2 The problem of proper SCCs number metric

The proper SCCs number metric was described in section 3.3.1.3. Despite the fact, that the metric seems to directly represent parallelisability inhibiting parts of the payload, figures in ... show that ICC compiler sometimes fails to parallelise loops even with 0 metric value. Listing **??** below, gives such example.

Loop in the listing **??** has a pretty simple PDG. There are no critical SCCs inside loop's payload. However, ICC compiler cannot parallelize this loop due to *randlc* function call. Compiler has to be conservative and assume, that this function call

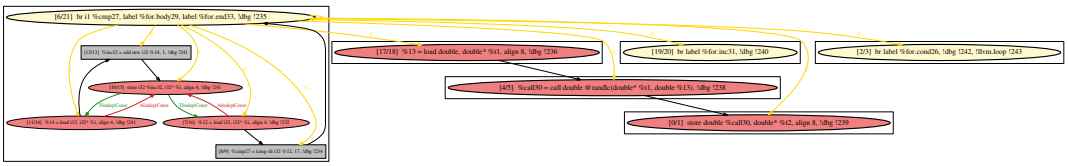

Figure 3.4: Program dependence graph (PDG) of the loop **??**, as built and visualized by the PPar tool 4.

introduces cross-iteration dependencies between iterations of the loop.

This example exposes drawback of proper SCCs number metric. This metric only considers structural properties of PDG and does not examine the nature of instructions, constituting the loop.

3.3.3 Loop Dependence Metrics

3.3.4 Loop Cohesion Metrics

As cohesion and coupling metrics have been proposed for computer software 2.1.4, these properties can also be extended in context of this work.

These properties characterize the degree of inter-dependence between different parts of loops. As was shown in the previous

The main motivation behind the metrics out of this group is the tighter the parts of a loop are coupled together (in terms of dependencies), the harder it is going to be to split and parallalize the loop.

Chapter 4

Software parallelisability metrics tool

This chapter describes the tool developed for software source code parallelisability metrics research, how to use it, its software architecture and all the underlying technologies and libraries used during its development.

The tool is developed with the C++ language and is almost completely based on the LLVM library of modular and reusable compiler technologies [20] [21]. The tool is implemented as a set of LLVM passes (see LLVM online documentation for further technical details [22]). The tool can be found at [23]. All parts of the tool rely heavily on the standard C++ template mechanism and C++ Standard Template Library (STL).

The tool operates on the level of LLVM intermediate representation [24] (LLVM IR) and completely decoupled from input languages as well as from target machine instruction sets. Theoretically, the tool can be used for source code parallelisability analysis of any arbitrary programming languages as it does not depend on any exact programming language concepts, data structures and constructs (such as conditional loops, for loops, range-for loops, goto statements, lists, maps, etc). The tool operates on the level of program dependencies **??** (data, control, etc), which are abstracted away from programming languages domain into a separate dependence analysis theory 2.3. In order to use the tool, one must provide a way of compiling input language into LLVM intermediate representation.

Conceptually the tool does the following. It accepts C/C++ programs as and input.

In this project all proposed concepts are being examined with the use of Clang/-Clang++ as a front end to transform input C/C++ source code into LLVM instruction set.

The remainder of the chapter is structured as follows. Section **??** briefly describes

parts of the LLVM library used in the project. Descriptions are mostly taken from the source code of LLVM and can be studied in more details at [25].

4.1 Tool implementation

.There are several LLVM provided analyses being used by the tool.

4.1.1 General software architecture

The tool is implemented withing LLVM pass framework (see [26]) and architected as a set of LLVM passes, dependent on each other and interacting through the standard mechanism LLVM pass manager provides. There are basically three types of passes in the tool, which are implemented as C++ template classes:

GraphPass<NODE,EDGE,PASS> Function analysis pass, which builds dependence graph of a function as well as depencence graphs of all function's loops. This pass stores all the built graphs in the process memory and makes them later accessible for subsequent passes. **NODE** and **EDGE** template parameters represent data, associated with each graph's node and edge respectively. **PASS** parameter is used to distinguish different passes, which use the same node and edge types.

GraphPrinterPass<NODE,EDGE,PASS> This pass depends on the **GraphPass** described above, and dumps its memory content into the files on the hard drive. Dumped files are formatted in accordance with the DOT graph description language and can be visualized with the corresponding tool (such as [27]).

DecoupleLoopsPass Function pass, implemented as a non-template C++ class. Pass runs on a function and computes information for every single function loop. Pass depends on the PDG C++ template specialization of the **GraphPass** and uses program dependence graphs (PDGs) of function loops to decouple latter into iterator and payload parts. Results are represented as sets of strongly connected components (SCCs). Those SCCs, which belong to the loop payload and those, belonging to the iterator of a loop (there should be only one such SCC). All this information is stored in the process memory and futher accessible for metric computing passes. Detailed algorithms and concepts, underlying the pass implementation, are described in the section 2.7 of the thesis.

MetricPass<METRIC> A C++ template to be specialized and instantiated for every single metric group to be computed. Metrics are computed as function passes, which depend on all passes described above. Different types of metrics, being computed by the tool are described in section 3.3 of the thesis.

MetricCollector This is a function pass located at the very output end of the whole metric computing pass pipeline. The primary task of that pass is to collect all metrics, computed by **MetricPass** set of passes, for the given function and report them in the file.

These passes rely on some standard LLVM analyses and facilities as well as on the functionality developed withing the current project. Standard LLVM passes, used by the tool are described in section **??** below. Representation of dependence graphs in the memory is described in the section **??** of this chapter. Section **??** describes graph visualisation facilities, provided by the tool. Exact specializations of pass templates, described above, correspond to program dependence graph theory given in section 2.6. LLVM details of these specializations are described in section **??**.

4.1.2 Standard LLVM analyses

The tool uses a number of standard LLVM analyses.

LoopInfo This analysis function pass identifies all natural loops withing the given function and assigns a loop depth to every function's basic block. This analysis calculates the nesting structure of loops in the function. For each natural loop identified, this analysis identifies natural loops, contained entirely within the loop and basic blocks that make up the loop.

DependenceAnalysis

PostDominatorTree

4.1.3 Graph representation

Since LLVM, as of version 6.0, does not currently provide a standard dependence graph (DG) implementation, custom graph building facilities were implemented in the project as a **Graph<NODE,EDGE>** C++ template. Template expects two parameters, which must be pointers to the **NODE** and **EDGE** classes. These classes represent

information assosiated with every graph's node and edge correspondingly. The tool uses several types of dependence graphs in its work and these parameters usually end up to be one of the following. NODE parameter is useually either llvm::Instruction or llvm::BasicBlock

4.1.4 Graph visualization facilities

While the main output of the tool is a set of software parallelisability metrics, the tool also accepts a number of side command line options that are useful for debugging to produce additional information, which can supplement bare metric values with some additional insights. Since the tool is based on a set of dependence graphs of programs, it is particularly useful to visualize these graphs.

GraphPrinterPass<NODE,EDGE,PASS> C++ template is designed for doing exactly that.

4.1.5 Template specializations

4.2 The tool workflow

The workflow of the tool can be conceptually divided into 4 phases, following each other in a pipelined fashion:

1. **LLVM part: C/C++ translation into LLVM IR, dependence analysis and loop identification.** The tool is operating on the level of LLVM intermediate representation (IR) [24]. Clang/Clang++ front-ends translate input C/C++ source code into this IR form. Then, LLVM performs a series of its standard analyses, required by the tool (see section **??**). LoopInfo identifies all the loops in program functions and provides convenient interface for further queries. LLVM builds def-use chains between LLVM IR-level instructions during IR construction. LLVM's dependence analysis identifies data dependencies between memory references in a function. Post-dominance analysis builds a post-dominator tree.

2. **PPar tool program dependence graph (PDG) building part.** In some sense, this part represents the front-end of PPar tool. The tool uses LLVM use-def chains, linking IR instructions, to build data dependence graph (DDG) of a program being examined. After that it uses LLVM dependence analysis and post-dominance tree to build memory dependence graph (MDG) and control dependence graphs (CDG)

respectively. The order of these passes does not matter. In principle, they could be done in parallel. Once all three graphs are built, the tool combines all dependencies present in them into a unified program dependence graph (PDG). Detailed descriptions of these graphs can be found in section 2.6. All that functionality is done by the corresponding specializations of **GraphPass**<**NODE,EDGE,PASS**> template (see 4.1.1) for every type of dependence graph.

3. **Iterator recognition and loop decoupling.** The tool uses results and algorithms, described in the paper [4] to decouple loops into iterator and payload parts (see section 2.7). This is done by DecoupleLoopsPass (see 4.1.1).

4. **PPar tool back-end.** This is the end of the pipeline. Here PPar tool produces its final results. Depending on the purpose, the tool runs here either a set of passes computing parallelisability metrics, or different graph printers (see 4.1.4) for visual graph analyses and tool debugging.

4.3 Tool use

Chapter 5

Benchmarks

NAS Parallel Benchmarks have been used withing this project. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications in the original ”pencil-and-paper” specification (NPB 1). The benchmark suite has been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids. Problem sizes in NPB are predefined and indicated as different classes. Reference implementations of NPB are available in commonly-used programming models like MPI and OpenMP (NPB 2 and NPB 3).

5.1 Benchmark descriptions

In the graph pass

5.1.1 IS - Integer Sort

5.1.2 EP - Embarrassingly Parallel

```
for (i = 0; i < MK + 1; i++) {
    t2 = randlc(&t1, t1);
}
```

Chapter 6

Analysis

After the set of parallelisability metrics has been devised and proposed (see chapter 3) and a working framework for metrics research and analysis has been implemented and set (chapter 4 describes developed PPar tool), software source code parallelisability metric values can be gathered and analysed. This analysis task in not that trivial. There is, principally, an unlimited number of ways in which this data can be visualized, interpreted and processed. This chapter describes taken analysis approaches and presents findings in the report.

Table 6.1 below presents the data, used for analysis. This data has been extracted from NAS parallel benchmarks (see chapter 5) and transformed into tabular format.

loop location	ICC parallel	loop absolute size	loop payload fraction	loop proper sccs number	loop critical payload fraction	iterator payload total cohesion	iterator payload non-CF cohesion	critical payload total cohesion	critical payload non-CF cohesion	payload total dependencies number	payload true dependencies number	payload anti dependencies number	critical payload total dependencies number	critical payload true dependencies number
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/add.c(45)	1	84	0.119	1	0.7	0.04444	0	0	0	20	10	10	20	10
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/add.c(46)	0	72	0.1389	1	0.7	0.05714	0	0	0	20	10	10	20	10
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/add.c(47)	0	60	0.1667	1	0.7	0.08	0	0	0	20	10	10	20	10
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/add.c(48)	1	48	0.8125	1	0.07692	0.4078	0.02913	0.07895	0.07895	38	37	1	4	3
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(218)	0	91	0.8242	1	0.1867	0.2651	0.01606	0.1049	0.08392	143	94	36	74	38
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(212)	0	14	0.4286	0	0	0.2581	0.06452	0	0	3	3	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(180)	0	20	0.25	0	0	0.1818	0.02273	0	0	0	0	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(175)	0	10	0.3	0	0	0.1905	0.04762	0	0	0	0	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(161)	0	10	0.3	0	0	0.1905	0.04762	0	0	0	0	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(133)	0	5	0.2	0	0	0.125	0	0	0	0	0	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/bt.c(131)	0	5	0.2	0	0	0.125	0	0	0	0	0	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(66)	1	46	0.4348	2	0.55	0.2113	0.02817	0.0625	0.0625	32	20	12	26	14
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(67)	0	25	0.72	1	0.3333	0.3455	0.01818	0.1579	0.1579	19	17	2	9	7
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(50)	0	97	0.03093	1	0.6667	0.03361	0.01681	0.3333	0.3333	3	2	1	2	1
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(52)	0	80	0.0375	1	0.6667	0.04124	0.02062	0.3333	0.3333	3	2	1	2	1
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(54)	0	63	0.3016	2	0.5263	0.1667	0.02381	0.0625	0.0625	32	19	13	26	13
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(58)	0	41	0.7561	1	0.3226	0.3571	0.04082	0.1081	0.1081	37	32	5	18	13
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/BT/src/error.c(46)	1	13	0.4615	0	0	0.2593	0.03704	0	0	3	3	0	0	0
.....														
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(190)	0	59	0.1695	1	0.7	0.08696	0	0	0	20	10	10	20	10
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(191)	0	49	0.8163	1	0.075	0.4095	0.02857	0.02564	0.02564	39	38	1	4	3
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(222)	0	128	0.5391	3	0.9565	0.1176	0.01103	0	0	216	113	89	214	112
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(225)	0	51	0.1373	1	0.8571	0.03846	0.01282	0.05882	0.05882	17	9	8	16	8
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(237)	1	27	0.6667	1	0.1667	0.339	0.0339	0.1765	0.1765	17	16	1	4	3
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(227)	1	26	0.6923	1	0.1667	0.3509	0.03509	0.1765	0.1765	17	16	1	4	3
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(278)	1	47	0.2128	1	0.5	0.12	0	0.07143	0.07143	14	8	6	12	6
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(279)	0	34	0.2941	1	0.5	0.1579	0	0.07143	0.07143	14	8	6	12	6
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/UA/src/utils.c(280)	0	21	0.619	0	0	0.3256	0.02326	0	0	10	10	0	0	0
/homes/s1736883/Work/PParMetrics/benchmarks/nauseous/common/randdp.c(122)	0	67	0.8806	1	0.7964	0.3571	0.05952	0.08989	0.08989	89	68	21	79	58

Figure 6.1: Analysis input table with computed metrics and ICC parallelizability classification labels.

This data is, essentially, a set of loops found in NAS benchmarks. For every single loop a vector of metrics (loop features) has been computed. All loops have passed through Intel C/C++ compiler parallelisability analyses and have been classified (labelled) as parallelizible or not. ICC compiler plays a role of an expert in this project.

The table 6.1 contains around 1400 loops with 13-dimensional feature vector for each.

This dataset has been analysed in several ways. First, the collected data has been visualized to see if there are any obvious correlations between loop parallelisability and metric values. Section 6.2 presents these visualizations and describes all found corerlations. This visualization has been done for every single metric (see subsection 6.2.1) as well as for the whole set of metrics altogether (see subsection 6.2.2). To visualize the whole combined set of metrics Principal Component Analysis (PCA) and clustering techniques have been used. Section 6.3 supplements these visualizations with some manually derived insights into benchmarks source code.

As another mean of parallelizability metrics examination, statistical analysis techniques have been applied to the data. Loop metrics have been viewed as machine learning features in the context of loop parallelizability classification problem. Standard state-of-the-art machine learning techniques (such as Support Vector Machines (SVM), desicion trees and neural network based Multi-layer Perceptron (MLP) scikit learn algorithm) have been applied to the data and all prediction errors have been compared against random predictor. Section 6.4 presents a report on this.

There has already been an attempt to apply statistical analysis techniques to see how software quality metrics, such as cyclomatic complexity 2.1.2 and Halstead's software science measures 2.1.3 behave on Mozilla Firefox browser and LLVM compiler components library open source codes [28]. Authors applied k-means clustering and got 3 clusters of software quality metric values for subroutines. But there were no classifications attached to the input data and no correlations have been examined. There could easily be subroutines of different software quality grades in the same cluster.

All results have been derived thanks to Python programming language and its packages: pandas [29], matplotlib [30] and scikit-learn [31].

6.1 Analyses preparation phase

Before we move onto the actual description of gathered results, there is a need to describe some preparatory data preprocessing procedures, which have been used throughout all analysis stages.

As it turned out, there are some outliers in the collected data that distort the final result. For example, figure 6.2 shows the plot of payload total dependencies number metric values on all NAS benchmark loops. The plot on the left shows metric values dispersion before outliers elimination. It can be seen that there is a non-parallelizible

(red dot) loop with almost 12000 dependencies in the payload which seriously shifts mean metric values for parallelizible and non-parallelizible subsets to the point of correlation inversion. Generally, it makes sense that the more dependencies we have in the payload, the harder it has to be to parallelize the loop. And it is seen from the whole dataset that loops with really high dependencies numbers have not been parallelized by the ICC compiler. But for majority of loops, encountered in NAS benchmarks, this tendency does not take place. Right plot contains only those loops, which have metric values withing 3 standard deviations from the mean. Here ICC parallelization ability does not really correlate with the amount of dependencies in the payload of the loop. The total number of dependencies in the payload of a loop is not the only metric, where we have some outliers. Similar filtering measures have been taken for all metrics.

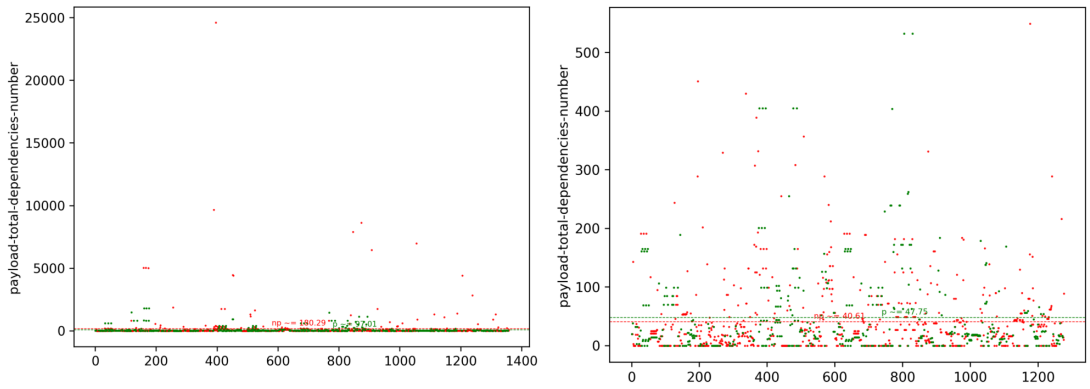

Figure 6.2: Distribution of total dependencies number metric values on all NAS benchmark loops before and after elimination of all cases beyond 3 standard deviations from the mean.

There was also a need to normalize the data before application of Principal Component Analysis (PCA) algorithm, in order to make a better graphical visualization.

6.2 Data interpretation and visualization

This section presents different attempts to find any patterns and correlations in the data from the table 6.1. First, every single metric (out of all 13 presented in the table metrics) has been examined alone. Section 6.2.1 contains all results of single metric parallelizability analysis. Then, the set of metrics has been considered altogether as a whole (as points in 13-dimensional space) in order to find any structural patterns they form. Principal Component Analysis and k-means clustering have been used for that

task. Results are presented in subsection 6.2.2.

6.2.1 Single loop metrics vs loop parallelisability analysis

The layout of this section corresponds to that of section 3.3, which describes proposed groups of software parallelizability metrics. Subsections below present studies of their parallelizability correlation.

6.2.1.1 Loop proportion metrics

Figures 6.3 and 6.4 present scattered plots of all loop proportion metric values, gathered on NAS benchmark loops. Numbers on vertical axes represent metric values, values on horizontal axes represent particular loops in the set. Dashed horizontal lines, crossing plots, represent mean values for subsets of parallelizible (green) and non-parallelizible (red) loops. Numbers show how many dots lie above and below mean values for red and green subsets. All outliers lying beyond 3 standard deviations have been filtered (see section 6.1).

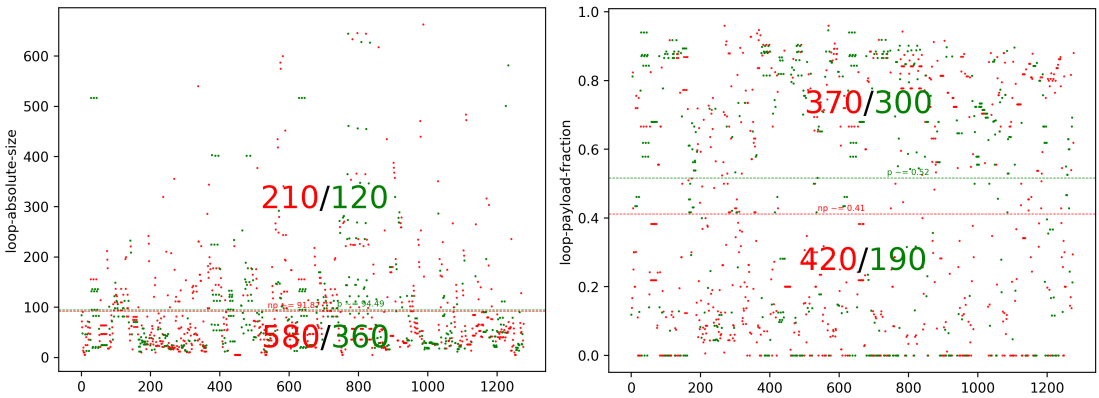

Figure 6.3: *Loop absolute size* metric on the left and *loop payload fraction* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly. Dashed lines show mean metric values for red and green subsets correspondingly.

As it can be seen from the plots, *loop absolute size* metric values do not really correlate with ICC parallelization ability. Mean values of that metric are almost identical, 91,87 and 94,49 LLVM-IR instructions, for non-parallelizible and parallelizible loop subsets correspondingly. Red and green dots are scattered above and below mean value lines in similar proportions (210/580≈0,36 and 120/360≈0,33) - majority of

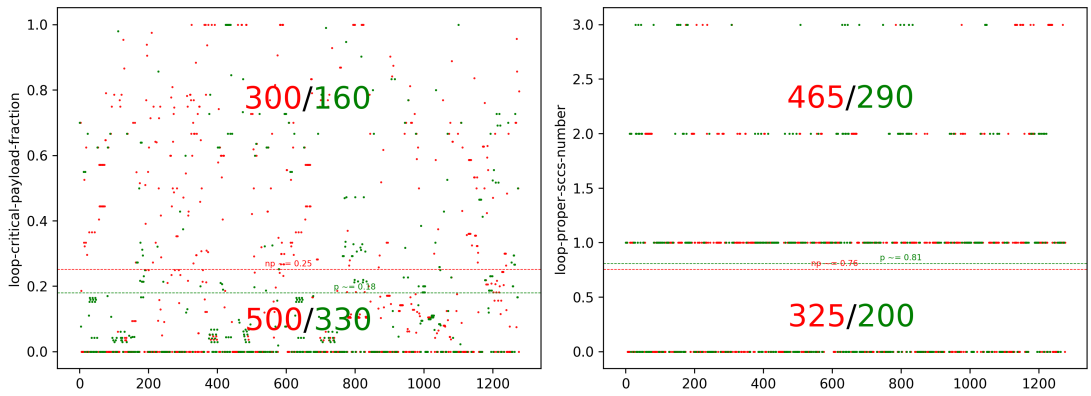

Figure 6.4: *Loop critical payload fraction* metric on the left and *loop proper SCCs number* metric on the right. Analogous to 6.3 figure.

NAS loops are smaller than 90 LLVM-IR instructions.

It must be clear from the nature of this metric, that size by itself does not prohibit loop parallelization. There might be a big loop with no cross-iteration dependencies, which must have a big portion of parallelism and should bring huge overall performance gains with its parallelization. And vice versa, there might be a small loop with cross-iteration dependency, which is non-parallelizible.

There is a pretty noticeable correlation between loop parallelisability and values of *loop payload fraction* metric. Green dots are scattered predominantly at the top of the plot. Dot numbers above and below mean lines are in inverse proportionality for red and greed dot subsets ($370/420 \approx 0{,}88 < 1$ and $300/190 \approx 1{,}58 > 1$). The bigger the payload of a loop in comparison with its iterator, the more seducing this loop for compiler to parallelize. Bigger payloads bring better performance gains with parallelization. Another consideration might also be applied here. Bigger iterators are common for loops, which, for example, perform traversal of more complex data structures like linked-lists or alike. Such loops contain memory operations linking iterator and payload - *iterator payload memory cohesion* metric should be non zero for such loops. Unfortunately, there are no such loops in NAS benchmark set and the latter metric has been excluded from consideration at all.

Despite the fact that *loop proper SCCs number* metric does not correlate with loop parallelizability that much, another metric of the same essense shows pretty good correlations. *Loop critical payload fraction* metric can be used to judge about loop parallelizability. Connection between this metric and parallelizability property follows just out of common sense. Critical payload parts represent strongly connected components with more than one instructions. There are PDG graph loops with forward

and back edges inside such SCCs. Usually it happens when two memory instructions reference the same location on different (or the same) loop iterations and introduce 2 inverse-directed edges with anti and true dependencies. Such critical loop payload parts introduce actual parallelization constraints. The left plot in the figure 6.4 illustrates this connection. The bigger the critical component in the payload, the harder it is to parallelize the loop.

6.2.1.2 Loop Dependence Metrics

Figures 6.5, A.2 and A.3 present plots of loop dependencies number metric values. These metrics do not seem to be particularly reliable, when it comes to judging about loop parallelizability. Due to this reason the two latter plots have been moved into appendix section.

As it can be seen from the figures, distributions of parallelizible and non-

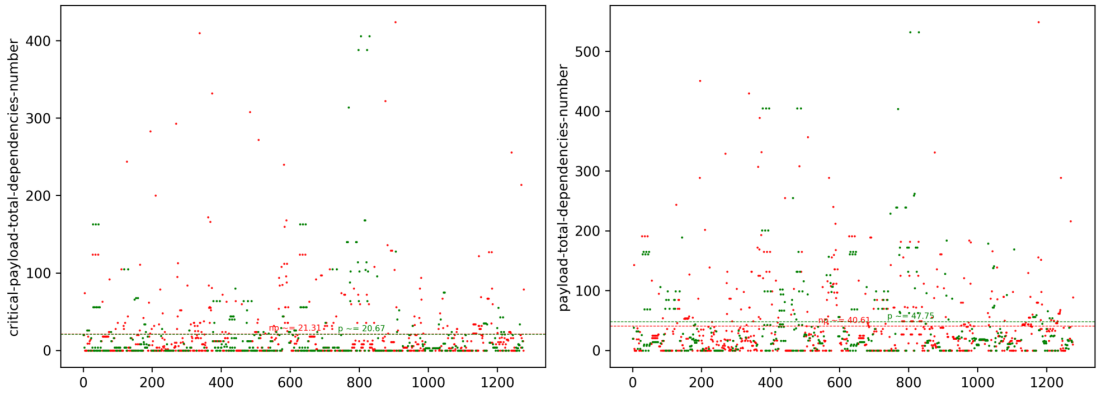

Figure 6.5: *Critical payload dependencies number* metric on the left and *total payload dependencies number* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly. The other 4 metrics show similar behavior and have been moved into appendix section (see plots A.2 and A.3).

parallelizible (green and red) dots are quite uniform throughout the whole spectrum of metric values. Mean metric values on green and red subsets are also quite close to each other. In cases, where they differ, that difference dissappears with addition of excluded outliers to consideration. Even for loops with a huge total dependencies number either in critical part or in the whole loop payload, ICC compiler sometimes is able to generate a parallel version. All these points show that loop dependencies number metrics cannot be used alone for loop parallelizability analysis.

6.2.1.3 Loop Cohesion Metrics

Loop cohesion metrics have been described in section 3.3.4. Figures 6.6 and 6.7 present plots of loop cohesion metric values on the set of NAS benchmark loops (roughly 1300 loops).

Critical payload cohesion metrics do not correlate that much with parallelizabil-

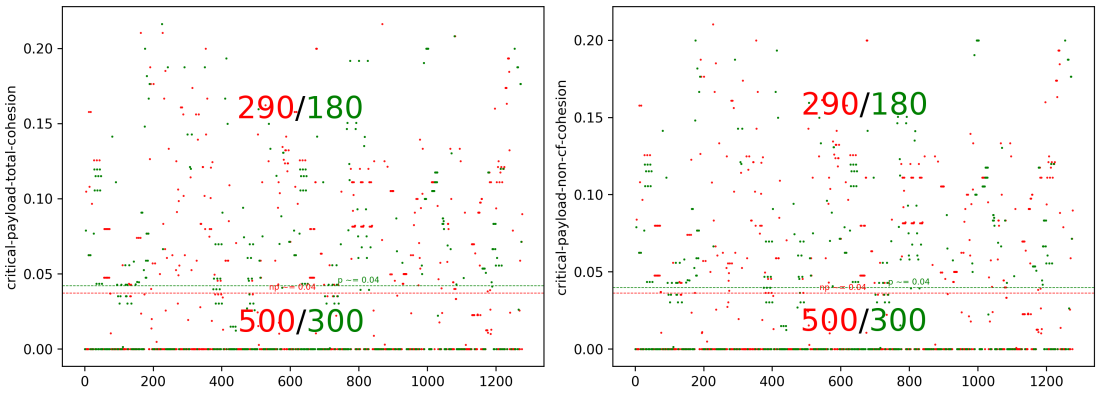

Figure 6.6: *Critical payload total cohesion* metric on the left and *critical payload non-cf cohesion* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly.

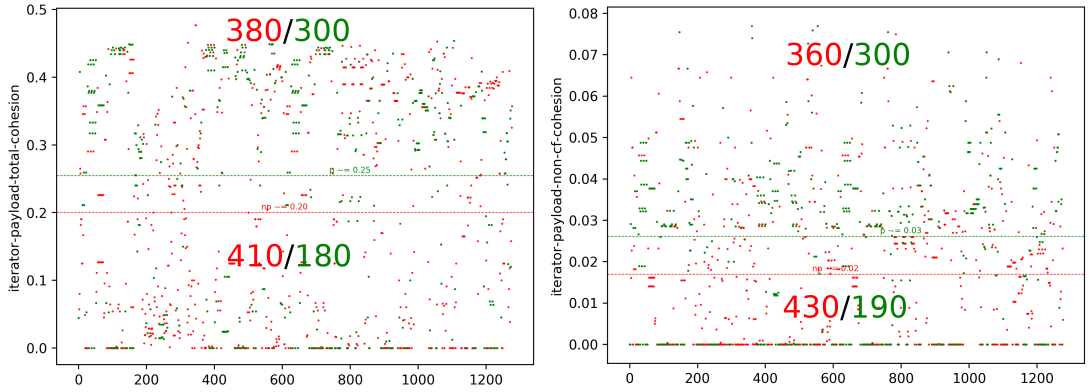

Figure 6.7: *Iterator payload total cohesion* metric on the left and *Iterator payload non-cf cohesion* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly.

ity property. Figure 6.6 shows approximately similar mean metric values on red and green subsets and it is also visible, that red and green dots are uniformly dispersed on the plane. These cohesion metrics represent the degree of interconnection between critical part of loop payload and its non-critical part.

On the other hand, *iterator payload cohesion* metrics show a certain degree of

correlation with loop parallelizability. It is visible from the two plots on figure 6.7, that green parallelizible dots are dispersed predominantly at the top of the plane. Mean metric values on red and green subsets tell exactly the same thing: mean metric values for parallelilazible loops tend to be higher than their counterparts on non-parallelizible subsets (0,25 versus 0,20 for *iterator payload total cohesion* and 0,03 versus 0,02 for *iterator payload non-cf cohesion*). Dot numbers above and below mean lines are also in inverse proportionality, as it is visible from figure 6.7.

6.2.2 Data clustering analysis

This section describes the results of dataset clustering analysis. Data from the table 6.1 can be viewed as a set of 13-dimensional vectors, describing NAS benchmark loops. These vectors map loops into 13-dimensional space. It might be interesting to see what spatial patterns and structural properties these mappings conform to. For every loop, thanks to Intel C/C++ compiler, we know the right answer regarding its parallelizability. This section reports studies about correlations between spatial and structural properties of these vector mappings and loops parallelizability.

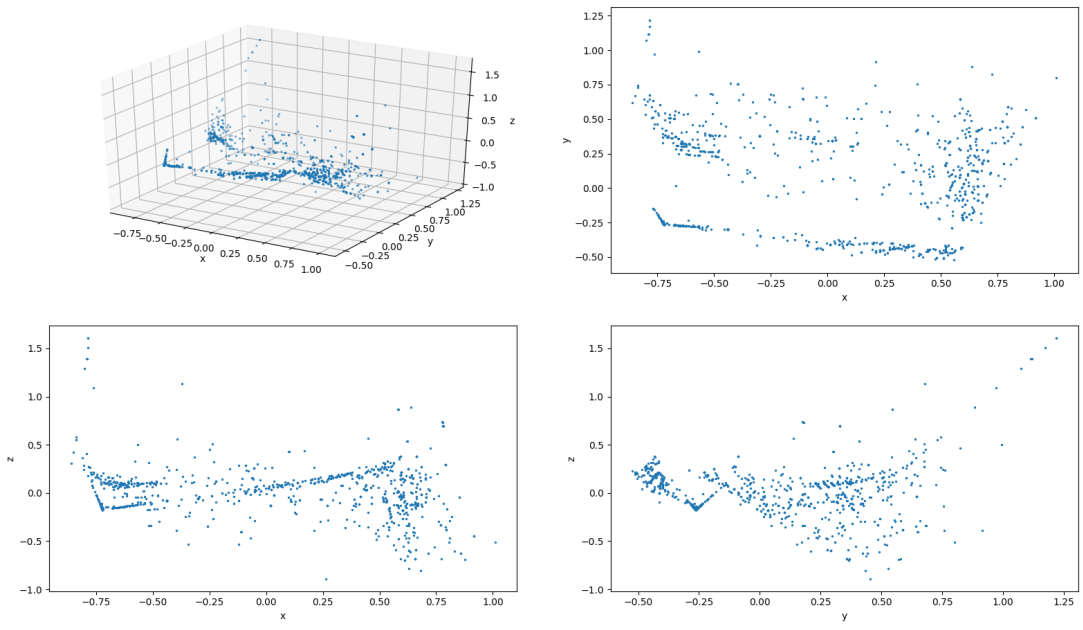

Figure 6.8: Visualisation of loop metrics dataset spatial distribution (13-dimensional metric vectors have been projected onto 3D space thanks to PCA algorithm) - blue dots correspond to metric values of single loops.

Since loops are represented by 13-dimensional vectors, direct dataset visualiza-

tion is unfeasible. Principal Components Analysis (PCA) has been used to project the data onto 3-dimensional space and visualize it. Figure 6.8 provides an illustration of distribution of metric values on all loops. It is visible from the figure, that data values do not form any apparent clusters, but there are some areas of increased density though. Metric dataset projection onto 2D plane looks like XY projection of 3D PCA mapping (see figure A.1).

Next step is to establish correlation between structural properties of dataset distribution and loop parallelizability. Figure 6.9 provides an illustration for it. Out of that figure it is visible that there is no apparent correlation between combined metric values and parallelizability of loops these metrics represent.

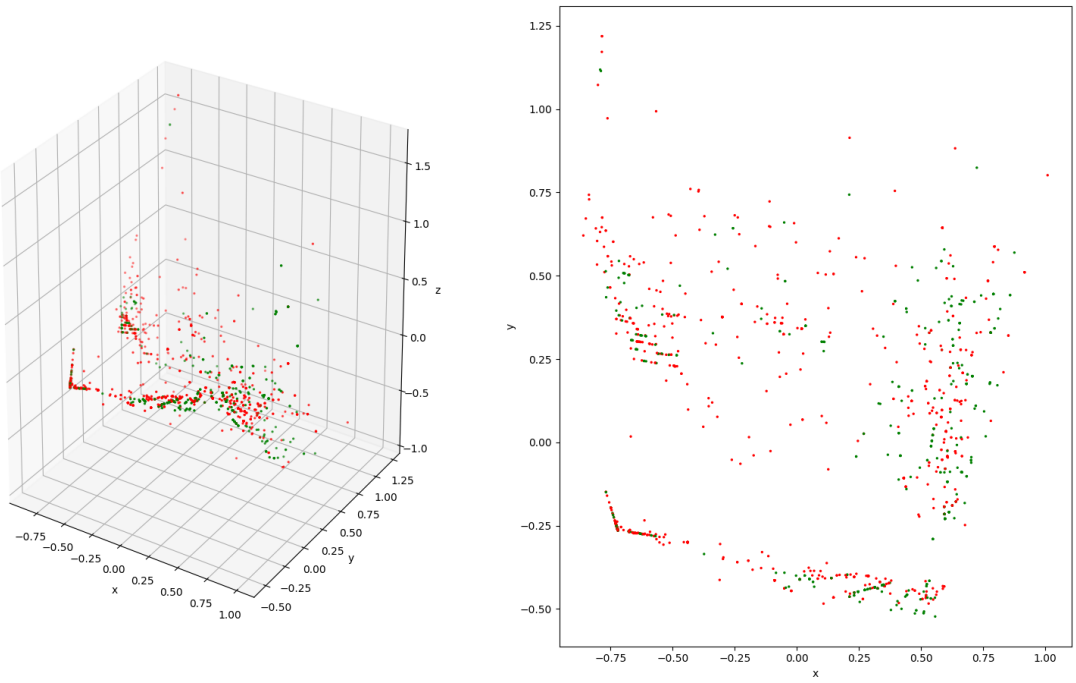

Figure 6.9: Visualisation of loop metrics dataset spatial distribution (3D PCA projection on the left, 2D PCA projection on the right) versus parallelizability property. Red dots - non-parallelizible loops; green dots - parallelizible loops

It would be interesting to see, what kind of loops and metric values form these areas of increased density. To get that information we can apply k-means clustering algorithm and get subsets of loops, which form these condensed clots. We need to pick the number of clusters. The number of clusters might be arbitrary, but from the figure it is visible that 3 or 4 clusters would be quite a close approximation to the given dots distribution. Figure 6.10 illustrates the results of k-means clustering algorithm run with 4 clusters. Algorithm has almost perfectly identified the structure of dots distribution.

Maybe yellow and blue subsets must be in the same subset. We can consider all these blue and yellow loops altogether. Some dots (the ones on the boundary of red cluster) also happened to be blue, while red classification would be the most appropriate for them. To exclude these must-be-red blue loops from considerations applied to the blue cluster, euclidean distances from cluster centers were computed for every dot. Only loops near cluster centers are considered in the manual source code study. Table 6.12 presents average values for all metrics in 4 clusters.

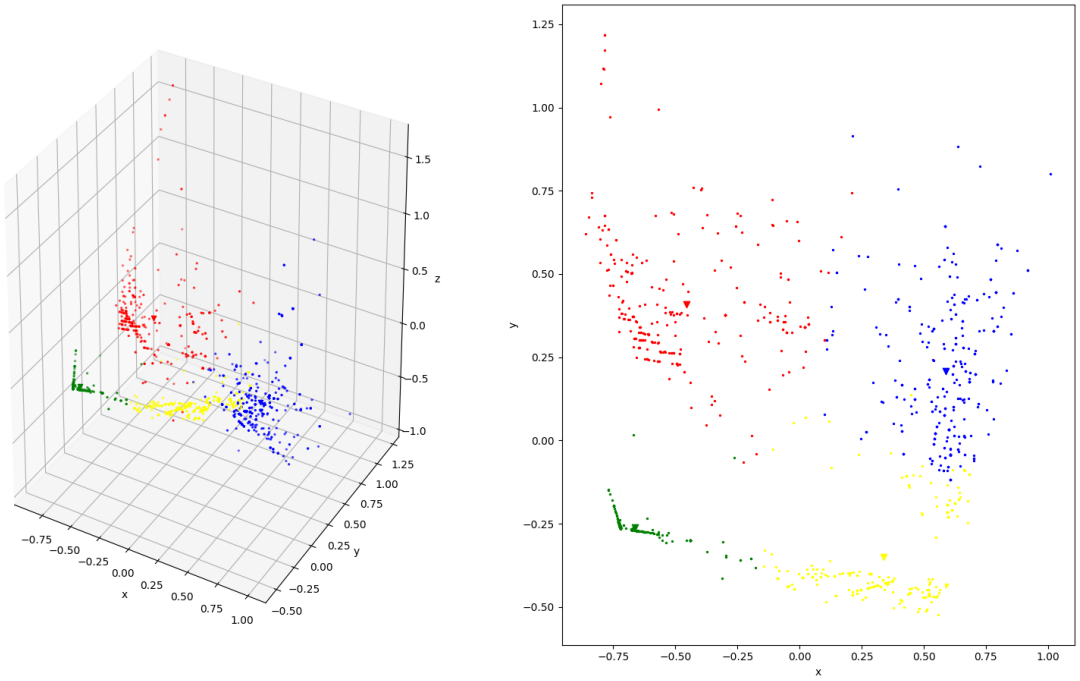

Figure 6.10: k-means algorithm classified the dataset into 4 separate clusters.

cluster	critical payload total cohesion	critical payload non cf cohesion	iterator payload total cohesion	iterator payload non cf cohesion	payload total dependencies number	payload true dependencies number	payload anti dependencies number	loop absolute size	loop payload fraction	loop proper sccs number	loop critical payload fraction	critical payload total dependencies number	critical payload true dependencies number	critical payload anti dependencies number
0	0.02	0.02	0.09	0.01	97.47	51.62	44.06	174.37	0.22	1.52	0.69	91.74	46.47	44.06
1	0.00	0.00	0.02	0.00	0.39	0.36	0.02	149.43	0.04	0.01	0.00	0.06	0.03	0.02
2	0.12	0.12	0.36	0.03	111.02	94.80	15.34	104.51	0.76	1.61	0.21	39.08	23.43	15.34
3	0.01	0.01	0.35	0.04	46.86	44.70	1.97	61.17	0.68	0.33	0.02	4.81	2.84	1.97

Figure 6.11: Average metric values for 4 different clusters

Out of that table 6.12 and figure 6.10 we can derive metric growth directions. All critical payload part related metrics show tendency to grow anti-clockwise, having minimum values in the green dot cluster and maximum values in the red one. We can combine all these metrics in one *critical component size* characteristic. Cohesion between loop iterator and payload parts grows from green cluster towards all 3 other

clusters, reaching its maximum values in blue and yellow clusters. *Loop absolute size* has a mirrored (in relation to *iterator payload cohesion* metric) pattern of growth. It starts to grow in the yellow cluster and takes on higher values as it reaches 3 other clusters. Figure 6.12 presents the final illustration for all considerations above.

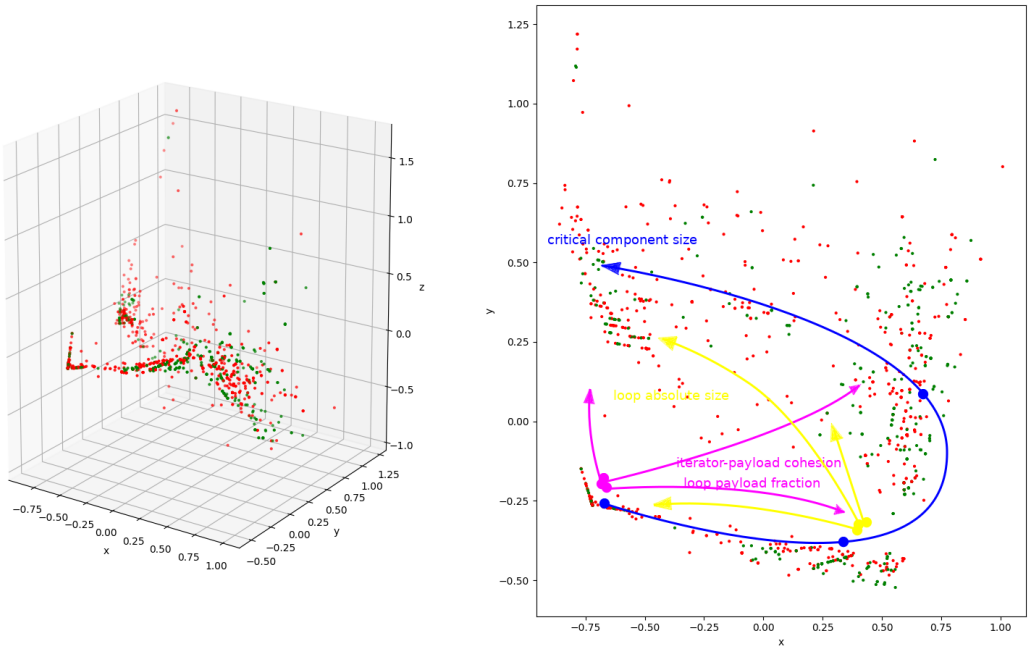

Figure 6.12: Correspondence between 3D PCA projected distribution and original metric values, set by growth arrowed lines.

The next section 6.3 provides examples of loops, which can be found in these 4 built clusters.

6.2.3 Decision tree based parallelisability classification

Decision tree is another human-comprehensible dataset visualization technique. Figure 6.13 has been produced with scikit-learn [31] decision tree classifier and represents a decision tree learned from metrics table dataset 6.1. Decision tree sorts metric vectors down the tree. Sorting starts from the root and goes along all the sorting rules down to the leaf. Leaf provides the final metric vector classification. For simplicity and feasibility of visualisation, the depth of the tree has been limited to 3. This limitation leads to some probabilistic errors in the final loop parallelizability classification, but is still capable of capturing the main trends in the data. 6.1.

The construction of decision tree is based on the notions of entropy and information gain. Decision tree algorithm finds and places in the root of the tree a rule, which separates the dataset in a way, that brings the biggest information gain (in other words, decreases entropy of the dataset the most). Then it goes along all the branches down the tree and places the most information gaining rule at every node, calculated on the dataset subset, corresponding to that node.

As we can see from the figure 6.13, scikit-learn implementation of decision tree algorithm picked *iterator-payload non-cf cohesion* as a metric, separating the input dataset in the best way. The input dataset contains 1277 metric vectors, 790 of them are classified as non-parallelizible and 487 as parallelizible. The entropy of such a dataset equals to $S = -\frac{790}{1277} \cdot \log_2 \frac{790}{1277} - \frac{487}{1277} \cdot \log_2 \frac{487}{1277} \approx 0.96$. After the rule in the root split the dataset into two subsets, corresponding to the left and right subtrees, the entropy became $S = \frac{790}{1277} \cdot S_{True} + \frac{487}{1277} \cdot S_{False} = \frac{793}{1277} \cdot (-\frac{591}{793} \cdot \log_2 \frac{591}{793} - \frac{202}{793} \cdot \log_2 \frac{202}{793}) + \frac{484}{1277} \cdot (-\frac{199}{484} \cdot \log_2 \frac{199}{484} - \frac{285}{484} \cdot \log_2 \frac{285}{484}) \approx$.

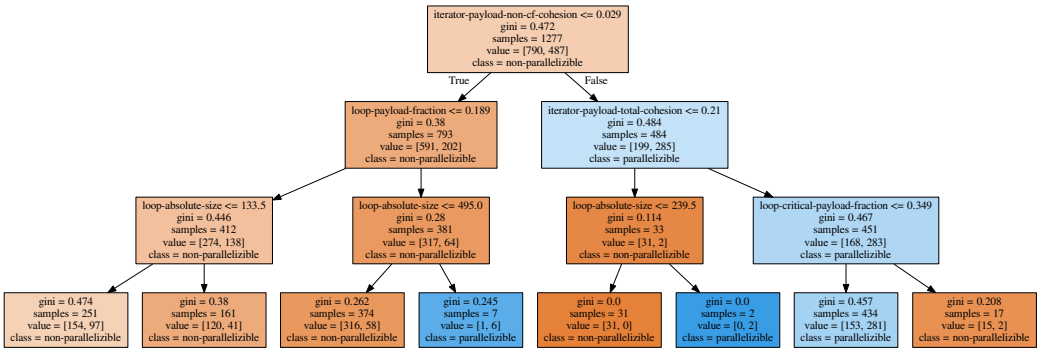

Figure 6.13: Dataset 6.1 decision tree, limited to the depth of 3.

For the purpose of our work we can interpret that decision tree in the following way. If we use *iterator-payload non-cf cohesion* metric to judge about loop parallelizability, then we should compare the value of that metric for a loop being considered against 0,03. If the value on the loop is grater than 0,03, then the loop is parallelizible with probability of $\frac{285}{484} \approx 60\%$ and non-parallelizible with probability of $\frac{199}{484} \approx 40\%$. That numbers do not give us a lot of information, but if value of the metric on a loop happens to be below 0,03, then the loop is most likely to be non-parallelizible with probability of $\frac{591}{793} \approx 75\%$. The latter signals loop code developer to reconsider his code, if he wants his code to take advantage of parallelization.

These are basically the same results as ones in the section 6.2.1 of the current

chapter, but decision tree methods have more solid mathematical grounds and foundation. Decision tree algorithm automatically finds a decision boundary in the range of single metric values, that splits the dataset in the most information gaining way. The tree on the figure 6.13 has been built from the set of all 13 metrics. According to the algorithm criteria *iterator-payload non-cf cohesion* metric is the best for loop parallelizability analysis task. If we exclude that metric from the set of metrics for the root rule search, or even use just one metric, we can find decision boundaries with corresponding probabilities for all more or less parallelizability correlating metrics. Figure 6.14 shows the result.

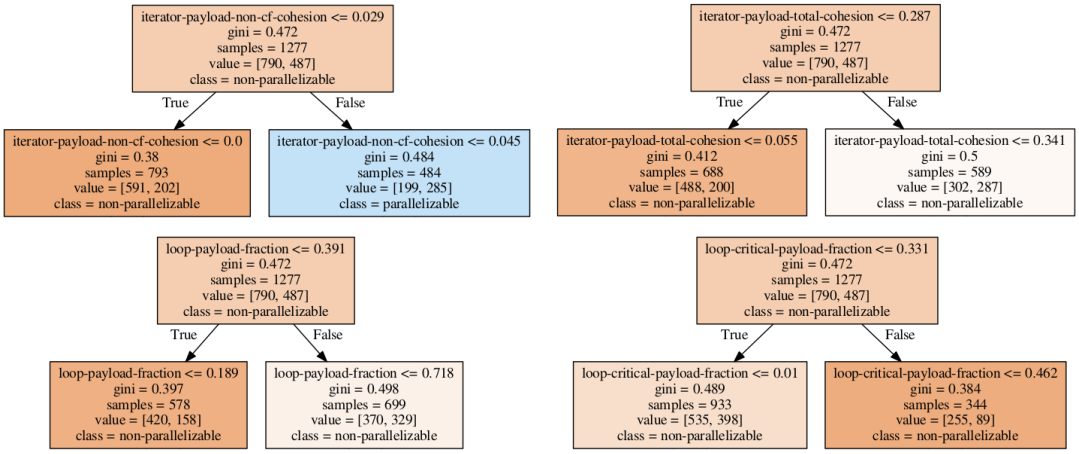

Figure 6.14: Decision boundaries and corresponding parallelizable/non-parallelizable probabilities for the best corerlating single metrics.

We can see from the above figure 6.14, that if we use *iterator-payload total cohesion* as a root node instead, we are going to get 0,287 decision boundary with 70% non-parallelizible and 30% parallelizible below 0,287 and 41% non-parallelizible and 59% parallelizible above the boundary. So, *iterator-payload total cohesion* is slightly worse than *iterator-payload non-cf cohesion*. It makes sence, since most iterators have control-flow structures, controlling conditions of the loop, but true dependencies are usually introduced by data edges from iterator to the payload. Metric *loop critical payload fraction* can be used in the inverse way. If the value of this metric for a given loop happens to be greater than 0,33 decision boundary, then this loop is 74% non-parallelizible. The latter result correlates with the common sense: the bigger critical payload part of the loop, the harder this loop is for dependence breaking and parallelization. The last metric in the figure 6.14 is *loop payload fraction*. We can see that the decision boundary for that metric is 0.391. So, if the payload of the loop happes to

be less than 40% of the total loop size, then this loop most likely is non-parallelizible with corresponding probability of approximately 73%. That also makes sense, since big iterators include too much computations in them, which makes such loops hard to parallelize. Computations should be concentrated in the payload of the loop and iterators should be relatively simple.

We don't necesserily need to use only single metrics. For a given loop we can compute a set of metrics and walk with their corresponding values down the decision tree in the figure 6.13, which has been taught by parallelizible Intel C/C++ compiler. Use of multiple features increases the accuracy of results. For example, lets look at the figure 6.13. Once we checked a loop being examing against the first root node rule and identified that the *iterator-payload non-cf cohesion* metric happened to be greater than 0,03 decision boundary, we can check the second rule, namely *iterator-payload non-cf cohesion* $\leq 0,21$. If we go along the blue (parallelizible) boxes, than we get $\frac{281}{434} \approx 65\%$ parallelizability chance against 59% after only *iterator-payload non-cf cohesion* boundary check. While parallelizability probabilities may not seem big enough for being worth the analysis, non-parallelizability probability in the left subtree can reach 85% for a relatively big (374 samples) subset. For getting that estimate, metric values on a loop must satisfy to a system of 3 inequalities:

$$\begin{cases} \textit{iterator-payload non-cf cohesion} \leq 0,029 \\ \textit{loop payload fraction} \leq 0.189 \\ \textit{loop absolute size} \leq 495 \end{cases}$$

If we limit the depth of decision tree to 5 instead of 3, decision tree algorithm is going to add less parallelizability correlating metrics to consideration as well. Non-parallelizible predictions are not that interesting as parallelizable. Decision tree of depth 5 can predict parallelizability with better probabilities. For example, the two systems of inequalities below are going to conclude that a given loop is $\frac{207}{322} \approx 64\%$ and $\frac{56}{61} \approx 92\%$ parallelizable correspondingly.

It would be of special interest to take a look at loops, which satisfy all these inequalitis, but nonetheless end up to be non-parallelizible. Such loops have not been parallelized by Intel ICC compiler, but have been classified as parallelizible by the set of metrics. If there are any among these loops, that are algorithmically parallel, but have not been parallelized by Intel compiler due to some lower-level technical issues, then these metrics find to be quite useful by telling software developed, that he should

not listen to Intel compiler and take a closer human look at them. Section conducts such an analysis.

$$\begin{cases} \textit{iterator-payload non-cf cohesion} \leq 0,029 \\ \textit{iterator-payload total cohesion} \leq 0,21 \\ \textit{loop critical payload fraction} \leq 0,349 \\ \textit{loop proper SCCs number} \leq 1,5 \\ \textit{critical payload true dependencies number} \leq 6,5 \end{cases}$$

$$\begin{cases} \textit{iterator-payload non-cf cohesion} \leq 0,029 \\ \textit{iterator-payload total cohesion} \leq 0,21 \\ \textit{loop critical payload fraction} \leq 0,349 \\ \textit{loop proper SCCs number} \leq 1,5 \\ \textit{loop payload fraction} \leq 0,575 \end{cases}$$

6.3 Manual analysis

Let's start from a trivial case and consider a loop, presented in listing 6.1. This loop fills timer records array *trecs[]* with timer reads. The latter are computed inside *timer_read()* function. Intel C/C++ compiler has to be conservative and does not parallelize that function, since it is not clear, whether there is a cross-iteration dependency introduced from inside the function, but the decision tree, presented in the figure 6.13 classifies that loop as parallelizible. Given that hint a programmer might check, if there are actually any states, preserved by *timer_read()* function from call to call, which might be distorted by the loop parallelization. This loop has *iterator payload non-cf cohesion* equal to 0.0645, which correspond to parallelizible loops, according to results presented in the section 6.2.1.

Another more elaborate case is presented in the listing 6.2. Intel compiler failed to parallelize that loop nest, because it assumed that there is flow and anti dependencies between load from rhs[k][j][i][m] and store to rms[m]. But decision tree of depth 3

```
for (i = 1; i <= t_last; i++) {
    trecs[i] = timer_read(i);
}
```

Listing 6.1: Loop, which has not been parallelized by Intel C/C++ compiler, but does seem algorithmically parallelizible, given absense of cross-iteration dependencies introduced by the function call.

```
for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            for (m = 0; m < 5; m++) {
                add = rhs[k][j][i][m];
                rms[m] = rms[m] + add*add;
            }
        }
    }
}
```

Listing 6.2: *SNU_NAS/BT/src/error.c:84*. Loop, which has not been parallelized by Intel C/C++ compiler due to conservative anti and flow dependencies assumptions between references to rhs and rms. But these are different arrays and the loop actually computes reduction.

(see figure 6.13) classifies that innermost loop as 65% parallelizible and 35% not, giving a programmer a little hope. Its classification could have been slightly better if there was not SCC inside the payload of the inner loop. Metric computing pass uses LLVM memory dependence analysis, which alike Intel compiler assumes dependencies between memory references to different arrays. Metric values for the innermost loop are: *iterator-payload non-cf cohesion* ≈ 0.0357, *iterator-payload total cohesion* ≈ 0.3571, *loop critical payload fraction* ≈ 0.2963.

Assuming that a programmer accepted that metric hint, he decided to analyze and rewrite the loop with OpenMP pragmas (see figure 6.3), since it it visible, that the loop computes reductions for all 5 rms array elements 6.1:

$$rms[m] = \sum_{k \in grid\_points[2]-2} \sum_{j \in grid\_points[1]-2} \sum_{i \in grid\_points[0]-2} rhs[k][j][i][m] \qquad (6.1)$$

```
#pragma omp for nowait
for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            for (m = 0; m < 5; m++) {
                add = rhs[k][j][i][m];
                rms_local[m] = rms_local[m] + add*add;
            }
        }
    }
}
for (m = 0; m < 5; m++) {
#pragma omp atomic
    rms[m] += rms_local[m];
}
```

Listing 6.3: Loop, which has not been parallelized by Intel C/C++ compiler, but does seem algorithmically parallelizible, given absense of cross-iteration dependencies introduced by the function call.

Let's consider some more examples. Listing 6.4 shows another loop example, which computes reduction 6.2.

$$key\_buff\_ptr[i] = \sum_{j=0}^{i-1} key\_buff\_ptr[i] \tag{6.2}$$

It is clear, that the loop in listing 6.4 should be rewritten in order to be parallelizible, but Intel compiler gives just answer "no", but programmer takes a look at the values of metrics: *iterator-payload-non-cf-cohesion*: 0.04, *iterator-payload-total-cohesion*: 0.3, *loop-critical-payload-fraction*: 0.3077. These values take the loop down the decision tree 6.13 into parallelizible classification. Programmer might take that hint and rewrite the loop.

NAS benchmarks contain a lot of such examples. Listings 6.5, 6.6 show some more of them.

```
for( i=0; i<MAX_KEY-1; i++ ) {
    key_buff_ptr[i+1] += key_buff_ptr[i];
}
```

Listing 6.4: *SNU_NAS/IS/is.c:508*. Algorithmically parallelizible problem ends up hidden from compiler behind unsuccessful implementation.

```
for( i=1; i< NUM_BUCKETS; i++ ) {
    bucket_ptrs[i] = bucket_ptrs[i-1] + bucket_size[i-1];
}
```

Listing 6.5: *SNU_NAS/IS/is.c:468*. Algorithmically parallelizible problem ends up hidden from compiler behind unsuccessful implementation. Metrics: *iterator-payload-non-cf-cohesion*: 0.0566, *iterator-payload-total-cohesion*: 0.3208, *loop-critical-payload-fraction*: 0.2143, *loop-payload-fraction*: 0.6087

```
for (k = 1; k < nz0-1; k++) {
    for (j = jst; j < jend; j++) {
        for (i = ist; i < iend; i++) {
            for (m = 0; m < 5; m++) {
                sum[m] = sum[m] +
                    v[k][j][i][m] * v[k][j][i][m];
            }
        }
    }
}
```

Listing 6.6: *SNU_NAS/LU/src/l2norm.c:57*. Intel compiler refuses to parallelize that reduction problem due to assumed true and anti dependencies between memory references to different sum and v arrays. Inner loop metrics: *iterator-payload-non-cf-cohesion*: 0.0331, *iterator-payload-total-cohesion*: 0.3967, *loop-critical-payload-fraction*: 0.1364, *loop-payload-fraction*: 0.8148

6.4 Statistical analysis

Statistical learning techniques can be used to compare relative performance of software source code metrics in the task of parallelizability classification. This section views loop parallelizability as a machine learning problem. Metrics represent features of loops. Intel C/C++ compiler gives an expert-opinion and labels (classifies) loops as parallelizible or not. Three ML algorithms have been applied to the dataset presented in figure 6.1: Support Vector Machines (SVM), decision tree classifier, neural network based Multi-layer Perceptron (MLP). All machine learning metric set studies have been conducted with the help of pandas [29] and scikit-learn [31] python packages. Detailed description of all algorithms, techniques and all underlying mathematical foundations can be found in the introduction to statistical analysis book [32].

Table 6.15 below presents results obtained for every single loop parallelizability metric out of proposed metric set (see section 3.3).

single metrics / Machine Learning algorithm	loop absolute size	loop payload fraction	loop proper sccs number	loop critical payload fraction	iterator payload total cohesion	iterator payload non-CF cohesion	critical payload total cohesion	critical payload non-CF cohesion	payload total dependencies number	payload true dependencies number	payload anti dependencies number	critical payload total dependencies number	critical payload true dependencies number	critical payload anti dependencies number
SVN	66%	62%	62%	62%	62%	62%	62%	62%	66%	67%	67%	67%	66%	67%
Decision Tree	67%	69%	62%	68%	72%	71%	67%	66%	68%	66%	66%	65%	66%	66%
MPL	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%

Figure 6.15: Average accuracies of parallelizability predictors. Each predictor has been trained with only one feature (table columns). Different machine learning techniques (SVN, Decision Tree, MPL) have been used for training. Percentages in the table represent mean accuracies. K-fold (K=15) cross validation technique has been used to partition samples from 6.1 into training and testing sets. Different K numbers (5, 10, 20, 30) produce approximately the same mean accuracies (see figure A.4).

Every single metric has been used as a loop feature, to train different machine learning models to classify loops as parallelizible or not. Above table 6.15 reports on accuracies of corresponding trained predictor models. In order to train every single model, the dataset 6.1 has been partitioned into training and testing subsets. Partitioning has been done with K-fold method. This method splits all data samples into K equal subsets. Then it chooses one of these spilts as a testing set and uses all the rest as a training set. This process is repeated K times to cover all possible cases. Then K-fold partitioning might be applied again, but with different partitions. That application might be repeated N times. For the table 6.15 above K=15, N=5. For every training and testing run predictors report accuracy. All these accuracies have been averaged to just one percentage, presented in the table 6.15 above.

Exactly the same experiment (with K=15 and N=5) has been conducted with different sets of metrics used as features of predictor models. Table 6.16 below shows collected results.

metric sets / Machine Learning algorithm	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number + critical-dependencies-number	iterator-payload-total-cohesion + iterator-payload-non-cf-cohesion + loop-critical-payload-fraction + loop-payload-fraction	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion	loop-proportions + payload-dependencies-number + critical-dependencies-number	iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number + critical-dependencies-number	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + critical-dependencies-number	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number
SVN	75%	62%	62%	75%	74%	71%	74%
Decision Tree	77%	77%	77%	75%	77%	77%	77%
MPL	62%	62%	62%	62%	62%	62%	62%

Figure 6.16: Average accuracies of parallelizability predictors. Each predictor has been trained from sets of metrics. Results are presented in the same form as in table 6.15 above. Full table, with all K values (5,10,15,20,30) is available in the figure A.5 in the appendix.

Results, presented in tables 6.15 and 6.16 do not change significantly with K and N variations. Corresponding tables A.4 and A.5 in the appendix show the whole sets of accuracies for K equal to 5, 10, 15, 20 and 30.

It is visible from the tables, that decision tree learning method works the best in both cases: for models with single loop metrics as featues, as well as for models with sets of metrics as features. For models with single metrics as features, decision tree makes predictions with accuracy around 65-70%, with 71-72% for two *iterator payload cohesion* metrics. These metrics seem to work a bit better than others, what corerlates with conclusions made in section 6.2.1 of this chapter.

When we use sets of metrics as machine learning model features, accuracies these machine learning predictors can achieve are a bit higher. Decision tree based algorithm can achieve accuracy of 77%. This learning algorithm works comparably for all sets. For the set in the first column of table 6.16 (with all 13 metrics out of 5 groups used as features) decision tree works as good as for the set in the second column (just 4 most parallelizability correlating metrics). On the other hand, accuracies of SVN based model tend to be noticably higher for models with more metrics as features. Table 6.16 shows, that when we add dependencies number metrics to feature sets we get increaced prediction accuracy. It might look weird in the light of absence of any correlations between dependencies number metrics and loop parallelizability, as was shown in the section 6.2.1. *Loop proportion* metrics combined with *loop cohision* metrics tend to work much worse, than *loop proportion* metrics combined with *loop dependencies* number metrics. This result is also a bit surprising, because *loop cohesion* metrics correlate with parallelizability better, that *loop dependencies number* metrics.

Results of decision tree learning algorithm corelate better with the results pre-

sented in section 6.2.1 of this chapter, than SVN results. MPL accuracies do not seem to change with different feature sets at all.

Chapter 7

Results

This chapter summarizes the main results of the undertaken MSc by Research 2018 project and describes its workflow as it happened. In short, the main results of the undertaken MSc by Research project can be described with several points:

1. **Software source code parallelizability metrics search has been conducted. MSc bt Research project has been set up.**

   A body of literature has been searched through in an attempt to find any software source code metrics, applicable to the software parallelizability problem. While there are a lot of metrics aimed at judging about software quality (maintainability, readability, etc.), none were proposed to judge about software parallelizability. The only metrics in the subfield of parallel computation represent different variations of program execution time speedup ratio. Short report is given in the sections 2.1 and 2.2 of the background chapter 2.

   It was decided, that Program Dependence Graph (PDG) (proposed in the paper [3]) is going to be an intermediate program representation, parallelizability metrics would be computed on. Moreover, parallelizability metrics would use loop decoupling and loop iterator recognition results (proposed in the paper [4]) as a prerequisite for their further computation. The first metric to be computed was decided to be *loop payload fraction* (see 3.3.1.2). LLVM compiler components library [21], [20] was chosen for the MSc project to be based on.

2. **Development of LLVM-based metric computing tool.**

   This stage took the most of the time and efforts. As a result PPar tool (described in 4 and hosted on the GitHub [23]) ($\approx$ 4750 C/C++ lines of code) has been developed. The tool development started straight after the first metric ideas were

conceived (end of January 2018).

The PDG intermediate representation and all algorithms, proposed and described in [3] and [4] have been implemented from the scratch. This took a significant project start-up overheads. Development of dependence graph intermediate representation on top of LLVM IR along with loop decoupling and iterator recognition algorithms (strongly connected components search) took more than a month of time. During that period LLVM DEBUG() prints served as the only debugging and validation means.

After the first graph visualization facilities (thanks to Graphviz and DOT) had been added the project, the first research work has actually started. Along with further validation, debugging and development of the tool, first metrics started to appear on the small hand-written tests. Metric values have been manually validated against PDG and its SCCs DOT graph visualizations. Visualizations of these PDG and their SCCs served as an inspiration for the proposal of a new metrics.

By June 2018, 17 metrics have been devised and integrated into the developed PPar tool framework. Metric values could be obtained on the small set of hand-written tests. These values have been manually verified with graph visualizations. This work has been reported during the Intermediate Progress Review held on the 7^{th} of June 2018.

3. **Devised metric values have been collected on the NAS benchmark suite.**

4. **Analysis of devised parallelizability metrics.** Once all metric values had been gathered for all NAS benchmark loops and all loops had been classified with Intel C/C++ compiler, the final stage of the project could be started.

The overall picture in the matter of software source code (loops in particular) metrics for parallelisability resembeles that of the software quality metrics. Software quality (say, maintenance) is a complex notion. To judge about good or bad software design one must posses vast software engeeniring expertise and skills. Metrics like cyclomatic complexity can be used as supplementary to manual analysis, but the values they give must be examined by a human with a deep understanding of the software quality question. Although, their values might sometimes correlate with the understanding of a sound software design, these metrics should not be applied blindly.

Software parallellizability property is not a simplier one. Despite the fact, that all examined metrics have grounds to be proposed and are not randomly selected, their parallelizability correlations are limited and there are always special cases, which break

generally established rules and patterns.

However, the working framework, developed withing that MSc by research project, is ready and can be used for further metrics research and analysis. It is quite easy to add new metrics to the tool. Tool provides visualization facilities for dependence graphs and loop iterator/payload decomposition. New metrics might be added. Alternatively, existing metrics might be fine-tuned as well. This work might be the one on the relatively new direction. Application of machine learning in compilers. Since all machine learning methods require some quantitative features, these loop metrics might be an attempt in their establishment. Modern compilers apply a set of optimizations: loop unrolling, peeling, splitting. Correlations between these metrics and these properties (like loop unrollability) might be examined towards development of machine-learning driven compiler optimizers.

A lot of time has been spent on the development of the tool itself and the first, suitable for analysis, results appeared quite late in the timeframe of the project.

Chapter 8

Future work and current limitations

The major inherent problem of the done MSc research work, is that under given timeframe it was decided to use Intel C/C++ compiler as a parallelizability expert. While ICC is pretty good in program parallelization, there are still cases, where it does not parallelize loops, which could be parallelized. If there was an ideal parallelizing expert available, then some of the red dots in the presented figures would be green, which could slightly change the results. Seoul National University implementation of NAS benchmarks has two versions of the benchmarks: sequential, and the one with added OpenMP pragmas. Ideally, these pragmas could be used as answers to the parallelizability question. It was just technically easier to use ICC compiler to get the answers.

While there were some little correlations between proposed metrics in their current form and loop parallelizability property, it is clear, that in order to get ideally expected correlation results (if it is principally possible to achieve it with certain precision), these metrics must be tuned, refined and probably supplemented with additional ones. In the current project state it is going to be a way easier. Working research framework has been developed in the form of PPar tool (see 4) and surrounding scripts. All the work from C/C++ source code at the input of LLVM to the final metric figures of Python machine learning scripts has been done. PPar tool is designed in such a way, that it can be easily extended with new metrics without much efforts. Graph visualization facilities can be used to study and refine currently proposed set of metrics.

It is possible to look at the done work from another more general angle. Computation of different numeric features lies at the basis of any machine learning technique. Some loop features have been computed in this work. These features have been examined (withing certain limits) against loop parallelizability property. The same features

can be examined against different loop properties, like applicability of different loop optimizations. This work might lie on the path towards machine learning driven optimizing compilers. Apart from loops, some numeric features can be computed for different objects like program data structures, which can enable their identification by compiler.

Appendix A

Appendix

The same dataset has been also projected onto 2D plane, as illustrated in the figure A.1.

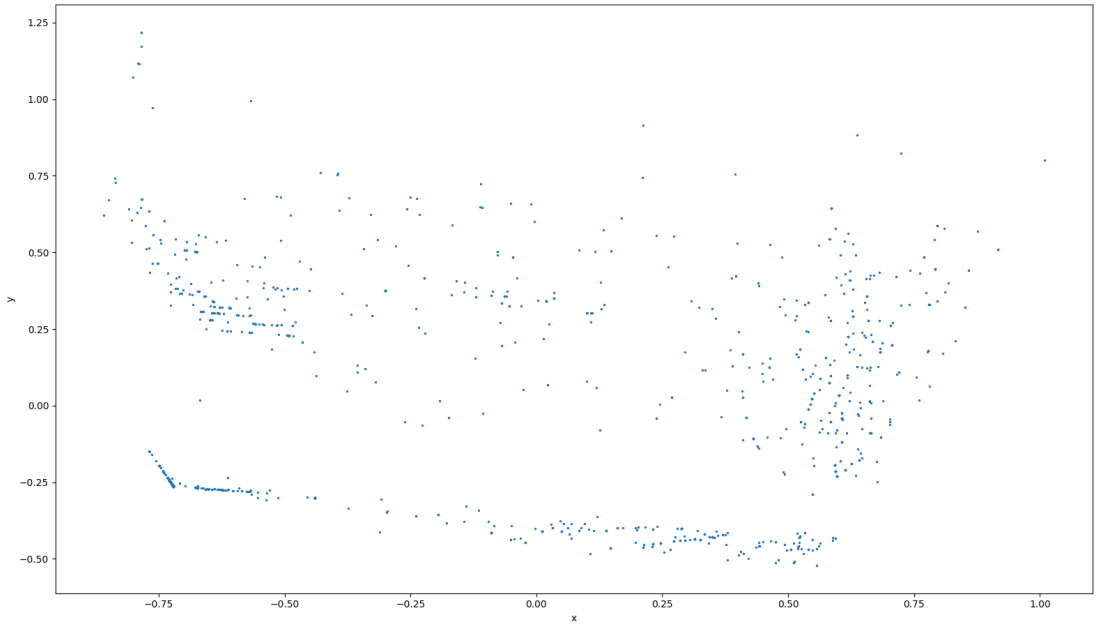

Figure A.1: Visualisation of loop metrics dataset (13-dimensional metric vectors have been projected onto 2d space thanks to PCA algorithm) - blue dots correspond to metric values on single loops.

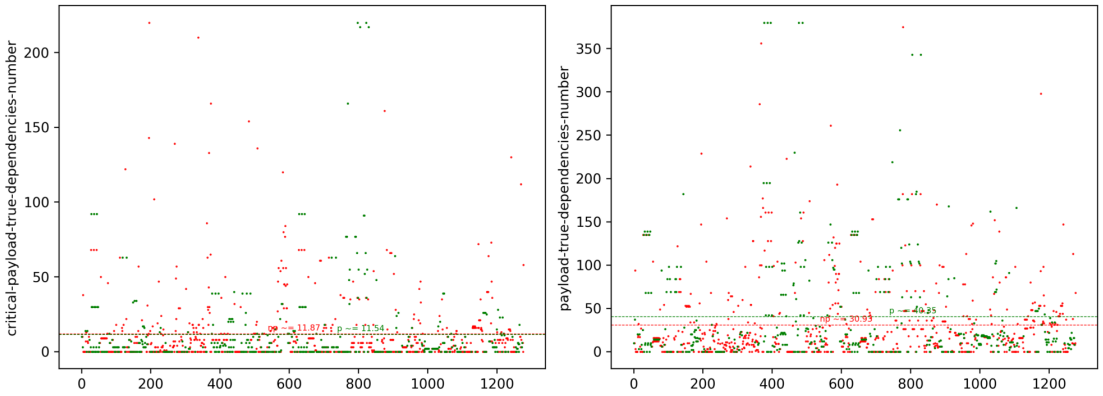

Figure A.2: *Critical payload true dependencies number* metric on the left and *payload true dependencies number* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly.

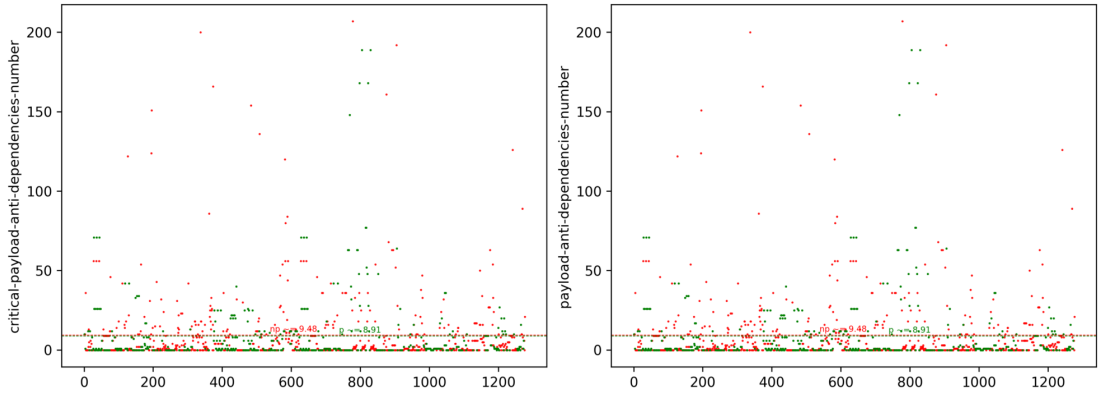

Figure A.3: *Critical payload anti dependencies number* metric on the left and *payload anti dependencies number* metric on the right. Red and green dots represent loops, which have not/have been parallelized by ICC compiler correspondingly.

single metrics		loop absolute size	loop payload fraction	loop proper sccs number	loop critical payload fraction	iterator payload total cohesion	iterator payload non-CF cohesion	critical payload total cohesion	critical payload non-CF cohesion	payload total dependencies number	payload true dependencies number	payload anti dependencies number	critical payload total dependencies number	critical payload true dependencies number	critical payload anti dependencies number
ML algorithm	k-fold number														
SVM	5	65%	62%	62%	62%	62%	62%	62%	62%	65%	66%	66%	66%	67%	66%
	10	66%	62%	62%	62%	62%	62%	62%	62%	66%	67%	67%	66%	66%	67%
	15	66%	62%	62%	62%	62%	62%	62%	62%	66%	67%	67%	67%	66%	67%
	20	66%	62%	62%	62%	62%	62%	62%	62%	66%	66%	67%	67%	66%	67%
	30	67%	62%	62%	62%	62%	62%	62%	62%	67%	66%	67%	67%	66%	67%
Decision Tree Classifier	5	66%	68%	62%	67%	71%	71%	66%	65%	66%	66%	65%	65%	66%	65%
	10	67%	69%	62%	68%	72%	71%	66%	65%	67%	66%	66%	65%	67%	66%
	15	67%	69%	62%	68%	72%	71%	67%	66%	68%	66%	66%	65%	66%	66%
	20	67%	69%	62%	68%	72%	71%	67%	66%	68%	66%	65%	65%	67%	65%
	30	67%	69%	62%	68%	72%	71%	67%	66%	68%	66%	66%	65%	66%	66%
MPL Classifier	5	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%
	10	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%
	15	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%
	20	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%
	30	62%	62%	62%	62%	62%	62%	62%	62%	62%	62%	63%	64%	64%	63%

Figure A.4: Application of machine learning (ML) techniques (table rows) to different single loop metrics (table columns). Table is populated with average accuracies of these ML methods on the chosen metric for different k-fold cross-validation parameters (5, 10, 15, 20, 30). Every single percentage number has been calculated as the mean of a set of ML classifier runs.

metric sets		loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number + critical-dependencies-number	iterator-payload-total-cohesion + iterator-payload-non-cf-cohesion + loop-critical-payload-fraction + loop-payload-fraction	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion	loop-proportions + payload-dependencies-number + critical-dependencies-number	iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number + critical-dependencies-number	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + critical-dependencies-number	loop-proportions + iterator-payload-cohesion + critical-payload-cohesion + payload-dependencies-number
ML algorithm	K-fold number							
SVM	5	73%	62%	62%	73%	73%	70%	73%
	10	74%	62%	62%	74%	73%	71%	74%
	15	75%	62%	62%	75%	74%	71%	74%
	20	74%	62%	61%	75%	74%	71%	74%
	30	75%	62%	61%	75%	74%	71%	74%
Decision Tree Classifier	5	76%	75%	76%	74%	76%	76%	76%
	10	77%	76%	77%	75%	77%	77%	77%
	15	77%	77%	77%	75%	77%	77%	77%
	20	77%	77%	77%	75%	77%	77%	77%
	30	77%	77%	78%	75%	77%	77%	77%
MPL Classifier	5	62%	62%	62%	62%	62%	62%	62%
	10	62%	62%	62%	62%	62%	62%	62%
	15	62%	62%	62%	62%	62%	62%	62%
	20	62%	62%	62%	62%	62%	62%	62%
	30	62%	62%	62%	62%	62%	62%	62%

Figure A.5: Application of machine learning (ML) techniques (table rows) to different single loop metrics (table columns). Table is populated with average accuracies of these ML methods on the chosen metric for different k-fold cross-validation parameters (5, 10, 15, 20, 30). Every single percentage number has been calculated as the mean of a set of ML classifier runs.

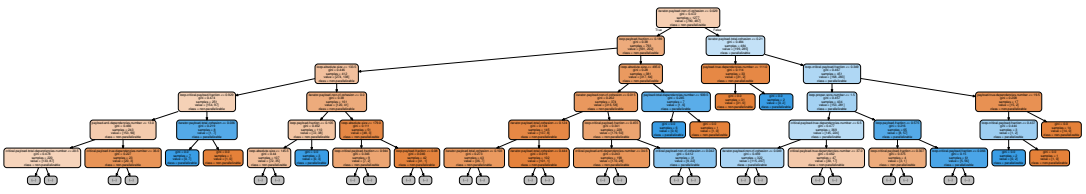

Figure A.6: Dataset 6.1 decision tree, limited to the depth of 5.

Bibliography

[1] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[2] Intel Parallel Studio XE 2018. `https://software.intel.com/en-us/parallel-studio-xe`.

[3] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[4] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. Generalized profile-guided iterator recognition. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 185–195, New York, NY, USA, 2018. ACM.

[5] Fabrizio Riguzzi and Fabrizio Riguzzi. A survey of software metrics. DEIS Technical Report no. DEIS-LIA-96-010., 1996.

[6] Eduardo Santana de Almeida Silvio Romero de Lemos Meira Aline Lopes Timteo, Alexandre lvaro. Software metrics: a survey. 2014.

[7] Jitender Chhabra and Varun Gupta. A survey of dynamic software metrics. 25:1016–1029, 09 2010.

[8] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[9] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[10] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, June 1974.

[11] Allan Albrecht. Measuring application development productivity. 1979.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[13] Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. Security metrics for source code structures. In *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, SESS '08, pages 57–64, New York, NY, USA, 2008. ACM.

[14] Sartaj Sahni and Venkat Thanvantri. Parallel computing: Performance metrics and models. Technical report, University of Florida, 1995.

[15] Intel(R). Intel Guide for Developing Multithreaded Applications. `https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications`.

[16] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[17] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[20] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[21] LLVM Official Website. `http://llvm.org/`.

[22] LLVM Online Documentation. `http://llvm.org/docs/`.

[23] Aleksandr Maramzin. Pervasive Parallelism (PPar) software parallelisability metrics tool implementation. `https://github.com/av-maramzin/PParMetrics`. University of Edinburgh, School of Informatics, MSc by Research, 2018.

[24] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[25] LLVM Doxygen Generated Documentation. `http://llvm.org/doxygen/`.

[26] Writing an LLVM Pass. `http://llvm.org/docs/WritingAnLLVMPass.html`.

[27] Graphviz - Graph Visualization Software. `https://www.graphviz.org/`.

[28] Petr Vytovtov and Evgeny Markov. Source code quality classification based on software metrics. In *Proceedings of the 20th Conference of Open Innovations Association FRUCT*, FRUCT'20, pages 71:505–71:511, Helsinki, Finland, Finland, 2017. FRUCT Oy.

[29] pandas: Python Data Analysis Library. `https://pandas.pydata.org/`.

[30] scikit-learn: Python Plotting Library - matplotlib. `https://matplotlib.org/`.

[31] scikit-learn: Machine Learning in Python. `http://scikit-learn.org/stable/`.

[32] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.