Refactoring for Introducing and Tuning Parallelism For Heterogeneous Multicore Machines in Erlang

VLADIMIR JANJIC, CHRISTOPHER BROWN, ADAM BARWELL
University of St Andrews, School of Computer Science, UK

KEVIN HAMMOND
Hylomorph Solutions, Glasgow, UK
(*e-mail:* vj32@st-andrews.ac.uk, cmb21@st-andrews.ac.uk, adb23@st-andrews.ac.uk, kevin@kevinhammond.net)

Abstract

This paper presents semi-automatic software refactorings to introduce and tune structured parallelism in sequential Erlang code, as well as to generate code for running computations on GPUs and possibly other accelerators. Our refactorings are based on the LAPEDO framework for programming *heterogeneous* multi-core systems in Erlang. LAPEDO is based on the PaRTE refactoring tool and also contains i) a set of *hybrid skeletons* that target both CPU and GPU processors; ii) novel *refactorings* for introducing and tuning parallelism; and, iii) a tool to generate the GPU offloading and scheduling code in Erlang, that is used as a component of hybrid skeletons. We demonstrate, on four realistic use-case applications, that we are able to refactor sequential code and produce heterogeneous parallel versions that can achieve significant and scalable speedups of up to 220 over the original sequential Erlang program on a 24-core machine with a GPU.

1 Introduction

Following initial stages of multi-core revolution, another big change has happened in the computer hardware world. Emergence of *accelerators*, such as graphic processing units (GPUs), field programmable gate arrays (FPGAs) and Intel Xeon PHIs, has resulted in a new class of devices, *heterogeneous multi-cores/many-cores*. These systems combine traditional multi-core CPUs with one or more accelerators, and are nowadays found in almost all high-performance systems, desktops, clouds, gaming consoles, tablets and mobile phones. These systems offer a significant computing potential that can deliver orders of magnitude more performance than traditional CPU-only based systems, and almost all of them support execution of *general-purpose* parallel applications.

However, we know that nothing comes for free. In order to fully exploit the potential that the heterogeneous multi-core/many-core systems offer, a programmer needs to combine different low-level programming models, e.g. CUDA or OpenCL for GPUs, WHDL or Verilog for FPGAs and OpenMP for CPUs. This involves combining different models for CPUs and different accelerators, explicitly manage data transfers between main memory and accelerator memory, schedule computations on the accelerators, fetch the results back

etc. Moreover, solutions that perform well are usually tied to a specific heterogeneous architecture and cannot easily be ported, yielding problems in terms of e.g. increased maintainance and lack of longevity. This makes programming heterogeneous many-core systems much harder than is already very-hard task of programming multi-core CPUs. What is needed are high-level programming models that will hide both hardware complexity of such systems, and also different models required to program them, while still achieving the performance comparable to that of hand-crafted low-level code. Functional programming, allowing a very high-level of abstraction without having to worry about the actual program flow, sounds like an ideal fit to program these systems. However, bridging the gap between high-level of abstraction and a complexity of the underlying hardware and programming models for it has so far proven to be a too big obstacle.

In our previous work (Janjic *et al.*, 2016), we have introduced the novel LAPEDO framework for programming heterogeneous multi-core systems in Erlang. LAPEDO is based on *hybrid skeletons* that can be used to abstract across low-level programming details. In this paper, we extend this framework with novel *refactorings* (programmer-directed source-level transformations). The extended framework enables the programmer to firstly focus on finding the most appropriate parallel *structure* for their application, and then on ensuring good, portable performance across a range of heterogeneous multicores. Given a OpenCL kernel, which the programmer has written by hand, LAPEDO automatically generates the tedious and error-prone code for creating buffers on accelerators, transferring the data to/from them, setting the necessary parameters, scheduling the execution of computations on accelerators, fetching the results back and so on. Furthermore, it semi-automatically, under the programmer's guidance, introduces calls to instances of hybrid skeletons into the sequential code after, also semi-automatically, shaping the code in order to choose e.g. the most appropriate data structures for the underlying hybrid skeleton.

The paper makes the following main research contributions:

1. We introduce new refactorings for introducing hybrid parallelism in Erlang (Section 5.1) , including novel *program shaping* (Section 5.2) refactorings to transform between different data structures in order to enable/improve parallelism;

2. We introduce new mechanisms to automatically generate the complex and tedious code to offload computations to the accelerators, shedule their execution there and retrieve the results (Section 5.3);

3. We describe how these skeletons, refactorings and tools can be integrated with the existing Wrangler refactoring tool to give a comprehensive framework for programming heterogeneous multi-core systems (Section 4);

4. We demonstrate that the LAPEDO framework, extended with the above refactorings, allows us to produce efficient and scalable code for heterogeneous systems, achieving real speedups of up to 220 over sequential Erlang programs on a 24-core machine with a GPU (Section 6).

2 Parallel (Heterogeneous) Programming and Algorithmic Skeletons

2.1 Parallel Programming in Erlang

Erlang (Armstrong *et al.*, 1993) is a strict, impure, functional programming language with built-in support for concurrency. It supports a *lightweight* threading model, where *processes* model small units of computation. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. This model thus allows the programmer to be explicit about processes and communication, but implicit about their placement and synchronisation.

One key aspect of the Erlang design is that it adopts a *shared-nothing* approach[1], in which processes do not implicitly share state. This allows easier parallel decomposition and has advantages in terms of e.g. parallel memory management, but requires shared data to be explicitly identified. In the Erlang model, data is communicated via channels, using explicit *send* and *receive* primitives.

We build on the lower-level process mechanisms to provide higher-level structured parallelism abstractions, in the form of *algorithmic skeletons* (described in Section 2.2), supporting common patterns of parallel programming. By engaging with the Erlang process model rather than replacing it with a completely new scheduler, as is often done, we are able to exploit the existing scheduling and distribution mechanisms, while accelerating specific components of an Erlang program using GPUs or other accelerators. In particular, we can exploit all the usual Erlang distribution mechanisms to build highly-distributed and scalable systems, where individual nodes can exploit GPUs and other accelerators using the LAPEDO mechanisms.

2.2 Algorithmic Skeletons and the Skel *Library*

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction (Cole, 1988; Cole, 2004) into parameterised templates. There has been a long-standing connection between the skeletons community and the functional programming community. In the functional world, skeletons are effectively higher-order functions that can be instantiated with specific user code to give some concrete parallel behaviour. For example, we might define a *parallel map* skeleton, whose functionality is identical to a standard *map* function, but which creates a number of Erlang processes (*worker processes*) to execute each element of the map in parallel.

Using a skeleton approach allows the programmer to adopt a top-down *structured* approach to parallel programming, where skeletons are composed to give the overall parallel structure of the program. This gives a flexible semi-implicit approach, where the parallelism is exposed to the programmer only through the choice of skeleton and perhaps some specific behavioural parameters (e.g. the number of parallel processes to be created, or how elements of the parallel list are to be grouped to reduce communication costs). Details such as communication, task creation, task or data migration, scheduling etc. are embedded within the skeleton implementation, which may be tailored to a specific architecture or class of architectures. This offers an improved level of portability over the typical low-level

[1] With some key exceptions, which we will cover later.

approaches. However, it will still be necessary to tune behavioural parameters in particular cases, and it may even be necessary to alter the parallel structure to deal with varying architectures (especially where an application targets different classes of architecture). A survey of algorithmic skeleton approaches can be found at (González-Vélez & Leyton, 2010).

The *Skel* (Brown *et al.*, 2014c) library defines a small set (currently seven) of classical skeletons for Erlang. Each skeleton operates over a stream of input values, producing a corresponding stream of results. That is, the general type of a fully instantiated skeleton, *skel*, is:

$$skel \ : \ \ll a \gg \ \rightarrow \ \ll b \gg$$

here $\ll \ldots \gg$ denotes a stream. Because each skeleton is defined as a streaming operation, they can be freely substituted provided they have equivalent types. The same property also allows simple composition and nesting of skeletons. This paper will consider the following subset of the *Skel* skeletons, and builds upon the previous work of Janjic (Janjic *et al.*, 2016):

- `func` is a simple wrapper skeleton that encapsulates a function, $f : a \rightarrow b$, as a streaming skeleton:

  $$\{\texttt{func}, f\} \ \ll x_1, x_2, \ldots, x_n \gg \ = \ \ll f\ x_1, f\ x_2, \ldots, f\ x_n \gg \ .$$

  When applied to some concrete function, `func` yields a streaming function:

  $$\texttt{func} : (a \rightarrow b) \rightarrow (\ \ll a \gg \ \rightarrow \ \ll b \gg\ ).$$

  For example, in Skel `{func, fun M:f/1}` denotes a `func` skeleton wrapping the Erlang function, `f`, with `M` denoting the module in which `f` is defined, and `f/1` denoting the arity of `f` (the explicit arity is necessary in Erlang, as one can define functions with different arities). In the remainder of the text, we will denote the `func` skeleton simply by `{func, fun f}`. We also note that, in *Skel*, the function specified for use in a `func` skeleton must have an arity of 1.
- `pipe` models a parallel pipeline over a sequence of skeletons $s_1, s_2, \ldots, s_m$ as a streaming skeleton:

  $$\{\texttt{pipe}, s_1, \ldots, s_m\} \ \ll x_1, x_2, \ldots, x_n \gg \ =$$

  $$\ll s_m(\ldots(s_1\ x_1)), s_m(\ldots(s_1\ x_2)), \ldots, s_m(\ldots(s_1\ x_n)) \gg$$

  Within the pipeline, each of the s_i is applied in parallel to the result of the s_{i-1}. If each s_i has type $a_{i-1} \rightarrow a_i$, then the pipeline will yield a streaming skeleton as its result:

  $$\texttt{pipe} : [a_{m-1} \rightarrow a_m, \ldots, a_0, \rightarrow a_1] \ \rightarrow \ (\ \ll a_0 \gg \ \rightarrow \ \ll a_m \gg\ )$$

  For example, in Skel:

  ```
  {pipe, [{func, fun f1}, {func, fun f2}, {func, fun f3}]}
  ```

  denotes a parallel pipeline with three stages. Each pipeline stage is a `func` skeleton, wrapping the Erlang functions, `f1`, `f2`, and `f3` respectively.

- `farm` models a parallel task farm with n workers, whose basic operation is the skeleton $s: \ll a \gg \rightarrow \ll b \gg$.

$$\{\texttt{farm}, n, s\} \ \ll x_1, x_2, \ldots, x_n \gg \ = \ \mathit{perm} \ \ll s\ x_1, s\ x_2, \ldots, s\ x_n \gg$$

  Here the *perm* operation returns some *permutation* of the results. Each of the n workers operates in parallel over independent values of the input stream. The farm skeleton has the type:

$$\texttt{farm} : \mathit{Nat} \rightarrow (a \rightarrow b) \rightarrow (\ \ll a \gg \ \rightarrow \ \ll b \gg\ )$$

  For example, in Skel:

```
{farm, 10, {pipe, [{func, fun f}, {func, fun g}]}}
```

  denotes a `farm` where the worker is a parallel pipeline with two stages (each a `func` skeleton wrapper for the functions `f` and `g`, respectively). Here, `farm` has 10 workers, as specified by the second parameter, therefore running 10 independent instances of the pipeline skeleton.
- `cluster` is data-parallel skeleton, which has a similar semantics to the `farm` in that it applies the same operation to a stream of input elements:

$$\{\texttt{cluster}, s, \texttt{decomp}, \texttt{recomp}\} \ \ll x_1, x_2, \ldots, x_n \gg \ =$$

$$\ll s\ x_1, s\ x_2, \ \ldots, s\ x_n \gg \ .$$

  The parallel behaviour is, however, significantly different than that of the `farm`. The parallelism in the `cluster` skeleton does not arise from applying the same operation to the different elements of the input stream, as in case of the `farm`, but rather from *decomposing* each element of an input stream into *chunks* (using a `decomp` function), applying the same operation in parallel to each individual chunk, and *recomposing* the chunks of a result into final outputs (using the `recomp` function). Therefore, the `cluster` skeleton requires all of the input elements $x_1, x_2, \ldots, x_n$ to be decomposable into smaller instances of the same type (e.g. lists or binaries). In the case when `decomp` and `recomp` are identity functions and the input stream consists of only one input, we get the usual `map` skeleton, where the function s is applied to each element of an input data-structure in parallel.
  An example, in Skel:

```
fun decompFun(X) ->
  case X of
    [] -> [];
    _ -> {Chunk, Rest} = lists:split(10, X),
         [ Chunk | Decomp(Rest) ]
end.

Skel = {cluster, {pipe, [{func, fun f}, {func, fun g}]},
                fun decompFun/1, fun lists:flatten/1}
```

  `Skel` denotes a parallel `cluster` skeleton that is applied to a stream of lists. Each worker is a two-stage parallel pipeline, the `decompFun` function is used to decompose

each input list into sublists (chunks) of 10 elements, and the `lists:flatten` function is used to recompose the final list of results into a single list.

- `feedback` wraps a skeleton s : $\ll a \gg \ \rightarrow \ \ll a \gg$, feeding the results of applying s back as new inputs to s, provided they match a filter function, f : $a \rightarrow bool$.

$$\texttt{feedback}\ f\ s\ \ll x_1, x_2, \ldots, x_n \gg\ =$$

$$\ll s'(x_1), s'(x_2), \ldots, s'(x_n) \gg\ ,$$

where

$$s'(x) = \left\{ \begin{array}{l} x, \text{ if } f(x) \text{ is true} \\ s'(s\ x), \text{ if } f(x) \text{ is false} \end{array} \right\}.$$

Operationally, `feedback` will apply the inner skeleton to each input in turn, merging results from the inner skeleton ahead of any subsequent inputs where they match the filter, and returning any results that do not match the filter. An example, in *Skel*:

```
{feedback, fun h, {pipe, [{func, fun f}, {func, fun g}]}}
```

denotes a `feedback` skeleton with a constraint checking function `h`, and a two-stage pipeline acting as an inner workflow. Elements of the input stream are passed through the inner `pipeline` workflow if the function `h` returns true when applied to the elements, otherwise they are released from the `feedback`.

Internally, *Skel* is implemented in terms of standard Erlang processes and channels, linked according to the skeleton structure, and using the Erlang "mailbox" to buffer messages between skeleton instances. At the top level, the `skel:do` function is used to generate a streaming skeletal program from a description of the skeletons and their parameters:

$$\texttt{skel:do}\ :\ \mathit{Workflow} \rightarrow (\ \ll a \gg\ \rightarrow\ \ll b \gg\ )$$

A *workflow* is an abstract description of a skeleton, that is used to generate the skeleton for the program as a whole. It is defined in terms of variable-sized tuples, whose first element is an atom that specifies the skeleton to be used, e.g. `func`, `map`, `farm`, or `pipe`; and whose remaining elements specify the skeleton parameters. Within a workflow description, wherever a single skeleton is allowed we will also use lists as syntactic sugar for the `compose` skeleton. For example,

```
skel:do({farm, 10, {pipe, [{func, fun f}, {func, fun g}]}}, Input).
```

This defines a farm with ten workers. Each worker is defined as a sequential composition of two skeletons, basic `funcs`, whose workers are respectively f and g. The `fun f` indicates that f is a function that is defined in the same module as the call to `skel:do`, i.e. equivalent to an undecorated partially-applied f in Haskell. The skeleton is applied to the input stream, *Input*. If *Input* has m elements, then the result will be:

$$\ll g(f\mathit{Input}_1),\ \ldots,\ g(f\mathit{Input}_m) \gg$$

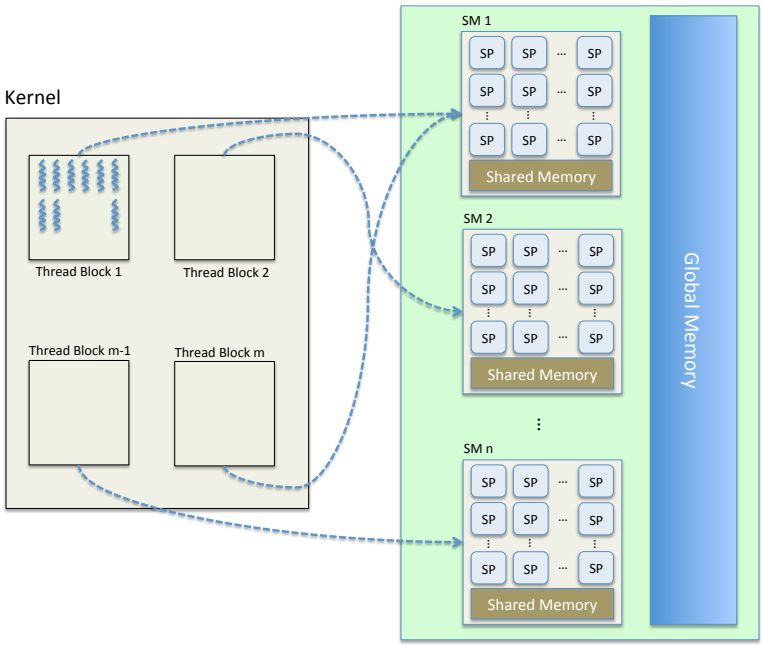


Fig. 1. Mapping of a GPU Kernel to the GPU device.

2.3 Heterogeneous Computing

Heterogeneous multicores comprise a mixture of CPUs and *accelerators*. These accelerators may be manycore processors, such as the Intel Xeon Phi, repurposed graphics processing units (GPUs), such as the NVidia Tesla, dedicated devices, e.g. Digital Signal Processors (DSPs), such as MSC81xx, or even "soft processors", e.g. Field Programmable Gate Arrays (FPGAs), such as the the Xilinx Virtex UltraScale. Our main focus in this paper is on the systems that comprise a multicore CPU and one or more GPUs; however, since the target code we generate uses OpenCL, the methodology is, in principle applicable to other classes of accelerators, although it might require additional tuning.

Compared with CPUs, GPUs offer higher performance at lower clock speeds. They also deliver advantages in terms of energy usage per unit of performance (but often have higher total energy usage). However, GPUs are restricted in terms of the parallelism model that is offered (data-parallel only), and are siginificantly more difficult to program than traditional CPU-only systems. Furthermore, GPUs typically only offer restricted memory sizes, increasing the effort that is required for large or complex problems, and potentially mitigating any performance gains. The recent class of Accelerated Processor Units (APUs), such as the AMD Mullins A10 and NVidia Jetson TX2, attempts to alleviate this by providing shared-memory access between the CPU and GPUs (and other on-board accelerators) as part of a single System-On-Chip. The most sophisticated current APUs (e.g. the AMD Mullins and Zynq Ulstrascale+ MPSoC) include CPUs and GPUs within the same device.

Conventional Approaches to GPU Programming The two most widely-used approaches to GPU programming, CUDA and OpenCL both aim to simplify the problem of program-

ming GPUs, providing similar portable, but low-level SIMD programming interfaces that can be executed on either CPUs or GPUs. In each case, the basic unit of GPU computation is a *kernel*, which corresponds to a function that contains inherent data-parallelism, and which is executed by each individual logical thread. Logical threads are grouped into *thread blocks*, which are mapped to the symmetric multiprocessors (SMs) of the GPU (see Figure 1). Threads within one thread block are scheduled automatically on their associated SM. Below is an example of a simple OpenCL kernel for adding two vectors of real numbers:

```
__kernel void vecAdd( __global double *a,
                      __global double *b,
                      __global double *c,
                      const unsigned int n)
{
    // get global thread id
    int id = get_global_id(0);
    // make sure we do not go out of bounds
    if (id < n)
       c[id] = a[id] + b[id];
}
```

When executing this kernel on a GPU, each GPU thread executes the kernel code independently. Different threads will get different ids assigned to them (variable `id`), and will, therefore, compute different elements of the output vector `c`.

Unfortunately, *programmability* is still generally lacking with CUDA and OpenCL: having initially restructured (or written) the program into a data-parallel form, suitable for SIMD execution, the programmer then needs to take care of a number of very low-level programming aspects, including the number of threads and thread blocks, data transfers between CPUs and GPUs, scheduling of computations on the GPUs, division of the work between CPUs and GPUs etc. For example, even the simple kernel above requires 100-150 lines of OpenCL code just for managing its execution on a GPU. This requires the programmer to have a deep understanding not only of the problem that is being solved, but also of the underlying hardware architecture. This usually results in a solution that is heavily optimised for a particular hardware system and which, therefore, lacks *performance portability*. Therefore, the code needs to be refactored for each new architecture. The GPU code is also often highly fragile and likely to be error-prone.

The LAPEDO framework aims to overcome the deficiencies of GPU programming models by exploiting functional language features and principles to provide high-level abstractions over different heterogeneous processor types, and a variety of parallel structure.

2.4 GPU Programming in Erlang

Erlang has no native support for GPU programming. However, a library containing OpenCL bindings is available[2] (Rogvall, n.d.). This provides an Erlang interface to low-level OpenCL functions to set up GPU computations, transfer data to/from the CPU, and launch GPU

[2] Available to download at `https://github.com/tonyrog/cl`

kernels implemented in OpenCL plus basic marshalling mechanisms between *binary* data structures in Erlang and C arrays. An example Erlang code that uses this library is given in 5.3. While enabling programmers to write their code in Erlang, this library does not simplify GPU programming, since the programmer is still required to write code that is equivalent to programming directly in OpenCL. In the LAPEDO framework, we build on this library and automate the process of generating the bookkeeping OpenCL code.

3 Refactoring Tool Support

Refactoring is the process of changing the internal structure of a program, while preserving its (functional) behaviour. In contrast to general program transformations, refactoring focuses on purely structural changes rather than on changes to program functionality, and is generally applied semi-automatically (i.e. under programmer direction), rather than fully automatically. This allows programmer knowledge about e.g. safety properties to be exploited, and so permits a wider range of possible transformations. Refactoring has many advantages over traditional transformation and fully automated approaches, including (but not limited to):

- Refactoring is aimed at improving the design of software. As programmers change software to meet new requirements, so the code loses structure; regular refactoring helps tidy up the code to retain a good structure (Fowler, 1999).
- Refactoring makes software easier to understand. Programmers often write software without considering future developers. Refactoring can enable the code to better communicate its purpose (Fowler, 1999).
- Refactoring encourages code reuse by removing duplicated code (Brown & Thompson, 2010).
- Refactoring helps the programmer to program faster and more effectively by encouraging good program design (Fowler, 1999).
- Refactoring helps the programmer to reduce bugs. As refactorings typically generate code automatically, it is easy to guarantee that this code is safe and correct (Sultana & Thompson, 2008).

The term "refactoring" was first introduced by Opdyke in his 1992 PhD thesis (Opdyke, 1992), but the concept goes back at least to Darlington and Burstall's 1977 *fold/unfold* transformation system (Burstall & Darlington, 1977), which aimed to improve code maintainability by transforming Algol-style recursive loops into a pattern-matching style commonly used today. Historically, most refactoring was performed manually with the help of text editor "search and replace" facilities. However, in the last couple of decades, a diverse range of refactoring tools have become available for various programming languages, that aid the programmer by offering a selection of automatic refactorings. For example, the most recent release of IntelliJ IDEA refactorer supports 35 distinct refactorings for Java (Fields *et al.*, 2006). Typical refactorings include *variable renaming* (changes all instances of a variable that are in scope to a new name), *parameter adding* (introduces a new parameter to a function definition and updates all relevant calls to that function with a placeholder), *function extraction* (lifts a selected block of code into its own function) and *function inlining*.

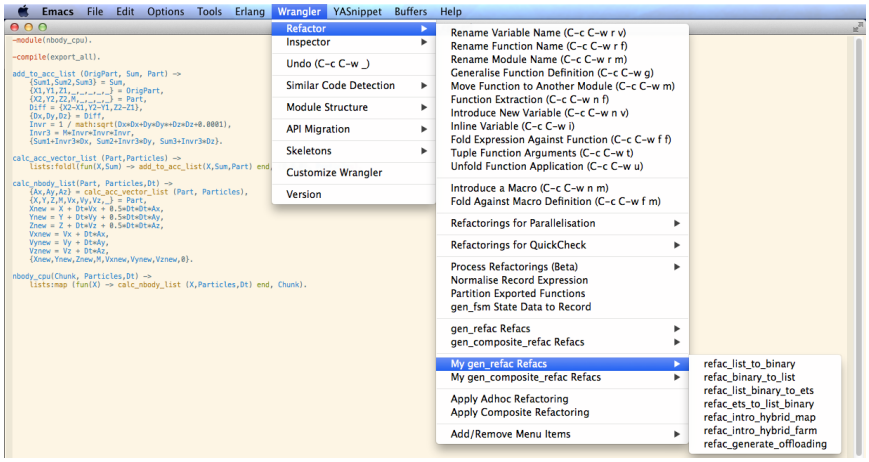


Fig. 2. Program Shaping and Hybrid Skeleton Refactorings in Wrangler.

3.1 Refactoring for Functional Languages

Whilst the refactoring community has produced a great deal of work on refactoring for the object-oriented paradigm (Dig *et al.*, 2009), the concept is nevertheless applicable to a wide range of programming styles and approaches. Functional programming is no exception to this. Indeed, Darlington and Burstall's transformation system for recursive functions produces code that would not be out of place in modern functional programs (Burstall & Darlington, 1977). The University of Kent has since produced the HaRe (Brown *et al.*, 2010) and Wrangler (Li *et al.*, 2008) refactoring tools for Haskell and Erlang respectively. Both tools are implemented in their respective languages, and offer a number of standard refactorings. Wrangler is implemented in Erlang and is integrated into both Emacs and Eclipse (Figure 2). We exploit a recent Wrangler extension which allows refactorings to be expressed as AST traversal strategies in terms of their pre-conditions and transformation rules.

3.2 Refactoring for Parallelism Introduction and Tuning

Some recent work has attempted to exploit refactoring technology to introduce and tune parallelism. For example, we have developed prototype refactorings for parallel Haskell using the HaRe (Brown *et al.*, 2010) system, and also extended Wrangler with a number of novel refactorings to introduce and tune parallelism using skeletons via the Skel library (Brown *et al.*, 2014c). We have extended this work to introduce and tune skeletons in C/C++ (Brown *et al.*, 2014a) using the FastFlow skeleton library, embedding our refactorings in the Eclipse IDE. In addition to the general advantages above, there are a number of specific advantages to using refactoring for parallelism introduction and tuning:

- Using a refactoring tool to introduce skeletons (instead of manual insertion) means that the programmer has to understand and remember less. This enables them to concentrate more on the design of the program.

- A refactoring tool can help minimise changes between different releases or implementations of skeletons.
- *By design*, a refactoring tool will not allow a user to break their programs. Introducing an incorrect skeleton, for example, is simply not allowed.

These advantages are exploited by LAPEDO (described in more detail in Section 4). In this paper, we will also take the concept of parallel refactoring further, however: by introducing novel refactorings that introduce *hybrid skeletons* for heterogeneous systems, and by using refactoring for *program shaping*.

3.2.1 The PaRTE Refactoring Tool for Erlang.

The prototype refactorings presented in this paper have been implemented in the *Paraphrase Refactoring Tool for Erlang* (PaRTE). PaRTE integrates capabilities of the RefactorErl (Bozo *et al.*, 2011) and Wrangler (Li *et al.*, 2008) refactoring/program analysis tools into a new parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns (Bozó *et al.*, 2014). Both the pattern candidate detection/assessment and semantics-preserving transformation steps require thorough syntactic and semantic analysis. This is achieved by exploiting the RefactorErl and Wrangler refactoring tools, which implement specific compile-time analyses, and support syntax-based transformations. RefactorErl implements a wide range of static semantic analyses, including scope analysis of various language entities, side-effect analysis and callee approximation of dynamic function calls. Most of these (higher-level) semantic analyses build on results from dataflow and type analyses. Based on the information uncovered by the semantic analyses, RefactorErl can determine non-trivial properties of code fragments. In the integrated PaRTE framework, it is therefore used to perform pattern discovery and evaluation. Conversely, Wrangler has a mature user interface for performing simple code rewriting, which makes it a good choice for a tool to carry out parallelisation transformations. In the integrated PaRTE framework, Wrangler is responsible for defining and executing shaping transformations, and for turning sequential computations into instances of parallel algorithmic skeletons.

Figure 2 shows PaRTE 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. PaRTE 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. PaRTE 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.

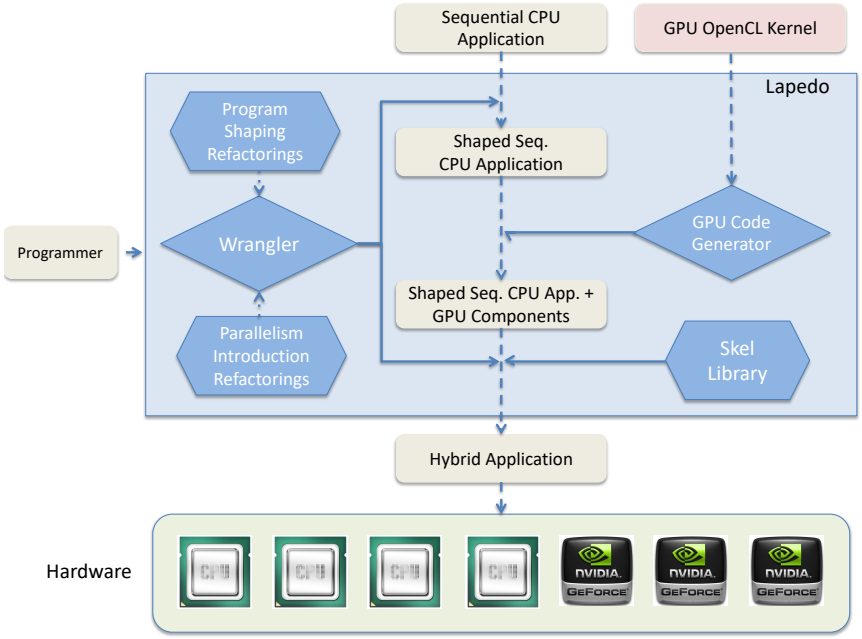


Fig. 3. Overview of the process of parallelisation of Erlang programs using LAPEDO.

6. The source code in the editor is automatically changed to reflect the refactored program.

This process is also described in (Brown *et al.*, 2014c).

4 The LAPEDO Framework

LAPEDO aims to guide the programmer in the process of converting a sequential or concurrent Erlang program into an equivalent parallel version that can be executed efficiently on a heterogeneous multicore system. While LAPEDO primarily targets CPU+GPU combinations, it can also, in principle, be used for other kinds of heterogeneous system that support the OpenCL model (e.g. ones with FPGAs and/or Xeon Phis as well). LAPEDO is built around the PaRTE refactoring tool (Section 3.2.1) and the *Skel* library (Section 2.2). Three main novelties that LAPEDO brings to these tools, and which enable programming of heterogeneous machines, are:

- *hybrid skeletons* that can have both CPU and GPU components (Section 4.1);
- new refactorings for introduction of hybrid skeletons and program shaping to convert the data into the format suitable for processing on GPUs (Section 5.1);
- a *GPU Code Generation* tool that generates the code for GPU components of hybrid skeletons (Section 5.3).

Figure 3 provides an overview of the process that is involved in transforming an existing Erlang application into a parallel equivalent using LAPEDO. This method is described in a EU deliverable that was released as part of the FP7 ParaPhrase Project (Brown, 2015)

in more detail. The programmer starts with a sequential (or concurrent) CPU application and an OpenCL kernel that provides a GPU implementation for suitable operations in the application. The programmer then performs the following steps (which may be repeated if necessary):

1. *Program shaping.* The programmer first uses Wrangler to apply any necessary program shaping transformations. For example, as described in Section 2.4, the basic Erlang OpenCL bindings only support the transfer of binary data to the GPU, so any data-structure that is to be used by the GPU kernel must be converted into binaries, together with the associated operations. These, and other program shaping transformations, are described in more detail in Section 5.2. This step is semi-automatic, requiring the programmer to select the regions of the code that need to be transformed, and to choose the appropriate transformations;
2. *GPU code generation.* In the next step, the programmer invokes the *GPU Code Generator* to generate Erlang code for the GPU components of the future hybrid application. This generates code to link with and execute GPU kernels (e.g. to manage data transfers to/from the GPU and to schedule the GPU computations), based on programmer-supplied information about the kernels. The GPU Code Generator is described in more detail in Section 5.3. This step is semi-automatic, requiring the programmer to supply information about the GPU kernels;
3. *Introduction of hybrid patterns.* Finally, the programmer uses Wrangler again to refactor the shaped CPU code in order to introduce hybrid skeletons. This results in a hybrid application that uses the GPU components that have been generated in the previous step. Hybrid skeletons are described in more detail in Section 4.1, and the refactorings that introduce them are described in Section 5.1. This step is semi-automatic, requiring the programmer to select the regions of the code that can be parallelised using high-level parallel patterns and to choose the appropriate hybrid pattern from the *Skel* library.

We will use the parallelisation of an *N-Body Simulation* to illustrate how the LAPEDO framework can be used in practice. This application is a simulation of a dynamic system of particles under the influence of external forces. It is commonly used in astronomy, for example, to simulate movements of the planets. The simulation proceeds in a number of iterations, each of which computes a new position for each particle, based on its interactions with all the other particles in the system. The core of the application is the `nbody` function:

```
nbody(Parts,Dt,0) -> Parts;
nbody(Parts, Dt, NrIters) ->
  NewParts = lists:map (fun(X) -> nbody_one_part_cpu (X,Parts,Dt) end, Parts),
  nbody(NewParts, Dt, NrIters-1).
```

This function applies the `nbody_one_part_cpu` function to each element of the input list of particles, `Part`. `Dt` is a constant that is passed to the `nbody_one_part_cpu` function. The `nbody_one_part_cpu` function is given below:

```
nbody_one_part_cpu(Part, Particles,Dt) ->
    {Ax,Ay,Az} = calc_acc_vector_list (Part, Particles),
    {X,Y,Z,M,Vx,Vy,Vz,_} = Part,
```

```
    Xnew = X + Dt*Vx + 0.5*Dt*Dt*Ax,
    Ynew = Y + Dt*Vy + 0.5*Dt*Dt*Ay,
    Znew = Z + Dt*Vz + 0.5*Dt*Dt*Az,
    Vxnew = Vx + Dt*Ax,
    Vynew = Vy + Dt*Ay,
    Vznew = Vz + Dt*Az,
    {Xnew,Ynew,Znew,M,Vxnew,Vynew,Vznew,0}.
```

The auxiliary `calc_acc_vector_list` function is a simple fold over the list of particles:

```
calc_acc_vector_list (Part,Particles) ->
    lists:foldl(fun(X,Sum) -> add_to_acc_list(X,Sum,Part) end,
                {0,0,0}, Particles).
```

We omit the definition of the `add_to_acc_list` function. We also assume that we have available an OpenCL kernel `nbodyKernel` that applies the `nbdoy_one_part_cpu` function to a chunk of particles. We can see that the main part of the code (in the `nbody` function) involves applying the same function to a list of particles. This suggests that it can be parallelised by using a `farm` or `cluster` skeleton. Since each individual computation on particles is relatively small but the total number of particles involved is potentially large, we have chosen to use a `chunking` pattern (so as to group particles into coarse-grained chunks that will be processed sequentially by worker processes), and use LAPEDO for parallelisation as follows:

1. We first need to perform some *program shaping* transformations in order to prepare the code for the introduction of the *chunking* pattern. Since the idea is to split the initial list of particles into a list of lists (chunks) and then apply sequentially the `nbody_one_part_cpu` function to these chunks, `nbody_one_part_cpu` will need to be applied to chunks of particles, rather than to just the individual particles. However, since we have a GPU kernel that calculates `nbody_one_part_cpu` for a chunk of particles, and since GPU kernels can only operate on binary data, our instance of the `chunk` skeleton needs to work on binaries rather than with lists. We, therefore, apply the *ListChunking* and *ListToBinary* program shaping refactorings from Section 5.2 to modify the `nbody_one_part_cpu` to: i) operate on chunks of particles; and ii) operate on binary data. Since Erlang binaries are essentially just streams of bytes, we need to specify the size (in bytes) of each data item (particle in our case), so that we can correctly apply the binary *map* and *fold* functions. The resulting code introduces the `nbody_chunk_cpu` function to apply `nbody_one_part_cpu` over individual chunks of the input data:

   ```
   nbody_chunk_cpu(Chunk, Particles,Dt) ->
     binary8Map(fun nbody_one_part_cpu/3, Chunk).
   ```

   The `binary8Map` function maps over a binary object where each data item is precisely 8 bytes long. Since it contains no explicit operations over binaries, the body of the `nbody_one_part_cpu` function remains unchanged. However, the `calc_acc_vector_list` must be changed to use a binary fold rather than the original list version:

   ```
   calc_acc_vector_list (Part,Particles) ->
       binary8Foldl(fun(X,Sum) -> add_to_acc_list(X,Sum,Part) end,
   ```

```
                {0,0,0}, Particles).
```

As with `binary8Map`, `binary8Foldl` operates over 8-byte wide binary data.

2. We can then invoke the GPU Code Generator to generate a wrapper around the GPU kernel, creating a GPU component, `nbody_chunk_gpu`, whose functionality is identical to that of `nbody_chunk_cpu`. Certain parameters, such as `K#kwork.localSize` and `K#kwork.globalSize`, cannot be automatically generated at present, however, and therefore need to be supplied by the programmer. The generated code is shown below:

```
nbody_chunk_gpu(Chunk, Particles, Dt) ->
    %% set up the environment
    E = clu:setup(all),
    {ok,Program} = clu:build_source(E, program(ok)),
    {ok,Kernel} = cl:create_kernel(Program, "nbodyKernel"),

    ...

    %% create buffers
    {ok,ChunkInBuffer} = cl:create_buffer(E#cl.context,[read_only],
                                          byte_size(Chunk)),
    {ok,ParticlesBuffer} = cl:create_buffer(E#cl.context,[read_only],
                                            byte_size(Particles)),
    {ok,ChunkOutBuffer} = cl:create_buffer(E#cl.context,[read_only],
                                           byte_size(Chunk)),

    ...

    %% data transfers
    {ok,E1} = cl:enqueue_write_buffer(K#kwork.queue,
                                      ChunkInBuffer,
                                      0, byte_size(Chunk),
                                      Chunk, []),
    {ok,E2} = cl:enqueue_write_buffer(K#kwork.queue,
                                      ParticlesInBuffer,
                                      0, byte_size(Particles),
                                      Particles, []),

    ...

    %% Set kernel arguments
    ok = cl:set_kernel_arg(Kernel, 0, ChunkInBuffer),
    ok = cl:set_kernel_arg(Kernel, 1, ParticlesInBuffer),

    ...

    %% enqueue the kernel
    Global = K#kwork.globalSize,
    Local = K#kwork.localSize,
    {ok,E6} = cl:enqueue_nd_range_kernel(K#kwork.queue,
                                         Kernel,
                                         [Global], [Local],
```

```
                                        [E1,E2]),

    ...

    %% read back from the GPU
    {ok,E7} = cl:enqueue_read_buffer(K#kwork.queue,
                                ChunkOutBuffer,
                                0, byte_size(Chunk), [E6]),
    Result = case cl:wait(E7,3000) of
            {ok, Data} ->
                Data;
            Res3 ->
                Res3
            end,

    ...

    %% return the result
    Result.
```

3. Finally, since we now have the CPU and GPU components (`nbody_chunk_cpu`) and `nbody_chunk_gpu`), and both operate over binary chunks, we can introduce the *cluster* skeleton. We apply the *IntroduceHybridCluster* refactoring from Section 5.1 to obtain the following parallel version of the main function.

```
Struct = {hyb_cluster, [{hyb_map, [{func, fun(X) ->
            nbody_cpu(X,Particles,Dt) end}],
                        [{func, fun(X) ->
                         nbody_gpu(X, Particles, Dt) end}]}],
            TimeRatio, fun struct_size/1, fun make_chunk/2,
            fun flatten/1,
            NrCPUWs, NrGPUWs},
Res = skel:do([Struct],[Particles]).
```

4. This code can then undergo further refactorings to make it more readable., using the Wrangler built in refactorings, such as *Function Extraction* (Li *et al.*, 2008). For example, extracting out the CPU and GPU components into top-level functions, as shown below. Further refactorings could also be applied to lift other definitions simplifying the code further, if desired.

```
Struct = {hyb_cluster, [{hyb_map, [{func, fun ncpu/1}],
                                  [{func, fun ngpu/1}]}],
            TimeRatio, fun struct_size/1, fun make_chunk/2,
            fun flatten/1,
            NrCPUWs, NrGPUWs},
Res = skel:do([Struct],[Particles]).
```

4.1 Hybrid Skeletons in Lapedo

Hybrid skeletons that Lapedo provides were described in detail in (Janjic *et al.*, 2016). Here, we just give a brief overview of them, relevant to the refactorings presented in Sections 5. The skeletons provided are:

- *Hybrid Farm* represents a hybrid variant of a *farm* skeleton, where the same operation is applied to a set of independent input values. The hybrid variant has different implementations of the operation for each of the different types of processors in the system. Currently, CPUs and GPUs are supported as the processor types. We support either explicit or implicit distribution of work between different kinds of workers, as described in (Janjic *et al.*, 2016). The syntax of the hybrid farm skeleton is

```
{hyb_farm, CPUWorkflow, GPUWorkflow, [NCPUWorkers, NGPUWorkers]}
```

  where `CPUWorkflow` and `GPUWorkflow` represent CPU and GPU implementations of the farm operation, and optional parameters `NCPUWorkers` and `NGPUWorkers` denote the number of CPU and GPU workers, in the case that the work needs to be explicitly divided between the two worflows (using `NCPUWorkers:NGPUWorkers` ratio between work executed on CPU and GPU).
- *Hybrid Cluster* skeleton accepts an input stream of tasks, decomposes each task into chunks of subtasks and passes them to one of the two inner workflows (CPU or GPU workflow, similar as in the hybrid farm skeleton). There are several different variants of the `hyb_cluster` skeleton, differing in whether the programmer provides the decomposition and recomposition functions, otherwise these are automatically provided by the library. In the paper, we consider two versions:
  — `{hyb_cluster, CPUfl, GPUfl, MinChSz, Ratio, NCPUW, NGPUW}`, where each element of an input stream is a list and `CPUfl` and `GPUfl` are CPU and GPU workflows that process list chunks (sublists). `MinChSz` is the minimum size of a list chunk, which can be chosen so that the `GPUfl` executed on a chunk of that size gives the maximum parallelism (e.g. `MinChSz` can be chosen to be a maximum number of list items that the GPU can process at the same time). `Ratio` is the ratio between the time it takes to process a chunk of `MinChSz` using `CPUfl` and `GPUfl`, and `NCPUW` and `NGPUW` are number of processes executing `CPUfl` and `GPUfl` workflows. In the case where there is no nesting, i.e. where `hyb_cluster` is the top-level skeleton and `CPUfl` and `GPUfl` are `func` skeletons, `NCPUW` and `NGPUW` should be set to the number of CPU cores and the number of physical GPU devices available in the system, respectively.
  — `{hyb_cluster, CPUfl, GPUfl, Ratio, SzFun, MkChFun, RecompFun}`. This version does not assume that the elements of an input stream are lists. Here, programmer provides functions for determining the size of an element (`SzFun`), creating a chunk of work with a specified size from an element of an input stream (`MkChFun`) and recomposing a list of chunks into a data structure of the output element (`RecompFun`). This skeleton uses a generic decomposition function, that successively calls `MkChFun` to decompose an element of an input stream into a list of chunks, tagging them in the process and passing them to the inner workflows.

5 LAPEDO Refactorings for Hybrid Skeletons and Program Shaping

All refactorings given in this paper will be described as semi-formal rewrite rules, operating over the abstract syntax tree (AST) of the source program. These definitions reflect the implementation of the below refactorings in Wrangler. Since Wrangler allows refactorings to be undone as part of its standard functionality, only the rules for introducing patterns, or otherwise applying the desired transformations, are described and not their inverse. Each refactoring has a set of conditions ensuring that the transformation is valid, a description of the syntax to be transformed, and a description of the syntax following successful transformation. Conditions are given as predicates to each rule.

Each rewrite rule operates within an environment, γ, allowing access and reference to the current scope of the rewrite rule within the source program. This includes a set of all available functions, $\mathbb{F}$. The skeleton library Skel, and the skeletons it provides, introduced as part of some below refactorings, are denoted by the set $\mathbb{S}$.

$$\mathbb{S} = \{skel, \bar{f}, pipe, farm, farm', farm'', cluster, cluster', cluster''\}$$

For all rewrite rules, $\mathbb{S}$ is assumed to be in scope; to ensure this, we extend γ:

$$\Gamma = \gamma \cup \mathbb{S}$$

We next define a series of semantic equivalences to allow for more concise rewrite rules. Each equivalence is subject to a series of predicates under which it is valid, and is defined in the form:

$$\bar{s}, xs \in \Gamma, xs : list\ a \quad \vdash \quad skel(\bar{s}, xs) \quad = \quad \texttt{skel} : \texttt{do}(\bar{s}, xs)$$

Where $\bar{s}$ represents any valid skeleton in $\mathbb{S}$, i.e. $\mathbb{S}/\{skel\}$; and xs evaluates to a list where all elements have the same type. Semantic equivalences are defined for `func`, `pipe`, `farm`, and `cluster` skeletons.

$$\begin{array}{rcll}
\Gamma, f \in \mathbb{F} & \vdash & \bar{f} & = \{\texttt{func}, \mathscr{F}_f\} \\
\bar{s}_1, \bar{s}_2 \in \Gamma & \vdash & pipe(\bar{s}_1, \ldots, \bar{s}_2) & = \{\texttt{pipe}, [\bar{s}_1, \ldots, \bar{s}_2]\} \\
\bar{s} \in \Gamma, n \in \mathbb{N}^+ & \vdash & farm(\bar{s}, n) & = \{\texttt{farm}, \bar{s}, n\} \\
\bar{s} \in \Gamma, p, c \in \mathbb{F} & \vdash & cluster(\bar{s}, p, c) & = \{\texttt{cluster}, \bar{s}, \mathscr{F}_p, \mathscr{F}_c\}
\end{array}$$

As Erlang functions may be referenced by variable, name, or definition of anonymous function, we denote $\mathscr{F}_f$ to be any valid reference to a given function f. Semantic equivalences for both the new hybrid farm and and hybrid cluster skeletons are defined thus:

$$\begin{array}{rcll}
\bar{s}_c, \bar{s}_g \in \Gamma & \vdash & farm'(\bar{s}_c, \bar{s}_g) & = \{\texttt{hyb_farm}, \bar{s}_c, \bar{s}_g\} \\
\bar{s}_c, \bar{s}_g \in \Gamma, n, m \in \mathbb{N}^0, n+m > 0 & \vdash & farm''(\bar{s}_c, \bar{s}_g, n, m) & = \{\texttt{hyb_farm}, \bar{s}_c, \bar{s}_g, n, m\}
\end{array}$$

$$\begin{array}{rcl} \bar{s}_c, \bar{s}_g \in \Gamma, & & \\ n \in \mathbb{N}^+, & & \\ x, y \in \mathbb{N}^0, & \vdash \quad cluster'(\bar{s}_c, \bar{s}_g, n, x, y, m) & = \quad \{\texttt{hyb\_cluster}, \bar{s}_c, \bar{s}_g, n, x, y, m\} \\ x + y > 0, & & \\ m \in \mathbb{R}^+ & & \end{array}$$

$$\begin{array}{rcl} \bar{s}_c, \bar{s}_g \in \Gamma, & & \\ g, h, i \in \mathbb{F}, & \vdash \quad cluster''(\bar{s}_c, \bar{s}_g, n, g, h, i) & = \quad \{\texttt{hyb\_cluster}, \bar{s}_c, \bar{s}_g, \mathscr{F}_g, \mathscr{F}_h, \mathscr{F}_i\} \\ n \in \mathbb{R}^+ & & \end{array}$$

Using these semantic equivalences, rewrite rules are defined for each refactoring introduced in the following sections.

5.1 Refactorings to Introduce Hybrid Skeletons

The *Introduce Hybrid Map* (which we also sometimes refer to as *IntroduceHybridCluster*) refactoring is used to insert an instance of a hybrid cluster.

$$\begin{array}{lcll} \Gamma, \bar{s}_g \in \mathbb{S}, n \in \mathbb{N}^+, x, y \in \mathbb{N}^0, m \in \mathbb{R}^+ & \vdash \quad \bar{f} & \mapsto & cluster'(\bar{f}, \bar{s}_g, n, m, x, y) \\ \Gamma, \bar{s}_g \in \mathbb{S}, n \in \mathbb{R}^+, g, h, i \in \mathbb{F} & \vdash \quad \bar{f} & \mapsto & cluster''(\bar{f}, \bar{s}_g, n, g, h, i) \end{array}$$

HybMapIntroSeq requires an environment, Γ; a skeleton, $\bar{s}_g$, that has been generated using the GPU offloading generator from Section 5.3; the number of CPU workers, x; the number of GPU workers, y; the minimum chunk size n; and the CPU to GPU ratio m, which could be generated using, e.g., the work division algorithm described in (Janjic *et al.*, 2016). The rule matches the highlighted expression, $\bar{f}$, and transforms it into a `hyp_cluster` skeleton with appropriate arguments. The pre-conditions verify: i) that the appropriate functions are in scope (i.e., defined within the environment, Γ); ii) that the number of CPU and GPU workers may individually be zero, but must sum to a strictly positive natural number; iii) that the minimum chunk size is also a strictly positive natural number; and iv) that the CPU to GPU ratio is a strictly positive real number. The second *Introduce Hybrid Map* rewrite rule acts in a similar way, but serves to instead introduce the second form of the Hybrid Cluster skeleton.

The *Introduce Hybrid Farm* refactoring is similar to *Introduce Hybrid Map* and it is used to insert an instance of a hybrid farm.

$$\begin{array}{rcll} \Gamma, \bar{s}_g \in \mathbb{S} & \vdash \quad \bar{f} & \mapsto & farm'(\bar{f}, \bar{s}_g) \\ \Gamma, \bar{s}_g \in \mathbb{S}, n, m \in \mathbb{N}^0 & \vdash \quad \bar{f} & \mapsto & farm''(\bar{f}, \bar{s}_g, n, m) \end{array}$$

It requires an environment, Γ, a skeleton $\bar{s}_g$ that has been generated using the GPU offloading generator from Section 5.3, and optionally the number of CPU and GPU workers, n and m. Unlike in the previous *Introduce Hybrid Map* refactoring, *Introduce Hybrid Farm* does *not* require user-supplied minimum chunk size or CPU to GPU ratio values.

Note that the pre-conditions listed above do not include checks to ensure that introduction of the farm skeleton does not also introduce race conditions in the code. These checks are outside of the scope of this paper and we currently rely on the programmer to ensure that the portion of the code where the farm will be introduced is, indeed, side-effects free.

In similar refactorings that we have implemented for C++ in other work (Brown *et al.*, 2019), we have a technolgy that checks for race conditions in loops that are turned into skeletons and, furthermore, in some cases suggests the code transformation that can avoid these race conditions. A similar technology could be applied to Erlang.

5.2 Refactorings for Tuning and Type Translation

These refactorings do not introduce any parallelism themselves, but facilitate tuning of parallelism and prepare the original source so that it is easier to introduce parallelism. Examples may include chunking inputs and altering data representations. These refactorings are useful for improving memory and space usage, and are also required for marshalling data to a GPU.

5.2.1 Introduce Chunking

Parallelisation introduces overheads in the form of copying and transmission costs. To maximise performance gains, these overheads should be kept to a minimum, and tasks should be large enough to make their copying and transmission worth it. Should tasks be too small, or are not as large as desired, grouping, or *chunking*, tasks together can be a simple solution. Chunking can be accomplished with the aid of the *Introduce Chunking* refactoring (Brown *et al.*, 2014c). The rules for it are shown below.

$$\begin{array}{rcll} xs : list\ a \in \Gamma, & & & \\ f \in \mathbb{F}, l \in \mathbb{N}^+ & \vdash & [f(x)\ ||\ x \leftarrow xs] & \mapsto \quad skel(map(\bar{f}), chunk(xs,l)) \\ \Gamma, l \in \mathbb{N}^+ & \vdash & skel(farm(\bar{s},n),xs) & \mapsto \quad skel(map(\bar{s}), chunk(xs,l)) \end{array}$$

The first rule matches a list comprehension, and taking a chunk size l, transforms the comprehension into a call to skel with a `map` skeleton over a `func` of the original comprehension function, f. The function, *chunk*, is introduced over the original input, xs, and takes l as a parameter. The *chunk* function is provided as part of the skel library; we include its semantic equivalence rule below.

$$xs : list\ a \in \Gamma, l \in \mathbb{N}^+ \quad \vdash \quad chunk(xs,l) \quad = \quad \texttt{skel} : \texttt{chunk}(xs,l)$$

The second rule may serve to tune already introduced skeletons, matching a task farm and transforming it into a `map` over the skeleton, $\bar{s}$ originally nested within the farm, with the original input chunked in the same manner as the first rule.

5.2.2 List to Binary.

The Erlang concurrency model is *shared-nothing*. This means that all data must be copied between processes. One exception being *binaries*, which are normally passed by reference. Converting a list that is shared between processes to a binary form can have a significant positive impact on performance, since only the reference to the binary is copied between processes and not the data (Barwell *et al.*, 2016). Conversion is also necessary when marshalling data to a GPU kernel.

Conversion between two data representations, such as lists and binaries, is made non-trivial in Erlang through a combination of differences in APIs between the two representations, and Erlang's dynamic typing. The latter allowing each element in lists and other

collection types to be of a different type. Accordingly, the programmer must know the type of each of the elements in the list to be transformed, in addition to the differences in API. Further still, should one part be changed to a different representation, the programmer must manually follow and transform all statements affected by that one transformation, a process during which error can easily be introduced and go unnoticed during compilation.

A list to binary refactoring has been developed to address these problems (Barwell *et al.*, 2016). The top-level rewrite rule for the refactoring is given below.

$$x \in \Gamma \quad \vdash \quad \mathscr{E}_{x:list\ a} \quad \mapsto^* \quad \mathscr{E}_{x:binary\ a}$$

The refactoring takes as argument a variable, x, of type list whose elements are of all the same type, to be transformed to a binary whose elements are of the same type and size. The rule discovers the set of all statements that define or influence x, denoted $\mathscr{E}$, and transforms them through application of a series of rewrite rules to their binary equivalents. This refactoring is a composite refactoring, meaning it applies a series of sub-rewrite rules in turn to each statement in such a way that the overarching change does not break functional correctness.

In order to illustrate this refactoring, consider the `get_particles` function:

```
get_particles(Data) ->
    lists:foldl(fun(X, Acc) -> {Str, _} = string:to_float(X),
                               [Str | Acc] end,
                [],string:tokens(Data,"␣")).
```

This is part of our N-Body example from Section 6.1. It takes a string, `Data`, that contains space-separated floating-point numbers, and returns a list of these numbers. Suppose that, instead of a list, we require `get_particles` to return a binary. We can achieve this by applying *ListToBinary* to the body of `get_particles`:

```
get_particles(Data) ->
    lists:foldl(fun(X, Acc) -> {Str, _} = string:to_float(X),
                               T = <<Str/float>>,
                               <<T/binary, Acc/binary>> end,
                               <<>>, string:tokens(Data, "␣")).
```

Here, the list pattern `[Str|Acc]`, and ultimate output of the `foldl` function is selected as a target for translation. The list pattern is replaced with an equivalent binary pattern. We also change the initial value to the `foldl` to ensure that functional correctness is maintained. Since the result of the `foldl` operation is also the result of the function itself, the refactoring must also transform, where necessary, any expressions where `get_particles` is called.

5.3 Generating GPU Worker Code (Offloading)

We now describe our automatic tool for generating GPU worker code (based on the work described in (Brown, 2015)). A GPU worker is essentially a wrapper for a GPU kernel that performs the actual useful work, and is executed on a CPU. The wrapper takes care of data transfers between the CPU and GPU, sets appropriate parameters of the GPU kernel, and finally schedules the kernel for execution on a GPU. For now, we will assume that the kernel is provided by the programmer, although in future we intend to extend our approach

to also automatically generate OpenCL kernels from (simplified) Erlang source code. The programmer starts by supplying an Erlang module that contains all the parameters that are needed for the generation process. This includes: the name of the Erlang module to be generated; its path; details of the GPU kernel arguments, including the names of the arguments, whether or not they are inputs or outputs to the kernel, and their size, in bytes; and finally a flag to indicate whether the parameter is `local`, `global` or `private`.

```
kernel_arguments() ->
  [ {input, "Chunk", "byte_size(Chunk)", global},
    ..., {input, "", "", local}, {input, "Dt", "", private} ].
```

Generating the `kwork` Record. A new Erlang module with the specified name is now generated by the tool. The module includes the definition of a record, `kwork`, whose fields contain all the necessary kernel buffers, work group sizes, clock frequency settings etc., that are needed by the GPU kernel. Here, `e1`, `e2` and `e2` are generated names that refer to the events that will capture the enqueueing of data into the three programmer-supplied buffers.

```
-record(kwork, { program, kernel, queue, local, freq, units, weight,
         e1,e2,e3, imem, omem, isize, idata,
         chunkInBuffer, particlesInBuffer, chunkOutBuffer, result }).
```

Generating the Erlang Function. A new Erlang function is then generated, whose arguments are the supplied global and local sizes for the kernel, plus the data for the input buffers to the kernel. The first step in setting up the kernel involves loading and building the kernel. When this is done, the next step is to set various kernel parameters, such as the clock frequency, etc.:

```
do_nbody_gpu(Chunk, Particles, Dt, ChunkSize) ->
    E = clu:setup(all),
    {ok,Program} = clu:build_source(E, program(ok)),
    {ok,Kernel} = cl:create_kernel(Program, "nbody_kern"),
    Kws = map( fun(Device) ->
                {ok,Queue} = cl:create_queue(E#cl.context,Device,[]),
                ...
                #kwork{ queue=Queue, ...} end, E#cl.devices),
```

The next step involves creating the buffers that are needed to marshal the data that is transferred to and from the GPU kernel. This code is generated automatically using the supplied kernel arguments.

```
Kws3 = map( fun(K) ->
   {ok,ChunkInBuffer} =
      cl:create_buffer(E#cl.context,[read_only],byte_size(Chunk)),
   {ok,ParticlesBuffer} = ...
   K#kwork {chunkInBuffer=ChunkInBuffer, ... } end, Kws),
```

The next step involves enqueueing the data into the buffers, and then setting the kernel arguments, so that each buffer is assigned to the correct kernel argument. Once the data is enqueued, the kernel can be run on the GPU.

```
Kws4 = map( fun(K) ->
{ok,E1} = cl:enqueue_write_buffer(K#kwork.queue, K#kwork.chunkInBuffer,
                0, byte_size(Chunk), Chunk, []),
{ok,E2} = ...
ok = cl:set_kernel_arg(K#kwork.kernel, 0, K#kwork.chunkInBuffer),
...
{ok,E6} = cl:enqueue_nd_range_kernel(K#kwork.queue, K#kwork.kernel,
                                     [Global], [Local], [E1,...]),
{ok,E7} = cl:enqueue_read_buffer(K#kwork.queue, K#kwork.chunkOutBuffer,
                                        0, byte_size(Chunk), [E6]),
ok = cl:flush(K#kwork.queue),
Result = cl:wait(E7)
K2#kwork {result = Result, e1=E1,... }
```

Finally, the buffers are released and deallocated, and the result is returned.

```
Bs = map( fun(K) ->
   cl:release_mem_object(K#kwork.particlesInBuffer), ... end, Kws4),
clu:teardown(E),
Bs.
```

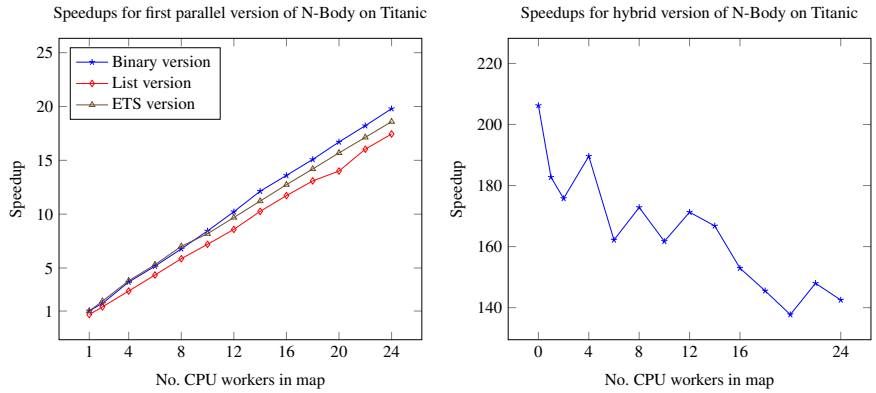
6 Experimental Evaluation

In this section, we evaluate our overall framework on four small but realistic examples, taken from different application domains: particle tracing (*N-Body*), evolutionary computing (*Ant Colony Optimisation*), image processing (*Image Merge*) and simulations (*Football Simulation*). For each example, we go through several phases in the process of parallelisation. Starting from an initial version that has no parallelism and a user-provided OpenCL GPU kernel: i) we first do an initial parallelisation by introducing CPU-only skeletons; ii) we refine this parallelisation using program shaping refactorings, in order to improve performance and/or to prepare the code for introducing hybrid skeletons; and, iii) we extend the initial parallelisation using hybrid skeletons. We omit the step where we generate Erlang GPU code from a given OpenCL kernel, as the resulting code is very similar for all of the examples. All of the steps are carried out semi-automatically using Wrangler and the refactorings from described in Section 5. We also evaluate the speedups that we obtain for various versions of the parallel code. We have used two experimental platforms: i) *titanic* experimental platforms which has two 2.3GHz 12-core AMD Opteron 6176 processors, and an NVidia Tesla C2050 Fermi GPU with 448 CUDA cores, running CentOS Linux; and, ii) *xookik*, with a 12-core 3.06 GHz Intel Xeon X5675 processor and NVidia Fermi GPU.

In addition to the refactorings for introducing hybrid skeletons, described in Section 5, we will also use refactorings to introduce/tune non-hybrid versions of skeletons, such as *ParMapIntroFunc*, *IntroduceTaskFarm*,*ParFeedbackIntro* and so on, described in (Brown *et al.*, 2014b).

6.1 N-Body

N-Body application was described in Section 4. As a reminder, the core of the application is the `nbody` function:



(a) Homogeneous Skeletons (b) Hybrid Skeletons

Fig. 4. Speedups for the N-Body Simulation on Titanic

```
nbody(Parts,0) -> Parts;
nbody(Parts, NrIters) ->
  NewParts = lists:map (fun(X) -> nbody_cpu (X,Parts) end, Parts),
  nbody(NewParts, NrIters-1).
```

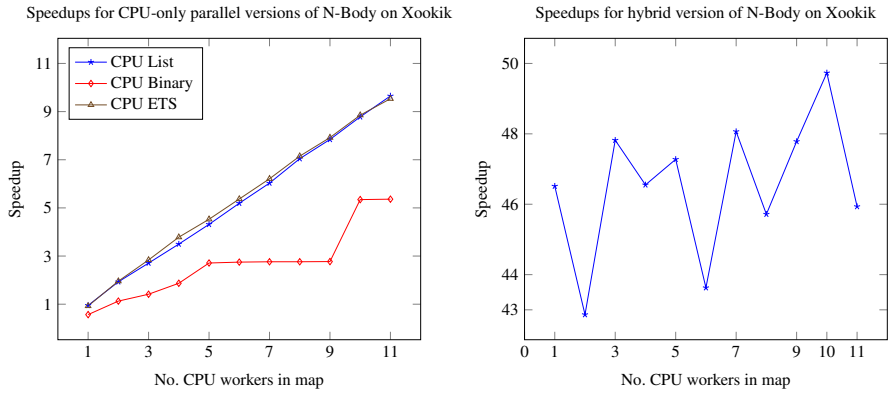
The code is parallelised using the `farm` (for CPU-only versions) and `hyb_farm` skeletons (for a CPU/GPU version). We have three different CPU-only versions: list version, binary version and ETS version. In the list version, particles, as well as chunks of works that need to be sent to the worker processes are kept in lists. Thus, when sending work to a worker process, the whole list needs to be copied. In the binary version, particles and chunks of work are kept in binaries, allowing us to avoid the costs of copying of the data to the worker processes, but slowing down access to individual particles. Finally, in the ETS version, particles and chunks of work are kept in ETS tables, allowing efficient access to individual particles and avoiding the costs of copying the data.

Figure 4(a) shows the speedups that we obtained for the CPU-only versions of the code. The input consists of 20 000 particles, with randomly chosen initial coordinates. We can observe that all the versions scale reasonably well, and that there is a small, but notable benefit in using binaries over lists, and ETS tables over binaries.

Speedups of the hybrid version of the N-Body simulation are given in 4(b). We can observe that the best speedup is obtained when using only one GPU. Adding CPU cores only slows down the execution. The reason for this is that the GPU is, for this particular computation, massively faster than the CPU cores.

6.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) (den Besten *et al.*, 2000) is a heuristic for solving NP-complete optimisation problems. In this paper, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP) optimisation problem, where we are given n jobs and each job, i, is characterised by its processing time, p_i (`Ps` in the code below), deadline, d_i (`Ds` in the code below), and weight, w_i (`Ws` in the code below). The goal is to find the schedule of jobs that minimises the total weighted *tardiness*, defined as



(a) CPU-only Versions (b) Hybrid Version

Fig. 5. Speedups for the N-Body Simulation on Xookik

$\sum w_i \cdot \max\{0, C_i - d_i\}$, where C_i is the completion time of the job, i. The ACO solution to the SMTWTP problem consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail* (`Tau` in the code below). The pheromone trail is stronger along previously successful routes and is defined by a matrix, τ, where $\tau[i, j]$ is the preference of assigning job j to the i-th place in the schedule. After all ants compute their solution, the best solution is chosen as the "running best"; the pheromone trail is updated accordingly, and the next iteration is started. The main part of the program is given below:

```
iterate_ants(_,_, _, _, _, _, 0) ->
    [];
iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, NrIters) ->
  Scheds = lists:map(fun(X) -> find_solution(NrJobs, Ps, Ws, Ds, Tau) end,
                     lists:seq(1, NrAnts)),
  {BestCost, BestSchedule} = pick_best(Scheds),
  NewTau = update(NrJobs, BestCost, BestSchedule, Tau),
  iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, NewTau, NrIters-1)].

ant_colony(InputFile, NrAnts, NrIters) ->
   {NrJobs, Ps, Ws, Ds, Tau} = init(InputFile),
   iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, NrIters).
```

The main function is `iterate_ants`, which is evaluated `NrIters` times. It first calculates `Scheds`, a list of schedules produced by individual ants. Then, it picks the running best schedule using `pick_best` function, updates the pheromone trail according to this schedule using `update` function, and then moves on to the next iteration. In the main function, `ant_colony`, an input is read into lists `Ps`, `Ws`, `Ds` and `Tau` using the `init` function, and then the `iterate_ants` function is called.

The overall structure of this application is similar to that of N-Body that was discussed in the previous section, though parallelisation is slightly more complicated. The initial parallelisation using Wrangler introduces a `chunk` skeleton in place of the original `lists:map` application in `iterate_ants`, using the *ParMapIntroFunc* refactoring. Ants are, therefore, grouped in chunks, with as many chunks as there are CPU workers in the instance of

the sekelton. This map is then linked, using a pipeline introduced by the *ParPipeIntro* refactoring, to a modified version of the `pick_best` function. The whole pipeline is then wrapped into a feedback skeleton, using the *ParFeedbackIntro* refactoring. Relevant parts of the code, after refactoring, are given below:

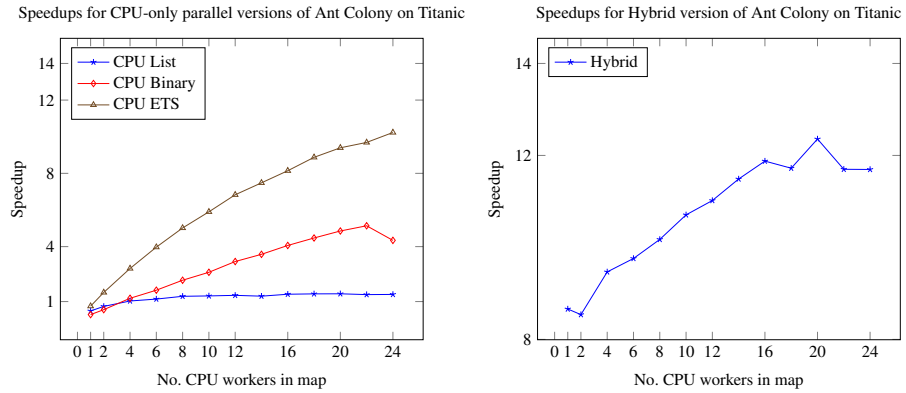
```
create_task_list({NrWs, NrAnts, NrJobs, Ps, Ws, Ds, Tau, NrIters}) ->
  Size = NrAnts div NrWs,
  Rem = NrAnts rem NrWs,
  ChunkSizes = lists:duplicate(Rem, {Size+1}) ++
               lists:duplicate(NrWs-Rem, {Size}),
  lists:map(fun(X) -> {X, NrJobs, Ps, Ws, Ds, Tau, NIters} end),
            ChunkSizes).


iterate_ants(NrWs, NrAnts, NrJobs, Ps, Ws, Ds, Tau, NrIters) ->
  Cluster = {cluster, [{func, fun find_solution_chunk/1}],
                        fun create_task_list/1,
                        fun lists:flatten/1},
  PickBestSpawn = {func, fun(X) ->
                           pick_best_spawn(X, NrAnts, NrJobs, Ps, Ws, Ds)
                         end}
  Pipe = {pipe, [Cluster, PickBestSpawn]},
  Feedback = {feedback, [Pipe], fun ant_feedback/1},
  skel:do([Feedback], [{NrWs, NrAnts, NrJobs, Ps, Ws, Ds, Tau, NrIters}]).

ant_colony(NrWorkers, InputFile, NrAnts, NrIters) ->
  {NrJobs, Ps, Ws, Ds, Tau} = init(InputFile),
  iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, NrIters).
```

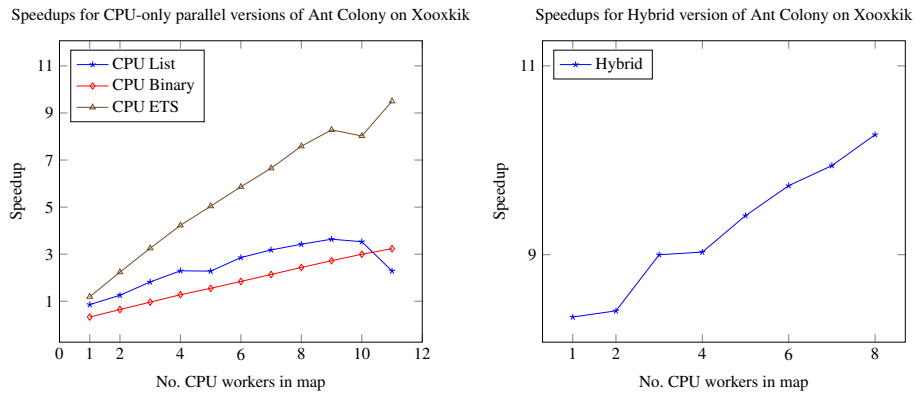
The most interesting in the above code is the `iterate_ants` function. Firstly, a cluster skeleton, `Cluster`, is created. Its inner workflow is a function similar to `find_solution` from the sequential version of the code, excepts that it computes solutions for multiple ants, computing a list of solutions. The cluster skeleton receives as its input a tuple containing number of CPU workers, number of ants, number of jobs, processing times, weights, deadlines, τ matrix and number of remaining iterations. Decomposition function, `create_task_list`, takes this tuple, calculates chunk sizes (based on the number of CPU workers), and returns a list of tasks that are sent to the instances of the inner workflow. Recomposition function simply takes the list of schedules produced by `find_solution_chunk` and flattens them into a single list using the `lists:flatten` function. Next, a `func` skeleton, `PickBestSpawn`, is created to pick the best schedule and update the τ matrix. This skeleton returns a new tuple which contains all of the input data for the next iteration of the `Cluster` skeleton (number of CPU workers, number of ants etc.) plus the updated τ matrix. A pipeline, `Pipe`, is the set up between `Cluster` and `PickBestSpawn` skeletons. Afterwards, the whole pipeline is wrapped into a feedback skeleton, `Feedback`, so that the output of `PickBestSpawn` is fed back as an input to the `Cluster` skeleton if `NrIters` is greater than 0.

To evaluate the performance of the basic parallelisation of Ant Colony Optimisation, we use a test input data with 100 jobs, using 64 000 ants. The speedups for this version are given in Figure **??** (on *titanic*) and Figure 7 (on *xookik*) as the "CPU List". We observe that this version gives relatively poor performance, compared to the original sequential



(a) CPU-only Versions (b) Hybrid Verisons

Fig. 6. Speedups for the Ant Colony Optimisation on *titanic*



(a) CPU-only Versions (b) Hybrid Verisons

Fig. 7. Speedups for the Ant Colony Optimisation on *xookik*

version. This is due to the fact that a large amount of data needs to be copied between processes when the input data is represented with lists. The entire `Ps`, `Ws`, `Ds` and `Tau` lists need to be copied, which amounts to huge amount of data if the number of jobs is big. In the subsequent step, we have therefore applied program shaping refactorings to produce both a version that uses binaries and a version that uses ETS tables, using the *ListToBinary* and *ListBinaryToETS* refactorings. The speedups of these two versions are also shown in Figure **??** as the "CPU Binary" and the "CPU ETS", respectively. We observe that the ETS version gives much better speedups (up to 10.23 on *titanic* and 9.49 on *xookik*) that the list version, due to only references to (global) ETS tables are passed between processes, coupled with very fast access to individual elements of the ETS tables. The binary version gives very modest speedups (up to 5.13 on *titanic* and 3.23 on *xookik*). As with the ETS version, only references to binaries as passed between processes. However, access to the individual elements of a binary is much slower than in the case of the ETS tables. Still, binary version proves a good starting point for a hybrid version.

The next steps are to generate the code for the GPU workers and to introduce hybrid skeletons (again, using the *HybMapIntroSeq* refactoring). Both of these steps are similar to those for N-Body, since the structure of the code is similar. Similar to particles in the N-Body example, a GPU kernel for Ant Colony can compute solutions for multiple ants in parallel – with one logical GPU thread per ant. This allows us to get much more parallelism on a GPU, where we can process many ants in parallel, then on a CPU core where processing of ants is sequential. Using profiling information, we discovered that the GPU is capable of processing about 64 ants in parallel without sequentialisation (both on *titanic* and on *xookik*), and that processing this many ants on the GPU is 25 times faster on *titanic*. than processing them sequentially on a CPU core. Therefore, using mechanisms for work division described in (Janjic *et al.*, 2016), we divide all ants into chunks, where the GPU chunk is a multiple of 64.

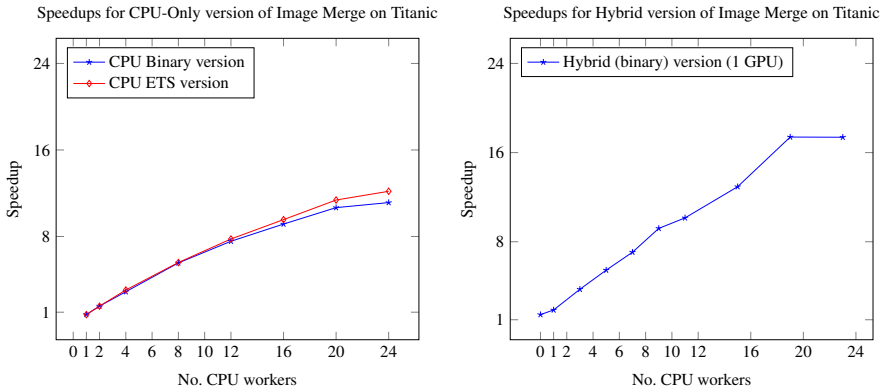
Speedups for the heterogeneous version of Ant Colony are given in Figures **??** and 7. As opposed to N-Body example, here we can see benefits when combining CPU and GPU Workers. On *titanic*, using 1 GPU and no CPU workers gives a speedup of approximately 8.5, while using 1 GPU and 20 CPU workers gives a better speedup of 12.2. In this case, since total number of ants is 64000, and the GPU is "only" 25 times faster in processing a batch of 64 ants than a CPU, so a lot of work will be allocated to the CPU cores also, thus speeding up the computation compared to when only the GPU is used. On *xookik*, improvements when combining CPUs and GPU are much more modest, from 4.66 with 1 GPU and no CPU cores to 5.41 with 1 GPU and 10 CPU cores. Also, on *xookik* we can observe that the CPU ETS version, when more than 5 CPU cores are used, outperforms the hybrid version. This is due to CPUs on *xookik* being much faster than on *titanic*, so the CPU ETS version gives very good speedup (9.49 using 11 workers) and is much faster than the CPU binary version. Since the hybrid version uses binary version as a base, CPU cores are much slower and assigning more work to the GPU cannot compensate for this.

Discussion From all of the experiments on Ant Colony Optimisation, we can make several conclusions. Firstly, the "obvious" parallelisation, without using GPU and program shaping transformations, gives very poor speedups, and in most of the cases actually results in a slowdown compared to the sequential version. Applying program shaping transformations results in the versions that use binary data and ETS tables that give either much better speedups (in the case of the ETS version) or provide a basis for hybrid version. As opposed to the NBody example, in this case GPU is not that much faster than CPU cores, so a hybrid version where CPU cores and GPUs are combined brings the benefits, compared to GPU-only and CPU-only versions. Which version is the fastest, CPU ETS or Hybrid, depends on the actual hardware.

6.3 Image Merge

Image Merge is an application from the computer graphics domain. It reads a stream of pairs of images from files, and merges images from each pair:

```
FinalImages = [convertMerge(readImage(Y)) || Y <- imageList(X)].
```



(a) CPU-Only Versions of Image Merge (b) Hybrid Verisons Versions of Image Merge

Fig. 8. Speedups for Image Merge on *titanic*

There are multiple ways in which this code can be parallelised. Since the output of `readImage` is an input to `convertImage`, we can set up a pipeline between these two functions. Also, each of the two functions can be farmed, so we read and process multiple images at the same time. We will, however, consider the simplest parallelisation here: the one where we set up a *farm* of workers that execute the composition of `readImage` and `convertMerge` functions. For this, we use the *IntroduceTaskFarm* refactoring. The parallel code for this is:

```
skel:do([{farm, [{seq, fun (Y) -> convertMerge(readImage(Y)) end}],
             <nr_workers>}],imageList(X)).
```

This version, however, has problems with memory consumption. Since images are kept in memory as lists, for larger numbers of large images, the required memory was too high for our testbed machine. Therefore, we needed to apply program shaping refactorings to first represent images as binaries or ETS tables (the *ListToBinary* and *ListBinaryToETS* refactorings). Figure 8 shows the speedups for the binary and ETS version of the code. We observe that both versions have very similar performance.

In the next step, we introduce a hybrid farm skeleton using the *HybMapIntroSeq* refactoring. The speedups for this version are also given in Figure 8. Unlike the first two examples, in this case the GPU code is not significantly faster than the CPU code, because of the large amount of data transfer that needs to take place. In this case, we can observe that the best results are obtained when both CPU and GPU workers are used. The best speedup of 17.4 is obtained for 20 CPU workers and 1 GPU worker. We were unable to obtain results on *xookik* for this example due to its large memory footprint exceeding the hardware capabilities of *xookik*.

6.4 Football Simulation

Football Simulation is an industrial application that predicts the outcomes of football games, and is used by betting companies to calculate the winning odds for matches. Each simulation accepts as arguments some information about the teams involved in a match (e.g. attack and defense strength), and uses a randomised routine to predict the end result

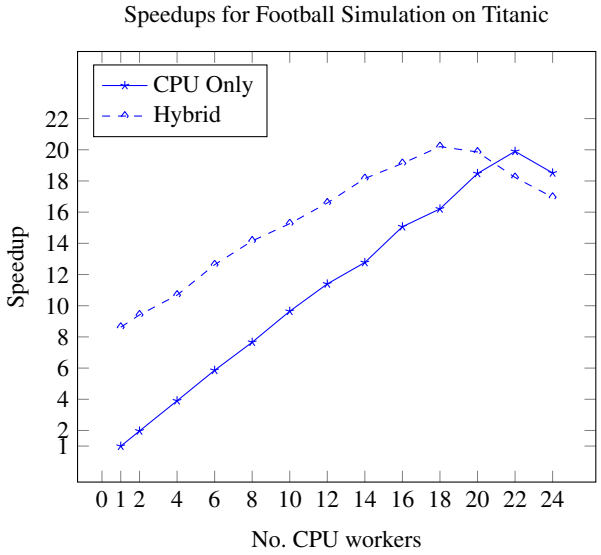


Fig. 9. Speedups for Football Simulation on *titanic*

of the match. The final prediction for each match is averaged over multiple simulations, so more time the simulation is repeated, more accurate the prediction will be.

The top-level structure of the code is

```
Results = [ get_average_score(simulate_match(Pair, NrSimulations))
          || Pair <- Pairs ]
```

`Pair` is a pair of tuples that contains the necessary information about one match, i.e. information about one pair of teams. In the simplest case, we provide just two floating point numbers for each team, attack and defense strength. For each pair of teams, `simulate_match` is called `NrSimulations` times, and then the average score is computed using the `get_average_score` function. We omit the details of the `simulate_match` and `get_average_score` functions, as they are both inherently sequential.

We can observe that the potential parallelism lies at two levels of the computation of the `Results` list. At the outer level, we can parallelise the `get_average_score` function over the list of pairs of teams. At the inner level, for each match we do `NrIterations` simulations, which are completely independent of one another and can, therefore, be executed in parallel. However, each individual simulation is very short, so parallelising the inner level results in too fine-grained parallelism. Therefore, we just parallelise the outer level. Similarly to ACO and NBody examples, here we also use hybrid map skeleton. Furthermore, since `simulate_match` operates over tuples, rather than lists, we do not have binary and ETS versions.

Figure 9 shows the speedups we obtained on *titanic* for both CPU-only and hybrid version of Football Simulation. The figure shows speedups when different number of CPU workers in used in the map skeleton, with 0 GPU workers for the CPU-only, and 1 GPU worker for the hybrid version. We can observe that the hybrid version significantly outperforms the CPU-only version when up to 20 CPU workers are used, but after that point, the CPU-only version gives better speedups. On the other hand, on *xooxik* (Figure 10), we can

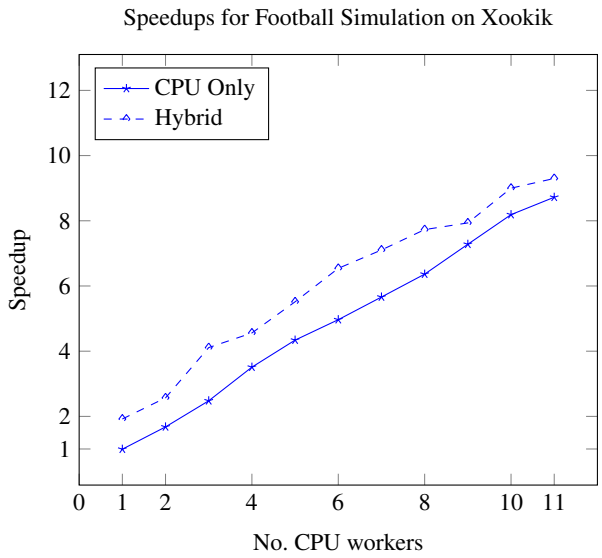


Fig. 10. Speedups for Football Simulation on *xookik*

observe that the hybrid version consistently outperforms the CPU-only version, achieving a speedup of around 10 on 11 CPU cores supported by the GPU.

7 Related Work

Algorithmic skeletons have been the focus of much research since the 1980s, with a number of skeleton libraries being produced in a range of languages (González-Vélez & Leyton, 2010), including our own Skel library (Brown *et al.*, 2014c), which is the only available skeleton library for Erlang, and, indeed, for BEAM systems in general, including Elixir. Despite this, there has been limited research on using skeletons for GPU programming, with most techniques being concerned with low-level language bindings to CUDA or OpenCL. In the functional programming community, there have been several attempts to simplify GPU programming, for example, using compile-time optimisations (Guo *et al.*, 2011; McDonell & et al., 2013). While these optimisations are fully automatic, rather than the semi-automatic refactorings we have introduced here, they are consequently limited to specific situations, and cannot take advantage of programmer knowledge or expertise, as with LAPEDO. A popular alternative approach is to define an embedded domain-specific language (EDSL) (Lee & et al., 2011; Claessen *et al.*, 2012; Grossman *et al.*, 2011). This can provide a structured approach to parallelism, similar to our approach using skeletons, but unlike LAPEDO, current approaches do not assist the programmer with the introduction of parallelism, including the important task of program shaping. Skeletons, moreover, have the advantage of representing *language-independent* patterns of parallelism, that can more easily be transferred to other language settings. Research into skeletons for GPU programming, while limited, has produced a number of skeleton libraries for a few languages, including: C++ (Le *et al.*, 2012), F# (Singh, 2010), and Haskell (Chakravarty *et al.*, 2011). However, all of these libraries only support GPUs. Unlike Lapedo, they do not provide *hybrid* CPU/GPU skeletons, and therefore do not provide flexible portability across both

homogeneous and heterogeneous systems, or allow the programmer to take advantage of CPUs where these are more efficient, without needing to reprogram the application.

A number of recent and highly successful "programming models" such as the well-known *Google MapReduce* derive and inherit from algorithmic skeletons (Buono *et al.*, 2010). Furthermore, skeletal methodologies inherently have a predictable communication and computation structure, since they capture the structure of the program. They provide, by construction, a foundation for performance modelling and estimation of parallel applications.

One novel feature of our work is the focus on how to introduce parallelism through (eventually) performance-directed refactoring that considers program shaping as well as the actual introduction parallelism. Most work on parallelism stops at the implementation of parallel libraries or language constructs, but fails to consider the actual programming process. In (Brown *et al.*, 2014c), we introduced a methodology for programming parallel Erlang programs, using refactoring to facilitate the introduction of parallel skeletons. In (Brown *et al.*, 2014b), we presented a similar refactoring methodology for C++ programs. This paper goes beyond that work in exploiting hybrid skeletons for heterogeneous multicores, and in considering program shaping for such architecture. Finally, we have considered program shaping refactorings to introduce the most appropriate data structures for parallelism, transforming lists to binary structures or ETS tables, as appropriate. In contrast to our work, most current research focuses on simple compile-time optimisations instead of source-to-source refactorings (Groot *et al.*, 2007). The approach presented in our paper therefore not only improves upon current methodologies by enabling their use on heterogeneous architectures, but also helps to introduce some automation to the previously-manual program shaping stage.

The static mapping problem is by no means a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures providing static partitioning (Kwok & Ahmad, 1999; Saraswat *et al.*, 2007; Subhlok *et al.*, 1993), using runtime scheduling (Ramamritham & Stankovic, 1984), heuristic-based mappings (Gordon *et al.*, 2006), analytical models (Navarro *et al.*, 2009). Each of these can improve the performance of the system. There are some heuristic based approaches which automate the process of mapping to multi-core architectures for specific frameworks, such as the learning approach used for partitioning streaming in the StreamIt framework (Wang & O'Boyle, 2010) or the runtime adaptation approach used in FlexStream (Hormati *et al.*, 2009) framework. Despite the amount of work done in the homogeneous environment, to our best knowledge there is little work done for mapping to heterogeneous (CPU/GPU) architectures. In (Goli *et al.*, 2013) we introduced a new mapping technique for heterogeneous multicore systems, but unlike the approach here, did not provide a usable programming methodology. Most of the work on GPUs is primarily focused on application performance tuning (Agrawal *et al.*, 2005) rather than orchestration. In (Udupla *et al.*, 2009) a method is provided to orchestrate the execution of heterogeneous StreamIt program presented on a multi core platform equipped with an accelerator.

RefactorErl also provides refactoring support for Erlang. Unlike Wrangler, **RefactorErl** (Tóth *et al.*, n.d.) focuses on program analysis rather than program transformation. RefactorErl implements a wide range of syntax-based transformations and static semantic analyses, including scope analysis of various Erlang language entities, side-effect analysis

and callee aproximation of dynamic function calls. Most of these high-level semantic analyses build on results from dataflow and type analyses. *PaRTE* (Bozó *et al.*, 2014) integrates capabilities from both the RefactorErl and Wrangler refactoring/program analysis tools into a new parallelisation framework that can be used to identify parallel patterns and to determine the best implementations of those patterns. PaRTE uses a novel *pattern discovery* technique to first find possible instances of skeletons, and then offers possible refactorings to transform these instances into actual skeletal code, using the Skel library. PaRTE uses RefactorErl to perform pattern discovery and the Wrangler tool to perform the actual refactorings. The refactorings allow for a initial, and basic, program shaping (e.g., transforming a recursive function call into a list comprehension) before applying refactorings to introduce skeletal code. While these are presently independent, but connected pieces of research, we envisage that the PaRTE framework could be used to complement LAPEDO by extending pattern discovery to also consider heterogeneous skeletal instances, for example. We intend to explore this in future.

8 Conclusions

In the paper we have introduced an extension to our LAPEDO Erlang framework for programming heterogeneous multi- and many-core systems comprising a mixture of CPUs and GPUs. LAPEDO consists of the Skel library of parallel skeletons, including hybrid skeletons that can have components for different kind of processors (e.g. CPUs and GPUs) and the PaRTE refactoring tool. In this paper we presented a novel set of refactorings that are implemented in Wrangler to support semi-automatic generation of problem-specific code. These refactorings allow both shaping of the sequential code in order to choose and incorporate the best data structure for hybrid computations, and introduction of hybrid parallelism in the form of instances of hybrid skeletons. We also described refactorings to *automatically* generate the tedious book-keeping code for transferring data to/from accelerators and executing computation kernels on them, which is a necessary part of hybrid computations. The use of algorithmic skeletons combined with functional programming offers us a very high-level of abstraction over the parallel structure of the program, and the use of refactoring technology allows systematic transformation and generation of code. In this way, the programmer can provide input in the process of parallelising their code, and can obtain a clear understanding of how the code is changed. One of the main strengths of the LAPEDO framework is that it produces highly *portable* code, which can be run both on CPU-only architectures, and on heterogeneous systems that also contain GPUs. This allows wider deployment of the same application in both homogeneous and heterogeneous multicore settings. We have also demonstrated that, for our examples, LAPEDO can deliver significant speedups over the corresponding sequential Erlang code (up to 220 for N-Body simulation), especially if GPUs are available. This is a major gain for relatively little work, allowing the performance of computationally intensive parts of an Erlang program to be improved both quickly and easily.

8.1 Further Work

In the future, we aim to extend LAPEDO with mechanisms to automatically generate GPU kernels from Erlang code. We will also incorporate previously developed mechanisms (Goli *et al.*, 2013) for automatic discovery of the optimal number of CPU and GPU workers for each skeleton of a given problem, and to link with work on automated pattern discovery that is being carried out in the ongoing EPSRC *Discovery* research project. We also aim to extend the Skel library to cover domain-specific skeletons, such as skeletons from computer algebra or from evolutionary computing, and to develop novel refactorings to address these skeletons. We also plan to perform larger-scale evaluation, including multicore systems containing multiple GPUs. Finally, we aim to develop larger and more complex use cases to test our framework, including larger simulations from computational physics, as well as additional artificial inteligence and deep learning computations.

Acknowledgments

This research has been generously supported by the European Union Framework 7 ParaPhrase project (**IST-288570**), EU Horizon 2020 projects RePhrase (H2020-ICT-2014-1), agreement number **644235**; Teamplay (H2020-ICT_2017-1) agreement number **779882**, and EPSRC Discovery, **EP/P020631/1**. EU COST Action IC1202: Timing Analysis On Code-Level (TACLe), and by a travel grant from EU HiPEAC. The University of Pisa generously provided us with access to their 24-core parallel machine, *titanic*, for experimental purposes.

References

Agrawal, Sitij, Thies, William, & Amarasinghe, Saman. (2005). Optimizing Stream Programs Using Linear State Space Analysis. *Pages 126–136 of: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '05. New York, NY, USA: ACM.

Armstrong, J., Virding, S.R., & Williams, M.C. (1993). *Concurrent Programming in Erlang*. Prentice-Hall.

Barwell, Adam D., Brown, Christopher, Castro, David, & Hammond, Kevin. (2016). Towards Semi-automatic Data-type Translation for Parallelism in Erlang. *Pages 60–61 of: Proceedings of the 15th International Workshop on Erlang*. Erlang 2016. New York, NY, USA: ACM.

Bozo, I., Horpocsi, D., Horvoth, Z., Kitlei, R., Koszegi, J., M., Tejfel., & Toth, M. 2011 (October). RefactorErl - Source Code Analysis and Refactoring in Erlang. *Pages 138–148 of: Proceedings of the 12th symposium on programming languages and software tools, isbn 978-9949-23-178-2*.

Bozó, István, Fordós, Viktoria, Horvath, Zoltán, Tóth, Melinda, Horpácsi, Dániel, Kozsik, Tamás, Köszegi, Judit, Barwell, Adam, Brown, Christopher, & Hammond, Kevin. (2014). Discovering Parallel Pattern Candidates in Erlang. *Pages 13–23 of: Proceedings of the thirteenth acm sigplan workshop on erlang*. Erlang '14. New York, NY, USA: ACM.

Brown, C. 2015 (March). *Pattern Transformation System, D4.4.* Deliverable for the EU Framework 7 ParaPhrase project.

Brown, Christopher, & Thompson, Simon. (2010). Clone Detection and Elimination for Haskell. *Pages 111–120 of: Proceedings of the 2010 acm sigplan workshop on partial evaluation and program manipulation*. PEPM '10. New York, NY, USA: ACM.

Brown, Christopher, Li, Huiqing, & Thompson, Simon. 2010 (May). An Expression Processor: A Case Study in Refactoring Haskell Programs. *Page 15 of:* Page, Rex (ed), *Eleventh symposium on trends in functional programming*.

Brown, Christopher, Janjic, Vladimir, Hammond, Kevin, Schöner, Holger, Idrees, Kamran, & Glass, Colin W. (2014a). Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. *Pages 36–43 of: Proceedings of the 22nd euromicro international conference on parallel, distributed and network-based processing, pdp 2014*.

Brown, Christopher, Janjic, Vladimir, Hammond, Kevin, Schoner, Holger, Idrees, Kamran, & Glass, Colin W. (2014b). Agricultural reform: More efficient farming using advanced parallel refactoring tools. *Pages 36–43 of: Parallel, distributed and network-based processing (pdp), 2014 22nd euromicro international conference on*. IEEE.

Brown, Christopher, Danelutto, Marco, Hammond, Kevin, Kilpatrick, Peter, & Elliott, Archibald. (2014c). Cost-Directed Refactoring for Parallel Erlang Programs. *Int. j. parallel program.*, **42**(4), 564–582.

Brown, Christopher, Janjic, Vladimir, Barwell, Adam M., & Garcia, Jose Daniel. (2019). Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++. *Under review for the 12th Interntional Symposium on High-Level Parallel Programming and Applications (HLPP 2019)*.

Buono, Daniele, Danelutto, Marco, & Lametti, Silvia. (2010). Map, Reduce and Mapreduce, the Skeleton Way. *Procedia computer science*, **1**(1), 2089–2097. ISSN 1877-0509, DOI: 10.1016/j.procs.2010.04.234.

Burstall, R. M., & Darlington, John. (1977). A Transformation System for Developing Recursive Programs. *J. acm*, **24**(1), 44–67.

Chakravarty, Manuel M.T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., & Grover, Vinod. (2011). Accelerating haskell array codes with multicore gpus. *Pages 3–14 of: Proceedings of the sixth workshop on declarative aspects of multicore programming*. DAMP '11. New York, NY, USA: ACM.

Claessen, Koen, Sheeran, Mary, & Svensson, Bo Joel. (2012). Expressive Array Constructs in an Embedded GPU Kernel Programming Language. *Pages 21–30 of: Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming*. DAMP '12. New York, NY, USA: ACM.

Cole, Murray. (2004). Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel comput.*, **30**(3), 389–406.

Cole, Murray I. (1988). *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Ph.D. thesis. AAID-85022.

den Besten, Matthijs, Stützle, Thomas, & Dorigo, Marco. (2000). Ant Colony Optimization for the Total Weighted Tardiness Problem. *Pages 611–620 of: Ppsn vi*. Lecture Notes in Computer Science, vol. 1917.

Dig, Danny, Marrero, John, & Ernst, Michael D. (2009). Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. *Pages 397–407 of: Proceedings of the 31st international conference on software engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society.

Fields, D.K., Saunders, S., & Belyaev, E. (2006). *IntelliJ IDEA in Action*. In Action. Manning.

Fowler, Martin. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley.

Goli, Mehdi, McCall, John A. W., Brown, Christopher, Janjic, Vladimir, & Hammond, Kevin. (2013). Mapping parallel programs to heterogeneous CPU/GPU architectures using a Monte Carlo Tree Search. *Pages 2932–2939 of: IEEE Congress on Evolutionary Computation*. IEEE.

González-Vélez, Horacio, & Leyton, Mario. (2010). A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. pract. exper.*, **40**(12), 1135–1160.

Gordon, Michael I., Thies, William, & Amarasinghe, Saman. (2006). Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGOPS Oper. Syst. Rev.*, **40**(5), 151–162.

Groot, Sven, van der Spek, Harmen L. A., Bakker, Erwin M., & Wijshoff, Harry A. G. (2007). The Automatic Transformation of Linked List Data Structures. *Page 408 of: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society.

Grossman, Max, Simion Sbîrlea, Alina, Budimlić, Zoran, & Sarkar, Vivek. (2011). CnC-CUDA: Declarative Programming for GPUs. *Pages 230–245 of: Languages and compilers for parallel computing*. Lecture Notes in Computer Science, vol. 6548. Springer Berlin Heidelberg.

Guo, Jing, Thiyagalingam, Jeyarajan, & Scholz, Sven-Bodo. (2011). Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler. *Pages 15–24 of: Proc. of damp*.

Hormati, Amir H., Choi, Yoonseo, Kudlur, Manjunath, Rabbah, Rodric, Mudge, Trevor, & Mahlke, Scott. (2009). Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. *Pages 214–223 of: Proceedings of the 2009 18th international conference on parallel architectures and compilation techniques*. PACT '09. Washington, DC, USA: IEEE Computer Society.

Janjic, Vladimir, Brown, Christopher, & Hammond, Kevin. (2016). Lapedo: Hybrid Skeletons for Programming Heterogeneous Multicore Machines in Erlang. *Pages 185–195 of: Parallel computing: On the road to exascale*. IOS Press.

Kwok, Yu-Kwong, & Ahmad, Ishfaq. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Acm comput. surv.*, **31**(4), 406–471.

Le, Duc Tung, Nguyen, Huu Duc, Pham, Tuan Anh, Ngo, Huy Hoang, & Nguyen, Minh Thap. (2012). An Intermediate Library for Multi-GPUs Computing Skeletons. *Pages 1–6 of: RIVF*. IEEE.

Lee, Hyouk Joong, & et al. (2011). Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *Micro, ieee*, **31**(5), 42–53.

Li, Huiqing, Thompson, Simon, Orosz, György, & Tóth, Melinda. (2008). Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. *Pages 61–72 of: Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG '08. New York, NY, USA: ACM.

McDonell, Trevor L., & et al. (2013). Optimising Purely Functional GPU Programs. *Sigplan not.*, **48**(9), 49–60.

Navarro, A., Asenjo, R., Tabik, S., & Cascaval, C. 2009 (Sept). Analytical modeling of pipeline parallelism. *Pages 281–290 of: Proceedings of the 18th international conference on parallel architectures and compilation techniques, 2009. pact '09.*

Opdyke, William F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. thesis.

Ramamritham, K., & Stankovic, J. A. (1984). Dynamic Task Scheduling in Hard Real-Time Distributed Systems. *Ieee softw.*, **1**(3), 65–75.

Rogvall, Tony. *OpenCL Binding for Erlang*. `https://github.com/tonyrog/cl`. Accessed: 20-05-2019.

Saraswat, Vijay A., Sarkar, Vivek, & von Praun, Christoph. (2007). X10: Concurrent programming for modern architectures. *Pages 271–271 of: Proceedings of the 12th acm sigplan symposium on principles and practice of parallel programming*. PPoPP '07. New York, NY, USA: ACM.

Singh, Satnam. (2010). Declarative Data-parallel Programming with the Accelerator System. *Pages 1–2 of: Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. DAMP '10. New York, NY, USA: ACM.

Subhlok, Jaspal, Stichnoth, James M., O'Hallaron, David R., & Gross, Thomas. (1993). Exploiting task and data parallelism on a multicomputer. *Sigplan not.*, **28**(7), 13–22.

Sultana, Nik, & Thompson, Simon. (2008). Mechanical Verification of Refactorings. *Pages 51–60 of: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '08. New York, NY, USA: ACM.

Tóth, M., Bozó, I., & Horváth, Z. *Utilization of RefactorErl during Development and Maintenance*. Tutorial at the 17th Erlang User Conference 2012, Stockholm, Sweden, May 2012.

Udupla, A., Govindarjan, R., & Thazhuthaveetil, M. J. (2009). Software Pipelined Execution of Stream Programs on GPUs. *Pages 200–209 of: International Symposium on Code Generation and Optimization*.

Wang, Zheng, & O'Boyle, Michael F.P. (2010). Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach. *Pages 307–318 of: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. New York, NY, USA: ACM.