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

2.1 Software Quality Metrics

2.1.1 McCabe's Cyclomatic Complexity

[9]

2.2 Dependence Theory

[5]

2.3 Program Dependence Graph

[4]

Chapter 3

Tool Implementation

The tool developed for source code parallelisability metrics research is completely based on the LLVM library of modular and reusable compiler technologies [6] [2] and implemented as a set of LLVM passes (see LLVM online documentation for futher technical details [3]). The tool can be found at [8] and is implemented as a set of LLVM passes, which heavily rely on the standard C++ template mechanism.

The tool operates on the level of LLVM intermediate representation [1] (LLVM IR) and completely decoupled from input languages as well as from target machine instruction set.

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

3.1 LLVM analyses used in the tool

There are several LLVM provided analysis passes being used in the tool.

3.1.1 LoopInfo analysis

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.

3.1.2 DependenceAnalysis analysis

3.2 Dependence Graphs

Since LLVM, as of version 6.0, does not currently provide a standard Dependence Graph (DG) implementation, custom graph building facilities were implemented 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 node and edge correspondingly.

3.2.1 Data Dependence Graph (DDG) pass

3.2.2 Memory Dependence Graph (DDG) pass

3.2.3 Control Dependence Graph (DDG) pass

3.2.4 Program Dependence Graph (DDG) pass

3.3 Loop Decoupling Pass

3.4 DOT graph printing facilities

3.5 Metric Groups

Chapter 4

Benchmarks

Tool developed as a set of LLVM passes

4.1 GraphPass

In the graph pass

4.1.1 Graph

4.1.1.0.1 Program Dependence Graph

Chapter 5

Analysis

Chapter 6

Results

Bibliography

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.