Pattern-Based Detection and Utilization of Potential Parallelism in Software Systems

Christian Wulf

Department of Computer Science
Kiel University
D-24118 Kiel
chw@informatik.uni-kiel.de

Abstract: Due to the paradigm shift from single-core to multi-core processors within the last ten years, software engineers not only need to have technical and domain-specific knowledge to add new features and solve any bugs. They also need to have knowledge about concurrency issues, e.g., to meet performance requirements.

Since introducing concurrency in existing software systems is often error-prone and difficult, we propose a semi-automatic, pattern-based approach to support the software engineer in the parallelization process and related concurrency tasks. We propose the use of patterns for both the detection of code regions with high potential of parallelism and for the corresponding parallel version utilizing information gathered by static and dynamic analysis. Besides describing the approach itself, we focus on our goals and research questions, and illustrate ideas on how to conduct a meaningful evaluation.

1 Introduction

Since processor performance cannot be improved anymore by increasing the clock frequency, many parallelization approaches have been proposed. For instace, parallel compilers [HAM+05, e.g.] or recommendation systems [MCGP07, e.g.] use the given structure of a software system either to detect parallelization potential or even to utilize such potential resulting in a parallel execution.

However, they often do not restructure the original source code by breaking dependencies to exploit further parallelization potential [URT11]. Moreover, fully automatic approaches need to over-approximate dependencies that are unknown or indeterminable at compile-time. Although semi-automatic approaches overcome this drawback with the help of dynamic analysis, all existing approaches require an parallelization expert instead of a general software engineer.

We present a semi-automatic parallelization approach for non-expert software engineers that provides solutions to the problems described above. Our approach allows to iteratively introduce parallelization by applying a pattern-matching restructur-

ing technique on the system dependency graph[1] of the given software system. In this paper, we focus on our goals, research questions, and the planned evaluation.

Structure of this paper: In Section 2, we describe the goals and research questions of our approach. Afterwards in Section 3, we present our approach. Finally, we present our planned evaluation in Section 4.

2 Goals and Research Questions

We envision a pattern-based, semi-automatic parallelization approach as solution to systematically guide and support the non-expert software engineer (in the following called *user*) in the parallelization process[2] without sacrificing flexibility and speedup for the sake of abstraction. This section provides an overview of the goals and research questions of the planned PhD thesis.

G1: Systematic Guidance and Support in the Parallelization Process We see a need for a systematic parallelization approach to guide and support the user in all the five phases in the parallelization process[2]: discovery, planning, transformation, code generation, and runtime management. *Q1: To what extent can we systematically guide and support the user in each of the five parallelization phases?*

G2: Hide Concurrency-Specific Aspects from the User Optimally, the approach should be executed automatically. If this is not possible, it should hide most of the concurrency-specific aspects from the user, e.g., the correct implementation of synchronization, to focus on the issues that are not automatically decidable.[3] *Q2: To what extent can we hide concurrency-specific aspects from the user?*

G3: Structure- and Language Independence Our approach should be able to parallelize any software system that can be represented as a system dependency graph, e.g., object-oriented software systems. In particular, it may not be tailored to one specific control or data structure, but should be open for all possible constructs. Furthermore, it should provide support for fine-grained as well as coarse-grained introduction of parallelism. *Q3: How to encapsulate structure- and language dependent information to provide a general parallelization concept?*

G4: Extensibility Our approach should be extensible to improve and enrich its parallelization phases with new insights from the research area. In particular, it should support adding new patterns at arbitrary levels of granularity without writing a single line of code. *Q4: How to achieve extensibility in each step?*

G5: Parallelism Finally, our approach should parallelize software systems to increase their performance. *Q5: To what extent can our approach parallelize software systems?*

[1] A system dependency graph represents a software system by nodes and edges where nodes are statements and edges are control or data dependencies between those statements.

[2] See [GJLT11] for the taxonomy of the five parallelization phases

[3] One example is when a code section is not parallelizable for all, but only for particular input values that are in fact garantueed, but not directly encoded in the software system.

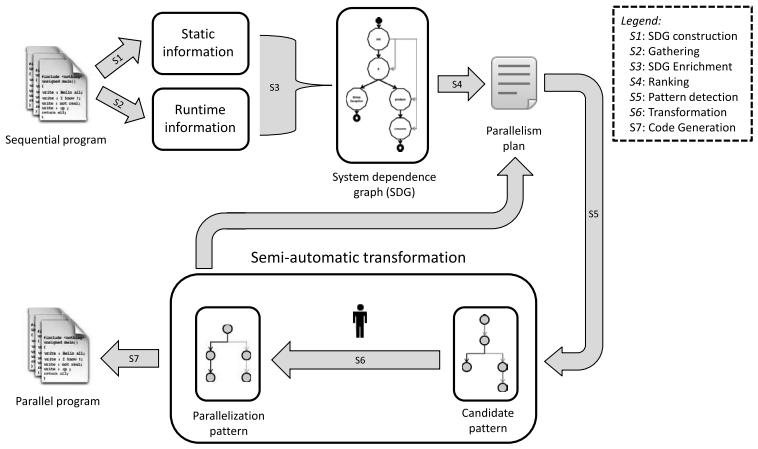

Figure 1: Overview of our semi-automatic parallelization approach

3 Approach

Our approach targets software engineers who need to parallelize existing software systems. It serves as guidance in the parallelization process and provides support for a pattern-based, iterative introduction of parallelism. Figure 1 gives an overview of the approach using seven steps to reveal and exploit parallelization potential.

The first three steps S1-S3 build a system dependency graph (SDG) representing the given software system using information gathered by static and dynamic analysis. It stores the control flow and data flow as well as further information about the structure and the runtime behavior.[4] In S4, a parallelism plan is constructed on the basis of the SDG. After construction, the plan consists of an ordered list of code sections that are most promising for a transformation to a parallel version. For example, assuming that long running methods have a higher parallelization potential, a simple plan would list all method declarations ordered by their average execution times.

The software engineer may then successively process the plan by executing the steps S5 and S6 on each code section. While S5 represents the pattern detection step to find code regions that have a high potential for parallelization, S6 constitutes the transformation from a matched instance of S5 to a semantically equivalent parallel version. For these two steps, we will provide an extensible pool of so-called candidate and corresponding parallelization patterns each represented as a dependency graph similar to the SDG. In this way, S5 and S6 can be executed automatically. However, before applying S6, the software engineer has the possibility to validate and adapt the proposed parallel version. The last step S7 is responsible for the code generation and can be executed after each iteration.

[4]For example, the type hierarchy and method execution times

Besides parallelizing loop iterations and array accesses, this approach also allows to parallelize, e.g., I/O accesses and to reveal further parallelization potential by restructuring and resolving dependencies with the help of runtime information.

4 Planned Evaluations

This section describes our planned evaluations for the goals mentioned in Section 2.

We evaluate **G1** and **G2** by implementing a prototype and conducting a questionaire survey. Our prototype will contain patterns each encapsulates as much concurrency-related knowledge as possible. We then let two professional software engineers and 30 master students parallelize several example applications (including a financial risk assessment application of a German bank). Finally, the subjects fill in a questionnaire that consists of questions about the interaction with and usability of our prototype.

We evaluate **G3** by parallelizing loop control structures, method invocations, and I/O operations with our prototype. We also implement support for Java and C# to show that our approach is not targeted at one specific programming language.

We evaluate **G4** by providing our prototype with at least two different ranking strategies for S4. Moreover, we define several candidate patterns for S5 and corresponding parallelization patterns for S6 with different levels of granularity.

We evaluate **G5** by conducting a performance evaluation of our prototype. We use several input programs from different application domains for which a manually parallelized and an unparallelized version exist. We then execute our prototype for each of the unparallelized version and measure their resulting speedups. Afterwards, we compare our performance results with those of the parallelized versions.

References

[GJLT11] S. Garcia, D. Jeon, C.M. Louie, and M.B. Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Lang. Design and Impl.*, 2011.

[HAM+05] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural Parallelization Analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27, 2005.

[MCGP07] T. Moseley, D.A. Connors, D. Grunwald, and R. Peri. Identifying Potential Parallelism via Loop-Centric Profiling. In *Proc. of the 4th Int. Conf. on Comp. Frontiers*, CF '07, pages 143–152. ACM, 2007.

[URT11] A. Udupa, K. Rajan, and W. Thies. ALTER: Exploiting Breakable Dependences for Parallelization. *SIGPLAN Notices*, 46, 2011.