Towards Semi-automatic Data-Type Translation for Parallelism in Erlang

Adam D. Barwell Christopher Brown David Castro Kevin Hammond

School of Computer Science, University of St Andrews, Scotland, UK.
{adb23,cmb21,dc84,kh8}@st-andrews.ac.uk

Abstract

As part of ongoing research into *programmer-in-the-loop parallelisation*, we are studying the problem of automatically introducing alternative data structures to support parallelism. Our goal is to make it easier to produce the best parallelisation for some given program, or even to make parallelisation feasible. We use a refactoring approach to choose and introduce these transformations for specific *algorithmic skeletons*, structured forms of parallelism that capture common patterns of parallelism.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering

Keywords Erlang, Refactoring, Program Transformation

1. Introduction

This paper studies the problem of automatically introducing alternative data structures to support parallelism. Our approach integrates with the *Wrangler* refactoring tool for Erlang, and uses the advanced Skel [4] skeleton library for Erlang. This library has previously been shown to give good parallelisations for a number of applications, including a multi-agent system [1] where we have achieved speedups of up to 142.44 on a 61-core machine with 244 threads. We have investigated three widely-used Erlang data structures: lists, binary structures and *ETS* (Erlang Term Storage) tables. In general, we have found that *ETS* tables deliver the best parallel performance for the examples that we have considered. However, our results show that simple lists may deliver similar performance to the use of *ETS* tables, and better performance than using *binary* structures. This means that we cannot blindly choose to implement a single optimisation as part of the compilation process. Our approach also allows the use of new (possibly user-defined) data structures and other transformations in future, giving a high level of flexibility and generality.

2. Recursive Descent Refactoring

In order to transform lists into binaries or *ETS* tables, we need to define a composite refactoring. Our *recursive descent refactoring* consists of a setup phase, converting the input list `Xs`, and a recursive phase that inspects each statement in the program, refactoring those that are relevant to `Xs` (Fig. 1). The refactoring is designed to convert the code block that contains `Xs`, expanding the set of terms that it looks for as `Xs` is manipulated, and duplicating and converting any programmer-defined functions that are invoked with `Xs` as argument. This results in an "island" of refactored functions whose interface(s) remain the same before and after refactoring (Fig. 2). Although we focus here on translating lists to binaries or ETS tables, the approach can be extended to either the reverse translation, or to translation to other data-types. Lists are a primitive data-type in Erlang that can contain potentially infinite elements of any type. Lists are *copied* when they passed between processes, and can be slow when randomly accessing elements. Erlang *binaries* have a similar syntax to lists, but have more numerous options during construction and pattern matching, where the type and size of individual elements can be specified. Binaries that are larger than 64 bytes are not copied between process heaps, but instead are passed by reference. They are also useful, and sometimes necessary, when interfacing both with GPUs and with other programming languages. Finally, ETS tables are global, mutable term storage for nodes. There are four types of ETS table, three of which provide constant time access to the contained data, and a range of construction options.

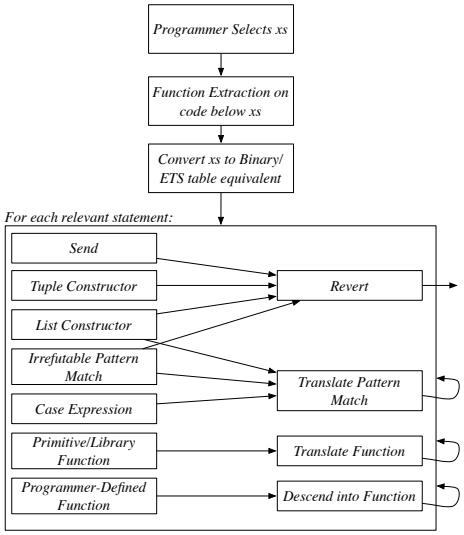

Figure 1. Recursive Descent Refactoring Approach

3. Illustrative Example: Image Merge

Image Merge takes a list of pairs of images (each represented as a two-dimensional list), and merges each pair. The main computation is defined by the `convertMerge/1` function.

Erlang'16, September 23, 2016, Nara, Japan
ACM. 978-1-4503-4431-9/16/09...$15.00
http://dx.doi.org/10.1145/2975969.2975978

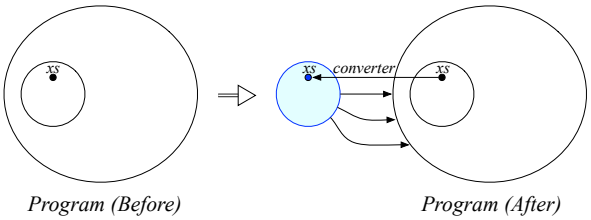

Figure 2. Result of applying recursive descent refactoring.

```
convertMerge({Xs, Ys, F1, F2, Name}) ->
    Xs_p = lists:map(
             fun(L) ->
                     removeAlpha(L, F1)
             end, Xs),
    Ys_p = lists:map(
             fun(L2) ->
                     removeAlpha(L2, F2)
             end, Ys),
    WhiteR =  lists:map(
                fun(Col) ->
                        convertToWhite(Col)
                end, Xs_p),
    Result = lists:zipwith(
                fun(L1,L2) ->
                        mergeTwo(L1, L2)
                end, WhiteR, Ys_p),
    {Result, length(Xs), Name}.
```

There are two refactoring opportunities. Both `Xs` and `Ys` are lists. We might choose either, or both, of these to translate into either an ETS or binary equivalent. When performed as a combination of a task farm and a two-stage pipeline, transmitting images between processes can result in significant memory usage. Indeed, this problem presented itself during testing on large numbers of images. This alone provides significant motivation to use our recursive descent refactoring to translate both `Xs` *and* `Ys`. When merging 100 pairs of 1024x1024 images, we observe that both the binary and ETS representations avoid the excessive memory usage of the list representation. Our experiments were performed on *Titanic*, a 2.3GHz AMD Opteron 6176 machine with 24 physical cores and 32 GB of RAM at the University of Pisa. Each experiment was repeated 10 times and we recorded the mean result. Fig. 3 gives speedups for varying number of cores compared with the original list sequential version. We observe a maximum speedup of 12.2 for the ETS representation, and 11.1 for the binary representation, where the original list sequential version takes on average $600,211,668.4\mu s$, or 10.0035 minutes. The slight advantage for the ETS version could be because the built-in fold operation, which it uses, is more efficient than the recursive function that is defined for binaries.

4. Conclusions

The correct choice of data structure can make a significant difference to parallel performance. To date, this has been generally a manual process. We have discussed a new type-translation refactoring that is designed to automatically translate Erlang list structures and operations to equivalent binary and ETS forms and applied our approach to the image merge example, implemented using the Skel parallel skeleton library for Erlang. For this example, our results show that ETS tables clearly deliver the best parallel performance: significantly better than lists and about 10% better than binaries. Whilst our results might therefore lead us to conclude there is little reason to use binaries, and that all lists should potentially be translated into ETS tables, this would be premature. As demonstrated

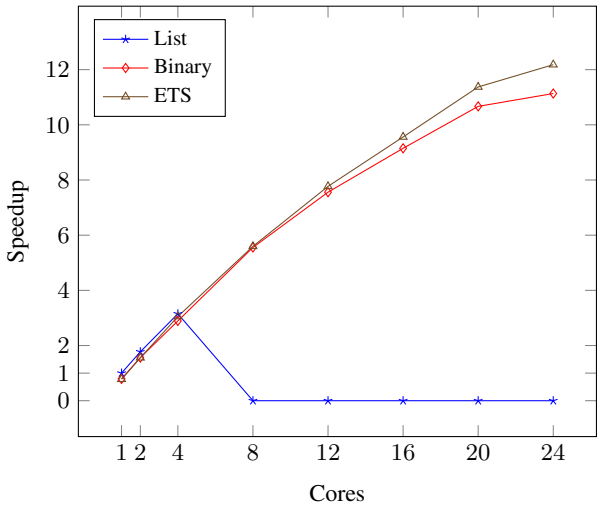

Figure 3. Speedups for Image Merge on *Titanic*

in [5], binaries are required when interfacing with other languages and hardware, e.g. OpenCL and GPUs. Furthermore, both lists and binaries can be passed across distributed systems with less administrative overhead than ETS tables, nor do they present a natural bottleneck with high frequency accesses across processes as with ETS tables. We therefore instead conclude that the correct choice of data-type is highly dependent upon the specific parallel program and its context, and should not be left to a blind optimisation process as part of a compilation phase.

Acknowledgments

This work has been partially supported by the EU Horizon 2020 grant "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach" (ICT-644235), by COST Action IC1202 (TACLe), supported by COST (European Cooperation in Science and Technology), and by EPSRC grant EP/M027317/1 "C^3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence".

References

[1] A. D. Barwell, C. Brown, K. Hammond, W. Turek, and A. Byrski. Using "Program Shaping" and Algorithmic Skeletons to Parallelise an Evolutionary Multi-Agent System in Erlang. *Computing and Informatics*, 2017. *to appear*.

[2] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, 2014.

[3] T. Brandes, S. Chaumette, M. C. Counilh, J. Roman, A. Darte, F. Desprez, and J. C. Mignot. Hpfit: A set of integrated tools for the parallelization of applications using high performance fortran. part i: Hpfit and the transtool environment. *Parallel Comput.*, 23(1-2):71–87, Apr. 1997.

[4] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming*, pages 1–19, 2013.

[5] V. Janijic, C. Brown, and K. Hammond. Lapedo : hybrid skeletons for programming heterogeneous multicore machines in Erlang. In *Parallel Computing: On the Road to Exascale*, ParCo '15, pages 185 – 195, 2015.

[6] R. Lämmel. Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, pages 15–28, 2002.