PhD Progression Document*

MSc by Research, CDT in Pervasive Parallelism

Student:
Aleksandr Maramzin

Supervision Team:
Björn Franke, Michael O'Boyle, Kenneth Heafield

30 August 2018

General words on Parallel Programming

In the modern computing world parallel programming is more actual than ever. From small embedded processors to warehouse-scale servers and supercomputers, all modern hardware is designed for running parallel computations. Unfortunately, the problem in the field of parallel computation is still present, and has shifted from the design of parallel hardware onto the development of parallel software.

Manual parallel programming is a complex task. It requires a combination of skills: solid algorithmic background, familiarity with the domain software is written for, general knowledge of parallel software development theory (deadlocks, race conditions, etc), familiarity with exact parallel programming frameworks to be used (MPI, OpenMP, etc.) and maybe even some knowledge of the exact underlying hardware architecture, software is to be run on (GPU, heterogeneous system).

The tuning of software mapping to such diverse and complex parallel hardware systems can be done with arbitrary granularity. Different tools have been developed for alleviating this hard and skill-requiring parallel software development process. Intel Parallel Studio 2018 [1] is probably the state-of-the-art. But these tools are quite complex and elaborate by themselves

*This document describes the lessons learnt from the undertaken MSc by Research project and revised PhD vision.

and require an intensive training, before they can be used. Intel C/C++ compiler on the other hand provides only short and simple reports, which are limited to binary "yes, parallelizible"/"no, non-parallelizible" answers. Moreover, compiler alias and memory dependence analyses are sometimes imprecise and conservative. The latter introduces confusion and mistakes into user reports.

Sometimes a software developer does not really need fine-grained tuning opportunities of Intel Parallel Studio 2018, but wants to know a bit more, than Intel C/C++ compiler can provide. Here we decided to conduct a research of a possible parallelizability user feedback forms.

Software Metrics for Parallelism

(MSc by Research)

Software source code metrics are not new to the field of software engineering. Numerous software source code quality metrics have been proposed (like McCabe′s Cyclomatic Complexity [2] and Halstead′s Software Science [3]). These metrics are somewhat controversial and some practitioners think, that these simplistic software quality measures can do more harm, than good. Others, integrate calculation of these metrics into regular software development subtasks. As our research has concluded, these metrics are not directly applicable to software source code parallelizability property and none metrics have been proposed to judge about source code parallelisability. The only metrics we could find in the field of parallel programming represent different variants of parallel program speedup and are not applicable to our task:

$$speedup = \frac{serial\ execution\ time}{parallel\ execution\ time} \tag{1}$$

In this MSc project we conducted a research of a set of software source code metrics and their applicability to the problem of loop parallelisation. We decided to base our metrics on compiler dependence analysis theory, and particularly on the structural properties of Program Dependence Graph (PDG) [4]. The undertaken project relied heavily on the loop iterator recognition work [5] as well.

As a result a set of loop parallelizability metrics has been devised and proposed. An LLVM-based tool has been developed for computation of these metrics. Values of proposed metrics have been computed for all loops of NAS Parallel Benchmarks [6], [7]. And after that different statistical analysis tech-

niques have been applied to computed data in order to evaluate devised software metrics for parallelism.

The outcomes of the project resemble the situation in the area of software quality. Like software quality metrics, software metrics for parallelism can be used to capture some parallelisability properties, but they cannot be used blindly without any human analysis. The main lessons learnt from the undertaked MSc by Research project could be enumerated as follows:

1. **Difficulty of parallelisability problem.** Software parallelizability is a complex task with a long, rich and vast research history, and it would be naive to expect perfect correlation between simplistic PDG structure based metrics and parallelizability property.

2. **Errors in parallelisability classification.** Devised metric give probabilistic answers (like this loop is likely to be parallelisible with probability of 65%) and are blurred. In other words, we do not expect precise decision boundary between parallelisible and not. Some problem inherent errors are always going to present.

3. **PDG structural properties + nature of PDG instructions.** While proposed metrics are based on the structural properties of PDG, it would be worthwhile to consider the nature of instructions, which constitute the graph (motivating example is a call instruction in a loop payload, hindering loop parallelisation without application of interprocedural analisis).

4. **The need for supplementary insights.** Even after introduction of better tuned parallelizability metrics and some more research efforts, we feel, that these metrics are still going to possess blurred and probabilistic nature. Thus, if we are talking about software developer feedback and assistance we need to research some other approaches.

Data-Centric Parallelisation

(PhD vision)

As the results of the MSc by Research project show, loop features alone might not be enough, to provide a software engineer with a comprehesive insight into parallelisability of his/her source code and its problems. Fortunately, there are other supplementary approaches to this problem.

```
while(ptr) {
    ptr->val++;
    ptr = ptr->next;
}
```

Listing 1: Irregular loop. Parallelisible algorithm is hidden behind suboptimal implementation.

```
std::vector<int> container;
#pragma openmp parallel for
for (int& val : container) {
    val++;
}
```

Listing 2: Linked-list based array in listing 1 has been replaced with C++ STL vector. Now we can parallelise the loop manually. Intel compiler can do that automatically as well.

10,000 foot problem view

Let's consider a loop, illustrated in the listing 1. Having figured out, that *ptr* points to C language structure with integer *val* field, which represent an element of an integer array, we can conclude, that this parallelisible algorithm is implemented with suboptimal underlying data structure. Intel compiler won't proceed with parallelisation of this loop. We are going to get "loop is not a candidate for parallelisation" feedback in the report.

Here the state-of-the-art compiler analyses and transformations cannot proceed any further. Although, a human programmer can easily see a solution. If we replace linked-list with just a regular array, we will immediately expose all inherent parallelism to the compiler and it will do its job. If we don't stop here and go even further we can "rejuvenate" the code, by replacing a simple C-style array with C++ standard library's *std::vector* like illustrated in the listing 2. If we can prove that increment operation order does not matter, then we can even replace *std::vector* with the *std::set*. The latter can bring some additional performance improvements.

Above example described the niche of this PhD project. We are going to concentrate our attention on "irregular" applications, like the one in the example. Irregular applications are abundant in legacy code. These applications work with "naked" C-style pointers and often contain suboptimal implementations. Such problems did not show up at the time of legacy applications development, but are critical for modern parallel hardware systems.

The value of the work

While linked-list example in the previous section is rather simple, it would still be useful to detect such suboptimalities automatically. Some software developers might not even spot that performance (and quality as well) problem of the code, and automatic feedback tool would be handy.

Moreover, sometimes data structures can be rather complex and one cannot fully understand how the data structure is being built and used, until he/she fully examines surrounding code with all data structure manipulations (element insertions, deletions, etc). Automatic abstract data structure recogniser would significantly help here. If data structure exhibit a random pattern of accesses, then this tool could advice a programmer to keep to a sequantial array, rather than a linked list. If, on the other hand, a data structure is being constantly modified, it becomes cumbersome to shift and realign an array in the memory and linked-list might be the optimal choice.

All these common sense considerations might be engraved into proposed static analysis tool. Intel compiler does not provide such reports. Intel VTune is supposed to fine-tune already parallelised applications and works on the very low level with a lot of details. High algorithmic level static discorery tool could take the market space in between.

The value of the work of course is not limited to a user feedback only. As presented in the previous section, rejuvenated code can be further parallelised and executed more efficiently. Performance motivation is another potential value of this work.

Outline of the 1st year activities [1]

1. **[25% - 3 months] Background study for the ongoing "Discovery" project. Startup overheads.**

   Before the first results and outputs on abstract data structures recognition can be collected, I need to get into the context of the field and get familiar with the work, which has already been done. As an example, this includes reading of papers on algorithmic skeletons and idioms (such as matrix multiplications, stencils, etc) recognition [8], [9], [10]. These works are based on the developed domain specific Compiler Analysis Description Language (CAnDL) [11]. I will need to get comfortable with the language and the tool, which implements it.

[1] All time estimates and workload percentages of stages are based on the projections of my first MSc by Research year experiences. These estimations are quite rough and are certainly going to change. Phases are going to overlap as well, and these numbers represent rather total efforts.

These papers will serve me as a starting point, from which I will go down deeper to study the foundations and the main results of surrounding fields, such as constraint programming, abstract data structures and algorithms, etc.

2. [**50% - 6 months**] **Feasibility study.**
   During this stage existent CAnDL tool will be extended with abstract data structures recognition related work. If necessary, supplementary tools and facilities will be developed.
   In this stage I plan to actually write my own CAnDL templates and constraints for different abstract data structures. I plan to start with linked-lists and gradually pass onto more complex data structures like binary and quad trees, different sorts of sets, etc.
   Small hand-written examples will be used to conduct feasibility study of ideas. Some benchmarks will be manually parallelised and checked for performance motivations.

3. [**10% - 1 month**] **Benchmarks run.**
   Once proposed and developed concepts will prove to work, a massive result collection will be conducted. I plam to use SPEC/Olden benchmarks for that purpose.

4. [**15% - 2 months**] **Analysis and results interpretation.** Once all results are collected, the data can be visualized, processed and studied in any other possible way in order to get correlations and present the results into the most suitable form.

1st year goal

The goal I am going to aim at throughout my 1st PhD year is to publish a paper "Towards Data-Centric Parallelisation" in Programming Languages Design and Implementation (PLDI) conference.

All 1st PhD year schedules and estimates are based on the experiences I got throughout my MSc by Research stage, when I mainly worked on "Software Metrics for Parallelism" topic. The skills (technical like LLVM) combined with better understanding of academic working environment, I acquired througout the first MSc by Research year, presence of publications [8], [9], [10] on the topic, as well as the fact, that abstract data structures recognition seem to be more fertile ground for an interesting research results make me feel highly optimistic about my goal.

2nd and 3nd year plans and the ultimate PhD goal

During these years the work on abstract data structures recognition and parallelisability feedback is supposed to gain a significant extension in the level of details and the amount of content. While my MSc by Reserach studied parallelisability metrics of source code loops, data structures can also be characterized by some quantitative properties (features). Machine learning techniques can also be applied here and we can try to classify types of abstract data structures with a certain precision.

Loop metrics work combined with abstract data structures discovery can shape the PhD into a final state of a static analysis tool, which can distantly resemble those of a software formal verification. The tool would be capable of giving parallel-wise advices to a programmer. Furthermore, these feedback directions can take optimization effect even for the serial hardware systems.

References

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

[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] 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] 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.

[6] NASA Parallel Benchmarks. `https://www.nas.nasa.gov/publications/npb.html`.

[7] Seoul National University NAS Parallel Benchmarks implementation. `http://aces.snu.ac.kr/software/snu-npb/`.

[8] Tobias J. K. Edler von Koch, Stanislav Manilov, Christos Vasiladiotis, Murray Cole, and Björn Franke. Towards a compiler analysis for parallel algorithmic skeletons. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 174–184, New York, NY, USA, 2018. ACM.

[9] Philip Ginsbach and Michael F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[10] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous apis: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 139–153, New York, NY, USA, 2018. ACM.

[11] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. Candl: A domain specific language for compiler analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 151–162, New York, NY, USA, 2018. ACM.