Automatic Parallelism in Mercury

Paul Bone*

Department of Computer Science and Software Engineering
The University of Melbourne
pbone@csse.unimelb.edu.au

National ICT Australia (NICTA)

Abstract

Our project is concerned with the automatic parallelization of Mercury programs. Mercury is a purely-declarative logic programming language, this makes it easy to determine whether a set of computations may be performed in parallel with one-anther. However, the problem of how to determine *which* computations should be executed in parallel in order to make the program perform optimally is unsolved. Therefore, our work concentrates on building a profiler-feedback automatic parallelization system for Mercury that creates programs with very good parallel performance with as little help from the programmer as possible.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases Program Optimization, Automatic Parallelism, Mercury

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.251

1 Introduction

The rate at which computers are becoming faster at sequential execution has dropped significantly. Instead their parallel processing ability is increasing, and multicore computers are now common. It is now necessary to use parallelism to get the most out of a modern processor. However, parallelization in imperative languages is very difficult, logic and functional languages make parallelism easier by supporting *deterministic parallelism*, which can prevent deadlocks and race conditions, and will always compute the same answer for the same inputs regardless of how execution is scheduled.

Unfortunately most programmers are not very good parallelizing their programs. It is very easy to slow a program down by creating a lot of fine-grained parallelism, in which the overheads are more expensive than the benefit of parallel evaluation. In other cases communicating threads may block waiting for one-another to produce some value, to minimize the runtime of the program, the programmer must understand how their program's threads will be scheduled and when during their execution values will be produced and consumed.

Our work attempts to solve these problems by automatically parallelizing programs written in the Mercury programming language. Mercury is a pure logic programming language, it already supports explicit parallelism of dependent conjunctions, as well as powerful profiling tools which generate data for our analysis.

* Paul is supported by an Australian Postgraduate Award and a NICTA top-up scholarship.

2 Background

Mercury supports dependant AND parallelism [4, 13], allowing programmers to request parallel evaluation of a conjunction by using an ampersand to separate conjuncts. We are extending this work to enable feedback directed automatic parallelization of Mercury programs.

Mercury's deep profiler [5] gathers detailed information about a programs execution. In particular, it records statistics not just for each predicate but for the chain of ancestor calls that represent the predicate's invocation. For example, if list.map/3 is used in multiple places within a program the deep profiler will record separate data for each use. This enables us to gather accurate information that we use to automatically parallelize the program.

3 Related Work

Mercury has strong mode and determinism systems [12], this makes it easy to detect how many solutions a goal may have as well as the locations within a predicate where variables are bound. Mercury only allows parallelization of goals that never fail, and never produce more than one solution, which is the most common type of goal in Mercury programs. Therefore, discovering producer-consumer relationships is easy compared to Prolog systems supporting AND-parallelism such as [6].

Most research in parallel logic programming so far has focused on trying to solve these problems of getting parallel execution to *work* well, with only a small fraction trying to find when parallel execution would actually be *worthwhile*. Almost all previous work on automatic parallelization has focused on granularity control: parallelizing only computations that are expensive enough to make parallel execution worthwhile [7, 9], and properly accounting for the overheads of parallelism itself [11]. Most of the rest has tried to find opportunities to exploit independent AND-parallelism during the execution of otherwise-dependent conjunctions [10, 3].

We have found that this is far from enough: the majority of conjunctions with two or more goals that are expensive enough to parallelize are dependant conjunctions, and most of these are dependant in such a way that they must almost be executed sequentially, usually because one goal's execution blocks on a variable that won't be produced until much later.

4 Goals

Our first goal is to detect all the parallelism implicit in a Mercury program — this information can be used by a parallelizing compiler to create efficient parallel Mercury by detecting all the profitable parallelism in a Mercury program. The more opportunities for parallelism that can be found, then the more optimally a program can be parallelized.

Given this, our second goal is to choose which set of these opportunities to take advantage of in order to automatically parallelize a program. This will involve parallelizing opportunities that have the best cost-benefit ratio, those that have the most parallelism due to their dependencies. When doing this, it is important to account for the effects that parallel evaluation of one opportunity will have on the benefits of parallelizing other opportunities.

5 Current Status

We have been able to automatically parallelize a number of small programs with abundant parallelism, including those with only dependant parallelism. For these programs we have recorded speed-ups that meet our expectations. [2]

We have also modified Mercury's deep profiler, making it possible to extract coverage information for every goal in a Mercury program. This is used by our analysis for measuring the parallel overlap between dependant conjuncts.

Based on ThreadScope [8] we are building tools to profile parallel Mercury programs. Note that this profiler is not related to the deep profiler. This enables us to improve Mercury's parallel runtime system and to improve our automatic parallelization tools. This will also help other developers profile their parallel Mercury programs.

We have also made a number of contributions to Mercury's parallel runtime system, making it more efficient.

6 Preliminary Results

Please see [2] for recent benchmarks. This paper was accepted into the ICLP 2011 conference as a full paper, and has been accepted for publication in TPLP.

Please also see [1] For a description of how we intend to modify ThreadScope to support the profiling of parallel Mercury programs. This paper was accepted into the WLPE 2011 workshop associated with ICLP.

I have also presented this research at the Multicore Miniconference associated with the Linux Conference Australia 2010 in Wellington, New Zealand. I have also been invited to speak at The University of New South Wales, and Google Australia.

7 Open issues

There are many ways in which we can improve our work. Firstly, we can use a best-first traversal of the call tree rather than a depth-first traversal. We can also use this to revisit nodes in the call tree after deciding to parallelize their siblings. We can also use parallelization as a specialization.

Often a loop may do very little work per iteration but may iterate many times. In these cases we should use granularity control to create fewer, larger parallel tasks. This should be tied to automatic parallelization so that the cost-benefit ratio of a granularity-controlled loop can be calculated.

We should also implement parallelization of dependant but commutative operations. When operations are commutative we can re-order them provided that the commutative operations are the only ones that are dependant. There are other cases where code can be re-ordered or transformed to improve the parallel speedup.

References

1 Paul Bone and Zoltan Somogyi. Profiling parallel Mercury programs with ThreadScope. In *Proceedings of the 21st Workshop on Logic-based methods in Programming Environments*, Lexington, Kentucky, 2011.

2 Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. In *Proceedings of the 27th International Conference on Logic Programming*, Lexington, Kentucky, 2011.

3 Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Annotation algorithms for unrestricted independent and-parallelism in logic programs. In *Proceedings of the 17th International Symposium on Logic-based Program Synthesis and Transformation*, pages 138–153, Lyngby, Denmark, 2007.

4 Thomas Conway. *Towards parallel Mercury.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, July 2002.

5 Thomas Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July 2001.

6 Daniel Cabeza Gras and Manuel V. Hermenegildo. Non-strict independence-based program parallelization using sharing and freeness information. *Theoretical Computer Science*, 410(46):4704–4723, 2009.

7 Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Notices*, 42(9):251–264, 2007.

8 Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 81–92, New York, NY, USA, 2009. ACM.

9 P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22(4):715–734, 1996.

10 Kalyan Muthukumar, Francisco Bueno, Maria J. García de la Banda, and Manuel V. Hermenegildo. Automatic compile-time parallelization of logic programs for restricted, goal level, independent AND-parallelism. *Journal of Logic Programming*, 38(2):165–218, 1999.

11 Kish Shen, Vïtor Santos Costa, and Andy King. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming*, 1999(1), 1999.

12 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

13 Peter Wang and Zoltan Somogyi. Minimizing the overheads of dependent AND-parallelism. In *Proceedings of the 27th International Conference on Logic Programming*, Lexington, Kentucky, 2011.