THE UNIVERSITY *of* EDINBURGH
informatics

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

MSc by Research
Informatics Research Proposal

Software Metrics for Parallelism

MSc Student	Aleksandr Maramzin (s1736883)
Primary Supervisor	Björn Franke
Domain Expert Co-Supervisor	Mike O'Boyle
Wild Card Co-Supervisor	Kenneth Heafield

Edinburgh, United Kingdom, 8 December 2017

1. 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.

2. Introduction

(Problem in the modern computing world)

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.

3. Motivating example

(“Bad” and ”Good” parallelisible code)

To illustrate some of the problems in the field of parallel programming, let's consider the task of multiplying two vectors **a** and **b**. Result of multiplication is stored into variable **c**.

Good score

```
int a[1000];
int b[1000];
int c;

// initialize arrays

c = 0;
for (int i = 0; i < 1000; i++) {
    c += a[i] * b[i];
}
```

Figure 1. C implementation of vector multiplication with array.

Bad score

```
Node *a;
Node *b;
int c;

// initialize linked lists

c = 0;
while (a && b) {
    c += a->val * b->val;
    a = a->next;
    b = b->next;
}
```

Figure 2. C implementation of vector multiplication with linked list.

Two code snippets, presented in Figure 2 on the left and Figure 3 on the right, implement vector multiplication in the C programming language. Left code snippet uses arrays, whereas right code snippet uses two linked lists for underlying vector representation. Both code snippets are functionally equivalent and produce the same result (provided that they are identically initialized), but differ from the point of code parallelisation. While array implementation is parallelisible, the while loop on the right has cross iteration dependency. We cannot proceed to the next iteration until we compute address of the next element in the list.

This example clearly demonstrates the problem. Suboptimal data structure choice made by the programmer hid all the inherent parallelism of the task and made it further impossible for compiler to automatically parallelise the program. If only a programmer had some parallelisation feedback tools, which could mark the code on the left as “good” and the code on the right as “bad”, then he would have paid more attention to this piece of code and gave it a second thought.

4. Software metrics in computer science

(Previous attempts to quantify the code)

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.

3.1 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 code

lines 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.

3.2 Cyclomatic complexity (CC)

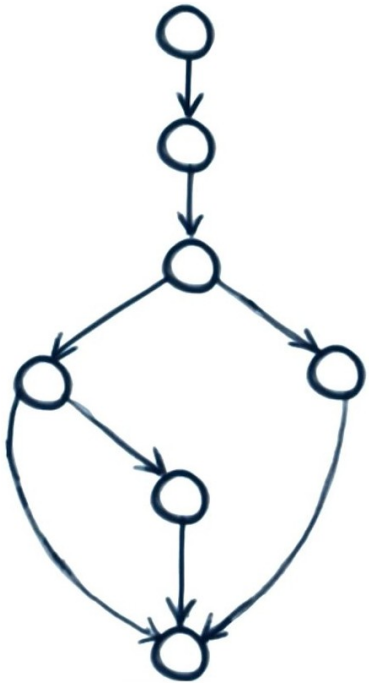

Figure 3. CFG of the code sample, CC equals to 3.

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** + 2**P**, 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.

If SLOC judges mostly about software product size and hence different sorts of associated efforts, CC is used to assess software source code complexity as perceived by the programmer (psychological comprehensibility, program's structuredness, etc.).

Sometimes CC is used to limit complexity of software modules during development. Programmers are required to work in accordance with the following practice: if CC of routine being developed grows higher than some pre-specified number (10-15 for instance), then that routine must be redesigned and split into smaller pieces of code.

CC found another application in the software testing area. As it is easy to notice, CC is equal to the number of test cases required to achieve thorough test coverage.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

3.7 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.

5. MSc by Research work plan

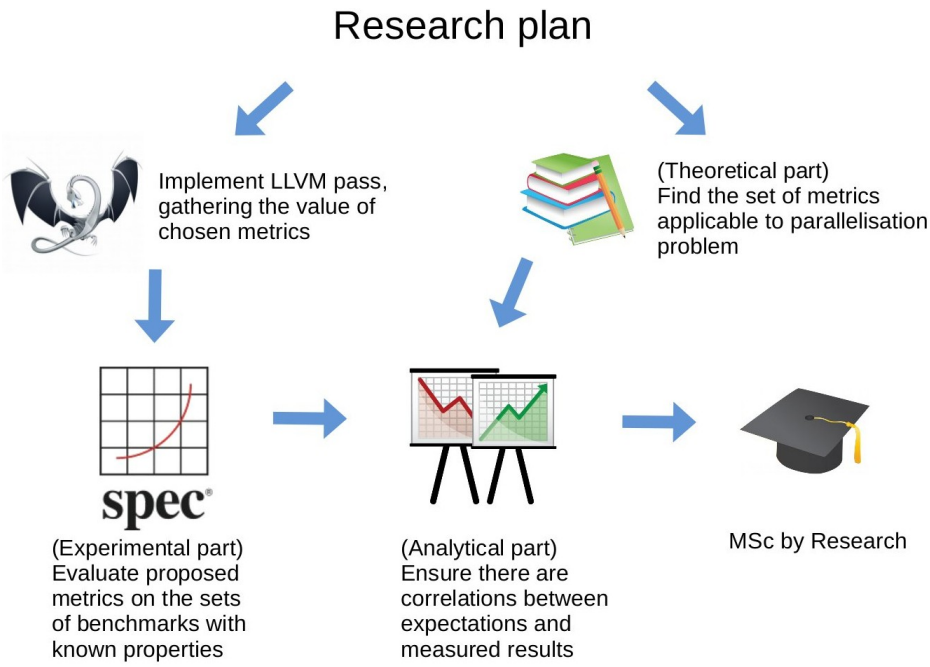

Figure 4. Illustration of the proposed work flow. Some separate parts of the work are interdependent on each other.

The nature of our work is similar in concept to that in [11]. The goal of the work [11] is to devise a working software source code quality assessment methodology. In that work authors explore correlation between existent software metrics (CC, Halstead's software science and some other low-level metrics) and known properties of the set of their hand-coded programs. Metrics are validated empirically. Authors implement an LLVM pass to collect values of the metrics. Statistical clustering techniques are further applied in order to classify studied programs into three groups: “good”, “normal” and “bad”. The same experiment is applied to Mozilla Firefox browser and LLVM compiler source codes.

While [11] studies software quality metrics, we are going to conduct a similar type of the work for software parallelisability. The proposed plan is almost the same as in work [11], except that we are additionally devising our own metrics, which are suitable for reasoning about software parallelisability. Figure 5 gives an illustration for the plan of the work, which consist of several parts:

Part 1. Benchmark search.

In this part decision regarding a set of benchmarks to use must be taken. There should be several types of benchmarks in the final test suite.

Some benchmarks must be specifically designed to run on parallel machines. These must be really well-designed benchmarks. Proposed metrics at the end should score the code of these benchmarks with the highest mark. NASA Advanced

Supercomputing (NAS) Division Parallel benchmarks seem to be a good choice. NAS benchmarks are targeted to run on highly parallel supercomputers. Reference implementations of NPB (NAS Parallel Benchmarks) use MPI and OpenMP programming models.

There must be benchmarks with opposite parallelisability properties in the final suite. There are compute intense SPEC benchmarks namely SPECint and SPECfp. SPECint and SPECfp benchmarks are usually run only on single CPU. If a system has multiple cores, then only one is used. Hyper-threading is also typically disabled. They test single processor's computing capabilities and are not supposed to run in parallel. These benchmarks should receive “bad” parallelisability scores.

Part 2. Software source code parallelisability metrics search.

Metrics search will rely on:
1) theoretical insight, based on considerable amount of parallelism related literature (compiler's data dependence and alias analyses, general parallel programming theory, etc.)
2) expert opinions in the pervasive parallelism field

Received knowledge will be used to formally work out a set (vector) of metrics to be verified.

Part 3. Empirical metrics verification.

In this part devised metrics will be empirically verified. LLVM appears to be the most suitable tool for that task. Software metrics verification consists of several steps:
1) Implement LLVM pass for metrics collection.
2) Run implemented pass on benchmarks chosen in the Part 1.
3) Receive and analyse results based on correlation between known properties of benchmarks and metrics behaviour.

If correlations are as expected, then go to the step 5), if metrics appear to give wrong results, then do the next step 4).
4) Re-design selected metrics and go to the step 1).
5) Represent results in the illustrative format. Statistical techniques can be used here. It must be clear from visualisations that there are “good parallelisable” benchmarks and “bad parallelisable” ones.
6) *(Optional step)* Integrate devised metrics into IDE of choice, try them out and write an end-user experience report.

6. The final research output expectations

(Benchmarks parallelisability classification)

As was already outlined in the previous sections of this proposal, the main final expected result is a vector of software parallelisability metrics. The values of these metrics must correlate with known behaviours of the chosen benchmarks. Table 1 below illustrates data to be collected.

	$Metric_1$ (M_1)	$Metric_2$ (M_2)	...	$Metric_{m-1}$ (M_{m-1})	$Metric_m$ (M_m)	$F(M_1,M_2, \ldots, M_{m-1}, M_m)$
$Benchmark_1$						
$Benchmark_2$						
...						
$Benchmark_{n-1}$						
$Benchmark_n$						

Table 1. Final expected research output. Benchmarks have different parallelisability properties. Hard-to-parallelise benchmarks must receive low scores, embarrassingly parallel ones must receive high scores. Metrics may be combined with some function F and mapped into some illustrative space.

While the ultimate and the main goal of the proposed research is development of software parallelisability metrics, research efforts applied in that direction must produce some side-results as well. The one such result is classification of benchmarks used in this research with regard to software parallelisability presented on the figure 5 below.

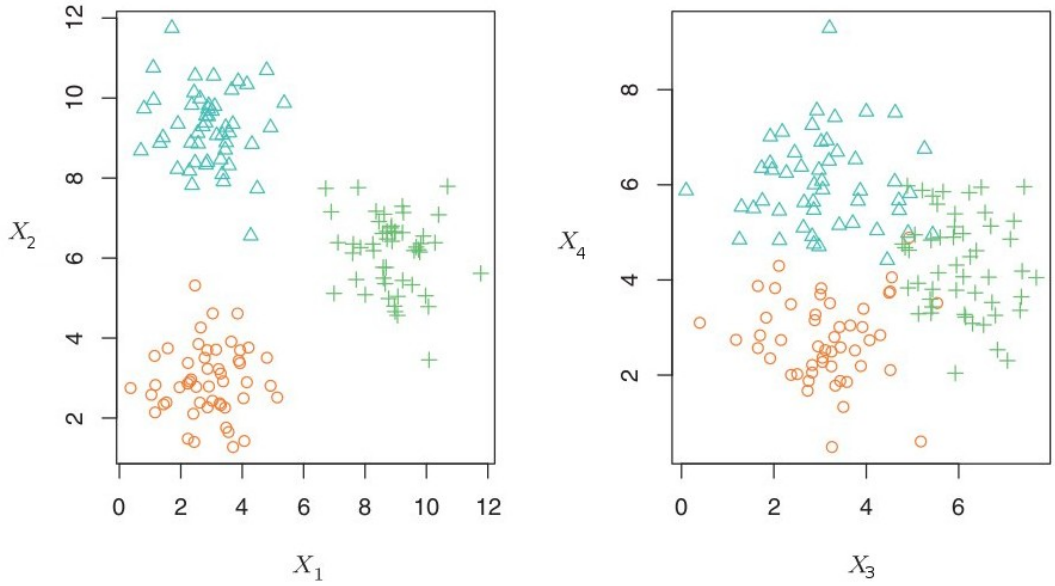

Figure 5. *(Illustration is taken from [12] + modified)* Clustering of benchmarks into distinct parallelisability groups based on the observed software parallelisability metrics X_1, X_2, X_3, X_4 values.

References

[1] Fabrizio Riguzzi. A survey of software metrics. July 1996: Universita degli Studi di Bologna, DEIS. DEIS Technical Report no. DEIS-LIA-96-010. LIA series no. 17.

[2] Aline Lopes Timóteo, Alexandre Álvaro, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira. “Software metrics: a survey”. Centro de Informática - Universidade Federal de Pernambuco (UFPE) and Recife Center for Advanced Studies and Systems (C.E.S.A.R), Brazil, 2014.

[3] Chhabra JK, Gupta V. A survey of dynamic software metrics. Journal of computer science and technology 25(5): 1016–1029, September, 2010. DOI 10.1007/s11390-010-1080-9.

[4] Thomas J. McCabe. “A complexity measure”. October, 1976. ICSE'76: Proceedings of the 2nd international conference on Software engineering.

[5] Maurice H. Halstead. Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA, 1977.

[6] Stevens, W. P.; Myers, G. J.; Constantine, L. L. (June 1974). “Structured design". IBM Systems Journal. 13 (2): 115–139.

[7] Allan Albrecht. “Measuring application development productivity”. 1979. IBM Corporation, White Plains, New York.

[8] S. Chidamber, C. Kemerer. “A Metrics Suite for Object Oriented Design”. IEEE Transactions Software Engineering, 20/6, 1994, pp. 263-265.

[9] Istehad Chowdhury, Brian Chan, Mohammad Zulkernine. “Security Metrics for Source Code Structures”. SESS’08, May 17–18, 2008, Leipzig, Germany.

[10] Chris Lattner, Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. Proceedings of the International Symposium on Code Generation and Optimization (CGO’04).

[11] Petr Vytovtov, Evgeny Markov. “Source code quality classification based on software metrics”. 20th Conference of FRUCT assosiation.

[12] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. An Introduction to Statistical Learning with Applications in R. Springer Texts in Statistics. © Springer Science+Business Media New York 2013. Chapter 2.1, Figure 2.8.