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, it gives a broad overview of various software source code metrics proposed in the field of computer science. Then, it gives an overview of the current software source code metrics, available in the subfield of parallel programming. This overview makes it clear, that there haven't been proposed any metrics, which can be used for judging about source code parallelisability. Mostly, proposed metrics represent different sorts of parallel speedup metrics.

Source code parallelisability metrics, examined in this work, are largely based on the generalized profile-guided iterator recognition work [1]. PPar tool, developed for parallelisability metrics collection is basically a pipeline of LLVM passes, which uses iterator recognition at the very beginning.

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 relatively new field is very vast. There are a lot of publications on different types of metrics as well as on their evaluation criteria, axioms the 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 ([1],[2],[3] to name a few). Work [2] 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 [1] dated as 1996 resembles [2], but also adds some critical insight. Jitender Kumar Chhabra and Varun Gupta in their paper [3] 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 [2]. 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 [3]. 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 [4], 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 [7]. 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 [8] 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 [9] gives some examples. Described metrics can be used at different stages of software development. Function points (3.5) can be used at initial stages of functional requirements specification. Software cohesion and coupling concepts (3.4) can be considered during later stages of high-level design specification (particular object-oriented software metrics (3.6)). Cyclomatic complexity (3.2), SLOC (3.1), Halstead's complexity measures (3.3) 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 parallelisability, these are not designed to directly judge about it.

2.2 Metrics in the area of parallel computing

[5]

The parallel speed-up for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc., but it will generally be less than linear relative to the number of threads used. Algorithmic parallelisability. For a whole program, speed-up depends on the ratio of parallel to serial computation (see any good textbook on parallel computing for a description of Amdahl's Law).

2.3 Modern parallelisability advisor tools

2.3.1 Intel(R) Parallel Studio XE 2018

Whithin the current project boundaries the tool is used in conjunction with Intel(R) Parallel Studio XE 2018 [6]. 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.

Intel C++ Compiler in Intel Parallel Studio XE

2.3.2 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 [7]. 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 comprehensive parallelisation reports.

Intel C/C++ compiler is used withing the scope and timeframe of the current MSc project as loop parallelisation expert. It's parallelisability reports are transformed into the following format, shown in figure below. That data is used for later statistical learning analysis as labels and parallelisability classifications for different loops of NAS benchmarks (see chapter 5).

2.4 Dependence theory

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 [8]. Here the brief descriptions fo notions are provided.

2.4.1 Types of dependencies

Generally speaking, a dependence is anything that introduces execution order constraints on statements or instructions of the sequential program. Statement S2 is dependent on statement S1, if statement S1 must be executed before statement S2. Dependencies may be broadly classified into two different categories: data and control dependencies. If statement S2 consumes the data, produced by S1, then this type of dependence is called data dependence. If whether S2 will be executed or not depends on the outcome of computation done in S1, then the statement S2 is control-dependent on statement S1.

Data dependencies are futher subdivided into four subcategories.

Read After Write (RAW) dependencies

Write After Read (WAR) dependencies

Write After Write (WAW) dependencies

Read After Read (RAR) dependencies

2.5 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 [9].

2.6 Control flow analysis

Control flow analysis [10]

2.7 Program Dependence Graph (PDG)

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

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.

2.7.1 Data dependence graph (DDG)

2.7.2 Memory dependence graph (MDG)

2.7.3 Control dependence graph (CDG)

2.7.4 Program dependence graph (PDG)

2.8 Loop decoupling

[?]

Chapter 3

Software Parallelisability Metrics

This chapter defines proposed software source code parallelisability metrics and gives the basic intuition behind them. Proposed metrics inherited dependence-based nature from the work [8]. This book is built on and describes the results gathered through countless years of research and tremendous amount of work done in the field of optimizing compilers and high-performance computer architectures.

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 3.3 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.

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

Loops and arrays the most fertile ground for optimizations

3.1.4 Dependence-based approach to metrics computation

Program 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 2.4 of the thesis.

3.2 Metrics use

Ideally, metrics should provide a quantitative measure of loop's algorithmic parallelisability (say, this loop is 80% parallelisible). While existent modern tools (like 2.3) give answers only in binary format: yes, this loop has been parallelised or no, it hasn't been parallelised.

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 3.1 given below. This loop is taken form EP NAS benchmark (see 5).

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

Listing 3.1: Example loop, taken from EP NAS benchmark

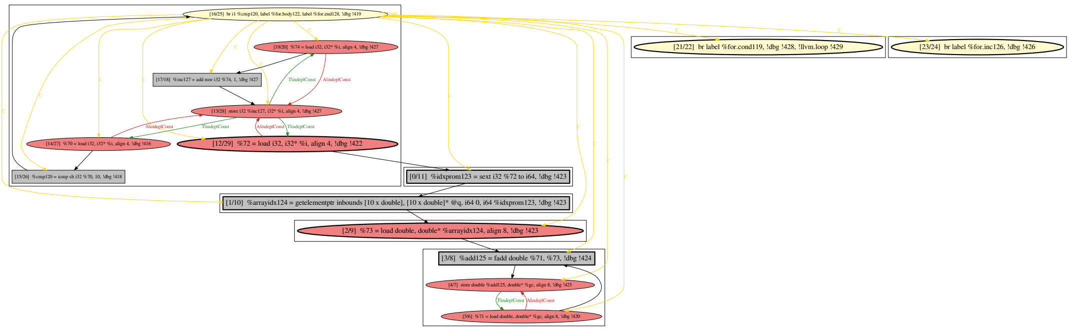

[16/25] br i1 %cmp120, label %for.body122, label %for.end128, !dbg !419
[19/20] %74 = load i32, i32* %i, align 4, !dbg !427
[17/18] %inc127 = add nsw i32 %74, 1, !dbg !427
TIindep|Const
AIindep|Const
[13/28] store i32 %inc127, i32* %i, align 4, !dbg !427
AIindep|Const
TIindep|Const
AIindep|Const
TIindep|Const
[14/27] %70 = load i32, i32* %i, align 4, !dbg !416
[12/29] %72 = load i32, i32* %i, align 4, !dbg !422
[15/26] %cmp120 = icmp slt i32 %70, 10, !dbg !418
[0/11] %idxprom123 = sext i32 %72 to i64, !dbg !423
[1/10] %arrayidx124 = getelementptr inbounds [10 x double], [10 x double]* @q, i64 0, i64 %idxprom123, !dbg !423
[2/9] %73 = load double, double* %arrayidx124, align 8, !dbg !423
[3/8] %add125 = fadd double %71, %73, !dbg !424
[4/7] store double %add125, double* %gc, align 8, !dbg !425
TIindep|Const
AIindep|Const
[5/6] %71 = load double, double* %gc, align 8, !dbg !420
[21/22] br label %for.cond119, !dbg !428, !llvm.loop !429
[23/24] br label %for.inc126, !dbg !426
C

Figure 3.1: Program dependence graph (PDG) of the loop 3.1, as built and visualized by the PPar tool 4.

Figure 3.1 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 2.8.

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 3.1 it is visible that the value of the metiric for the given loop 3.1 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 2.8, 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 3.1 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 3.1. Despite the presense of cross-iteration dependency in the loop, Intel C/C++ compiler is capable of its parallelization with reduction techniques.

Simplier loops, like the one shown on the listing 3.2, 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 3.2 provides illustration.

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

Listing 3.2: Parallelizible loop, with no cross-iteration dependencies. Taken from EP NAS benchmark.

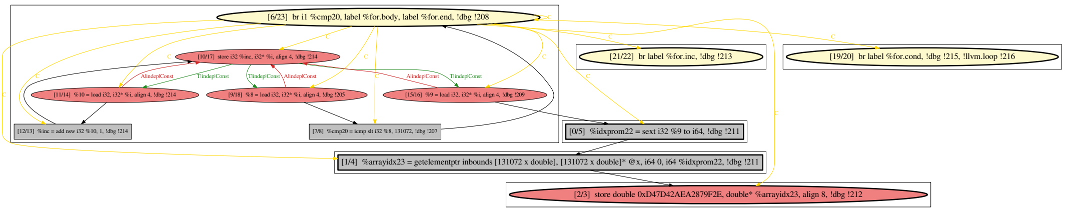

[6/23] br i1 %cmp20, label %for.body, label %for.end, !dbg !208
[10/17] store i32 %inc, i32* %i, align 4, !dbg !214
[11/14] %10 = load i32, i32* %i, align 4, !dbg !214
[9/18] %8 = load i32, i32* %i, align 4, !dbg !205
[15/16] %9 = load i32, i32* %i, align 4, !dbg !209
[12/13] %inc = add nsw i32 %10, 1, !dbg !214
[7/8] %cmp20 = icmp slt i32 %8, 131072, !dbg !207
[21/22] br label %for.inc, !dbg !213
[19/20] br label %for.cond, !dbg !215, !llvm.loop !216
[0/5] %idxprom22 = sext i32 %9 to i64, !dbg !211
[1/4] %arrayidx23 = getelementptr inbounds [131072 x double], [131072 x double]* @x, i64 0, i64 %idxprom22, !dbg !211
[2/3] store double 0xD47D42AEA2879F2E, double* %arrayidx23, align 8, !dbg !212
A|indep|Const
T|indep|Const
C

Figure 3.2: Program dependence graph (PDG) of the loop 3.2, 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 Loop Dependence Metrics

3.3.3 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 [11] [12]. The tool is implemented as a set of LLVM passes (see LLVM online documentation for further technical details [13]). The tool can be found at [14]. 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 [15] (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 2.4.1 (data, control, etc), which are abstracted away from programming languages domain into a separate dependence analysis theory 2.4. 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 [16].

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 [17]) 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 [18]).

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 2.8 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 2.7. 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) [15]. 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 2.7. 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 [1] to decouple loops into iterator and payload parts (see section 2.8). 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

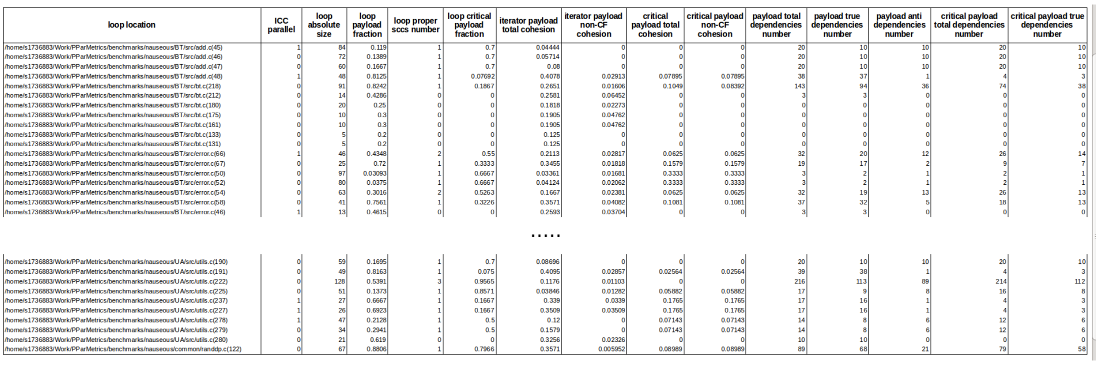
```
for (i = 0; i < MK + 1; i++) {
    t2 = randlc(&t1, t1);
}
```

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.

Figure 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. This data is, essentially, a set of loops found in NAS benchmarks. For every single loop a set 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 expert in this project. The table contains around 1400 loops with and 13-dimensional feature vector for each.



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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
						.....								
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/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
/home/s1736883/Work/PParMetrics/benchmarks/nauseous/common/randdp.c(122)	0	67	0.8806	1	0.7966	0.3571	0.005952	0.08989	0.08989	89	68	21	79	58

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

This chapter describes analysis approaches and presents findings in the report. First, the collected data has been visualized to see any obvious correlations between loop parallelisability and metric values. Section 6.2 provides data interpretation and visualizations. Section 6.4 contains some insights into analysis results. Then, statistical analysis techniques have been applied to the data. Loop metrics have been viewed as machine learning features in the context of loop parallelisability classification problem. Standard state-of-the (such as Support Vector Machines (SVM), desicion trees, etc) machine learning techniques have been applied to the data and all prediction errors have been compared agains random predictor. Section 6.5 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 [19]. 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.

6.1 Analyses preparation phase

As it turned out, there are some outliers in the collected data that distort the final result.

6.2 Data interpretation and visualization

The first step in the analysis of collected loop metrics and parallelisability properties is to visualize the data and manually check for existent data patterns and correlations. Python packages pandas [20] and matplotlib [21] have been used for the task.

6.3 Data clustering analysis

6.3.1 Combined metrics data visualization

This section describes an attempt to conduct structural data analysis and identify separate clusters in the data. Since loops are represented by 13-dimensional vectors, direct dataset visualization is unfeasible. Principal Components Analysis (PCA) has been used to project the data onto 3-dimensional space and visualize it. Figure 6.2 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.

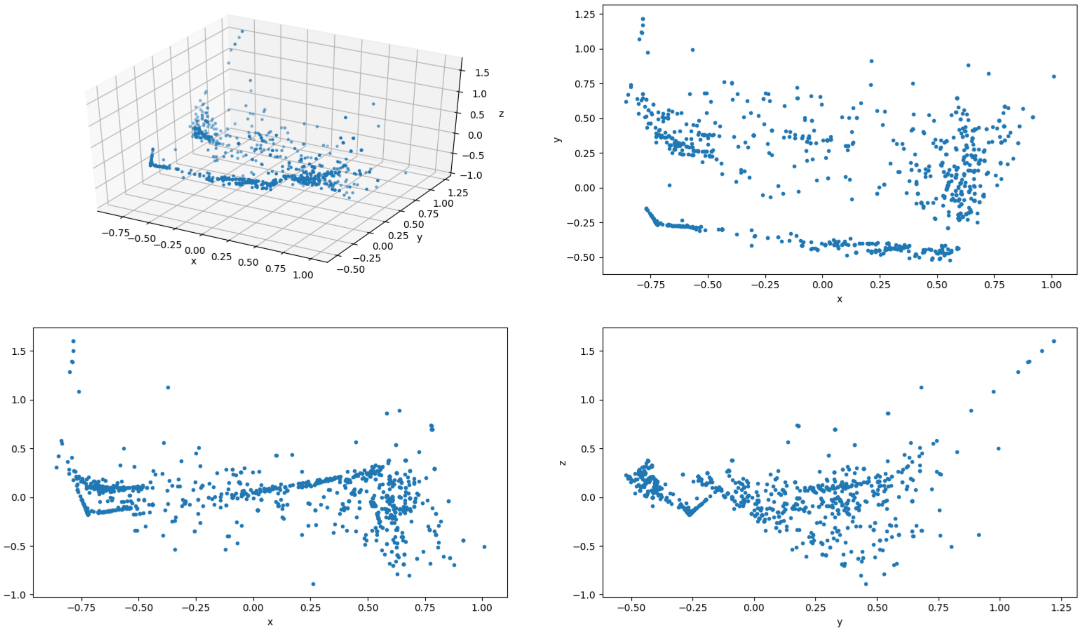


Figure 6.2: Visualisation of loop metrics dataset (13-dimensional metric vectors have been projected onto 3d space thanks to PCA algorithm) - blue dots correspond to metric values on single loops.

Metric dataset projection onto 2D plane looks alike XY projection of 3D PCA mapping (see figure A.1).

6.4 Manual analysis

6.4.1 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 6.1 below, gives such example.

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

Listing 6.1: Non-parallelizible loop. Function call inside loop's body prevents ICC compiler from parallelizing it. Loop taken from EP NAS benchmark.

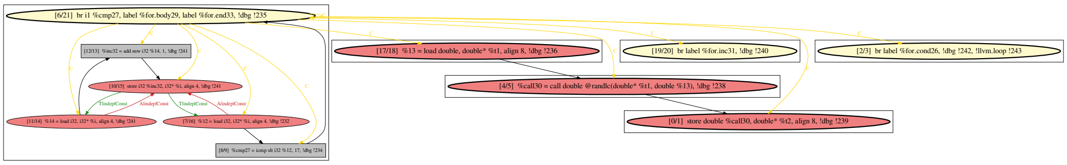

[6/21] br i1 %cmp27, label %for.body29, label %for.end33, !dbg !235
[12/13] %inc32 = add nsw i32 %14, 1, !dbg !241
[10/15] store i32 %inc32, i32* %i, align 4, !dbg !241
T|indep|Const
A|indep|Const
T|indep|Const
A|indep|Const
[11/14] %14 = load i32, i32* %i, align 4, !dbg !241
[7/16] %12 = load i32, i32* %i, align 4, !dbg !232
[8/9] %cmp27 = icmp slt i32 %12, 17, !dbg !234
[17/18] %13 = load double, double* %t1, align 8, !dbg !236
[19/20] br label %for.inc31, !dbg !240
[2/3] br label %for.cond26, !dbg !242, !llvm.loop !243
[4/5] %call30 = call double @randlc(double* %t1, double %13), !dbg !238
[0/1] store double %call30, double* %t2, align 8, !dbg !239
C

Figure 6.3: Program dependence graph (PDG) of the loop 6.1, as built and visualized by the PPar tool 4.

Loop in the listing 6.1 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 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.

6.5 Statistical analysis

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.

Statistical analysis of loop parallelisability metrics has been conducted with the help of pandas [20] and scikit-learn [22] python packages. Detailed description of all algorithms, techniques and all underlying mathematical foundations can be found in the introduction to statistical analysis book [23].

6.5.1 K-Means clustering

In this work k-means clustering techniques have been used as the first method of statistical analysis.

6.5.2 SVM-based parallelisability analyzer

Chapter 7

Results

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, they exibit only minor correlations with general parallelisability property 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

Appendix A

Appendix

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

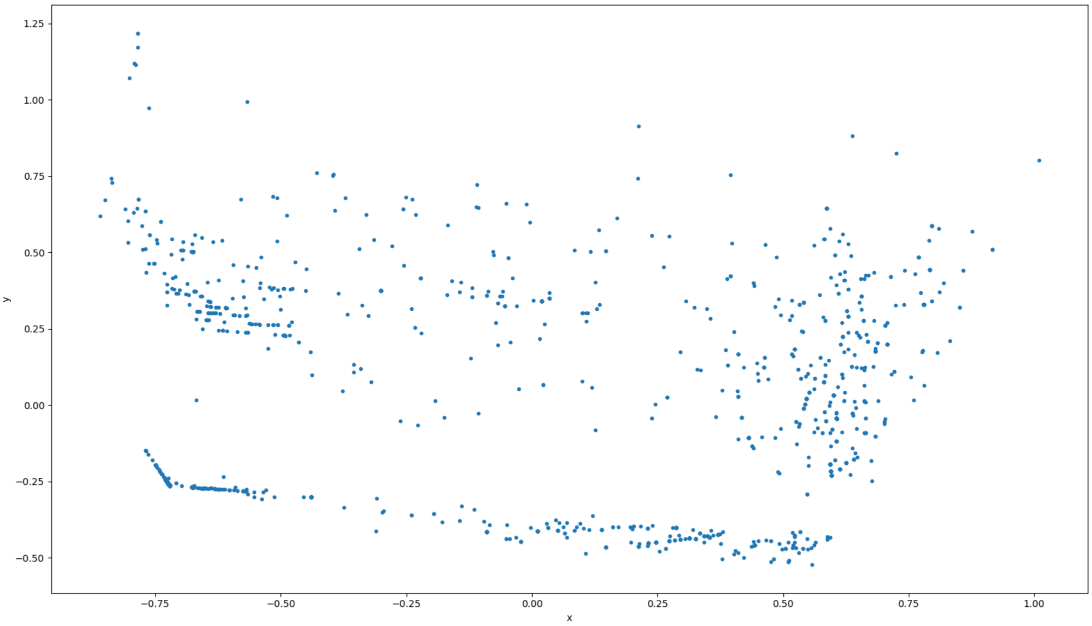


Figure A.1: Visualisation of loop metrics dataset (13-dimensional metric vectors have been projected onto 2d space thanks to PCA algorithm) - blue dots correspond to metric values on single loops.

Bibliography

[1] 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.

[2] 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.

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

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

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

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

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

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

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

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

[11] 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.

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

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

[14] 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.

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

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

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

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

[19] 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.

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

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

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

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