Invasive Interactive Parallelization

Mikhail Chalabine
Programming Environments Laboratory
Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
mikch@ida.liu.se

ABSTRACT

In this thesis proposal we suggest an interactive method of parallelizing legacy software that is based on separation of concerns and aspect weaving. We statically inject parallelizing code into sequential cores by means of reusable rewrite rules. We view parallelization as a generic refactoring process with programmable control automated by inferences on structured parallelization categories.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent Programming—*Parallel programming*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program modification, and program transformation*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancemen—*Restructuring, reverse engineering, and reengineering*

Keywords

Performance, multicore, processor, reasoning

1. INTRODUCTION

Over a number of years parallelism has become an indispensable tool in physics, chemistry and other areas of science and business. Shared-memory machines and distributed-memory clusters of various capacity are widely employed today for parallel execution of heavy computations and simulations. The underlying technology is developing fast. Together with powerful clusters we see new multi-core chips emerging on the market [16]. This new paradigm places parallelism inside an individual processor and, thus, inside ordinary of-the-shelf computers. The number of users of parallelism and the spectrum of its application are growing.

Despite this development, programming for a parallel computer remains a prerogative of a few. Today it is the same individual, *e.g.*, a physicist, who writes both the core and the parallelizing code. From her perspective parallelization is a burden. It obscures her algorithms, complicates understanding and creates difficulties in the development process.

In our work we start from the fact that any parallelized application can be viewed as consisting of two parts: a serial core and parallelizing statements. We intend to separate the job of a *domain expert* who develops and maintains the core from the job of a *parallelization expert* who prepares it for execution on a parallel machine.

In our method we view the result of parallelization as a property of a program – a property that does not change the semantics (output) of an algorithm being parallelized. In our hypothesis we investigate the possibility of injecting parallelizing code into sequential cores. We term this injection as *weaving* or *composition* similar to Aspect-oriented Programming (AOP) [15] and Invasive Software Composition (ISC) [2].

The rest of this proposal is organized as follows. In Section 2 we review parallelization techniques existing today. In Section 3 we summarize the challenges that still need to be addressed. In Section 4 we set the research question and in Section 5 we discuss our solution to it. Section 6 discusses expected results, Section 7 presents methodology and Section 8 reveals the goals of the evaluation.

2. RELATED WORK

Today parallelization can be performed in four different ways. The most widely accepted method is *manual refactoring*. Manual interference assumes absolute freedom in reengineering at the level of program statements. While it is often the only way to reach acceptable performance levels, manual parallelization remains the most complex approach for both parallelization and domain experts. The process cannot be automatized, no automated assistance in coding can be received, and the systematic reuse of parallelizing solutions is limited. Furthermore, the life cycle of the core application is terminated as its parallelization requires two separate branches: one for shared- and one for distributed-memory platforms. This causes chaos in maintenance.

An alternative is *automatic parallelization* by the compiler – either its classical fully-automated version or an extension based on abstract interpretation and program comprehension. A parallelizing compiler is good for the domain expert as it keeps her core application intact. Its capabilities are limited, however, by the complexity of the underlying pat-

tern matching and static analysis [10]. In the classical version, therefore, the compiler will conservatively process only a subset of parallelizable data structures, such as, nested loops with statically analyzable data dependencies [10, 13, 20]. Program comprehension helps to extend this set by enabling domain-specific pattern recognition through, *e.g.*, pluggable pattern-identifying productions [9]. Although the method is shown effective in certain structured domains, such as, scientific computing [9], pattern-matching remains limited given unstructured (sparse) patterns whose identification and refactoring must be manually encoded off-line; no interactive navigation is possible.

The third alternative is re-designing the core using, so-called, parallel components or *skeletons* [8]. The method has two serious drawbacks. First, it requires excessive manual refactoring, and, second, skeletons limit the parallelization since they cover only well-structured parallelism [8, 17]. Our case studies show, however, that in many domains with a strong demand for parallelism (telecommunication industry, for instance) we often deal with unstructured parallelism or parallelism that cannot be expressed in terms of skeletons, such as, producer-consumer coordination [5, 7]. Note also that consolidating performance over a composition of two or more optimally parallelized components is not always possible as skeletons do black-box composition of parallelism only. This problem calls for an automated adaptation mechanism that depending on the composition environment changes individual parallel components generating effectively parallelized units (*i.e.*, it supports gray-box composition [2]). Such a mechanism is not available today.

The remaining option is to use one of the available interactive parallelization tools, such as, [1, 12, 14]. These turn to the expert's knowledge who directs the refactoring in situations where the capabilities of a classical parallelizing compiler do not suffice. To our knowledge, the interaction in these tools is triggered only when an automatically identified pattern cannot be entirely understood by static analysis. They do not allow to pinpoint a pattern and to select an action to apply to it. Furtermore, some of these systems, such as, [1] allow to store and to replay parallelization history, but even they lack reasoning that builds and uses causal relations between actions such a history contains and between sparsely-distributed code fragments it affects.

3. CHALLENGES IN CONTEMPORARY PARALLELIZATION

We conclude that one of the major challenges in modern parallelization consist in detaching the complexity due to the original core from the complexity due to the parallelizing code. Such separation directly facilitates the work of a domain expert. In the ideal case it permits free manipulations in the core independent of the parallelization executed in the background. Automated refactoring by the compiler remains the only technique to approach this today.

Simplifying the parallelization process, assisting the parallelization expert with decisions and allowing of more qualitative reuse of parallelizing solutions are yet other significant challenges in the development perspective. A good tool should help the programmer to build efficient implementation of effective parallelization strategies. *Efficiency* here embraces optimizations in the parallelizing code while *effectiveness* deals with modifications at the algorithm level including those affecting the core. Assume, for instance, that substituting a pair of $D - C_1 - G$ *and* $D - C_2 - G$ patterns by a single $D - \{C_1; C_2\} - G$ improves efficiency by reducing communication overhead.[1] An effective strategy goes beyond this optimization and replaces $D - \{C_1; C_2\} - G$ by a more effective parallel pattern $D' - C' - G'$.

A great challenge with this respect is the development of mechanisms of abstract interpretation and inference that allow of optimizations at a more abstract level than the local control- and data-flow analysis or domain- and structure-specific pattern matching. At this new level each block being parallelized should be causally coupled with other blocks in the program triggering proactive and reactive adaptation in the parallelization process.

We may conclude at this point that today's manual and to some extent existing interactive parallelization are the only methods allowing of both efficient and effective parallelism. They, nevertheless, expose the domain expert to the joint complexity of the core and the parallelizing code. Neither manual nor semi-automatic parallelization have mechanisms that assist the parallelization expert and allow of interactive identification of parallelizable fragments while the fully-automated refactoring by the compiler remains limited.

4. RESEARCH QUESTION

We investigate, therefore, whether a new unifying model of parallel programming can be developed where: (P_1) the programmer receives automatically inferred assistance in selecting parallelization strategies; (P_2) serial code is preserved in its original form abstracted from the parallelizing statements; (P_3) maintenance of the core and the parallelizing code are isolated and require different expertise; (P_4) parallelization is turned into an interactive semi-automated on-demand process easy to activate and de-activate; (P_5) parallelization becomes adaptable, *i.e.*, supports an easy transition between parallelizing for shared- and distributed-memory platforms; (P_6) parallelizing code is kept in a form where larger blocks of it can be reused. With this in mind we define the following hypothesis:

HYPOTHESIS. Although applying AOP to parallelization is recognized as a difficult task, see, *e.g.*, [11, 18], AOP (or equally ISC) together with the theory of defaults and defeasible reasoning studied in Artificial Intelligence and Knowledge Representation [3] form an environment sufficient for the development of systems for on-demand weaving of parallelism that supports a number of useful features missing in the existing methods. In particular, a system built on top of these technologies in a way sketched in Section 5 supports the properties (P_1) through (P_6) outlined above, addresses the challenges stated in Section 3 and while being capable of all the major parallelizing refactoring operations found in the contemporary methods discussed in Section 2 simplifies the job of both parallelization and domain experts.

5. INVASIVE INTERACTIVE PARALLELIZATION

[1]Data distribution, computations, data gathering.

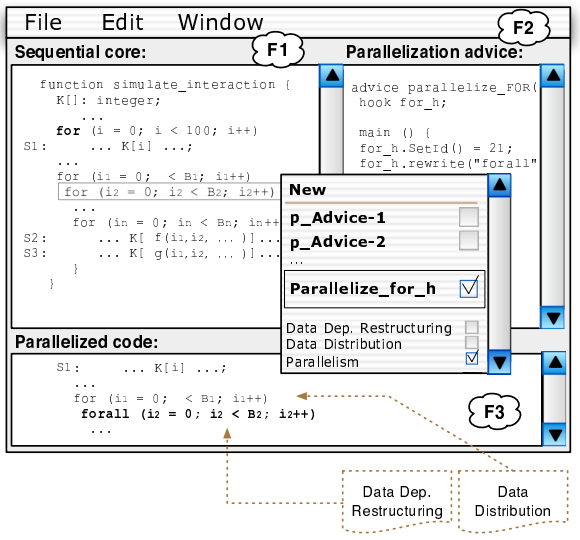

Figure 1: The main interface with a `for` header being right-mouse clicked in the F_1 frame.

To test our hypothesis we develop Invasive Interactive Parallelization (IIP) model where sequential sources are incrementally prepared for execution on a parallel machine [4]. IIP is aimed as an automated mechanism that isolates the development of the core from the development of the parallelizing code. An expert leads the process and interactively receives intelligent (automatically inferred) support in terms of feasible parallelization strategies. IIP extends a statement, basic block, function, class, fragment, component or any other program unit with special functionality that is separable from the core (plug-and-unplug) and composable at (interactively) defined program points, called *hooks*. Thus, we talk about a programmable unit extended with a *parallelization property* which has a hierarchical structure and is composed out of, so-called, parallelization concerns [5].

5.1 A blueprint of the system

An IIP system that we develop consists of three components: an *interaction engine*, a *parallelization engine* and a *weaving engine*. The interaction engine implements dialog functionality. It reacts to user actions and navigates her in the parallelization process. For example, given the user creates a data distribution aspect [5] the engine must remind of data gathering, synchronization or any other causally related parallelization concern. The engine also protects the code affected by the parallelization and closed for direct editing. It should also allow of marking (sparsely distributed) code fragments and relating them to instances of known parallelizable problems that match in a deliberate type system.

The parallelization engine is a reasoning element in the system. Its main task is to infer strategies and control causal dependencies within the aspect hierarchy [5]. In particular, it regulates interferences in the set of aspects and optimizes their compositions. It also sets restrictions on code, decides on parallelizability, and infers recommendations.

The weaving engine carries the refactoring process. It matches patterns rewrites, inserts and deletes code. The weaver operates on an Intermediate Representation (IR) of a program, *e.g.*, a Concrete Syntax Tree [6].

5.2 Parallelization process

Serial core is displayed in one of the three frames comprising the user interface – F_1 in Figure 1. Initially the core is subjected to an extended automated parallelization procedure where the system builds its intermediate representation, analyses and refactors it. Beyond these canonical operations the system assigns unique identifiers to IR nodes and records their relative positions as $(file, line, column)$ triplets allowing for automated round-trip software engineering [6]. It also highlights parallelizable code segments, as exemplified by the first `for`-header token shown bold in F_1.

The parallelization continues as the expert pinpoints further parallelizable fragments relating new hooks to sequences of rewrite rules organized as refactoring programs in parallelization aspects. Hooks are identified either interactively (on-line) or declaratively (off-line). For example, clicking the third consecutive `for` lexeme in F_1 returns a set of aspects applicable to a `for` node, among which *parallelize_for_h* rewrites a sequential `for` into a parallel `forall`, see Figure 1. The click triggers creation of a hook (`for_h`) identifying a node by its unique IR id as shown in F_2.

5.3 Automation and reasoning

Besides pattern recognition the system should automate the parallelization process. As an example note that the n-dimensional loop nest in Figure 1 represents a generally undecidable problem in automatic parallelization.[2] If an expert decides, however, that it can be parallelized then the refactoring is done automatically on the basis of what normally constitutes parallelization. This knowledge is encoded as environment-specific sequences of aspects and their compositions. In particular, parallelization of a `for` in a shared memory environment with no loop-carried data dependencies corresponds to a straightforward rewrite into a `forall`. Parallelizing loops with dependencies requires extra loop transformations or synchronization [13, 19, 20]. Parallelizing for distributed memory will further require data to be partitioned, distributed, and gathered before and after the loop body. The corresponding aspects are woven at the right positions and in the right order inferred under the following simple rule: *Aspects are woven in a default order as long as no other order is explicitly specified.*

This rule is followed in a defeasible manner, *i.e.*, allowing for exceptions given the interaction with the expert [3]. Parallelization is complete if it reaches a final state of a given aspect sequence. Computing such sequences requires new reasoning mechanisms on aspect hierarchies that we develop.

6. EXPECTED CONTRIBUTIONS

Specification of inference mechanisms for reasoning on parallelization concerns is one of the expected contributions of this work. We plan to investigate whether defeasible reasoning is a suitable technique for that. Thus, while the holy

[2]The most common problem is statically undecidable data dependencies caused, *e.g.*, by statically irresolvable access strides in loop bounds or indirect array indexing.

grail of contemporary automated parallelization is a conservative control and data flow analysis, this work will provide an insight into the usability of non-conservative defaults in automation of the parallelization process. To do that we first expect to generate a set of default rules (concretizing the general rule above) for a set of parallelizable structures, such as, nested loops.

In the area of software engineering we shall deliver a set of requirements for constructing a parallelizing weaving system that will clarify how AOP (ISC) should be extended to allow for weaving of parallelism.

In terms of technical contributions we expect to develop a prototype of an interaction engine, reasoning engine, and a weaving engine supporting a simple Pascal-like language, Diesel. We also expect to introduce and test the concept of automated roundtrip engineering in AOP systems.

7. METHODOLOGY AND RESULTS

Following a major case study of parallelizing a large telecommunication product [7] we observed that, to reduce the impact of the parallelization on the domain expert, parallelizing code can be abstracted to parallelization concerns injected into sequential cores on demand. As a formalization in [4] we suggested a general IIP model where we motivate the approach, introduce the related techniques, including [2], and propose the structure of a weaving engine. As a step toward typed weaving we narrowed the set of parallelization concerns and structured them in a concern model described in [5]. We observed that aspects implementing parallelization concerns strongly depend on the underlying environment while abstract concerns do not. We use this fact to encode the parallelization process in terms of sequences of concern applications. As the most recent step we introduced automated roundtrip engineering to be included in the final IIP system prototype [6]. This prototype constructed in a way discussed in Section 5 will help to answer whether the set of chosen technologies suffices. It will also be the means to demonstrate that IIP is capable of transformations supported by the existing approaches. This work will be pursued in a number of case studies to be individually selected for each of the four techniques presented in Section 2. Results will be shared as a collection of conference papers.

8. EVALUATION

The extent to which the technologies mentioned in the hypothesis suffice will be evaluated by the level of difficulties during the development of the prototype. The claim that the approach simplifies the job of both experts will be evaluated in case studies with the following focus. First of all, we are interested in the usability of the defaults. In this respect we shall record the amount of inferred assistance a programmer receives on average together with its subjective usefulness. Quantitatively we shall record how parallelization using non-conservative defaults deviates from the classical conservative one. As another metric we shall study the degree of correctness of the recommendations based on the default rules. This will be further supported by a subjective qualitative metric reflecting the degree to which the parallelization is simplified (whether it is easier to program parallelism). This will be measured in terms of the time to parallelize a problem including the time to switch between parallelization for shared and distributed memory. Finally, the remaining qualitative metric will measure the outcome of the parallelization in terms of the performance of the emitted code.

ACKNOWLEDGMENTS: Christoph Kessler, CUGS, RISE.

9. REFERENCES

[1] Parawise widening accessibility to efficient and scalable parallel code. *Parallel Software Products (PSP), White Paper*, 2004.

[2] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, Berlin, Germany, 2003.

[3] R. Brachman and H. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004.

[4] M. Chalabine and C. Kessler. Parallelisation of sequential programs by invasive composition and aspect weaving. In *Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies (APPT'05)*. LNCS, 2005.

[5] M. Chalabine and C. Kessler. Crosscutting concerns in parallelization by invasive software composition and aspect weaving. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS'39)*. IEEE, 2006.

[6] M. Chalabine, C. Kessler, and P. Bunus. Automatic round-trip software engineering in aspect weaving systems. In *ASE 2006*, Tokyo, Japan, 2006. (Short paper).

[7] M. Chalabine, C. Kessler, and S. Wiklund. Optimising intensive interprocess communication in a parallelised telecommunication traffic simulator. In *Proc.of the Int. High-Performance Computing Symposium, Advanced Simulation Technology Conference*, Orlando, Florida, USA, 2003. SCS.

[8] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. MIT Press, 1989.

[9] B. di Martino and C. W. Keßler. Two program comprehension tools for automatic parallelization. *IEEE Concurrency*, 8(1):37–47, 2000.

[10] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. Habilitation thesis, University of Passau, Germany, 2004.

[11] B. Harbulot and J. R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM, 2004.

[12] M. Ishihara, H. Honda, and M. Sato. Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: iPat/OMP. *IEICE Trans Inf Syst*, E89-D(2):399–407, 2006.

[13] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, San Francisco, CA, USA, 2002.

[14] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*. Springer-Verlag, 1997.

[16] D. Pham et al. The design and implementation of a first-generation cell processor. In *ISSCC 2005 Digest of Technical Papers*, pages 184–185, 2005.

[17] F. A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

[18] N. W. Winstanley. *Staged Methodologies for Parallel Programming*. PhD thesis, University of Glasgow, Sept. 2000.

[19] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1994.

[20] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM Press, New York, NY, USA, 1991.