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 describes

2.1 Software metrics in computer science

The idea of software 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)

[1] Another well-known software metric is cyclomatic complexity (CC). The metric was first developed by Thomas J. McCabe in 1976 [4]. 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 [5]. 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 [6], 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

2.3 Dependence theory

[2]

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 [3].

2.5 Control flow analysis

Control flow analysis [4]

2.6 Program dependence graph (PDG)

[5]

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 decoupling

[6]

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 [2]. 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 3.2 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

3.1.2 The modern role of compilers

3.1.3 The famous 80/20 rule

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 2.3 of the thesis.

3.2 Metric Groups

The whole set of proposed metrics is divided into several concepual groups.

3.2.1 Loop Proportion Metrics

3.2.1.1 Loop Absolute Size

3.2.1.2 Loop Payload Fraction

3.2.1.3 Loop Proper SCCs number

3.2.2 Loop Dependence Metrics

3.2.3 Loop Cohesion Metrics

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 [7] [8] and implemented as a set of LLVM passes (see LLVM online documentation for futher technical details [9]). The tool can be found at [10]. 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 [11] (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. In order to use a 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 [12].

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 [13]) 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 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).

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 3.2 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 4.1.2 below. Representation of dependence graphs in the memory is described in the section 4.1.3 of this chapter. Section 4.1.4 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 4.1.5.

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.

4.1.5 Template specializations

4.2 Tool use

4.2.1 Intel(R) Parallel Studio XE 2018

Whithin the current project boundaries the tool is used in conjunction with Intel(R) Parallel Studio XE 2018 [14]. 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

Chapter 5

Benchmarks

NAS Parallel Benchmarks have been used withing this project.

5.1 Benchmark descriptions

In the graph pass

5.1.1 EP - Embarrassingly Parallel

Chapter 6

Analysis

6.1 K-Means clustering

6.2 SVM-based parallelisability analyzer

Chapter 7

Results

Bibliography

[1] 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.

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

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

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