Automatic Parallelization using AutoFutures

Korbinian Molitorisz, Jochen Schimmel, Frank Otto

Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
{molitorisz|schimmel|frank.otto}@kit.edu

Abstract. Practically all new computer systems are parallel. The minds of the majority of software engineers are not, and most of existing source code is still sequential. Within only a few years, multicore processors changed the system landscape, but the competence to reengineer for computer systems of today is shared among a small community of software engineers.
In this paper we present AutoFuture, an approach that automatically identifies parallelizable locations in sequential source code and reengineers them for multicore. This approach demands minimal change to sequential source code. AutoFutures make parallel code easy to understand and increase the acceptance of parallel software.

1 Introduction

Parallelization is hard. After 10 years of multicore commodity systems, the vast majority of source code still is sequential. The knowledge about multicore software engineering is still shared among a small circle of software engineers. We see the necessity to address software engineers that do not have this knowledge.

[1] and [2] show that most of the time-consuming work is encapsulated in functions or methods. Critical path analysis methods also operate on statement blocks, so any automatic parallelization concept has to consider that the highest parallelization potential lies in statement blocks.

Automatic parallelization has the potential to make multicore systems available to software engineers of all competence levels. However, the parallelization process is still very time-consuming and skill-intensive. The most promising regions have to be located, the appropriate parallelization has to be identified and the parallel code has to be checked [3, 4]. AutoFutures simplify and automate the manual parallelization process.

2 Scenario for asynchronous parallelization

To emphasize the necessity of our research we introduce a real-world scenario. Many real-world projects do not offer time for parallelization, so even semi-automatic approaches are too expensive. One way to gain performance boosts

on multicore systems are fast and fully automatic parallelization approaches. This defines the first mission of AutoFutures. At the same time, code correctness must be preserved. Because of these two constraints we can only use a small set of automatic parallelization techniques. Hence, evidently less performance gains can be achieved compared to dedicated parallelization.

AutoFutures' second mission is to achieve a broad acceptance of multicore programming in the target group. We see three flavors to accomplish this mission: Recognition, recurrence and correctness.

Recognition: As we face inexperienced engineers, we have to manage the parallelization process without their involvement. The software engineer must be able to directly recognize the parallelized code. Hence, our parallel code needs to be as similar to the original version as possible. The precondition for our parallelization concept is to be as unobtrusive as possible which naturally comes at the cost of lower speedup expectations. But to us any speedup is worthwhile when achieved without user interaction.

Recurrence: Another aspect to fulfill our mission to raise the acceptance is to use a well-defined set of recurring patterns. All parallel regions should follow a very small set of parallelization patterns. With this we lower the entry threshold among incompetent programmers. As we attain a higher recognition value we expect a rising acceptance rate.

Correctness: Parallel code that is faster but incorrect is preposterous. As this would lead to even lower acceptance rates concerning multicore programming, it is crucial to only parallelize where code correctness can be guaranteed. As a proof of concept, we implemented the pattern shown in figure 1.

3 AutoFutures: Automatic asynchronous method calls

The first goal in our concept is to have an automatic parallelization technique that is very fast in execution. We use a static analysis to identify code that can be executed asynchronously. This precondition constraints the search space for parallelization potential down to code that can verifiably be executed in parallel without any data dependencies. This could be part of a compiler.

After code hotspots have been identified, we have to address synchronization. We make use of the widely known concept Future. A Future serves as a placeholder for the result of an asynchronous computation. Futures offer an easy way to specify asynchronicity and hide synchronization code.

Without a Future synchronization code has to be added at the end of concurrent activities. This breaks code readability and violates to be as unobtrusive as possible. Furthermore, Futures hide whether the result of a computation is already available or not. Figure 1 shows a simple example: Two consecutive methods *solve*() and *statements*() operate on different objects and do not have data dependencies. We suggest to transform the invocation of *solve*() using a Future. *statements*() is then executed in parallel to *solve*(). The variable x in the call to *print*() is an automatically added synchronization point, as the result of *solve*() has to be available here.

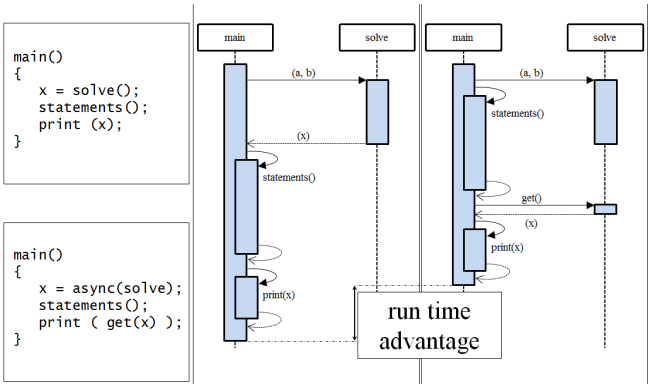

Fig. 1. Asynchronous method invocation

4 First results

As a first experimental approach we implemented AutoFutures in Java. We base on the object-oriented paradigm so AutoFutures automatically convert synchronous method invocations to asynchronous Futures and insert synchronization points. Our implementation currently searches for the consecutive method invocation pattern shown in figure 1. We used the Soot framework [5] for the static analysis and developed two heuristics to detect the synchronization point.

Our key finding is that with AutoFutures we can detect parallel potential in sequential code and re-engineer it to parallel code with only little code changes. We evaluated our concept in 5 real-world applications and could achieve an average speedup of 1.81 (min: 0.76, max: 3.34) on an Intel Core2-Quad machine. In table 1 we present our results. In order for the parallel version of ImageJ to function properly, the store procedure had to be altered manually. PMD and ANTLR already run in parallel. We used the DaCapo-benchmark suite [6] and with AutoFutures we could almost reach the manual parallelization performance.

	MergeSort	Matrix	PMD	ANTLR	ImageJ
Source lines	34	81	44782	36733	93899
Methods	3	5	3508	1998	4505
Input data	array	matrix	rules	files	images
Speedup min	2.16	2.61	—	—	1.74
Input size	1.600.000	400x400	—	—	512x512
Speedup max	2.70	3.34	0.91	0.76	2.04
Input size	8.000.000	600x600	13	18	1448x1448

Table 1. Evaluation results for AutoFutures

5 Perspectives

We feel confirmed to investigate further patterns for the application of AutoFutures. Currently we conduct an empirical study to manually detect additional patterns in sequential source code by selecting programs from different application domains to explore applicability in different scenarios. This leads to an additional aspect: Efficiency. Not every parallelizable construct should effectively be parallelized. With the pattern approach we try to distinguish parallelization potential.

Our results so far reveal three additional patterns: Loop parallelization with and without in-order-execution, method extraction from blocks of statements and speculative value calculation as used in instruction-level parallelism. We consider extending our analysis method to include runtime information, as our static analysis leads to a very small search space. With the extension to include runtime information in our analysis the identification of design patterns comes into reach: We see the chance to extend the AutoFuture concept for the automatic detection of Master/Worker from sequential source code. The potential, limitations and tradeoffs for this approach need further research.

6 Conclusion

Although multicore systems are omnipresent software engineers are still afraid of parallelism. With systems becoming more and more complex we must address software engineers that don't have any knowledge about parallelization and don't have the time to learn.

In this paper we introduced the concept AutoFuture that enables a fully automatic code refactoring to help software engineers to familiarize with multicore software engineering. We argue that with AutoFutures the acceptance rate for multicore software engineering can be raised. Additionally, we see the potential to establish a parallelization process with variable granularity levels. For this reason it is possible to also address competent software engineers, provide them with more information and offer them higher speedup potentials. The free lunch might be over, but free snacks are still available.

References

1. S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32nd PLDI*, 2011.
2. W. C. Benton, "Fast, effective program analysis for object-level parallelism," Ph.D. dissertation, University of Wisconsin at Madison, 2008.
3. G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th PACT*, 2010.
4. C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *Proceedings of the 2nd IWMSE*, 2009.
5. [Online]. Available: http://www.sable.mcgill.ca/soot/
6. [Online]. Available: http://dacapobench.org/