Int J Parallel Prog
DOI 10.1007/s10766-013-0266-5

Cost-Directed Refactoring for Parallel Erlang Programs

Christopher Brown · Marco Danelutto · Kevin Hammond · Peter Kilpatrick · Archibald Elliott

Received: 3 March 2013 / Accepted: 5 August 2013

Abstract This paper presents a new programming methodology for introducing and tuning parallelism in Erlang programs, using source-level code refactoring from sequential source programs to parallel programs written using our skeleton library, *Skel*. High-level cost models allow us to predict with reasonable accuracy the parallel performance of the refactored program, enabling programmers to make informed decisions about which refactorings to apply. Using our approach, we demonstrate easily obtainable, significant and scalable speedups of up to 21 on a 24-core machine over the sequential code.

1 Introduction

Software development approaches are not keeping pace with the current trend towards increasingly parallel multi-core/many-core hardware, as epitomised by the recent announcement of Intel's 60-core Xeon Phi x86 co-processor. Most current applications

C. Brown (✉)· K. Hammond · A. Elliott
School of Computer Science, University of St Andrews, Saint Andrews, Scotland, UK
e-mail: chrisbrown.guitar@googlemail.com; cmb21@st-andrews.ac.uk

K. Hammond
e-mail: kh8@st-andrews.ac.uk

A. Elliott
e-mail: ashe@st-andrews.ac.uk

M. Danelutto
Department of Computer Science, University of Pisa, Pisa, Italy
e-mail: marcod@di.unipi.it

P. Kilpatrick
School of Electronics, Electrical Engineering and Computer Science, Queen's University, Belfast, UK
e-mail: p.kilpatrick@qub.ac.uk

programmers are not experts in parallelism, so knowing when and where to effectively deploy parallelism can seem *ad-hoc* or even impossible. Most parallel programs are therefore written using simple, primitive and small-scale *threading* techniques. While fully automatic approaches may work in specific cases, they lack generality and scalability. What is needed is a more systematic methodology for introducing and tuning parallelism. This paper describes such a methodology, and explains how it can be used in Erlang [1]. We introduce a novel programming methodology that uses advanced semi-automatic software refactoring techniques coupled with algorithmic skeletons and supported by high-level cost models, to provide a structured approach for reasoning about parallel programming. We use cost models to predict the outcome of the refactorings, allowing a programmer to intelligently decide which refactorings to apply. Automated refactoring tools are used as part of the normal parallel tool-chain to introduce and tune the parallelism. A set of high-level skeletons [2] provides the *palette* of possible parallelisations to choose from; corresponding cost models are used to provide the *evidence* that is needed to make informed decisions; and refactoring tools provide the programmer with *choice* and enforce the required *discipline* to correctly implement those decisions. Used properly, refactoring can improve programmer productivity by: *helping* the programmer to make the right implementation decisions; *hiding* unnecessary implementation decisions and detail; *warning* of common pitfalls; and *ensuring* that only correct transformations are applied. In order to demonstrate the principle of our methodology, we have chosen to use Erlang as an example of a high-level language where parallelism has been fairly unexplored, but which has an expanding programmer community that is keen to embrace the multi-core era.

This paper makes the following main contributions:

1. we introduce a new set of refactorings for Erlang, prototyped in the Wrangler refactoring tool [3], that introduce and tune parallelism;
2. we formally define rewrite rules for these refactorings for Erlang;
3. we introduce, for the first time, a domain-specific language for expressing skeletons in Erlang;
4. we provide and instantiate cost models for our skeletons in Erlang; and,
5. we show, using our refactoring approach, how to easily achieve a speedup of up to 21 on a 24-core machine for one example.

In addition to the above contributions, this paper also shows an example of effective parallelisation in Erlang, adding to previous attempts, such as Parallelising Dialyzer [4] and a scalability benchmark suite for Erlang [5]. Although it is common to provide cost models and abstract rewrite rules for specific algorithmic skeletons, this paper represents the first attempt, of which we are aware, to provide a concrete and tool-supported programming methodology that builds on these lower-level mechanisms to develop parallel Erlang programs. While this paper considers only Erlang, it is important to note that the principles of the refactorings, the skeleton implementations and the associated cost models that are described here can be carried over to other skeleton frameworks for other languages, including C, C++, Java and Haskell. Skeletons provide a structural expression for parallelism, where the structure is common across different languages. Refactoring, on the other hand, relies on a disciplined structural approach. While the skeletons themselves are implemented in terms of different syntaxes depending on the language being used, the underlying abstract structure common to the skeletons, and the rewritings associated with them, remains the same.

2 A Cost-Directed Parallel Programming Methodology

Our parallel programming methodology aims to support the inexperienced parallel programmer who may have little knowledge of how to apply/introduce skeletons or tuning; and also the more experienced programmer who simply wants an automated tool that checks that the transformations are correct, supplies cost information, and automatically rewrites source code under their direction. Our proposed refactoring methodology is shown in Fig. 1. The programmer commences with a sequential program, without any introduced skeletons or parallelism. The first step is to gain evidence about the program, using profiling information and cost models (such as those from Sect. 3.3). Using this evidence, the programmer can then use *refactoring* to automatically introduce skeletons into the program.

Refactoring is the process of changing the structure of a program while preserving its functional semantics in order, for example, to increase code quality, programming productivity and code reuse. The term *refactoring* was first introduced by Opdyke in his PhD thesis in 1992 [6], and the concept goes at least as far back as the fold/unfold system proposed by Burstall and Darlington in 1977 [7].

Following a refactoring step, it is possible that further factorisations of a program can be made available or that a further tuning/refinement step is required to get the desired parallelism. The programmer may therefore re-evaluate the factorised program, gathering further evidence about the predicted performance and then applying further refactorings, as required. These steps can be repeated in order to introduce nested skeletons or to refine the program based on different inputs, for example. At any step the programmer can undo and redo their changes using the refactoring tool; this may be in order to try a different set of factorisations that may lead to better performance, where the input set changes, for example. In this way the methodology supports both naive and experienced programmers, directing the former and assisting the latter to achieve good parallelisations. Section 5 gives two simple case studies that illustrate the use of this methodology.

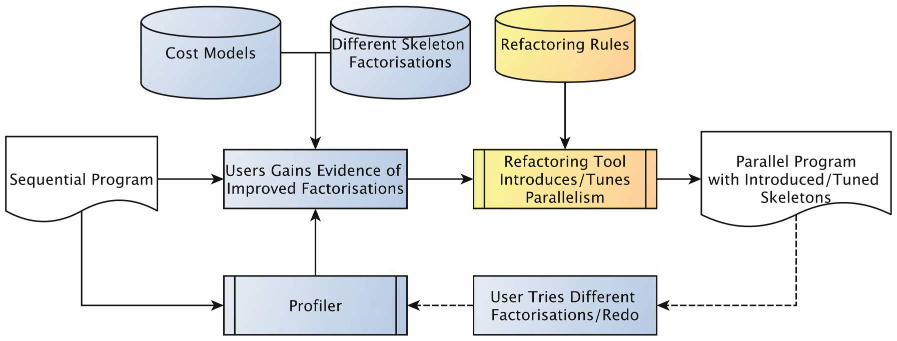

Cost Models
Different Skeleton Factorisations
Refactoring Rules
Sequential Program
Users Gains Evidence of Improved Factorisations
Refactoring Tool Introduces/Tunes Parallelism
Parallel Program with Introduced/Tuned Skeletons
Profiler
User Tries Different Factorisations/Redo

Fig. 1 The cost-directed parallel refactoring methodology

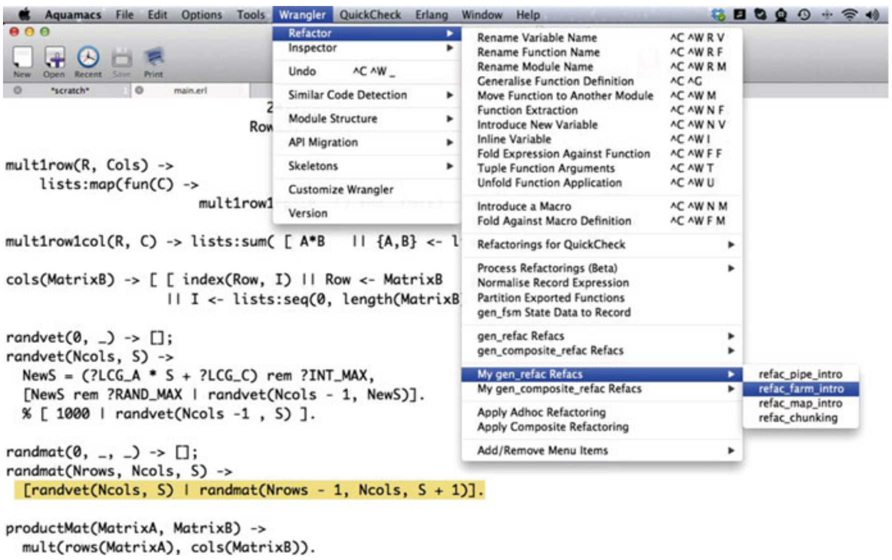
3 Background

3.1 Wrangler

The refactorings are applied automatically to a selected piece of syntax in the refactoring editor and are fully implemented in the Wrangler [3] refactoring tool for Erlang, as shown in Fig. 2. Wrangler itself is implemented in Erlang and is integrated into both Emacs and Eclipse. Figure 2 shows Wrangler in Emacs, presenting a menu of refactorings for the user. Like most interactive semi-automatic refactoring tools, the user follows a number of steps in order to invoke a refactoring:

1. The user starts with either an Emacs or an Eclipse session, with their sequential or parallel code.
2. The user identifies and highlights a portion of code that is amenable for refactoring in the text editor.
3. The appropriate refactoring is then selected from the Wrangler drop down menu. This step requires user-knowledge.
4. Wrangler will then ask the user for any additional parameters, such as the number of workers, any additional functions, or any additional information, such as new names for any new definitions that may be introduced by the refactoring process.
5. Wrangler then checks the pre-conditions, and if the pre-conditions hold, the program is transformed by the tool, depending on the refactoring rule being invoked.
6. The source code in the editor is automatically changed to reflect the refactored program.

We exploit a recent Wrangler extension that allows refactorings to be expressed as AST traversal strategies in terms of their pre-conditions and transformation rules. The



Aquamacs
File
Edit
Options
Tools
Wrangler
QuickCheck
Erlang
Window
Help
New
Open
Recent
Save
Print
scratch
main.erl
Refactor
Inspector
Undo ^C ^W _
Similar Code Detection
Module Structure
API Migration
Skeletons
Customize Wrangler
Version
Rename Variable Name ^C ^W R V
Rename Function Name ^C ^W R F
Rename Module Name ^C ^W R M
Generalise Function Definition ^C ^G
Move Function to Another Module ^C ^W M
Function Extraction ^C ^W N F
Introduce New Variable ^C ^W N V
Inline Variable ^C ^W I
Fold Expression Against Function ^C ^W F F
Tuple Function Arguments ^C ^W T
Unfold Function Application ^C ^W U
Introduce a Macro ^C ^W N M
Fold Against Macro Definition ^C ^W F M
Refactorings for QuickCheck
Process Refactorings (Beta)
Normalise Record Expression
Partition Exported Functions
gen_fsm State Data to Record
gen_refac Refacs
gen_composite_refac Refacs
My gen_refac Refacs
My gen_composite_refac Refacs
Apply Adhoc Refactoring
Apply Composite Refactoring
Add/Remove Menu Items
refac_pipe_intro
refac_farm_intro
refac_map_intro
refac_chunking

```
                                          2
                                        Row

mult1row(R, Cols) ->
    lists:map(fun(C) ->
                    mult1row1

mult1row1col(R, C) -> lists:sum( [ A*B   || {A,B} <- l

cols(MatrixB) -> [ [ index(Row, I) || Row <- MatrixB
                   || I <- lists:seq(0, length(MatrixB

randvet(0, _) -> [];
randvet(Ncols, S) ->
  NewS = (?LCG_A * S + ?LCG_C) rem ?INT_MAX,
  [NewS rem ?RAND_MAX | randvet(Ncols - 1, NewS)].
  % [ 1000 | randvet(Ncols -1 , S) ].

randmat(0, _, _) -> [];
randmat(Nrows, Ncols, S) ->
  [randvet(Ncols, S) | randmat(Nrows - 1, Ncols, S + 1)].

productMat(MatrixA, MatrixB) ->
  mult(rows(MatrixA), cols(MatrixB)).
```

Fig. 2 The Erlang refactorer, Wrangler, showing a list of parallel refactorings

extension comes in two parts: a user-level language for describing the refactorings themselves [8]; plus a Domain-Specific-Language to compose the refactorings [9].

3.2 Skeletons

We take a pattern-based approach where a parallel application is developed by composing *algorithmic skeletons*. An *algorithmic skeleton* [2] is an abstract computational entity that models some common pattern of parallelism (such as the parallel execution of a sequence of computations over a set of inputs, where the output of one computation is the input to the next; i.e., a pipeline). A skeleton is typically implemented as a high-level function that takes care of the parallel aspects of a computation (e.g. the creation of parallel threads, communication and synchronisation between these threads, load balancing etc.), and where the programmer supplies sequential problem-specific code and any necessary skeleton parameters. These skeletons may be specialised by providing (suitably wrapped) sequential portions of code implementing the business logic of the application.[1] For this paper, we restrict ourselves to four classical parallel skeletons, that are among the most common and most useful:

- `seq` is a trivial wrapper skeleton that implements the sequential evaluation of a function, $f :: a \rightarrow b$, applied to a sequence of inputs, $x_1, x_2, \ldots, x_n$.
- `pipe` models a parallel pipeline applying the functions $f_1, f_2, \ldots, f_m$ in turn to a sequence of independent inputs, $x_1, x_2, \ldots, x_n$, where the output of f_i is the input to f_{i+1}. Parallelism arises from the fact that, for example, $f_i(x_k)$ can be executed in parallel with $f_{i+1}(f_i(x_{k-1}))$. Here, each f_i has type $a \rightarrow b$.
- A `farm` skeleton models the application of a single function, $f :: a \rightarrow b$, to a sequence of independent inputs, $x_1, x_2, x_3, \ldots, x_n$. Each of the $f(x_1)$, $f(x_2)$, $f(x_3)$, $\ldots$, $f(x_n)$ can be executed in parallel.
- A `map` skeleton is a variant of a farm where each independent input, x_i, can be $x_1, x_2, x_3, \ldots, x_n$, is partitioned $(p :: a \rightarrow [b])$ into a number of sub-parts that can be worked upon in parallel, a worker function, $f :: [b] \rightarrow [c]$, is then applied to each element of the sublist in parallel, finally the result is combined $(c :: [c] \rightarrow d)$ into a single result for each input.

3.3 Skeleton Cost Models

In this section we give a corresponding high-level cost model for each skeleton in Sect. 3.2, derived after those presented in [10,11]. These cost models capture the service time of our skeletons and will be used to drive the refactoring process. In order to demonstrate the principles of our methodology, the cost models that we consider here are intentionally high-level, abstracting over many language- and architecture-specific details. If desired, more complex models could be used to yield possibly more accurate predictions for a specific architecture, without changing the general methodology. A suitable cost model for the parallel pipeline with m stages is:

[1] Our skeleton implementations can be found at https://github.com/ParaPhrase/skel.

$$TC_{pipeline}(L) = max_{i=1\dots m}(T_{stage_i}(L)) + T_{copy}(L) \tag{1}$$

where L represents the maximum size of the input tasks, x_i, and T_{copy} is the time it takes to copy data between the pipeline stages. This defines the cost of a steady-state pipeline as the maximum execution time for any of the stages in the pipeline. The corresponding cost model for the map skeleton is:

$$\begin{gathered} TC_{map}(L) = T_{distrib}(N_w, L) + \frac{T_{Fun}(L)}{Max(N_p, N_w)} + T_{gather}(N_w, L) \\ \texttt{where } N_W = npartitions(L) \end{gathered} \tag{2}$$

where $T_{distrib}$ and T_{gather} are the times to distribute the computations and gather the results, respectively (see below), $npartitions(L)$ is the number of partitions in L created by the partition function, and $T_{Fun}(L)$ is the time it takes to compute the entire sequential map computation. Here we employ N_p as the number of processors available in the system. For our Erlang definition, more accurate definitions of $T_{distrib}$ and T_{gather} are:

$$\begin{aligned} T_{distrib}(N_w, L) &= N_w \cdot T_{spawn} + N_w \cdot \left(T_{setup} + T_{copy}\left(\frac{L}{N_w}\right)\right) \\ T_{gather}(N_w, L) &= N_w \cdot \left(T_{setup} + T_{copy}\left(\frac{L}{N_w}\right)\right) \end{aligned}$$

where, T_{setup} is the time it takes to set up N_w Erlang processes, and T_{copy} is the time it takes to copy L items of data to N_w processes.

For the farm skeleton, assuming that each worker task has a similar granularity and that all workers are fully occupied, the corresponding cost model is similar to that for the map skeleton, except that N_W is a fixed parameter:

$$TC_{farm}(N_w, L) = max\{T_{emitter}(N_p, N_w, L), \frac{T_{Fun}(L)}{Max(N_p, N_w)}, T_{collector}(N_w, L)\} \tag{3}$$

3.4 Erlang

Erlang is a strict, impure, functional programming language with support for *first-class concurrency*. Although Erlang provides good concurrency support, there has so far been little research into how this can be used at a higher level to effectively support *deterministic*, *structured* parallelism. The Erlang concurrency model allows the programmer to be explicit about processes and communication, but deals implicitly with task placement and synchronisation. It provides three concurrency primitives:

- `spawn` to execute functions in a new lightweight Erlang process;
- `!` to send messages between Erlang processes; and,
- `receive` to receive messages from another Erlang process.

Process scheduling is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms.

4 Skeleton Equivalences and Refactoring

This section gives a number of high-level structural equivalence relations for skeletons and derives corresponding refactorings for Erlang. The rules given here define *language-independent* structural equivalences between two (possibly nested) skeletons, that can be used to improve performance by rewriting from one form to the other. Figure 3 shows four well-known skeleton equivalences (see, e.g. [12,13]). Here *Pipe*, $\circ$ and *Map* refer to the skeletons for parallel pipeline, function composition and parallel map, respectively, and *S* denotes any skeleton. All skeletons work over streams. $Partition$ and $Combine$ are primitive operations that work over non-streaming inputs. p and c are simple functions that specify a partition ($p :: a \rightarrow [b]$), and a combine operation ($c :: [b] \rightarrow a$). Working from left to right in these equivalences increases the parallelism degree, while working from right to left reduces it. Reading from *left to right*, we can therefore interpret the rules as follows:

1. *pipe intro*. A sequential function composition with n function applications can be rewritten as a parallel pipeline with n stages.
2. *map fission*. A single parallel map over a sequential composition can be rewritten as a composition where each stage is a parallel map. Here we state that the partitioning and combining functions for the map on the left hand side are replicated in the maps on the right hand side. The inverse of this is *map fusion*, read from right to left.
3. *farm intro*. A skeleton or sequential computation over a stream can be rewritten to a farm computation over the stream.
4. *data2stream*. A parallel map can be rewritten as a parallel farm. This requires the introduction of partitioning and combining stages before and after the farm.
5. *map intro*. A skeleton or sequential computation over a stream can be rewritten to a map, given the combining and partitioning functions c and p, and provided there is a translation between the skeleton S_1 and a skeleton, S_1', where S_1' works over each of the partitions produced by p.

Their inverses, read from right to left, are *pipe elimination*, *map fusion*, *farm elimination*, *stream2data* and *map elimination*. Any combination of these rules can be used as part of a (reversible and undoable) parallel refactoring process.

$$\begin{array}{lcll}
S_1 \circ S_2 & \equiv & Pipe(S_1, S_2) & \text{pipe intro/elim} \\
Map(S_1 \circ S_2, p, c) & \equiv & Map(S_1, p, c') \circ Map(S_2, p', c) & \text{map fission/fusion} \\
S & \equiv & Farm(S) & \text{farm intro/elim} \\
Map(F, p, c) & \equiv & Pipe(Partition(p), Farm(F), Combine(c)) & \text{data2stream} \\
S_1 & \equiv & Map(S_1', p, c) & \text{map intro/elim}
\end{array}$$

Fig. 3 Some standard skeleton equivalences

4.1 Refactoring Erlang

This section presents new refactorings that introduce parallelism for Erlang, implementing the rules from Fig. 3. Each refactoring is described as a transformation rule and any associated preconditions, operating over the abstract syntax tree (AST) of the source program.

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

where $x_0, \dots, x_n$ are the arguments to the refactoring. The refactoring rules are defined as functions over nodes that represent expressions in the AST:

$$\mathcal{E}[\![.]\!] :: Expr \rightarrow Expr$$

Choice is denoted by $a \oplus b$ where rule a is tried first, and if it fails to match the program segment under consideration, b is tried instead. The refactoring rules are defined in terms of choice as typically the rule to be applied depends upon the source code highlighted by the user. For example, refactoring a list comprehension requires a different rule from refactoring a recursive definition. Quasi-quotes are used to denote code syntax, so that $[\![\mathtt{f} = \mathtt{e}]\!]$ denotes a function in the AST of the form `f = e`, for example. In order to use the cost models to direct the refactoring process, each cost model is implemented as a simple Erlang computation. The refactorer stores the programmer-provided values for the various parameters of the cost models, such as the cost of the sequential computation of a skeleton and the predicted partitioning and combine costs. The cost models are then instantiated with these profile costs, and the result of the cost models are presented to the user, who can thereby make an informed decision about which refactoring to choose.

4.1.1 Pipeline Introduction

Refactoring rules for the *Introduce Pipeline Refactoring* are shown below.

$$\begin{array}{l}
PipeComp(\rho, \mathtt{e}) = \\
\quad \mathcal{E}[\![[\ \mathtt{f_1}\ (\ \mathtt{f_2}\ (\dots \mathtt{f_n}(\mathtt{Input}) \dots))\ ||\ \mathtt{Input} \leftarrow \mathtt{Inputs}\ ]\ ]\!] \Rightarrow \\
\qquad [\![\ \mathtt{skel : run([\{pipe, [\{seq, fun\ ?MODULE : f_n/1\}, \dots \{seq, fun\ ?MODULE : f_2/1\},}} \\
\qquad\qquad\qquad\qquad \mathtt{\{seq, fun\ ?MODULE : f_1/1\}]\}], Inputs)}\ ]\!] \\
\qquad \{\mathtt{skel} \in \mathtt{imports}(\rho), \mathtt{run} \in \rho, \mathtt{pipe} \in \rho,\ \mathtt{seq} \in \rho, \mathtt{f_1}, \mathtt{f_2}, , \dots, \mathtt{f_n} \in \rho, \mathtt{Inputs} \in \rho\} \\
\oplus PipeSeq(\rho, \mathtt{e}) = \\
\quad \mathcal{E}[\![\ \{\mathtt{seq}, (\mathtt{fun(X)} \rightarrow \mathtt{f_1}\ (\ \mathtt{f_2}\ (\dots \mathtt{f_n}(\mathtt{X}) \dots))\ \mathtt{end})\}]\!] \Rightarrow \\
\qquad [\![\ \{\mathtt{pipe, [\{seq, fun\ ?MODULE :} f_n/1\},\ \dots \{seq, \mathtt{fun\ ?MODULE :} f_2/1\}, \\
\qquad\qquad \mathtt{\{seq, fun\ ?MODULE : f_1/1\}]\}}\ ]\!] \\
\qquad \{\ \mathtt{pipe} \in \rho, \mathtt{seq} \in \rho, \mathtt{f_1}, \mathtt{f_2}, , \dots, \mathtt{f_n} \in \rho\}
\end{array} \tag{4}$$

Each rule takes an environment, ρ, and an expression, $\mathtt{e}$, denoting the syntax phrase to be matched. The first rule, $PipeComp$, matches an expression that forms a list comprehension in Erlang, e.g.:

```
[ f1 (f2 (f3( Input ))) || Input <- Inputs ]
```

and transforms the expression into a call to the skeleton DSL, with a pipeline:

```
skel:run([{pipe, [{seq,fun ?MODULE:f3/1}, {seq,fun ?MODULE:f2/1}
                  {seq,fun ?MODULE:f1/1}]}],Inputs)
```

Here we introduce a call to the skeleton library (in module `skel`), calling the top-level function, `run`, which takes a list of skeletons to execute. Each skeleton is expressed as a tuple: `pipe` denotes the skeleton is a pipeline, and `seq` denotes a sequential computation. Here the pipeline has three stages, where the first stage executes `f1`, the second, `f2` and the third, `f3`. The input stream to the pipeline is preserved as a list, `Inputs`. `?MODULE:f/1` is required in Erlang in order to locate the module defining the function, `f` (with an arity of `1`); `?MODULE` means that the module is to be found from the namespace, rather than being named explicitly. In the preconditions, the refactoring checks that the newly introduced skeletons, `pipe` and `seq`, the stages of the pipeline, and the list of inputs, `Inputs`, are in scope. In the second rule, $PipeSeq$, we match an existing sequential skeleton (as part of a nesting of skeletons inside a `run` call), and transform it into a pipeline skeleton.

4.1.2 Parallel Map Introduction

Here we show the refactoring rules to introduce a parallel map skeleton into Erlang, where we use `parmap` to distinguish between the parallel `map` skeleton and the Erlang built-in sequential `lists:map` function. The corresponding refactoring rules for the *Introduce Parallel Map Refactoring* are:

$$
\begin{array}{l}
\mathit{ParMapIntroSeq}(\rho, \mathtt{e}, g, p, c) = \\
\quad \mathcal{E}[\![\{\mathtt{seq}, \mathtt{F}\}]\!] \Rightarrow \\
\quad [\![\{\mathtt{parmap}, [\{\mathtt{seq}, \mathtt{fun\ ?MODULE}:g/1\}], \mathtt{fun\ ?MODULE}:p/1, \mathtt{fun\ ?MODULE}:c/1\}]\!] \\
\qquad \{\mathtt{seq} \in \rho, \mathtt{parmap} \in \rho, p \in \rho, c \in \rho, g \in \rho\} \\
\oplus \mathit{ParMapIntroComp}(\rho, \mathtt{e}, g, p, c) = \\
\quad \mathcal{E}[\![[\ \mathtt{f(Input\ )} \mid\mid \mathtt{Input} \leftarrow \mathtt{Inputs}\ ]]\!] \Rightarrow \\
\quad [\![\mathtt{skel:run}([\{\mathtt{parmap}, [\{\mathtt{seq}, \mathtt{fun\ ?MODULE}:g/1\}], \mathtt{fun\ ?MODULE}:p/1, \mathtt{fun\ ?MODULE}:c/1\}], \\
\hfill \mathtt{Inputs})]\!] \\
\qquad \{\mathtt{skel} \in \mathtt{imports}(\rho), \mathtt{parmap} \in \rho, \mathtt{run} \in \rho, \mathtt{seq} \in \rho, \mathtt{Inputs} \in \rho, p \in \rho, c \in \rho, g \in \rho\}
\end{array}
\tag{5}
$$

The $ParMapIntroSeq$ rule takes an arbitrary sequential skeleton, `{seq, F}`, that computes the result of a sequential function, `F` (the input to `F` is implicit here, and is handled at a higher-level by the `skel:run` function), and transforms it into a `parmap` skeleton, passing a function, g, into the map, together with partition and combine functions, p and c. These functions should be supplied as arguments to the refactoring, and must be in scope after the transformation. The second rule in Eq. 5, $ParMapIntroComp$, allows a list comprehension to be transformed into a new skeleton composition. For example, suppose that we have:

```
[ f(Input) || Input <- Inputs ].
```

Using the $MapIntroComp$ rule, we can transform this into:

```
skel:run([{parmap, [{seq, fun ?MODULE:f'/1}],
      fun ?MODULE:partition/1, fun ?MODULE:combine/1}], Inputs)
```

Here we transform the list into a call to `skel:run` where `parmap` denotes the `parmap` skeleton, `f'` is the user supplied worker for the `parmap`, and `partition` and `combine` are user supplied functions for the partitioner and combiner.

4.1.3 Farm Introduction

Refactoring rules for the *Introduce Task Farm Refactoring* are shown below:

$$\begin{array}{l}
\textit{FarmIntroSeq}(\rho, \texttt{e}, \texttt{Nw}) = \\
\quad \mathcal{E}[\![\{\texttt{seq}, \texttt{E}\}]\!] \Rightarrow \\
\quad [\![\{\texttt{farm}, [\{\texttt{seq}, \texttt{E}\}], \texttt{Nw}\}]\!] \\
\qquad \{\texttt{seq} \in \rho, \texttt{farm} \in \rho\} \\
\oplus \textit{FarmIntroMap}(\rho, \texttt{e}, \texttt{Nw}) = \\
\quad \mathcal{E}[\![\texttt{lists} : \texttt{map(fun ?MODULE : f/1, List)}]\!] \Rightarrow \\
\quad [\![\texttt{skel} : \texttt{run([\{farm, [\{seq, fun ?MODULE : f/1\}], Nw\}], List)}]\!] \\
\qquad \{\texttt{seq} \in \rho, \texttt{farm} \in \rho, \} \\
\oplus \textit{FarmIntroComp}(\rho, \texttt{e}, \texttt{Nw}) = \\
\quad \mathcal{E}[\![\texttt{[f(Input) || Input} \leftarrow \texttt{Inputs]}]\!] \Rightarrow \\
\quad [\![\texttt{skel : run([\{farm, [\{seq, (fun(Input)} \rightarrow \texttt{?MODULE : f(Input)end\}], Nw\}], Inputs)}]\!] \\
\qquad \{\texttt{skel} \in \texttt{imports}(\rho), \texttt{run} \in \rho, \texttt{seq} \in \rho, \texttt{farm} \in \rho\} \\
\oplus \textit{FarmIntroComp2}(\rho, \texttt{e}, \texttt{Nw}) = \\
\quad \mathcal{E}[\![\texttt{[f}_1\texttt{ (f}_2\texttt{ (... f}_\texttt{n}\texttt{(Input)...)) || Input} \leftarrow \texttt{Inputs]}]\!] \Rightarrow \\
\quad [\![\texttt{skel : run([\{farm, [\{seq, (fun(Input)} \rightarrow \texttt{f}_1\texttt{ (f}_2\texttt{ (... f}_\texttt{n}\texttt{(Input)...)) end)\}], Nw\}],} \\
\hfill \texttt{Inputs)}]\!] \\
\qquad \{\texttt{skel} \in \texttt{imports}(\rho), \texttt{run} \in \rho, \texttt{seq} \in \rho, \texttt{farm} \in \rho, \texttt{Input}\ \textit{fresh}\}
\end{array} \tag{6}$$

The rules take three parameters: an environment, ρ, a selected expression, `e`, and the number of workers for the farm, `Nw`. The first rule, $FarmIntroSeq$, states that if we have a `seq` skeleton wrapping an expression, `E`, then we can transform it into a `farm` skeleton, with an additional constraint, `Nw`, governing the number of task farm workers. We note that the $ParMapIntroSeq$ rule in Eq. 5 does not require an `Nw` parameter, as the partition function, p, divides the input into n components according to its definition. In the rule, `Input` $fresh$ denotes that a new name, `Input`, will be introduced, and that the name will be new and not conflict with any other names in the scope in which `Input` is defined and used. The $FarmIntroMap$ rule allows a farm to replace an existing sequential map. In this case, the function passed into the map, `f`, becomes the worker function of the farm; the input list, `List`, becomes the input stream of tasks to the farm. Moreover, the refactoring rule may also match against a list comprehension, shown by $FarmIntroComp$, converting it into a task farm. In this case, if there is a function composition as the farm worker, then an anonymous function is introduced in order to pass the tasks into the task `farm` (this is shown in Rule $FarmIntroComp2$). For example, suppose that we have the following list comprehension:

```
[ f ( f2 ( Input )) || Input <- Inputs ]
```

The refactoring will transform this into:

```
skel:run([{farm, [{seq, (fun(Input) -> f ( f2 (Input)) end)}],
                                              NW}], Inputs)
```

Here, `farm` denotes the task `farm` skeleton, where we supply a list of worker skeletons, denoted by `[ {seq, ...} ]`, where here each worker function is a `seq` skeleton, wrapping the function composition, `f ( f2 (Input))`. An anonymous function, that takes `Input` as an argument has been introduced, in order to pass the task into the worker.

4.1.4 Introduce Chunking

Having introduced parallelism via skeletons, often the programmer needs to further tune the parallel performance. Here we discuss a new refactoring that enables programmers to increase the granularity of the parallelism. The rules for the *Introduce Chunking Refactoring* are shown below:

$$\begin{array}{l}
\mathit{IntroChunkComp}(\rho, c, \mathtt{e}) = \\
\quad \mathcal{E}[\![\, [\ \mathtt{f(\ Input\ )} \ || \ \mathtt{Input} \leftarrow \mathtt{Inputs}\ ]\,]\!] \ \Rightarrow \\
\quad [\![\mathtt{skel : run([\{parmap, [\{seq, fun\ ?MODULE : f/1\}], fun\ skel : partition/1,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{fun\ skel : combine/1\}], skel : chunk(Inputs,}c\mathtt{))}]\!] \\
\qquad \{\mathtt{skel} \in \mathtt{imports}(\rho), \mathtt{parmap} \in \rho, \mathtt{run} \in \rho, \mathtt{seq} \in \rho\} \\
\oplus \mathit{IntroChunkFarm}(\rho, c, \mathtt{e}) = \\
\quad \mathcal{E}[\![\mathtt{skel : run([\{farm, [\{seq, fun\ ?MODULE : f/1\}], Nw\}], Inputs)}]\!] \ \Rightarrow \\
\quad [\![\mathtt{skel : run([\{parmap, [\{seq, fun\ ?MODULE : f/1\}], fun\ skel : partition/1,} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{fun\ skel : combine/1\}], skel : chunk(Inputs,}c\mathtt{))}]\!] \\
\qquad \{\mathtt{skel} \in \mathtt{imports}(\rho), \mathtt{run} \in \rho, \mathtt{parmap} \in \rho, \mathtt{seq} \in \rho, \mathtt{farm} \in \rho, \mathtt{Nw} \in \rho\}
\end{array} \tag{7}$$

The first rule, $IntroChunkComp$, takes, along with an environment, ρ, and a selected expression, e, a chunk size, c. The refactoring then matches a list comprehension and transforms it into a new `parmap` skeleton. Here we *chunk* up the tasks in the input list, by calling a function, `chunk`, located in the skeleton library. `chunk` is parameterised by the input list and the chunk size, and returns a new list of tasks, grouped together in sizes of c. The function `partition` simply passes the chunks to the map workers and `combine` appends the results into a result list.

The second rule, $IntroChunkFarm$, allows the programmer to introduce a chunk for a skeleton that is already farmed. In this case, the refactoring simply rewrites the `farm` skeleton into a `parmap`, again with the input list chunked into c chunks. It is important to note that, in both of these refactoring rules, the function `f` remains the same when passed as an argument to the `parmap` skeleton. Here, the `partition`, `combine` and `chunk` functions that are introduced are supplied by the Skel library and are defined:

```
partition(X) -> X.
combine([])->[];
combine([X|Xs]) -> lists:append(X, recomp(Xs)).
chunk([], ChunkSize) -> [];
chunk(List, ChunkSize) ->
   case (length(List) < ChunkSize) of
     true -> [List];
     false -> Chunk = lists:sublist(List, ChunkSize),
           NewList = lists:sublist(List, ChunkSize+1),
           [ Chunk | chunk(NewList, ChunkSize) ]
  end.
```

Here, `chunk` groups together `ChunkSize` elements in a list; `partition` acts as the identity function (in the `parmap` skeleton, the `partition` function decomposes a single element of the input stream; in this case a single element is a group of elements (a chunk) where we want each worker thread to operate over a whole chunk, and so we disable the partition function by using the identity instead), and `combine` concatenates the partitioned results into a single list.

To illustrate how this chunking refactoring works, consider the following Erlang code, which applies a fine-grained function, `f`, to each element in an input list, `Inputs`, using a list comprehension:

```
[ f(Input) || Input <- Inputs ].
```

We can rewrite this into an equivalent program using a combination of the `chunk`, `combine` and `partition` functions, therefore increasing granularity:

```
skel:run({parmap, [{seq, fun ?MODULE:f/1}], fun skel:partition/1,
                             fun skel:combine/1}],
                             skel:chunk(Inputs, C)).
```

Here we also have to introduce a call to `lists:map` inside the list comprehension, as `f` now operates over a `partition` of list elements (rather than a single element, as before). This is now equivalent to the following `parmap` skeleton using *Skel*:

```
skel:run({parmap, [{seq, fun ?MODULE:f/1}], fun skel:partition/1,
                             fun skel:combine/1}],
                             skel:chunk(Inputs, C)).
```

Here, we lose the explicit `lists:map` when calling `f`, as the skeleton implicitly assigns an instance of `f` to each partition, performing an implicit map. It is possible to use the cost model defined for the map to calculate a chunk size, as the model takes into account the tradeoff between the process spawn overhead, which increases with *smaller* chunk sizes, and the advantage of executing more and more things in parallel.

5 Refactoring Case Studies

We present two worked examples, showing how the rules above can be applied as part of our methodology to introduce skeletons and also to further refine, introduce and tune the skeleton program. Our first source program is a denoising algorithm, `Denoise`, where we initially introduce a pipeline factorisation, and then, following cost analysis, refine this factorisation into a nesting of skeletons that also include a `parmap` and a task `farm`. Our second example is from the domain of symbolic computation, `SumEuler`, which computes the sum over the Euler totient function, applied to a list of integer numbers. We initially develop a data-parallel version of `SumEuler` using refactoring, and then further refactor to introduce chunking.

5.1 Denoising

In this section we illustrate the refactoring process on a simple example of denoising a stream of satellite images. The basic program comprises a two-stage function composition. In the first stage, `geoRef`, we apply an algorithm to each image as it is received to consolidate geo-referencing information. These images are then passed to a second stage, `filter`, where they are denoised. For the purposes of our experiment,

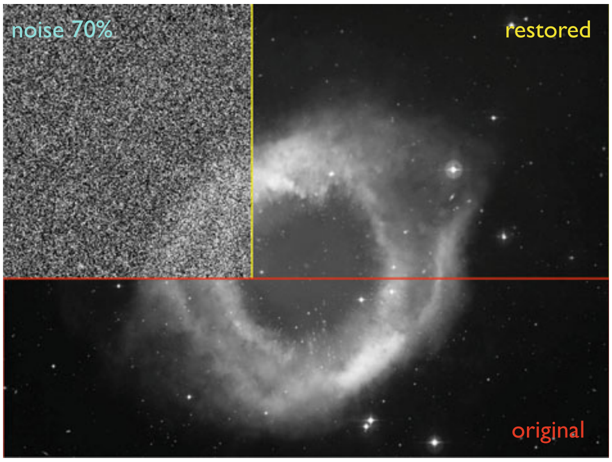


Fig. 4 Example of a satellite image before and after denoising, compared with original image

we abstract over the algorithm, providing a synthetic reproduction of the sequential denoiser code in Erlang. Figure 4 shows an example of a satellite image with 70 % noise on the left hand side, and on the right hand side after the denoising. The bottom portion of the figure shows the original image. The programmer uses information about the costs of each computation stage, plus basic metrics for T_{gather} and $T_{distrib}$ (obtained using profiling) to instantiate the cost models and the choice of refactoring.

Stage 1: Introduce a Pipeline The basic structure of the denoiser is:

```
denoise(Ims) -> [  filter (geoRef ( Im ) ) || Im <- Ims ].
```

Here a simple function composition is applied to a list of images. Our timings show that the `GeoRef` stage takes 171 ms to compute one image, and the `Filter` stage takes 466 ms for one image. The cost of the composition is simply the sum of the costs of the stages:

$$TC_{comp} = (T_{stage1} + T_{stage2})$$

Thus to denoise 1,024 images takes $(171 + 466) * 1,024$ ms. Based on these calculations, the programmer applies the *Introduce Pipeline Refactoring* (Rule $PipeComp$) to transform the function composition into a parallel pipeline, in order to reduce the overall runtime by the first stage:

```
denoise(Ims) -> skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                                   {seq, fun ?MODULE:filter/1}]}], Ims).
```

Using the parallel pipeline cost model, we can determine the total completion time for the pipeline to be 477 s for 1,024 images ($Max(171, 466) * 1,024$), plus some small overhead to fill the pipeline and to send messages.

Stage 2: Introduce a Parallel Map Using the cost models given in Sect. 3.3 it can be determined that the next stage of the refactoring process is to exploit data parallelism in either, or both, of the pipeline stages. The first stage of the pipeline, `geoRef`, does not have a corresponding partition function to transform into a map skeleton. The partitioner and combiner for the second `filter` stage, however, can be easily derived from the implementation of `filter`. The programmer therefore first introduces a new definition, `filter'`, plus associated partition and combine functions. This new `filter'` function works over smaller portions of the image, with `partition` breaking down the image into 16 smaller partitions (where each partition goes to a single worker operating in a thread). Based on these new functions, we profile the new function, `filter'` over all the images, giving us an average time of 50 ms. Although, $466/16 = 29$ ms, the computational cost of `filter'` is not directly proportional to the size of the input data. This could be to do with other system overheads in the Erlang system, such as Garbage collection or task creation/communication sizes. Profiling the costs for the distribution, combine, gathering and copying stages of the `parmap` appear to be have an approximate uniform value of 0.001 ms. Using the new costs of `filter`, T_{gather} and $T_{distrib}$, the programmer applies the *Introduce Parallel Map Refactoring* (Rule $ParMapIntroSeq$) to produce:

```
denoise(Ims) ->  skel:run([{pipe, [{seq, fun ?MODULE:geoRef/1},
                     {parmap, [{seq,fun ?MODULE:filter'/1}],
                              fun ?MODULE:partition/1,
                              fun ?MODULE:combine/1}]}], Ims).
```

This gives us a predicted service time of 175,104 ms for 1,024 images.

Stage 3: Introduce a Task Farm Although the `geoRef` stage does not lend itself to easy partitioning, we can still use the cost models to determine that it would be beneficial to apply the `geoRef` function to several images in parallel. Therefore the next stage in the refactoring process is to apply the *Introduce Task Farm Refactoring* (Rule $FarmIntroSeq$).

```
denoise(Ims) -> skel:run([{pipe,[{farm,[{seq,fun ?MODULE:geoRef/1}],
                    Nw}, {parmap, [{seq, fun ?MODULE:filter'/1}],
                              fun ?MODULE:partition/1,
                              fun ?MODULE:combine/1}]}], Ims).
```

Based on the cost model for the farm skeleton in Section 3.3, the programmer predicts an approximate service time of 34,153 ms with 8 workers (we introduce 8 workers to soak up the remaining cores on the machine) by adding a farmed `geoRef` stage. This represents a predicted speedup factor of 19.09 on a 24 core machine, compared to the original sequential version.

5.2 Sum Euler

Stage 1: Introducing a Task Farm Introducing parallelism here is done by first identifying a sub-expression in the program that generates a compound data structure, such as a list, and where each operation on the list could be computed in parallel. `sumEuler` is actually an *irregular* problem, with different granularities for the task sizes. This means that we profile the range of granularities, which range between 70 μs and 104 ms. Using the largest granularity to predict execution times allows us to predict a *worst case* sequential execution path and a *best case* prediction for the parallelism.

Our performance measurements indicate that the predicted sequential execution time for `sumEuler` will be 1,040 s, where `N` $= 10,000$. Introducing a task farm with 24 workers (one per core) will give a predicted execution time of 43 s on 24 cores. This is a speedup prediction of 23.88. In our example, we select the `lists:map` subexpression in `sumEuler` in order to introduce a farm:

```
sumEuler(N) -> result = lists:map(fun ?MODULE:euler/1,mkList(N)),
               lists:sum(result).
```

To introduce a task farm, we then use the refactoring tool to apply the $FarmIntroMap$ rule from the *Introduce Task Farm Refactoring*:

```
sumEuler(N) -> result=skel:run([{farm,[{seq, fun ?MODULE:euler/1}],
                                                   24}], mkList(N)),
               lists:sum(result).
```

Stage 2: Chunking While using a task farm for `sumEuler` creates a reasonable amount of parallelism, the parallelism is too fine-grained and the program does not scale as we expect. This is a common problem in the early stages of writing parallel programs. To combat this, we use the $IntroChunkFarm$ rule from the *Introduce Chunking Refactoring*. This refactoring allows us to group a number of small tasks into one larger parallel task, where each parallel thread operates over a sub-list, rather than just one element. We want each worker thread to be busy, so we chunk by assigning 416 to the parameter `C` below for groups of 416 elements ($10,000\ tasks/24\ workers$). By chunking in this way, we also decrease the communication costs, and reduce parallel overheads. Chunking can generally be achieved in a variety of different ways. In our example, we refactor our task farm into a map with a chunking and de-chunking stage:

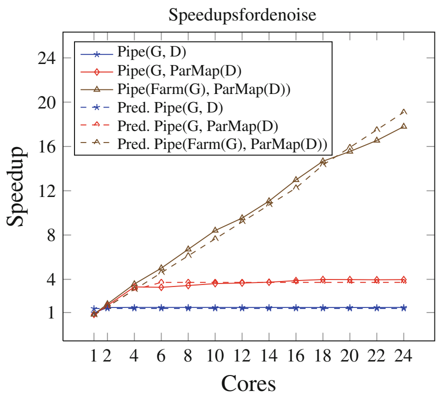
```
sumEuler(N) -> skel:run([{parmap, [{seq, fun ?MODULE:euler/1}],
                         fun ?MODULE:partition/1,
                         fun ?MODULE:combine/1}],
                         chunk(List, C)),
```

The refactoring also introduces new functions, `combine` and `partition`:

```
partition(X) -> X.
combine([])->[];
combine([X|Xs]) -> lists:append(X, combine(Xs)).
```

5.3 Predicted Versus Actual Times

All measurements have been made on an 800 MHz 24 core, dual AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.el5. and Erlang 5.9.1 R15 B01, averaging over 10 runs. Figure 5, left, compares the predicted (dashed) speedups against the actual (solid) speedups. The overall predicted speedup for *denoise* on 24 cores for the $Pipe(Farm(G), ParMap(D))$ version is 19.09 versus an actual speedup of 17.79, a 93 % accurate prediction. In the second refactoring step $Pipe(G, ParMap(D))$, where the second stage of the pipeline is transformed into a `parmap`, the predicted speedup is 3.73 (175 s) versus an actual speedup of 3.97 (132 s) on 24 cores. The overall predicted speedup for *sumEuler* is (shown in Figure 5, in the right column) 23.88 (4.35 s), with a predicted sequential time of 104 s versus an actual sequential



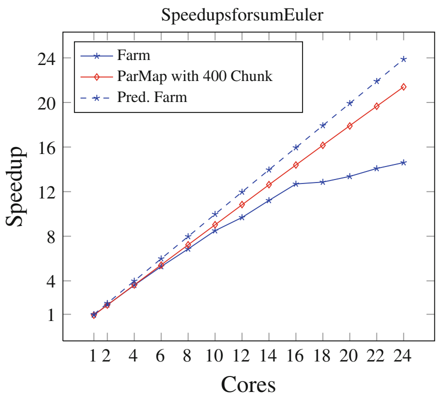


Fig. 5 Predicted (*dashed*) versus actual speedups (*solid*) for `denoise`(1,024) and `sumEuler`(10,000)

time of 179 s, a speedup of 14.6 (12.29 s) for the task farm version and 21.39 (8.38 s) for the chunked version with a `parmap`. In both graphs, the speedup seems to tail off as we add more cores. This is evident above 18 cores for the `denoise` and above 16 cores for the `SumEuler` task farm. Although more investigation is needed, we suspect that the `denoise` result is a combination of a hyper-threading effect and scheduling overheads in the Erlang VM. For the `sumEuler` example, the application seems to scale well for a task farm up to 16 cores, mostly because the system has an abundance of parallelism, with many tasks being very fine-grained. As we add more worker processes, overall communication and synchronisation costs begin to dominate the computation (evident through profiling). When we apply chunking, we can control the size and quantity of tasks for each worker thread, reducing the overall amount of parallelism, communication and synchronisation and ensuring that the granularity of each worker is maximised but reducing load balancing.

5.4 Using Refactoring Tools for Introducing Parallelism

Using a refactoring tool to introduce parallelism in a program instead of manually inserting the parallelism has many advantages. *Refactoring helps the programmer to program faster*. Using a parallel refactoring tool to introduce skeletons instead of manual insertion means that the programmer has to remember and understand less, allowing them to concentrate on the program design. Also, if the skeleton interfaces change, a refactoring tool can help automate the process of modifying the program to reflect the new interfaces. *Refactoring provides correctness-by-construction*. A refactoring tool, by virtue of its design, will not allow a user to break their programs. Introducing the wrong skeleton into a program is simply disallowed, assuming the preconditions are correct. Furthermore, the refactoring tool will only provide a list of refactorings that are applicable to a selected portion of code, saving time and frustration. *Refactoring encourages a consistent software engineering discipline*. Programmers often write a program without thinking about future developers. They may understand their code at the moment that they are writing it, but in a short time the code may become difficult to understand. Refactoring helps the programmer make their code more readable and consistent with a particular style. This helps other programmers to read and understand the skeleton code, making it more amenable for future tuning and modification. *Refactoring helps the programmer find bugs*. As parallel refactoring helps improve the understanding of algorithmic skeletons, it also helps the programmer to verify certain assumptions they have made about the program.

6 Related Work

Program transformation has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [14], and Mens and Tourwé producing a survey of refactoring tools and techniques in 2004 [15]. The first refactoring tool system was the *fold/unfold* system of Burstall and Darlington [16] which was intended to transform recursively defined functions. There has so far been only a limited amount of work on refactoring for parallelism [17]. Hammond et al. [18] used Template Haskell [19] with explicit cost models to derive automatic farm skeletons for Eden [20]. Unlike the approach presented here, Template-Haskell is compile-time, meaning that the programmer cannot continue to develop and maintain his/her program after the skeleton derivation has taken place. Other work on parallel refactoring has mostly considered loop parallelisation in Fortran [21] and Java [22]. However, these approaches are limited to concrete structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites as we have described here. We have recently extended HaRe, the Haskell refactorer [23], to deal with a limited number of parallel refactorings [24]. This work allows Haskell programmers to introduce data and task parallelism using small structural refactoring steps. However, it does not use pattern-based rewriting or cost-based direction, as discussed here. A preliminary proposal for a language-independent refactoring tool was presented in [25], for assisting programmers with introducing and tuning parallelism. However, that work focused on building a refactoring tool supporting multiple languages and paradigms, rather than on refactorings that introduce and tune parallelism using algorithm skeletons, as in this paper.

Since the 1990s, the skeletons research community has been working on high-level languages and methods for parallel programming [26–29,2,30]. A rich set of skeleton rewriting rules has been proposed in [31–34]. When using skeleton rewriting transformations, a set of functionally equivalent programs exploiting different kinds of parallelism is obtained. Cost models and evaluation methodologies have also been proposed that can be used to determine the best of a set of equivalent parallel programs [13,34]. The methodology presented in this paper extends and builds on this and similar work by providing refactoring tool-support supplemented by a programming methodology that aims to make structured parallelism more accessible to a wider audience.

7 Conclusions and Future Work

This paper has described a new cost-directed refactoring approach for Erlang. The approach builds on pluggable algorithmic skeletons for Erlang, using cost models to derive performance information that can be used to choose among alternative parallel implementations. Benchmark results show that the cost information derived gives good predictions of parallel performance. While our work is described in terms of Erlang, the approach taken is generic: the refactorings demonstrate general principles that could easily be adapted for other languages, such as Haskell and C++. In particular, the cost models can be adapted to other settings and the refactoring rules given here are essentially generic. Our intention is that we will, in time, construct a generic refactoring environment capable of using a common set of refactoring rules to cover a range of different programming languages, parallel patterns and structured parallel implementations. In future we intend to extend the range of skeletons to cover parallel workpools, divide-and-conquer, map-reduce, bulk synchronous parallelism and other parallel patterns. Each of these patterns must be supported by corresponding cost models and refactoring rules. Finally, while we are confident that the transformations described here preserve functional correctness, we have not yet formally proved the correctness of our refactorings. Preliminary work on proving refactorings has been undertaken by Li and Thompson [35]. While there are complications concerning parallel and concurrent execution, we anticipate that their approach could be adapted to the setting described here.

Acknowledgments This work has been supported by EU Framework 7 Grants IST-288570 “ParaPhrase” (http://www.paraphrase-ict.eu), and IST-248828 “ADVANCE” (http://www.project-advance.eu).

References

1. Cesarini, F., Thompson, S.: ERLANG Programming, 1st edn. O’Reilly Media, Inc., Sebastopol, CA (2009)
2. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
3. Li, H., Thompson, S.: A Comparative Study of Refactoring Haskell and Erlang Programs. SCAM 2006, pp. 197–206. IEEE (2006)
4. Aronis, S., Sagonas, K.: On using Erlang for parallelization: experience from parallelizing dialyzer. In: Loidl, H.-W. (ed.) Trends in Functional Programming 2012 (TFP 2012). Springer, St Andrews (2012)
5. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.: A scalability benchmark suite for Erlang/OTP. In: Proceedings of 11th ACM SIGPLAN Workshop on Erlang Copenhagen, Denmark. pp. 33–42. ACM. NY, USA (2012)
6. Opdyke, W.: Refactoring object-oriented frameworks. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
7. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
8. Li, H., Thompson, S.: Let’s Make Refactoring Tools User Extensible! The Fifth ACM Workshop on Refactoring Tools. Rapperswill, Switzerland (2012)
9. Li, H., Thompson, S.: A domain-specific language for scripting refactorings in Erlang. In: Proceedings 15th International Conference on Fund. Approaches to Software Engineering, pp. 501–515. (2012)
10. Caromel, D., Leyton, M.: Fine tuning algorithmic skeletons. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par, vol. 4641. Lecture Notes in Computer Science. Springer, Rennes (2007)
11. Pelagatti, S.: Structured Development of Parallel Programs. Taylor and Francis, London (1999)
12. Aldinucci, M.: Automatic program transformation: the meta tool for skeleton-based languages. In: Gorlatch, S., Lengauer, C. (eds.) Constructive Methods for Parallel Programming, Advances in Computation: Theory and Practice, chap. 5, pp. 59–78. Nova Science, NY (2002)
13. Aldinucci, M., Coppola, M., Danelutto, M.: Rewriting Skeleton Programs: How to Evaluate the Data-Parallel Stream-Parallel Tradeoff, pp. 44–58. CMPP, Germany (1998)

14. Partsch, H., Steinbruggen, R.: Program transformation systems. ACM Comput. Surv. **15**(3), 199–236 (1983)
15. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
16. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
17. Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., Gonzalez-Velez, H., Kilpatrick, P., Keller, R., Natschlager, T., Shainer, G.: The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. FMCO, Turin (2012)
18. Hammond, K., Berthold, J., Loogen, R.: Automatic skeletons in template Haskell. Parallel Process. Lett. **13**(3), 413–424 (2003)
19. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. SIGPLAN Not. **37**, 60–75 (2002)
20. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. J. Funct. Program. **15**(3), 431–475 (2005)
21. Wloka, J., Sridharan, M., Tip, F.: Refactoring for Reentrancy. ESEC/FSE '09, pp.. 173–182. ACM, Amsterdam (2009)
22. Dig, D.: A refactoring approach to parallelism. IEEE Softw. **28**, 17–22 (2011)
23. Brown, C., Li, H., Thompson, S.: An expression processor: a case study in refactoring Haskell programs. In: Eleventh Symposium on Trends in Functional Programming (2010)
24. Brown, C., Loidl, H., Hammond, K.: Paraforming: forming Haskell programs using novel refactoring rechniques. In: 12th Symposium on Trends in Functional Programming, Spain (2011)
25. Brown, C., Hammond, K., Danelutto, M., Kilpatrick, P.: A language-independent parallel refactoring framework. In: Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12), pp. 54–58. ACM, New York, USA (2012)
26. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. InfoScale '06. Article 13, Hong Kong. ACM, NY, USA (2006)
27. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: a structured high level program language and its structured support. Concurr. Pract. Exp. **7**(3), 225–255 (1995)
28. Botorog, G.H., Kuchen, H.: Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In: Proceedings of the 5th International Symposium on High Performance Distributed Computing (HPDC '96), pp. 243–252. IEEE Computer Society Press, Syracuse, NY, USA (1996)
29. Cole, M.: Algorithmic skeletons: structured management of parallel computations. In: Research Monographs in Par. and Distrib. Computing. Pitman(1989)
30. Darlington, J., Guo, Y., Jing, Y., To, H.W.: Skeletons for Structured Parallel Composition. In: Proceedings of the 15th Symposium on Princ. and Prac. of Parallel Programming (1995)
31. Aldinucci, M., Danelutto, M.: Stream Parallel Skeleton Optimization. In: Proceedings of the International Conference on Parallel and Distributed Computing and Systems (PDCS), pp. 955–962. IASTED, ACTA Press, Cambridge, MA, USA (1999)
32. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithm Appl. **16**(2–3), 87–121 (2001)
33. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: Proceedings of the 13th International Symposium on Par. Proceedings and the 10th Symposium on Par. and Dist. Proc., pp. 492–499, Washington, DC, USA (1999)
34. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. J. Parallel Distrib. Comput. **28**(1), 65–83 (1995)
35. Li, H., Thompson, S.: Formalisation of Haskell refactorings. Trends Funct. Program. (2005)