

A Language-Independent Parallel Refactoring Framework

Christopher Brown
Kevin Hammond

School of Computer Science,
University of St Andrews,
St Andrews, UK.

{chrisb,kh}@cs.st-andrews.ac.uk

Marco Danelutto

Dept. of Computer Science, Univ.
Pisa, Largo Pontecorvo 3, 56127,
PISA, Italy.

marcod@di.unipi.it

Peter Kilpatrick

Dept. of Computer Science,
Queen's University Belfast, UK.
p.kilpatrick@qub.ac.uk

Abstract

Recent trends towards increasingly parallel computers mean that there needs to be a seismic shift in programming practice. The time is rapidly approaching when most programming will be for parallel systems. However, most programming techniques in use today are geared towards sequential, or occasionally small-scale parallel, programming. While refactoring has so far mainly been applied to sequential programs, it is our contention that refactoring can play a key role in significantly improving the programmability of parallel systems, by allowing the programmer to apply a set of well-defined transformations in order to parallelise their programs. In this paper, we describe a new *language-independent* refactoring approach that helps introduce and tune parallelism through high-level design patterns targeting a set of well-specified parallel *skeletons*. We believe this new refactoring process is the key to allowing programmers to truly start *thinking in parallel*.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Programming Environments]: Integrated Environments

General Terms Languages, Design, Performance

Keywords Refactoring, Erlang, C/C++, Skeletons, Patterns, ParaPhrase, Parallelism, Concurrency

1. Introduction

It is hard to over-state the importance that parallelism will play for future generations of programmers. For reasons of both performance and energy usage, the single-processor CPU that has dominated more than half-a-century of com-

puting is quickly becoming obsolete. Machines with dual, quad or even hexa-core CPUs are already commonplace, and plans for single-chip CPUs with more than 50 cores have been announced¹.

However, software techniques for programming such parallel systems have not caught up with this trend. Many software developers still use small-scale, high-effort methods for parallel programming such as locks and explicit threading, mechanisms that have effectively been bolted on to mainstream languages as an afterthought. What is needed is an approach that helps the programmer *think parallel* so that they can take advantage of the massive amounts of parallelism that will soon be available to them.

In this paper we present a new *software refactoring* approach that aims to increase the *programmability* of parallel systems in a language-independent way. The EU Framework-7 PARAPHRASE project² will use refactoring combined with high-level parallel design patterns that will introduce parallelism into C/C++ and Erlang programs. The paper makes three main contributions:

1. we introduce a new design for a language-independent refactoring tool;
2. we show how to use this design to introduce parallelism for both C and Erlang; and
3. we define a set of formal rewrite rules (formal in the sense that the rules are machine readable and with greater scope for reasoning about correctness) that can be used by our refactoring tool to introduce parallelism for a divide-and-conquer skeleton, in both C and Erlang.

2. The Parallel Refactoring Tool

The parallel refactoring framework introduced is language-independent. It may be specialised to work over different languages, including Erlang, C and C++. The specialisation process is performed by providing grammars and refactoring rules specific to the target language, whilst preserving

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WRT '12 June 01 2012, Rapperswil, Switzerland.

Copyright © 2011 ACM 12/06-978-1-4503-1500-5 ...\$10.00

¹ e.g. Intel's Many Integrated Core family.

² <http://www.paraphrase-ict.eu>

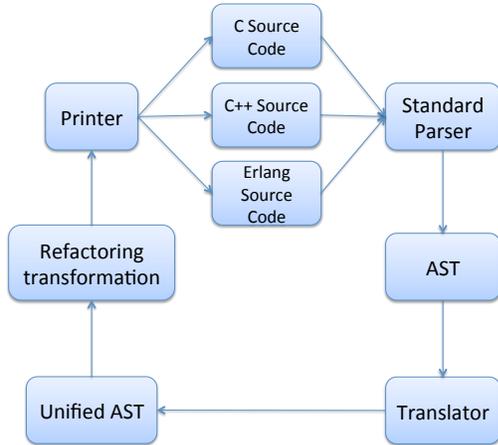


Figure 1. The Parallel Refactorer.

the correctness of the target language by user-defined pre-conditions. By targeting C/C++ and Erlang we can explore the advantages and limitations of both the imperative and the functional paradigms, whilst also contributing significantly to both user domains. Currently, however, we do not plan to support the refactoring of programs written using a mix of the possible target languages. Rather, we aim to apply similar refactoring techniques in different, language specialised versions of the framework.

Figure ?? presents an overview of the general workflow of the parallel refactorer. The user begins with his/her sequential program implemented in C/C++ or Erlang which is then *parsed* into a standard Abstract Syntax Tree (AST). The AST is then *translated* into a unified intermediary language. This unified AST is refactored by applying a set of user-driven rewrite rules which applies a formal set of transformation rules and checks for pre- and post- conditions. The refactored intermediary language is then *pretty printed* into the program source language. We envisage that the rewrite rules for the refactorings will work over the unified core language, with explicit rules for informing the refactoring system how to translate the refactored source back into the source language. The refactorings will be modelled on a set of well-defined parallel patterns, which model parallelism in a high-level abstract way.

3. Language Independent Refactoring

In this section we explore a classical *Divide and Conquer* skeleton and show how it is possible to use refactoring to introduce parallelism (we use divide and conquer here for its familiarity and simplicity). The skeleton itself is shown at the language-independent level in the form of pseudo code, which we intend to be similar to our unified intermediary language. The skeleton code for divide and conquer is shown in Algorithm ?? (where the highlighted parts reflect the user highlighted code in the native program source) as a normalised intermediate representation.

Algorithm 1 A Classical Divide and Conquer Skeleton

```

fun dnc ( p )
  if base_case ( p )
    then return solve ( p )
  else
    (p1, p2 ) = divide ( p )
    (r1, r2 ) = (dnc p1, dnc p2)
    combine (r1, r2)
  end if

```

Algorithm 2 A Classical Divide and Conquer Skeleton with Introduced Parallelism via Refactoring

```

fun dnc ( p )
  if base_case ( p )
    then return solve ( p )
  else
    (p1, p2 ) = divide ( p )
    emitPara ( dnc, p1, 1 )
    emitPara ( dnc, p2, 2 )
    c1 = collectPara ( 1 )
    c2 = collectPara ( 2 )
    combine ( c1, c2 )
  end if

```

In the algorithm, divide and conquer is defined in terms of a function, `dnc`, which takes a computation to be solved, `p`. Typically, in a divide and conquer, we test to see if a straightforward base case is met, to determine whether or not to solve the computation directly. Otherwise, we divide the computation into a number of sub-components, (p_1, \dots, p_n) . In our example, we show just two sub-components to illustrate the basic principles of divide and conquer. We divide and conquer each sub-component, and then combine the results. We show a possible refactored version of the divide and conquer in Algorithm ?. In the algorithm, we have replaced $(r_1, r_2) = (dnc\ p_1, dnc\ p_2)$ with code that first *introduces* (or *emits*) the tasks `dnc p1` and `dnc p2` as *parallel* tasks. The results of these parallel sub-tasks are also *collected*. This divide and conquer skeleton is very similar to a *map reduce*, and the same reasoning could also be applied to a map reduce skeleton. The refactored AST contains placeholders, `emitPara` and `collectResult` that are further transformed when the unified language is pretty printed, using the rules defined in Figure ?, therefore generating the source-level refactorings automatically.

Figure ? shows the rule to introduce parallelism for a set of tasks. Each rule describes a possible refactoring as a function that is applied to a node of the AST and corresponds to traversals of the AST (rules are generally top-down, unless otherwise stated). We define our refactoring

$$\begin{aligned}
&IntroEmitCollect(\rho, \mathbf{f}, \mathbf{g}) = \\
&\mathcal{E}[(r_1, \dots, r_n) = (\mathbf{f} p_1, \dots, \mathbf{f} p_n)] \Rightarrow \\
&\quad \llbracket \text{emitPara}(\mathbf{f}, p_1, id_1); \dots; \text{emitPara}(\mathbf{f}, p_n, id_n); \\
&\quad c_1 = \text{collectPara}(id_1); \dots; c_n = \text{collectPara}(id_n) \rrbracket \\
&\quad \{c_1, \dots, c_n \text{ fresh}, p_1, \dots, p_n \in \rho, \\
&\quad id_1, \dots, id_n = \text{generateIds}, \text{pure}(\mathbf{f})\} \\
&\triangleright \\
&\mathcal{E}[\llbracket \mathbf{g}(r_1, \dots, r_n) \rrbracket] \Rightarrow \llbracket \mathbf{g}(c_1, \dots, c_n) \rrbracket
\end{aligned} \tag{1}$$

Figure 2. Introduce Emit and Collect for Parallelism

function:

$$Refactoring(x_0, \dots, x_n) = \{Rule \times \{Condition\}\}$$

where x_0, \dots, x_n are the arguments to the refactoring. The rewrite rules are defined as functions over types of nodes in the AST:

$$\mathcal{E}[\llbracket \cdot \rrbracket] :: Expr \rightarrow Expr$$

Each rewrite rule has its own set of conditions which are enclosed by $\{\}$. If a condition fails for a rewrite rule, then subsequent rules to be applied will also fail automatically. Sequencing states that the rules should be applied in a strict sequence. It is denoted by $a \triangleright b$ where rule a is applied first and then rule b . Code syntax is separated from the rule semantics by quasi quotes, so that $\llbracket \mathbf{f} = \mathbf{e} \rrbracket$ denotes a function in the AST of the form $\mathbf{f} = \mathbf{e}$. The rewrite rules are described in much more detail in [?]. We envisage that, in the scope of the PARAPHRASE project, this rewrite language would form the basis of a language-independent abstract DSL for expressing refactorings, together with their appropriate conditions and transformation rules.

In Figure ??, we introduce a new rule, called *IntroEmitCollect*, which takes an environment (ρ); a function (\mathbf{f}) that will be computed in parallel; and a combine function (\mathbf{g}). We can read the rewrite as two traversals over an AST in sequence. The first traversal matches the expression between the quasi quotes $\llbracket (r_1, \dots, r_n) = (\mathbf{f} p_1, \dots, \mathbf{f} p_n) \rrbracket$, and transforms it into a sequence of applications of `emitPara` (passing in the function to be parallelised, \mathbf{f} , and any arguments, followed by an identifier tag). The pre-conditions to this transformation are that the new variables (c_1, \dots, c_n) are fresh, the arguments (p_1, \dots, p_n) are in scope, and also that the function \mathbf{f} is side-effect free (pure). We also generate the ids to use as the channels or process ids. This id generation is implemented at the source language level. This is then followed by a sequence of `collectPara` applications, which receive messages from the corresponding parallel process on the node or channel with id_n . If this transformation succeeds, then the next rule is applied, which transforms the function application previously acting on (r_1, \dots, r_n) (this function is passed as an argument, and corresponds to the combine function in Algorithm ??) into a function over the results of the calls of `collectPara`.

$$\begin{aligned}
&IntroParEmit(\rho) = \\
&\mathcal{E}[\llbracket \text{emitPara} \mathbf{f} \mathbf{x} \mathbf{n} \rrbracket] \Rightarrow \\
&\quad Erlang : \llbracket \text{spawn}(\text{?MODULE}, \text{worker}, [\mathbf{f}, \text{Self}(), \mathbf{x}]), \llbracket \cdot \rrbracket \rrbracket \{\} \\
&\quad \triangleright \\
&\quad \mathcal{D}[\llbracket \text{decls} \rrbracket] \Rightarrow \llbracket \text{worker}(\mathbf{F}, \mathbf{Args}, \mathbf{P})\{\mathbf{P} ! \mathbf{F}(\mathbf{Args})\}, \text{decls} \rrbracket \\
&\quad \mathbf{C} : \llbracket \text{MPI.Send}(\&\mathbf{x}, 1, \text{type}, i + 1, i, \text{MPI_COMM_WORLD}); \rrbracket \\
&\quad \quad \{i = \text{modulo}(\mathbf{n}, \text{no. of workers}), \text{type} = \text{typeof}(\mathbf{x})\} \\
&IntroParCollect(\rho) = \\
&\mathcal{E}[\llbracket \mathbf{x} = \text{collectPara} \mathbf{n} \rrbracket] \Rightarrow \\
&\quad Erlang : \llbracket \text{receive} \mathbf{R} \rightarrow \mathbf{R} \text{ end}, \llbracket \mathbf{R} \text{ fresh} \rrbracket \rrbracket \\
&\quad \triangleright \\
&\quad \mathcal{E}[\llbracket \mathbf{x} \rrbracket] \Rightarrow \llbracket \mathbf{R} \rrbracket \{\} \\
&\quad \mathbf{C} : \llbracket \text{MPI.Recv}(\&\mathbf{x}, 1, \text{type}, \text{MPI_ANY_SOURCE}, \text{MPI_ANY_TAG}, \\
&\quad \quad \text{MPI_COMM_WORLD}, \&\text{status}); \rrbracket \\
&\quad \quad \{\text{status} \in \rho, \text{type} = \text{typeof}(\mathbf{x})\}
\end{aligned} \tag{2}$$

Figure 3. Introduce Emitter and Collector for Erlang and C

In Figure ?? we give an example for the rewrite rules for Erlang and C. The rules match over expressions of the unified language, and instruct the refactoring tool how to convert between the refactored intermediary language and the source language. For example, for introducing the emitters, in Erlang we simply introduce calls to `spawn`. In C, using MPI, we need to do something more complex, but in essence the basic principle is to set up a number of worker threads, and then pass messages to them to act on a particular task. We shorten the rewrite rules in this position paper to give the reader an idea of the basic principles of using rewrite rules to transform the *intermediary* level into the *source* level.

3.1 Discussion

The proposed approach is intended to separate the concerns between the *application* programmer, who identifies the portions of the code amenable to parallelisation and refactoring; and the *systems* programmer, who encapsulates the parallelism by defining rewrite rules to allow the construction of parallel skeletons.

Although we intend to deal with language-independence, there are a number of potential pitfalls that we will need to consider carefully. The semantics of each language will have to be carefully reasoned about in both the intermediary language and in the rewrite rules. It may also be necessary to capture static semantic information such as scope of variables and types. For example, some languages like C++ have very specific scoping semantics, where destructors are called when variables go out of scope. The tool-user also needs to be careful to ensure that the components that are parallelised preserve functional correctness, i.e. that the portions of code identified by the application programmer are side-effect free. This requires that the program source be organised into a well-defined system of components before

refactoring, where each component has a documented list of inputs and outputs. Preserving the layout of the refactored program will also have to be considered, so that the refactored output of the tool closely matches the programming style of the user (this is particularly important so that the tool does not produce code that the user suddenly finds incomprehensible). Furthermore, different languages have different ways in which parallelism is modelled, and this will also have to be taken into account in the intermediary language and in the rewrite rules.

4. Related Work

Program transformation has a long history with Mens and Tourwé producing an extensive survey of refactoring tools and techniques in 2004 [?]. There has been a limited amount of work in parallel refactoring in general, mostly with loop parallelisation in Fortran [?]. In the Java community, there has been a recent trend for refactoring for parallelism. In particular, in [?], Dig *et al.* introduce three new refactorings for introducing concurrency into Java programs, including refactorings for divide and conquer. In [?], Radoi, et al. introduce new refactorings for loop parallelism. The recent work of Kjolstad *et al.* [?] and Schäfer *et al.* [?] investigate the complexity of thread safety issues when refactoring parallel Java programs. However these approaches are limited to concrete structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites. Paraforming [?] is a technique by Brown *et al.* aimed at exploring the feasibility of using refactoring techniques to form Parallel Haskell programs by presenting a number of refactorings to parallelise and tune data and task parallelism. O'Donnell and Rüniger derived a formal methodology for deriving parallel programs using transformation steps with a family of parallel abstract machines [?]. Here the idea is that any parallel programming language provides its own unique model of parallelism, where the programmer writes the parallelism directly in that language. Abstract machines provide a way of deriving the program in a sequence of steps, where each step is based on a specific model of parallelism, called an abstract parallel machine. However this technique is not user-driven and can only be applied to very particular sets of problems, unlike our approach that will apply to a wide range of parallel problems. Hammond *et al.* [?] gives a more detailed overview on refactoring and skeleton techniques for parallel programming. Finally, the application and derivation of the divide and conquer skeleton has been researched in detail, with key work by Smith [?] on applying divide and conquer to specific numerical problems. The CYPRESS system [?] also deals with the problem of deriving instances of divide and conquer for quicksort and mergesort examples.

5. Conclusions and Future Work

This paper has described a new prototype parallel (language-independent) refactoring tool for introducing—and tuning—

parallelism into systems of software components. We envisage the tool supporting the refactoring of programs written in languages such as C/C++ and Erlang (although we do not expect our tool to be limited to just these: other desirable target languages include Haskell and Python), by applying a set of well-defined high-level rewrite rules that introduce new patterns. We illustrated our tool design using a simple divide and conquer skeleton implementation, showing how we can model the refactoring process in a *language independent* way, using a unified intermediary language and formal rewrite rules to describe the refactorings. The refactoring tools will eventually allow for the introduction of pre-defined parallel patterns as calls to existing algorithmic skeleton libraries. Parallelism can then be exploited by:

1. the application programmer, who identifies the portions of code submitted to the refactorings via well-known parallel design patterns. The refactoring tools consequently introduce calls to existing skeleton library entries; and
2. the system programmer, who programs the algorithmic skeleton library entries according to the most efficient *state-of-the-art* techniques, again via refactoring.

We expect the work described here will continue in a number of directions:

1. we expect to implement a number of refactorings for Erlang, implemented for the Wrangler tool [?] that will introduce and tune parallelism by introducing parallel skeletons via refactoring techniques;
2. we will implement a new language-independent refactoring tool that will initially support C/C++ and Erlang based on the framework described in this paper; and
3. we will develop a new framework that will allow users to describe refactorings using a formal rewrite rule DSL that will be language-independent;

Refactoring tool support gives many advantages to the programmer for introducing and tuning parallel programs: it can *guide* the programmer through the necessary steps required to introduce parallelism; it can *simplify* the process by automating a large bulk of the boilerplate detail (such as MPI code in C); it can *warn* the user against potential common pitfalls and in general can promote a structured approach to writing parallel programs. Moreover, with additional tools such as QuickCheck [?] for Erlang, it is possible to prove functional correctness of the refactored parallel programs automatically, as part of the refactoring process. This is an *enormous* saving in effort, and we envisage such tool support will dramatically increase the *programmability* of such systems.

It is already apparent that each language has its own advantages and limitations, and as new multi-core hardware is emerging we need multiple parallel programming models to program them effectively. Having a *one stop shop* offering programmers appropriate tool support to help them build

parallel programs *no matter which paradigm they choose* has huge potential for helping programmers *think in parallel*.

Acknowledgments

This work has been supported by the European Union grants RII3-CT-2005- 026133 SCIENCE: Symbolic Computing Infrastructure in Europe, IST-2010- 248828 ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, and IST-2011-288570 ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP: High Performance Computational Algebra. We would also like to thank the anonymous reviewers for their invaluable comments.

References

- [] T. Arts, L. M. Castro, and J. Hughes. Testing Erlang Data Types with Quviq QuickCheck. In *Proceedings of 7th ACM SIGPLAN Erlang Workshop (Erlang'08)*. Victoria, British Columbia, Canada. September 27, 2008., pages 1–8. Association for Computing Machinery (ACM), September 2008. ISBN 978-1-60558-065-4.
- [] C. Brown, H. Loidl, and K. Hammond. Paraforming: Forming Parallel Haskell Programs using Novel Refactoring Techniques. In *Twelfth Symposium on Trends in Functional Programming*, Madrid, Spain, May 2011.
- [] D. Dig, J. Marrero, and M. D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *ICSE'09, Proceedings of the 31st International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.
- [] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: Refactoring for Loop Parallelism in Java. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 793–794, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640018. URL <http://doi.acm.org/10.1145/1639950.1640018>.
- [] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danellutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, N. T., and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. FMCO 2012. Submitted, February 2012.
- [] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for class immutability. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985803. URL <http://doi.acm.org/10.1145/1985793.1985803>.
- [] H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. D. Penta and L. Moonen, editors, *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE, September 2006. ISBN 0-7695-2353-6.
- [] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004. ISSN 0098-5589.
- [] J. O'Donnell and G. RÄnger. A Methodology for Deriving Parallel Programs with a Family of Parallel Abstract Machines. In *Euro-Par'97 Parallel Processing, volume 1300 of LNCS*, pages 662–669. Springer, 1997.
- [] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring Java Programs for Flexible Locking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 71–80, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985804. URL <http://doi.acm.org/10.1145/1985793.1985804>.
- [] D. Smith. Readings in Artificial Intelligence and Software Engineering. chapter Top-down Synthesis of Divide-and-Conquer Algorithms, pages 35–61. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986. ISBN 0-934613-12-5. URL <http://dl.acm.org/citation.cfm?id=31870.31872>.
- [] D. R. Smith. Applications of a Strategy for Designing Divide-and-Conquer Algorithms. *Sci. Comput. Program.*, 8(3):213–229, June 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90034-7. URL [http://dx.doi.org/10.1016/0167-6423\(87\)90034-7](http://dx.doi.org/10.1016/0167-6423(87)90034-7).
- [] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *ESEC/FSE '09*, pages 173–182, Amsterdam, 2009. ACM. ISBN 978-1-60558-001-2.