Automatic Parallelisation for Mercury

Paul Bone

Submitted in total fulfilment of the requirements of the degree of
Doctor of Philosophy

December 2012

Department of Computing and Information Systems
The University of Melbourne

Abstract

Multicore computing is ubiquitous, so programmers need to write parallel programs to take advantage of the full power of modern computer systems. However the most popular parallel programming methods are difficult and extremely error-prone. Most such errors are intermittent, which means they may be unnoticed until after a product has been shipped; they are also often very difficult to fix. This problem has been addressed by pure declarative languages that support explicit parallelism. However, this does nothing about another problem: it is often difficult for developers to find tasks that are worth parallelising. When they can be found, it is often too easy to create too much parallelism, such that the overheads of parallel execution overwhelm the benefits gained from the parallelism. Also, when parallel tasks depend on other parallel tasks, the dependencies may restrict the amount of parallelism available. This makes it even harder for programmers to estimate the benefit of parallel execution.

In this dissertation we describe our profile feedback directed automatic parallelisation system, which aims at solving this problem. We implemented this system for Mercury, a pure declarative logic programming language. We use information gathered from a profile collected from a sequential execution of a program to inform the compiler about how that program can be parallelised. Ours is, as far as we know, the first automatic parallelisation system that can estimate the parallelism available among any number of parallel tasks with any number of (non-cyclic) dependencies. This novel estimation algorithm is supplemented by an efficient exploration of the program's call graph, an analysis that calculates the cost of recursive calls (as this is not provided by the profiler), and an efficient search for the best parallelisation of N computations from among the 2^{N-1} candidates.

We found that in some cases where our system parallelised a loop, spawning off virtually all of its iterations, the resulting programs exhibited excessive memory usage and poor performance. We therefore designed and implemented a novel program transformation that fixes this problem. Our transformation allows programs to gain large improvements in performance and in several cases, almost perfect linear speedups. The transformation also allows recursive calls within the parallelised code to take advantage of tail recursion.

Also presented in this dissertation are many changes that improve the performance of Mercury's parallel runtime system, as well as a proposal and partial implementation of a visualisation tool that assists developers with parallelising their programs, and helps researchers develop automatic parallelisation tools and improve the performance of the runtime system.

Overall, we have attacked and solved a number of issues that are critical to making automatic parallelism a realistic option for developers.

Declaration

This is to certify that:

- the thesis comprises only my original work towards the PhD except where indicated in the Preface,
- due acknowledgement has been made in the text to all other material used,
- the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed:

Date:

Preface

The dissertation begins with an introduction (Chapter 1) that motivates the need for automatic parallelism systems. It provides an analysis of the related literature; using this to show why automatic parallelism is important and why the problem is still unsolved.

Chapter 2 provides background material used throughout the rest of the dissertation. In particular it describes prior work that I did as part of my Honours project for the degree of Bachelor of Computer Science Honours at the University of Melbourne. This work occurred before my Ph.D. candidature commenced. This work included: the profiler feedback framework (Section 2.4), an initial rudimentary version of the automatic parallelisation tool (also Section 2.4), an algorithm for determining when a sub-computation first uses a variable (Section 2.4.4), and the initial version of the coverage profiling support in Mercury's deep profiler (Section 2.4.3).

Chapter 3 discusses a number of improvements that were made to the runtime system, in order to make parallel Mercury programs perform better. Chapter 3 describes some joint work with Peter Wang. Wang contributed about 80% of the initial work stealing implementation described in Section 3.4; I contributed the other 20%. An improved version of work stealing is described in Section 3.6, but the work in that section is solely my own.

The next three chapters are based on published papers, of which I am the first author and contributed the vast majority of the work. Chapter 4 describes our new automatic parallelism analysis tool and its novel algorithms. It is based on the following journal article.

> Paul Bone, Zoltan Somogyi and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. *Theory and Practice of Logic Programming,* 11(4–5):575–591, 2011.

Chapter 5 describes a code transformation that fixes a long standing performance problem in the parallel execution of most recursive code. It is based on the following conference paper.

> Paul Bone, Zoltan Somogyi and Peter Schachte. Controlling Loops in Parallel Mercury Code. *Proceedings of the 7th Workshop on Declarative Aspects and Applications of Multicore Programming,* Philadelphia PA USA, January 2012.

Chapter 6 describes our use of the ThreadScope visual profiler that was developed for use with Haskell and how we use it with Mercury. It is based on the following workshop paper.

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

Finally Chapter 7 concludes the dissertation, providing a discussion that unifies the work presented in the four main chapters and describes potential further work.

Acknowledgements

First I will thank my supervisors, Zoltan Somogyi, Peter Schachte and Aaron Harwood for all their support, hard work, and on occasion weekends. I have learnt a lot from you and enjoyed your company and our interesting off-topic conversations. I also want to thank my Mercury, G12 and programming languages colleagues: Julien Fischer, Mark Brown, Ralph Becket, Thibaut Feydy, Matt Guica, Matt Davis, Andreas Schutt, Leslie DeKoninck, Sebastian Brand, Leon Mika, and Ondrej Bojar; and Mercury developers outside of the University: Peter Wang, Ian MacLarty, and Peter Ross. I also want to thank the helpful and supportive staff in the Computing and Information Systems department at the University of Melbourne, especially Linda Stern, Bernie Pope, Harald Søndergaard, and Lee Naish.

I would also like to acknowledge the support of the functional and logic programming research community. Firstly, I want to thank everyone who has contributed to the development of ThreadScope, helped organise the ThreadScope summit, and answered many of my questions, in particular Simon Marlow, Simon Peyton-Jones, Eric Kow and Duncan Coutts. Simon Marlow and his family were also incredibly hospitable during my visit to Cambridge. I also want to thank those at the University of New South Wales who invited me to present a seminar to their research group: Ben Lippmeier, Gabi Keller, and Manuel Chakravarty.

I received financial support from the following scholarships, prizes and organisations: Australian Postgraduate Award, NICTA Top-up Scholarship, NICTA travel support, Melbourne Abroad Travelling Scholarship, Google Travel Prize, Association for Logic Programming (ICLP), and Open Parallel.

I would like to offer special thanks to those who contributed time and effort in other ways. Ben Stewart for shell access to his multicore system early in my candidature, Chris King for his spectralnorm benchmark program, Michael Richter for assistance with proof reading.

I would like to offer a whole lot of thanks to my friends: Amanda & Chas Dean, Lucas & Jen Wilson-Richter, Sarah Simmonds, John Spencer, Jeff Beinvenu, Mangai Murugappan, Terry Williamson, Heidi Williams, Tom Wijgers, Ha Le, Emil Mikulic, Dave Grinton, Marco Maimone, Michael Slater, Geoff Giesemann, Enzo Reyes, Dylan Leigh, and Marco Mattiuzzo.

I want to thank my parents, Keith and Faye Bone, for their support and encouragement, but most of all, for teaching me that my poor vision might be an impairment, but should never be a limitation. Thanks to my brother, James Bone, for always having time to listen, even when I use incomprehensible jargon ("nerd-words").

Finally, Liz Bone, my wife, for her love patience understanding and caring and being generally awesome.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In 1965 Moore [86] predicted that the transistor density on processors would grow exponentially with time, and the manufacturing cost would fall. The smaller transistors are, the faster they can switch (given adequate power), and therefore manufacturers can ship faster processors. The industry celebrated this trend, calling it Moore's Law. However as faster processors require more power, they create more heat which must be dissipated. Without novel power saving techniques (such as Bohr et al. [16]), this limits increases of processors' clock speeds.

Around 2005 it became clear that significant improvements in performance would not come from increased clock speeds but from multicore parallelism [103]. Manufacturers now build processors with multiple processing cores, which can be placed in the same package, and usually on the same die. Individual cores work separately, communicating through the memory hierarchy.

Other methods of improving performance without increasing clock speed have also been tried.

- Modern processors perform super-scalar execution: processor instructions form a pipeline, with several instructions at different stages of execution at once, and by adding extra circuitry, several instructions may be at the same stage of execution. However, we have just about reached the limits of what super-scalar execution can offer.
- Manufacturers have also added Single Instruction Multiple Data (SIMD) instructions to their processors; this allows programmers to perform the same operation on multiple pieces of data. In practice however, SIMD is useful only in some specific circumstances.
- Multicore computing has the potential to be useful in many circumstances, and does not appear to have limitations that cannot be overcome. Cache coherency could be a limitation for processors with many cores. However there are solutions to this such as directory based memory coherency; there are also research opportunities such as making compilers responsible for cache management.

Multicore computing is a special case of multiprocessing. Most multiprocessing systems are symmetric multiprocessing (SMP) systems. An SMP system consists of several homogeneous processors and some memory connected together. Usually all processors are equally-distant from all memory location. Most multicore systems are SMP systems; they may have more than one CPU each with any number of cores. Some multiprocessing systems use a non-uniform memory architecture (NUMA). Usually this means that each processor has fast access to some local memory and slower access to the other processors' memories. A new type of architecture uses graphics

programming units (GPUs) to perform general purpose computing, they are called GPGPU architectures. However they are not as general purpose as their name suggests: they work well for large regular data-parallel and compute-intensive workloads, but do not work well for more general symbolic processing. GPGPUs give programs access to small amounts of different types of memory that must be allocated statically, however most symbolic programs rely on dynamic allocation of unpredictable amounts of memory. Additionally, symbolic programs often include code with many conditional branches; this type of code does not perform well on GPGPUs. GPGPU architectures are not as general purpose as SMP systems and SMP systems are vastly more common than NUMA systems. Therefore, in this dissertation we are only concerned with SMP systems as they are both more general and more common, making them more desirable targets for most programmers. Our approach will work with NUMA systems, but not optimally.

To use a multicore system, or multiprocessing in general, programmers must parallelise their software. This is normally done by dividing the software into multiple threads of execution which execute in parallel with one another. This is very difficult in imperative languages as the programmer is responsible for coordinating the threads [104]. Few programmers have the skills necessary to accomplish this, and those that do, still make expensive mistakes as threaded programming is inherently error prone. Bugs such as data corruption, deadlocks and race conditions can be extremely tedious to find and fix. These bugs increase the costs of software development. Software companies who want their software to out-perform their competitors will usually take on the costs of multicore programming. We will explain the problems with parallelism in imperative languages in Section 1.1.1.

In contrast to imperative languages, it is trivial to express parallelism in pure declarative languages. Expressing this parallelism creates two strongly-related problems. First, one must overcome the costs of parallel execution. For example, it may take hundreds of instructions to make a task available for execution on another processor. However, if that task only takes a few instructions to execute, then there is no benefit to executing it in parallel. Even if the task creates hundreds of instructions to execute, parallel execution is probably not worthwhile. Most easy-to-exploit parallelism is *fine grained* such as this. Second, an abundance of coarse grained parallelism can also be a problem. Whilst the amount of parallelism the machine can exploit cannot increase beyond the number of processors, the more parallel tasks a program creates, the more the overheads of parallel execution will have an effect on performance. In these situations, there is no benefit in parallelising many of the tasks, and yet the overheads of their parallel executions will still have an effect. This often cripples the performance of such programs. This is known as an *embarrassingly parallel* workload. Programs with either of these problems almost always perform more *slowly* than their sequential equivalents. Programmers must therefore find the parts of their program where parallel execution is profitable and parallelise those parts of their program *only*, whilst being careful to avoid embarrassing parallelism. This means that a programmer must have a strong understanding of their program's computations' costs, how much parallelism they are making available, and how many processors may be available at any point in the program's execution. Programmers are not good at identifying the hotspots in their programs or in many cases understanding the costs of computations, consequently programmers are not good at manually parallelising programs. Programmers are encouraged to use profilers to help them identify the hotspots in their programs and speed them up; this also applies to parallelisation.

Automatic parallelism aims to make it easy to introduce parallelism. Software companies will not need to spend as much effort on parallelising their software. Better yet, it will be easier for

programmers to take advantage of the extra cores on newer processors. Furthermore, as a parallel program is modified its performance characteristics will change, and some changes may affect the benefit of the parallelism that has already been exploited. In these cases automatic parallelism will make it easy to *re-parallelise* the program, saving programmers a lot of time.

1.1 Literature review

In this section we will explore the literature concerning parallel programming.

The languages in which parallelism is used have a strong influence on the semantics and safety of the parallelism. We group languages into the following two classifications.

Pure declarative programming languages are those that forbid *side effects*. A side effect is an action that a computation may perform without a declaration of the possibility of the action. The effect of executing any computation must be declared in the computation's signature. Because all effects are declared, none of them are *side* effects. This has numerous benefits, but we will restrict our attention to the specific benefit that this makes it easy for both compilers and programmers to understand if it is safe to parallelise any particular computation. Examples of pure declarative languages are Mercury, Haskell and Clean.

Impure programming languages are those that allow side effects. This includes imperative and impure declarative languages. Determining if parallelisation is safe is usually intractable and often requires whole program analysis. The analysis is almost always incomplete for programs whose size makes parallelism desirable. Thus parallelisation of programs written in impure languages is notoriously difficult. Examples of impure languages are C, Java, Prolog and Lisp; even though the last two of these are declarative languages, they still allow side effects, and are therefore impure.

There are many ways to introduce parallelism in computer programming. We find it useful to create the following categories for these methods.

- Parallelism via concurrency
- Language support for parallelism
- Feedback directed automatic parallelism

We explore the literature in this order as each successive category builds on the work of the previous category. Our own system is a feedback directed automatic parallelism system; it makes use of concepts described in all four categories. Therefore we cannot discuss it meaningfully until we have explored the literature of the giants upon whose shoulders we are standing.

1.1.1 Parallelism via concurrency

It is important to distinguish between parallelism and concurrency.

Concurrency is parallelism in the problem domain.

As an example consider a web server running on a single core, which is required to handle multiple requests at once. Concurrency is used by the programmer to help him write the webserver in a more natural style.

Parallelism is parallelism in the implementation domain.

> Consider a program that uses SIMD instructions to multiply matrices, performing several basic operations with a single instruction. Parallelism is used to reduce the elapsed execution time of the program.

Concurrency may be implemented using parallelism, because two independent tasks may be executed at the same time. Languages which have been designed for concurrent programming, and happen to be implemented using parallelism, can be used by programmers to achieve parallelism. Because the programmer uses concurrency to achieve parallelism, we call this "parallelism via concurrency". Unfortunately, this means that parallelism is both at the top and bottom levels of the software stack, with concurrency in the middle. The middle layer is superfluous, and should be avoided if possible. In particular, programmers should not have to shoe-horn their programs into a concurrent model if that model is not a natural fit for the program. This is particularly bad because it can make the program difficult to read, understand and debug. When a program can naturally be expressed using concurrency, then it should probably be written to use concurrency anyway, regardless of the benefits of parallelism. In this section we will discuss systems that allow parallelism through concurrency. Our critique of these systems should not be misunderstood as a critique of concurrency, which is outside the scope of this dissertation.

Concurrency notations such as Hoare's communicating sequential processes (CSP) [61] or Milner's π-calculus [85] provide the programmer with a natural ways to express many programs, such as webservers. In these cases concurrency is (and should be) used to make the program easier to write. If concurrency uses parallelism provided by the hardware, many programs will speed up due to this parallelism, but not all.

The most common method of introducing concurrency is with *threads*. Some languages, such as Java [88], support threads natively; others, such as C [65], support it via libraries, such as POSIX Threads [24] or Windows Threads [33]. These libraries tend to be low level; they simply provide an interface to the operating system. Different threads of execution run independently, sharing a heap and static data. Because they share these areas where data may be stored, they can, and often do, use the same data (unlike CSP). Threads communicate by writing to and reading from the shared data. Regions of code that do this are called *critical sections*, and they must protect the shared data from concurrent access: a thread reading shared data while another writes to it may read inconsistent data; and when two threads write to the same data, some of the changes may be lost or corrupted. Synchronisation such as *mutual exclusion locks* (mutexes) [37] must be used to manage concurrent access. Upon entering a critical section a thread must acquire one or more locks associated with the critical section, upon leaving it must release the locks. The use of locks ensures that only one thread may be in the critical section at a time. (There are other ways that locks may be used to allow more than one thread inside the critical section, such as the standard solution to the readers-writers problem.) The programmer is responsible for defining critical sections and using locks correctly.

Mutual exclusion locking is problematic, because it is too easy to make mistakes when using locks. One frequent mistake is forgetting to protect access to a critical section, or making the critical section too small. This can lead to incorrect program behaviour, inconsistent memory states and crashes. Making it too large can lead to deadlocks and livelocks. When a critical section uses multiple shared resources, it is normal to use multiple locks to protect access to a critical section. The locks must be acquired in the same order in each such critical section, otherwise deadlocks

can occur. This requirement for order prevents critical sections from being nestable, which causes significant problems for software composability. Failing to acquire locks in the correct order or failing to avoid nesting can also cause deadlocks. All of these problems are difficult to debug because their symptoms are intermittent. Often so intermittent that introducing tracing code or compiling with debugging support can prevent the problem from occurring; these things are known as *heisenbugs* and are very frustrating. While it is possible to deal with these problems, we believe that it is better to design problems out of systems; doing so guarantees that the programmer takes on fewer risks.

Another popular model of concurrency is message passing. Unlike threads, message passing's processes do not share static or heap data; processes communicate by passing messages to one another. The major benefit of message passing is that it does not make assumptions about where processes are executing: they can be running all on one machine or spread-out across a network. Therefore, message passing is very popular for high performance computing, where shared memory systems are not feasible. Message passing also avoids the problems with threading and mutexes. However message passing suffers from some of the same problems that threading suffers from. For example it is possible to create deadlocks with cyclic dependencies between messages: a process a will send a message to b but only after it has received a message from b, and b will send a message to a but only after it has received its message. (We can create a similar situation with locks.) Most deadlocks are more subtle and harder to find than this trivial example.

Notable examples of message passing application programming interfaces (APIs) for C and Fortran are MPI [82] and PVM [41]. Every process can send messages to and receive messages from any other process. We can think of this in terms of *mailboxes*: each process owns (and can read from) its own mailbox and can place messages into any mailbox. MPI and (to a lesser extent) PVM support powerful addressing modes such as multicasting the same message to multiple processes, or distributing the contents of an array evenly between multiple processes. These features make MPI and PVM very powerful interfaces for high performance computing. They make the programmer is responsible for encoding and decoding messages. While this adds extra flexibility, it can also be very tedious.

Communicating sequential processes [61] is a notation for concurrency using message passing. Although we introduced PVM and MPI first, CSP predates them by 11 years. Like MPI and PVM, CSP allows messages to be sent to and from processes using mailboxes. But unlike MPI and PVM, CSP allows processes to be created dynamically, making it more flexible. The first language based on CSP was occam [9, 73]. The π-calculus [85] was developed later. It uses *channels* rather than mailboxes for communication, although each model is a dual of the other. Unlike CSP, the π-calculus makes channels first class; they may be sent over a channel to another process. This allows programmers to compose a dynamic communication network.

The Erlang language [5] supports native message passing, and unlike MPI and PVM, encoding and decoding is implicit. Erlang is typically used where fault tolerance is important; whereas MPI and PVM are used for high performance computing. All three systems support distributed computing.

There are many other languages and libraries based on these process algebras such as Jocaml [78] (an Ocaml [70–72, 115] derivative) and Go [42]. Go is similar in style to the π-calculus. However it also uses shared memory without providing any synchronisation for communicating through shared memory. In particular data sent in messages may contain pointers, and therefore different processes may end up referring to the same memory locations. If one process modifies

such a memory location, other processes can see the modification. This creates the possibility that modifications to this data can cause a process to read inconsistent data or cause updates get lost or corrupted. This makes several classes of bug possible through accidental use of shared memory. This is a result of the language specification's inclusion of pointers and the implementation of channels.

All the systems we have introduced so far are prone to deadlocks, and many are prone to other problems. Many problems including deadlocks can be avoided using software transactional memory (STM) [97]. STM is similar to the threading model, in that it includes the concept of critical sections. STM does not protect critical sections using mutexes, instead it uses optimistic locking. Variables that are shared between threads are known as STM variables. They may only be modified within a critical section and a good implementation will enforce this. STM works by logging all the reads and writes to STM variables within a critical section, and not actually performing the writes until the end of the critical section. At that point the STM system acquires a mutex for each STM variable (in a pre-defined order to avoid deadlocks) and checks for sequential consistency by comparing its log of the observed values of the variables with the current values of the variables. If no inconsistencies are found (the variables' values are all the same), it then performs any delayed writes and unlocks the variables. Otherwise it must retry the critical section or fail. This is much safer than the threading model because the user does not need to use mutexes, and critical sections are now nestable. Code containing nested sections will perform the commit only when the outermost section is ready to commit. However, programmers must still place their critical sections correctly in their program. A common criticism of STM is that it performs poorly when there is a lot of contention as a failed critical section must be re-executed, and this often adds to the amount of contention. Including side-effecting computations inside an STM critical section can also be a problem. This can be avoided in a pure declarative language such as Haskell [64] or Mercury [101]. Harris, Marlow, Peyton-Jones, and Herlihy [54] describes an implementation for Haskell, and Mika [83] describes an implementation for Mercury.

1.1.2 Language support for parallelism

Parallelism can also be achieved without concurrency. This can be achieved with language annotations that do not affect the (declarative) semantics of the program. The annotations usually describe *what* should be parallelised, and rarely describe *how* to parallelise each computation. We will refer to this as *explicit parallelism* since parallelism is expressed directly by the programmer by the explicit annotations. In this section we also discuss *implicit parallelism*, which covers cases where programming languages make parallel execution the default method of execution.

OpenMP [35] is a library for C++ and Fortran that allows programmers to annotate parts of their programs for parallel execution. In OpenMP most parallelism is achieved by annotating loops. C++ is an imperative language and therefore the compiler generally cannot determine which computations are safe to parallelise. The programmer is responsible for guaranteeing that the iterations of the loops are independent, in other words, that no iteration depends on the results of any previous iterations. If an iteration makes a function call, then this guarantee must also be true for the callee and all transitive callees. The programmer must also declare the roles of variables, such as if they are shared between iterations or local to an iteration. This is necessary so that the compiled code is parallelised correctly. The programmer may specify how the iterations of the loop may be combined into parallel tasks (known as *chunking*). This means that to some

extent the programmer still describes *how* to parallelise the loop, although the programmer has much less control than with parallelism via concurrency systems. Provided that the programmer can annotate their loops correctly, explicit parallelism with OpenMP avoids the problems with threading and message passing.

In pure declarative languages a function has no side effects: all possible effects of calling a function are specified in its declaration. This is often done by expressing effects as part of the type system; Haskell [64] uses Monads [69], Mercury[1] [101] and Clean [21] use uniqueness-typing, and Disciple [74] which uses effect-typing. In any of these systems both the programmer and compiler can trivially determine if executing a particular computation in parallel is safe. In Haskell, Mercury and Clean (but not Disciple) the concept of a variable is different from that in imperative programming, it is not a storage location that can be updated, but a binding of a name to a value. Therefore variables that are shared (those that we would have to declare as shared in OpenMP) are really shared immutable values. Explicit parallelism in declarative languages is much easier and safer to use than in imperative languages. The next two sub-sections describe parallelism in functional and logic languages respectively.

Parallelism in functional languages

Pure functional languages like Haskell and Clean can be evaluated using a process known as *graph reduction*. Graph reduction works by starting with an expression and a program and applying reductions to the expression using the program. Each reduction uses an equation in the program to replace part of the expression with an equivalent expression. In a lazy language (or lazy implementation of a non-strict language) each reduction step performs *outermost reduction*. This means that the arguments of a function are not normally evaluated before a reduction replaces the function call with the right hand side of equation for that function. Some reductions require the evaluation of an argument, these include primitive operations and pattern matching. The system continues applying reductions until outermost symbol is a value. The resulting value may have parameters which are still expressions; they have not been evaluated. This is called weak head normal form (WHNF). Graph reduction itself can be parallelised. If a subgraph (an argument in the expression) is known to be required in the future, then it can be evaluated in parallel using the same graph reduction process. Several projects have experimented with this technique [7, 22, 89].

Because the programmer does not need to annotate their program this parallelism is implicit. While implicit parallelism is always safe[2] there is a serious performance issue. The cost of spawning off a parallel evaluation is significant, and so is the cost of communication between parallel tasks. There are also distributed costs throughout the execution of these systems: for example many parallel graph reduction systems introduce locking on every graph node. Such locking increases the overheads of evaluation even when no parallel tasks are created. Therefore parallelised computations should be large enough that the benefit of evaluating them in parallel outweighs the costs of doing so. Unfortunately most parallelism in an implicit parallel system is very fine grained, meaning that it is not large enough to make parallel evaluation worthwhile. There is also a lot of this parallelism. This means that the system spends most of its time managing the parallel tasks rather than evaluating the program. This is called embarrassing parallelism as the

[1] Mercury's uniqueness typing is part of its mode system. Despite this, it is still considered uniqueness typing.

[2] There are some systems that use explicit parallelism and allow side-effects such as Prolog implementations with extra-logical predicates and Multilisp with destructive update. Nevertheless the *intention* is that implicit parallelism should always be safe

amount of fine-grained parallelism can cause the evaluation to perform *more slowly* than sequential evaluation. Most other implicitly parallel systems share these problems. Many systems such as Peyton-Jones et al. [89] go to lengths to exploit only coarse grained parallelism. In our opinion the problem with implicit parallelism is that it approaches the parallel programming challenge from the wrong direction: it makes everything execute in parallel *by default* and then attempts to introduce sequential execution; rather than make sequential execution the default and then attempt to introduce parallel execution *only where it is profitable*. The former creates thousands if not millions of parallel tasks, whereas the latter creates far fewer, dozens or hundreds. Since most systems have only a handful of processors the latter situation is far more manageable having far fewer overheads; therefore we believe that it is the correct approach. In contrast to implicit parallelism, explicit parallelism allows the programmer to introduce parallelism only where they believe it would be profitable. This means that the default execution strategy is sequential, and far fewer parallel tasks are created.

Multilisp [49, 50] is an explicitly parallel implementation of Scheme. Like Scheme, Multilisp is not pure: it allows side effects. This means that as programmers introduce parallelism, they must be sure that parallel computations cannot be affected by side effects, and do not have side effects themselves. Therefore Multilisp is not safe as side effects may cause inconsistent results or corrupt memory. Multilisp programmers create parallelism by creating delayed computations called *futures*. These computations will be evaluated concurrently (and potentially in parallel). When the value of a future is required, the thread blocks until the future's value is available. Operations such as assignment and parameter parsing do not require the value of a future.

Haskell is a pure functional language. Being pure it does not have the problem with side effects that Multilisp does. The Haskell language standard specifies that it is non-strict[3]; in practice, all implementations use lazy evaluation most of the time. A subexpression may be evaluated eagerly if it is always required in a particular context. Haskell programmers can introduce explicit parallelism into their programs with two functions (`par` and `pseq`). Trinder et al. [107] first added these functions to GUM (A parallel Haskell implementation). Then Harris et al. [53] added them to the Glasgow Haskell Compiler (GHC). The `par` and `pseq` functions have the types:

```
par :: a -> b -> b
pseq :: a -> b -> b
```

They both take two arguments and return their second. Their declarative semantics are identical; however their *operational* semantics are different. The `par` function may spawn off a parallel task that evaluates its first argument to WHNF, and returns its second argument. The `pseq` function will evaluate its first argument to WHNF *and then* return its second argument. We can think of these functions as the `const` function with different evaluation strategies.

Unfortunately lazy evaluation interacts poorly with parallel evaluation. Because parallel tasks are only evaluated to WHNF, most tasks do not create enough work to make parallelisation worthwhile. Trinder et al. [108] tried to reduce this problem by allowing programmers to specify an evaluation strategy, which describes how deeply to evaluate expressions. This description is written in Haskell itself, This support is implemented as a library and it can describe higher order evaluation strategies. This does not solve the problem, it only gives the programmer a way to deal with it when they encounter it. It would be better to make it impossible for the problem to occur.

[3]Non-strict evaluation is more broadly defined than lazy evaluation

An alternative Haskell library called Monad.Par [80] requires parallel computations to be evaluated in full. However the programmer must provide typeclass instances for each of their own types that describe how to completely evaluate that type. While this still requires a non-trivial amount of effort from the programmer; it makes it harder for the programmer to make mistakes, since their program will not compile if they forget to provide a typeclass instance.

Regardless of how we manage lazyness in parallelism we expose, it still introduces a runtime cost. Every variable must be tested to determine if it is a thunk or a value, and while this cost occurs in sequential programs, it can be higher in parallel programs. Furthermore, this means that it is not always clear (even when one does use `pseq`, strategies or Monad.Par) which threads will perform which computations.

Parallelism in logic languages

Different logic programming languages use different evaluation strategies, but we will restrict our attention to those that use selective linear resolution with definite clauses (SLD resolution) [67]. SLD resolution attempts to answer a query by finding a Horn clause whose head has the same predicate name as the selected atom in the query, performing variable substitution and then performing a query on each conjunct in the clause's body. If a query of a conjunct fails then the clause fails and an alternative clause may be tried. Multiple clauses may succeed meaning that there are several solutions, and if no clauses succeed it means that there are zero solutions.

Prolog is the best known logic programming language based on SLD resolution. OR-parallel Prolog implementations may attempt to resolve a query against multiple clauses in parallel. In most logic programs that we have seen have very little OR-parallelism. In cases where there is abundant OR-parallelism, it is usually because the program uses a significant amount of non-deterministic search. Large search problems can be implemented more efficiently in constraint programming languages, even sequential ones. Additionally OR-parallelism is speculative: the current goal may be executed in a context where only the first result will be used, therefore searching for more results in parallel may be a waste of resources. Systems that attempt to exploit OR-parallelism include Aurora [77], Muse [3] and ACE [45]. (ACE also supports AND-parallelism.)

AND-parallel Prolog implementations introduce parallelism by executing conjuncts within clauses in parallel. AND-parallelism comes in two varieties, independent and dependent. Independent AND-parallelism is much simpler, it only parallelises goals whose executions are independent. There are various ways to prove the independence of goals with varying degrees of completeness. If we cannot prove that a group of goals is independent, then we must assume the goals are dependent. The simplest and weakest way to prove independence is to show that the goals have no variables in common, including aliased variables (free variables unified with one another). This can be done statically. However very few groups of goals are trivially independent. We can improve this by allowing the sharing of ground variables. Systems allowing this initially used runtime tests to check the groundness and independence of variables. These tests may traverse arbitrarily large data structures. The overheads of executing these checks (which must be executed regardless of their outcome) can outweigh the benefit of parallel execution. Some systems used compile time analysis to reduce the costs of the runtime checks. Despite this the remaining runtime tests' costs were still unbounded. More advanced systems such as Hermenegildo and Greene [58], Gras and Hermenegildo [44] and Hermenegildo and Rossi [59] perform more complicated compile time analyses. Some of these analyses require whole program analysis for completeness, making them

impractical for large programs. Furthermore, the Prolog language allows some operations that can prevent an analysis from being complete even if whole program information is available.

Prolog supports *logic variables*. When two or more free variables are unified with one another they behave as one variable; any later successful unification on any of the variables implicitly modifies the state of the other variables. During sequential execution part of the search space may bind variables to values. If this branch of the search fails, execution must backtrack and retract any bindings that were made on the failed branch. This is required so that alternative branches of the search tree are free to bind the variables to other values. With OR-parallelism a variable may be bound to distinct values in different parts of the search tree at once. There are many different ways to implement this, for example Muse stores variables for different parallel parts of the search tree in different address spaces. While this is simple to implement it performs poorly: extra work is required to copy the states of computations as new computations are executed in parallel. It is difficult to implement systems that perform well.

Also, in independent AND-parallel systems variable bindings are easy to manage correctly, but difficult to manage efficiently. Systems that combine independent AND-parallelism and OR-parallelism, such as PEPSys [8] and ACE [45], combine the costs of managing variables in either parallelism type.

Unfortunately, just as most programs do not contain much OR-parallelism, very few programs contain much independent AND-parallelism. For example, in the Mercury compiler itself, there are 69 conjunctions containing two or more expensive goals, but in only three of those conjunctions are the expensive goals independent (Section 4.1), Supporting dependent AND-parallelism in Prolog is much more difficult *because* of the management of variable bindings. Any conjunct in a dependent AND-parallel conjunction may produce bindings for any given variable. If different conjuncts produce different values for the same variable, then one of them must fail. Handling this correctly and efficiently is extremely difficult. In a system such as Ciao [57], which attempts to infer modes for predicates (but allows them to be specified), the producer of a variable may be known, greatly simplifying the management of that variable's bindings. The next problem is how variable communication can affect other conjuncts running in parallel. Most systems allow parallel conjuncts to communicate variable bindings as soon as they are produced. This allows conjoined code running in parallel, in particular non-deterministic search computations, that consume those bindings to detect failure earlier and therefore execute more efficiently. However if the producer of a binding is non-deterministic, and it produces the binding within the non-deterministic code, then a failure may require it to retract variable bindings that are shared in the parallel conjunction. Implementing this efficiently is difficult. Kernel Andorra Prolog [51] solves this problem by performing either AND-parallel execution or OR-parallel execution. It attempts to execute deterministic[4] conjoined goals in parallel. For Andorra's purposes, deterministic means "succeeds at most once". If all the goals in the resolvent have more than one matching clause, then it uses OR-parallelism to execute the matching clauses in parallel. This means that non-determinism cannot occur in parallel conjunctions.

Another solution to this problem involve restricting the Prolog language by removing backtracking. This creates committed-choice logic languages such as Concurrent Prolog [96, 106]; Parlog [29, 30] and Guarded Horn Clauses (GHC) [109]. When these systems encounter multiple clauses during execution they commit to only one of the clauses. Each of these system uses guarded clauses, which have the form: $H \leftarrow G_1 \wedge \ldots \wedge G_n : B_1 \wedge \ldots \wedge B_m$ where G_i and B_i are atoms

[4] Andorra uses a different definition for determinism than Mercury does. Section 2.1 explains Mercury's definition.

in the guard and body of the clause; H is the head of the clause. The evaluation of the body is committed to if and only if all unifications in the head and the guard are true. Declaratively the guard and body are conjoined. All three systems have two types of unifications:

***ask* unifications** occur only in one direction: one side of the unification is a term provided by the environment (the resolvent), the other side is a term in the clause head or guard. An ask unification may bind variables in the second term but not in the first.

***tell* unifications** allow binding in either direction; they can instantiate the resolvent's term.

Each system implements these differently: Concurrent Prolog uses a read-only annotation to identify ask unifications, and GHC uses scoping rules (tell unifications to non-local variables cannot occur in the head or guard). When an ask unification is used and a ordinary unification in its place would normally bind a variable, then the computation is suspended until the variable becomes instantiated allowing the ask unification to run without binding the variable. This mechanism ensures that enough information is available before the program commits to a particular clause. When a tell unification is used, the binding is made locally and is re-unified as the clause commits. This re-unification can add to the costs of execution, especially if failure is detected as speculative work may have been performed in the meantime. Note that in GHC, there cannot be any tell unifications in the guard and so re-unification is not required. However GHC suspends computations more often, which also adds to the costs of parallel execution; especially when most unifications can cause a task to suspend or resume. Newer *flat* versions of these languages do not allow predicate calls to be used in a guard [40, 106, 110], this means that unifications in clause bodies can only be tell unifications. Whilst this reduces the expressiveness of the languages, it allows them to use slightly simpler unification code in more places. However clause bodies can still include tell unifications can may provide a variable instantiation that allows some other blocked computation to resume. Therefore most unifications still incur these extra costs.

Pontelli and Gupta [91] describe a system that supports explicit dependent AND-parallelism in a non-deterministic language. Analyses determine a conservative set of shared variables in dependent conjunctions and then *guess* which conjunct produces each variable (usually the leftmost one in which the variable appears) and which conjuncts consume it. The consumers of a variable are suspended if they attempt to bind it and the producer has not yet bound it. The compiler names apart all the potentially shared variables so that bindings made by the producer can be distinguished from bindings made by consumers. As these analyses are incomplete (in particular mode analysis), the predicted producer may not be the actual producer as execution may follow a different path of execution than the one that was predicted. If this happens then the runtime system will dynamically change the guess about which conjunct is the producer. This system still requires plenty of runtime overhead and since it is non-deterministic it must handle backtracking over code that produces the values of shared variables. The backtracking mechanism used is described by Shen [98].

Most of the systems we have discussed use implicit parallelism. These are: Aurora, Muse, ACE, Andorra, Concurrent Prolog, Parlog, GHC and DDAS; while PEPSys uses explicit parallelism and Ciao can use either. The implicitly parallel systems parallelise almost everything. As we mentioned earlier this can lead to embarrassing parallelism. Therefore, it is better to create fewer coarse-grained parallel tasks rather than many fine-grained tasks. To achieve this, some systems use *granularity control* [36] to parallelise only tasks large enough such that their computational cost makes up for the cost of spawning off the task. While it is easy to either spawn off or not

spawn off a task, it is much more difficult to determine a task's cost. There are multiple different ways to estimate the cost of a task. For example King et al. [66] and Lopez et al. [76] construct cost functions at compile time which can be evaluated at runtime to estimate whether the cost of a goal is above a threshold for parallel execution given the sizes of certain terms. During compile time the Lopez et al. paper uses the type, mode and determinism information that the Ciao [57] compiler makes available. The King et al. paper uses a convex hull approximation to create its cost functions. This approximation is very useful in this domain, it makes the analysis of recursive code feasible in modest memory sizes, and not too much information is lost during the approximation. Shen et al. [99] created a different method for granularity control, which measures the "distance" between the creation of successive parallel tasks. They propose two methods for increasing this distance which can create more coarse grained parallelism and usually spawns off less parallel work (reducing the cost of parallel execution in the program). Their first method works at compile time by transforming a recursive predicate so that it spawns parallel tasks less frequently. Their second method works at runtime to achieve the same means, it uses a counter and will not spawn off parallel work until the counter reaches a predetermined value. Once it reaches such a value, then the program will spawn off some parallel work and reset the counter. The counter can be incremented based on some indication of how much work is being done between tasks, such as abstract machine instructions executed. These methods can help to reduce the amount of embarrassing parallelism, but the compiler-based method can be difficult to implement correctly for all programs, and the runtime-based method introduces extra runtime costs. Finding profitable parallelism is still a difficult problem.

Mercury is a pure logic/functional language, with strong and static type, mode and determinism systems. We will discuss Mercury in more detail throughout Chapter 2. Conway [31] introduced explicit independent AND-parallelism to Mercury in deterministic code, code with exactly one solution (see also Section 2.2). Later Wang [113]; Wang and Somogyi [114] implemented support for dependent AND-parallelism. Mercury's mode system allows the compiler to determine the exact locations of variable bindings. This allowed Wang to implement a transformation that efficiently handles dependent AND-parallelism (see also Section 2.3). Because Mercury only supports parallelism in deterministic code, variable bindings made by dependent conjuncts can never be retracted. Some of our critics (reviews of the paper Bone et al. [19] on which Chapter 4 is based) are concerned that this reduces the amount of parallelism that Mercury is able to exploit *too much*. However, roughly 75% of Mercury code is deterministic, and most of the rest is semi-deterministic (it produces either zero or one solutions). We gathered this determinism information from the largest open source Mercury program, the Mercury compiler; our experience with other Mercury programs shows a similar pattern. It may appear that we are missing out on parallelising 25% of the program, but most of the parallelism in this code is speculative and therefore more difficult to parallelise effectively. Furthermore, the 75% that we can parallelise accounts for 97% of the runtime (Table 4.1) when tested on a single program (the compiler). Parallelising the remaining 3% would require supporting parallelism in at least semi-deterministic code which has higher runtime costs. It is also unlikely that this part of the program contains much profitable parallelism as most semi-deterministic code is in the goals of if-then-else goals. The Prolog systems that support parallelism in goals that can fail and goals with more than one solution must manage variable bindings in very complicated ways; this adds a lot of runtime cost. In contrast, Mercury's simpler implementation allows it to handle a smaller number of predicates, *much* more efficiently.

In Prolog, the management of variable bindings in dependent AND-parallel programs adds extra

overhead to unifications of all variables as most Prologs cannot determine which variables are shared in parallel conjunctions, and none can determine this accurately. Mercury's implementation, which uses futures (Section 2.3), increases the costs of unification only for shared variables. A related comparison, although it is not specific to parallelism, is that unification is an order of magnitude simpler in Mercury than in Prolog as Mercury does not support logic variables. (The various WAM unification algorithms have roughly 15–20 steps, whereas the most complicated Mercury unification operation has 3 steps.) This was also a deliberate design decision.

Other language paradigms

Some languages allow parallelism in other forms, or are created specifically to make parallelism easy to expose. Parallel languages that do not fall into one of the above paradigms are assessed here.

Sisal [38] was a data-flow language. Its syntax looked like that of a imperative language, but this was only syntactic sugar for a pure functional language. It was also one of the first single assignment languages; using single assignment to implement its pure functional semantics. Sisal supported implicit data-flow parallelism. Sisal's builtin types included arrays and streams. Streams have the same form as cons-lists (Lisp style) except that as one computation produces results on the stream, another computation may be reading items from the stream. The concept of streams is not new here, it has been seen in Scheme [1] and probably other languages. A stream may have only one producer, which makes them different from channels. The other source of parallelism was the parallel execution of independent parts of loop iterations; as one iteration is being completed, another iteration can be started. It could also handle completely independent parallel loops. Sisal's for loops included a reduction clause, which stated how the different iterations' results were summarised into a single overall result, which may be a scalar, an array or a stream. A loop's body must not have any effects or make any variable bindings that are visible outside the loop, so a loop's only result is the one described by the result clause. The Sisal project is long dead, which is unfortunate as it appeared to be good for expressing data-flow parallel programs. Sisal performed very well, giving Fortran a run for its money: in some tests Sisal outperformed Fortran and in others Fortran outperformed Sisal [38].

Data-parallel languages such as Data Parallel Haskell (DpH) [27, 90], NESL [12] and C* [93] parallelise operations on members of a sequence-style data structure. DpH calls them parallel arrays, NESL calls them sequences, and C* calls them domains. DpH and NESL can handle some forms of nested parallelism. Any operations on these structures are executed in parallel. NESL and C* have been targeted towards SIMD hardware and DpH has been targeted towards multicore hardware. Data-parallelism's strength is that its computations are *regular*, having a similar, if not identical, shape and operations. This makes targeting SIMD hardware, and controlling granularity on SMP hardware easier in these systems than most others.

Data parallel and data-flow parallel languages do not have the same problems (such as embarrassing parallelism) as implicit parallel languages. This is because programs with data-parallelism (and to a lesser extent data-flow parallelism) have a more regular structure, which allows these systems to generate much more efficient parallel code. However, very few programs have either data-parallelism or data-flow parallelism and cannot be parallelised using these systems. Therefore, these techniques cannot be used to parallelise most programs.

Reform Prolog [11] handles data-parallelism in a logic language. It finds singly recursive Prolog

predicates, separating their recursive code into the code before the recursive call and the code after. It compiles them so that they execute in parallel the code before the recursive call on all levels of the recursion, then they execute the base case, followed by parallel execution of the code after the recursive call on all levels of the recursion. Reform Prolog only does this for deterministic code (which we do not think is a problem, as we do the same in Mercury). However Reform Prolog has other limitations: the depth of the recursion must be known before the loop is executed, parallel execution cannot be nested, and no other forms of parallelism are exploited. This means that like the languages above, Reform Prolog cannot effectively parallelise most programs.

1.1.3 Feedback directed automatic parallelisation

As we mentioned above, we believe that it is better to start with a sequential program and introduce parallelism only where it is beneficial, rather than parallelising almost all computations. Programmers using an explicit parallelism system could follow this principle. However for each computation the programmer considers parallelising, they must know and keep track of a lot of information in order to determine if parallel execution is worthwhile. In the context of profiling and optimisation, it is widely reported that programmers are poor at identifying the hotspots in their programs or estimating the costs of their program's computations in general. Programmers who are parallelising programs must also understand the overheads of parallelism such as spawning parallel tasks and cleaning up completed parallel tasks. Other things that can also affect parallel execution performance including operating system scheduling and hardware behaviour such as cache effects. Even if a programmer determines that parallel execution is worthwhile in a particular case, they must also be sure that adding parallel execution to their program does not create embarrassing parallelism. They must therefore understand whether there will be enough processors free at runtime to execute each spawned off computation. When parallel computations are dependent, such as when futures are used in Multilisp or dependent parallelism is used in Mercury, programmers must also estimate how the dependencies effect parallelism. In practice programmers will attempt to use trial and error to find the best places to introduce parallelism, but an incorrect estimation of the magnitude of any of the above effects can prevent them from parallelising the program effectively.

We therefore advocate the use of feedback directed parallelism. This technique gathers cost information from a previous typical execution of the program, and uses that information to estimate the costs of the program's computations in future executions of the program, parallelising it accordingly. This is important because the cost of a computation often depends on the size of its input data. Feedback directed parallelism avoids the runtime cost of such a calculation (as the size of the input data is not available except at runtime or through profiling). It also avoids the human-centric issues with explicit parallelism.

Harris and Singh [52] developed a profiler feedback directed automatic parallelisation approach for Haskell programs. They reported speed ups of up to a factor of 1.8 compared to the sequential execution of their test programs on a four core machine. However they were not able to improve the performance of some programs, which they attributed to a lack of parallelism available in these programs. Their partial success shows that automatic parallelisation is a promising idea and requires further research. Their system works by measuring the execution time of thunks and using this information to inform the compiler how to parallelise the program. Any thunk whose execution time is above a particular threshold is spawned off for parallel evaluation. This has

two problems, both due to Haskell's lazy evaluation. The first is that their system does not also introduce calls to `pseq`, which would force the evaluation of thunks in order to improve granularity. Therefore Harris and Singh's method has the same problems with lazyness that the introduction of `pseq` attempts to overcome (see Section 1.1.2). The second problem is that thunk execution is speculative: when a thunk is created it is usually unknown whether the thunk's value will actually be used; and if it is known at compile time that the thunk will be used, the Glasgow Haskell Compiler (GHC) will optimise it so that the computation is not represented by a thunk at all and is evaluated eagerly.

Tannier [105] previously attempted to automatically parallelise Mercury programs using profiler feedback. His approach selected the most expensive predicates of a program and attempted to parallelise conjunctions within them. This is one of the few pieces of work that attempts to estimate how dependencies affect the amount of parallelism available. Tannier's approach counted the number of shared variables in a parallel conjunction and used this as an analogue for how restricted the parallelism may be. However in practice most producers produce dependent variables late in their execution and most consumers consume them early. Therefore Tannier's calculation was naive: the times that these variables are produced by one conjunct and consumed by the other usually do not correlate with the number of dependent variables. Tannier's algorithm was, in general, too optimistic about the amount parallelism available in dependent conjunctions. Tannier did made use of compile-time granularity control to reduce the over-parallelisation that can occur in recursive code.

To improve on Tannier's methods, my honours project [17] aimed to calculate when the producing conjunct is most likely to produce the dependent values and when the consuming conjunct is likely to need them. To do this I modified Mercury's profiler so that it could provide enough information so that I could calculate the likely production and consumption times. This early work was restricted to situations with a single dependent variable shared between two conjuncts. It is described in more detail in Section 2.4.

1.2 General approach

Unfortunately automatic parallelisation technology is yet to be developed to the point where it is generally useable. Our aim is that automatic parallelisation will be easy to use and will parallelise programs more effectively that most programmers can by hand. Most significantly, automatic parallelism will be very simple to use compared with the difficulty of manual parallelisation. Furthermore as programs change, costs of computations within them will change, and this may make manual parallelisations (using explicit parallelism) less effective. An automatic parallelisation system will therefore make it easier to maintain programs as the automatic parallelisation analysis can simply be redone to re-parallelise the programs. We are looking forward to a future where programmers think about parallelism no more than they currently think about traditional compiler optimisations.

In this dissertation we have solved several of the critical issues with automatic parallelism. Our work is targeted towards Mercury. We choose to use Mercury because it already supports explicit parallelism of dependent conjunctions, and it provides the most powerful profiling tool of any declarative language, which generates data for our profile feedback analyses. In some ways our work can be used with other programming languages, but most other languages have significant barriers. In particular automatic parallelism can only work reliably with declaratively pure languages, the

language should also use a strict evaluation strategy to make it easy to reason about parallel performance, and in the case of a logic language, a strict and precise mode system is required to determine when variables are assigned their values. Mercury's support for parallel execution and the previous auto-parallelisation system [17] is described in Chapter 2. In this dissertation we make a number of improvements to Mercury's runtime system that improve the performance of parallel Mercury programs (Chapter 3). In Chapter 4 we describe our automatic parallelism analysis tool and its algorithms, and show how it can speedup several programs. In Chapter 5 we introduce a new transformation that improves the performance of some types of recursive code and achieve almost perfect linear speedups on several benchmarks. The transformation also allows recursive code within parallel conjunctions to take advantage of tail recursion optimisation. Chapter 6 describes a proposal to add support for Mercury to the ThreadScope parallel profile visualisation tool. We expect that the proposed features will very useful for programmers and researchers alike. Finally in Chapter 7 we conclude the dissertation, tieing together the various contributions. We believe that our work could also be adapted for other systems; this will be easier in similar languages and more difficult in less similar languages.

Chapter 2

Background

In this chapter we present background material for the remainder of the thesis. This material starts with a description of the relevant parts of the Mercury programming language (Section 2.1), and how explicit parallelism in Mercury is implemented (Sections 2.2 and 2.3). Lastly, Section 2.4 describes preexisting automatic parallelisation implementations for Mercury, which includes algorithms that we use later in the thesis.

2.1 Mercury

Mercury is a pure logic/functional programming language intended for the creation of large, fast, reliable programs. Although the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are different due to Mercury's purity and its type, mode, determinism and module systems.

Mercury programs consist of modules, each of which has a separate namespace. Individual modules are compiled separately. Each module contains predicates and functions. Functions are syntactic sugar for predicates with an extra argument for the their result. In the remainder of this dissertation we will use the word predicate to refer to either a predicate or a function.

A predicate or function P is defined in terms of a goal G:

$$P \quad : \; p(X_1, \ldots, X_n) \; \leftarrow \; G \quad \text{predicates}$$

Mercury's goal types are shown in Figure 2.1. The atomic goals are unifications (which the compiler breaks down until each one contains at most one function symbol), plain first-order calls, higher-order calls, typeclass method calls (similar to higher-order calls), and calls to predicates defined by code in a foreign language (usually C). The compiler may inline foreign code definitions when generating code for a Mercury predicate. To allow this, the representation of a foreign code construct includes not just the name of the predicate being called, but also the foreign code that is the predicate's definition and the mapping from the Mercury variables that are the call's arguments to the names of the variables that stand for them in the foreign code. The composite goals include sequential and parallel conjunctions, disjunctions, if-then-elses, negations and existential quantifications. Section 2.2 contains a detailed description of parallel conjunctions.

The abstract syntax does not include universal quantifications: they are allowed at the source level, but are transformed into a combination of two negations and an existential quantification

G	: $X = Y$	unifications
	$\mid p(X_1, \ \ldots, \ X_n)$	predicate calls
	$\mid X_0(X_1, \ \ldots, \ X_n)$	higher order calls
	$\mid m(X_1, \ \ldots, \ X_n)$	method calls
	$\mid$ foreign$(p, [X_1 : \ Y_1, \ \ldots, X_n : \ Y_n], \mathit{foreign\ code})$,	foreign code
	$\mid (G_1, \ \ldots, \ G_n)$	sequential conjunctions
	$\mid (G_1 \ \& \ \ldots \ \& \ G_n)$	parallel conjunctions
	$\mid (G_1; \ldots; G_n)$	disjunctions
	$\mid$ switch $X \ (f_1 : \ G_1 \ ; \ \ldots \ , f_n : \ G_n)$	switches
	$\mid (\mathit{if}\ G_{cond}\ \mathit{then}\ G_{then}\ \mathit{else}\ G_{else})$	if-then-elses
	$\mid \mathit{not}\ G$	negations
	$\mid \mathit{some}\ [X_1, \ldots, X_n]\ G$	existential quantification
	$\mid \mathit{promise\_pure}\ G$	purity promise
	$\mid \mathit{promise\_semipure}\ G$	purity promise

Figure 2.1: The abstract syntax of Mercury

(Equation 2.1). Universal quantifications do not appear in the compiler's internal goal representation. Similarly, a negation can be simplified using the constants $true$ and $false$, and an if-then-else (Equation 2.2). However the compiler does not make this transformation and negations may appear in the compiler's goal representation. Nevertheless, we will not discuss negations in this dissertation, since our algorithms' handling of negations is equivalent to their handling of the below if-then-else.

$$\mathit{all}\ [X_1, \ \ldots, \ X_n]\ G \ \rightarrow \ \mathit{not}\ (\ \mathit{some}\ [X_1, \ \ldots, \ X_n]\ (\ \mathit{not}\ G\ )) \tag{2.1}$$

$$\mathit{not}\ G \ \rightarrow \ (\mathit{if}\ G\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true}) \tag{2.2}$$

A switch is a disjunction in which each disjunct unifies the same bound variable with a different function symbol. Switches in Mercury are thus analogous to switches in languages like C. If the switch contains a case for each function symbol in the switched-on value's type, then the switch is said to be *complete*. Otherwise the switch is *incomplete*. Note that the representation of switches in Figure 2.1 is not the same as their source code representation; Their source code syntax is the same as a disjunction. See page 23 for a description of the purity promises.

Note that because a unification goal contains at most one function symbol then any unifications that would normally have more than one function symbol are expanded into a conjunction of smaller unifications. This also happens to any unifications that appear as arguments in calls. A goal for a single expression such as the quadratic equation:

```
X = (-B + sqrt(B^2 - 4*A*C)) / (2*A)
```

Will be expanded into 12 conjoined goals:

```
V_1  = 2,              V_2  = B ^ V_1,        V_3 = 4,
V_4  = V_3 * A,        V_5  = V_4 * C,        V_6 = V_2 - V_5,
V_7  = sqrt(V_6),      V_8  = -1,             V_9 = V_8 * B,
V_10 = V_9 + V_7,      V_11 = V_1 * A,        X   = V_10 / V_11
```

As shown here, constants such as integers must be introduced as unifications between a variable

```
:- pred append(list(T), list(T), list(T)).

append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :-
    append(Xs, Ys, Zs).
```

Figure 2.2: `append/3`'s definition and type signature

and a function symbol representing the constant. Goals flattened in this way are said to be in *super homogeneous form.*

Mercury has a type system inspired by the type system of Hope [23] and is similar to Haskell's type system [64]; it uses Hindley-Milner [60, 84] type inference. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference). Types can be parametric, type parameters are written in uppercase and concrete types are written in lower case. As an example, the maybe type is defined as:

```
:- type maybe(T)
    --->    yes(T)
    ;       no.
```

A value of this type is a discriminated union; which is either `yes(T)`, indicating the presence of some other value whose type is `T`; or simply `no`. The maybe type is used to describe a value that may or may not be present. The type `maybe(int)` indicates that an integer value might be available. The list type uses a special syntax for nil and cons and is recursive:

```
:- type list(T)
    --->    [T | list(T)]       % cons
    ;       [].                 % nil
```

`append/3`'s type signature and definition are shown in Figure 2.2. Mercury also has a strong mode system. At any given point in the program a variable has an *instantiation state*, this is one of free, ground or clobbered. Free variables have no value, ground variables have a fixed value, clobbered variables once had a value, but it is no longer available. Partial instantiation can also occur, but the compiler does not fully support partial instantiation and therefore it is rarely used. Variables that are ground may also be described as unique, meaning they are not aliased with any other value. When variables are used, their instantiation state may change; such a change is described by the initial and final instantiation states separated by the `>>` symbol. The instantiation states are the same if nothing changes. Variables can only become *more instantiated*: the change 'free to ground' is legal, but 'ground to free' is not. Similarly, uniqueness cannot be added to a value that is already bound or ground. Commonly used modes such as input (`in`), output (`out`), destructive input (`di`) and unique output (`uo`) are defined in the standard library:

```
:- mode in  == ground >> ground.
:- mode out == free >> ground.
:- mode di == unique >> clobbered.
:- mode uo == free >> unique.
```

A predicate may have multiple modes. Here are two mode declarations for the `append/3` predicate above; these are usually written immediately after the type declaration.

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

When a predicate has a single mode, the mode declaration may be combined with the type signature as a single declaration such as the following declaration for `length/2`, the length of a list:

```
:- pred length(list(T)::in, int::out) is det.
```

The Mercury compiler generates separate code for each mode of a predicate or function, which we call a *procedure*. In fact, the compiler handles individual procedures as separate entities after mode checking. Mode correct conjunctions, and other conjoined goals such as the condition and then parts of an if-then-else, must produce values for variables before consuming them. This means that variables passed as input arguments must be ground before and after the call and variables passed as output arguments must be free before the call and ground after the call. Similarly, a variable passed in a destructive input argument will be destroyed before the end of the call; it must be unique before the call and clobbered afterwords. Referencing a clobbered variable is illegal. A variable passed in a unique output argument must initially be free and will become unique[1].

Unifications can also modify the instantiation state of a variable, Unifying a free term with a ground term will make the first term ground, and the second term non-unique. Unifying a ground term with a ground term is an equality test, it does not affect instantiation. Unifying two free terms is illegal[2]. The instantiation state of each variable is known at each point within a procedure, therefore the compiler knows exactly where each variable becomes ground. The mode checking pass will minimally reorder conjuncts (in both sequential and parallel conjunctions) so that multiple procedures created from the same predicate can be mode correct.

The mode system enforces three invariants:

Conjunction invariant: In any set of conjoined goals, including not just conjuncts in conjunctions but also the condition and then-part of an if-then-else, each variable that is consumed by any of the goals is produced by exactly one earlier goal. The else part of an if-then-else is not conjoined with the condition part so this invariant does not apply to the condition and else parts.

Branched goal invariant: In disjunctions, switches or if-then-elses, the goal types that contain alternative branches of execution, each branch of execution must produce the exact same set of variables that are consumed from outside the branched goal, with one exception: a branch of execution that cannot succeed (see determinisms below) may produce a subset of this set of variables.

Negated goal invariant: A negated goal may not bind any variable that is visible to goals outside it, and the condition of an if-then-else may not bind a variable that is visible anywhere except in the then-part of that if-then-else.

[1] The `di` and `uo` modes are often used together to allow the compiler to destructively update data structures such as arrays.

[2] Aliasing free variables is illegal, Mercury does not support logic variables. This is a deliberate design decision; it makes unification in Mercury much faster than unification in Prolog.

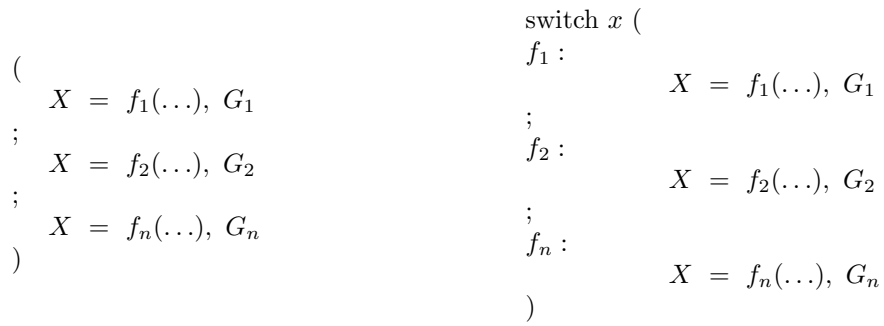


Figure 2.3: Switch detection example

Each procedure and goal has a determinism, which puts upper and lower bounds on the number of the procedure's possible solutions (in the absence of infinite loops and exceptions). Mercury's determinisms are:

`det` procedures succeed exactly once (upper bound is one, lower bound is one).

`semidet` procedures succeed at most once (upper bound is one, no lower bound).

`multi` procedures succeed at least once (lower bound is one, no upper bound).

`nondet` procedures may succeed any number of times (no bound of either kind).

`failure` procedures can never succeed (upper bound is zero, no lower bound).

`erroneous` procedures have an upper bound of zero and a lower bound of one, which means they can neither succeed nor fail. They must either throw an exception or loop forever.

`cc_multi` procedures may have more than one solution, like `multi`, but they commit to the first solution. Operationally, they succeed exactly once.

`cc_nondet` procedures may have any number of solutions, like `nondet`, but they commit to the first solution. Operationally, they succeed at most once.

Examples of determinism declarations can be seen above in the declarations for `append/3` and `length/2`. In practice, most parts of most Mercury programs are deterministic (`det`). Each procedure's mode declaration typically includes its determinism (see the mode declarations for `append/3` above). If this is omitted, the compiler can infer the missing information.

Before the compiler attempts to check or infer the determinism of each procedure, it runs a switch detection algorithm that looks for disjunctions in which each disjunct unifies the same input variable (a variable that is already bound when the disjunction is entered) with any number of different function symbols. Figure 2.3 shows an example. When the same variable is used with multiple levels of unification in a disjunction switch detection will rely on a form of common subexpression elimination to generate nested switches.

The point of this is that it allows determinism analysis to infer much tighter bounds on the number of solutions of the goal. For example, if each of the G_i is deterministic (i.e. it has determinism `det`) and the various f_i comprise all the function symbols in X's type, then the switch can be inferred to be deterministic as well, whereas a disjunction that is *not* a switch and produces at least one variable cannot be deterministic, since each disjunct may generate a solution.

```
:- pred main(io::di, io::uo) is det.

main(S0, S) :-
    io.write_string("Hello ", S0, S1),
    io.write_string("world\n", S1, S).
```

Figure 2.4: Hello world and I/O example

Disjunctions that are not converted into switches can be classified into two separate types of disjunction. If a disjunction produces one or more variables then it may have more than one solution (it will be `nondet`, `multi`, `cc_nondet` or `cc_multi`). If a disjunction does not produce any variables then it has at most one solution (it will be either `semidet` or `det`). Even when there are different execution paths that may succeed, the extra execution paths do not affect the program's semantics. In practice disjunctions that produce no values are `semidet` (if they where `det` the programmer would have not written them) therefore we call them *semidet disjunctions*. The compiler will generate different code for them, since they never produce more than one solution.

Mercury has a module system. Calls may be qualified by the name of the module that defines the predicate or function being called, with the module qualifier and the predicate or function name separated by a dot. The `io` (Input/Output) module of the Mercury standard library defines an abstract type called the `io` state, which represents the entire state of the world outside the program. The `io` module also defines a large set of predicates that do I/O. These predicates all have determinism `det`. and besides other arguments, they all take a pair of `io` states whose modes are respectively `di` and `uo` (as discussed above, `di` being shorthand for *destructive input* and `uo` for *unique output*). The `main/2` predicate that represents the entire program (like `main()` in C) also has two arguments, a `di,uo` pair of `io` states. A program is thus given a unique reference to the initial state of the world, every I/O operation conceptually destroys the current state of the world and returns a unique reference to the new state of the world, and the program must return the final state of the world as the output of `main/2`. Thus, the program (`main/2`) defines a relationship between the state of the world before the program was executed, and the state of the world when the program terminates. These conditions guarantee that at each point in the execution there is exactly one current state of the world.

As an example, a version of "Hello world" is shown in Figure 2.4. `S0` is the initial state of the world, `S1` is the state of the world after printing "`Hello `", and `S` is the state of the world after printing "`world\n`" as well. Note that the difference e.g. between `S1` and `S` represents not just the printing of "`world\n`", but also all the changes made to the state of the world by *other* processes since the creation of `S1`. This threaded state sequentialises I/O operations: "`world\n`" cannot be printed until the value for `S1` is available.

Numbering each version of the state of the world (or any other state that a program may pass around) is cumbersome. Mercury has syntactic sugar to avoid the need for this, but this sugar does not affect the compiler's internal representation of the program. This syntactic sugar is shown below, along with the convention of naming the I/O state `!IO`. The compiler will transform the example below into the example in Figure 2.4.

```
:- pred main(io::di, io::uo) is det.

main(!IO) :-
```

```
io.write_string("Hello ", !IO),
io.write_string("world\n", !IO).
```

Any term beginning with a bang (`!`) will be expanded into a pair of automatically named variables. The variable names created sequence conjunctions from left to right in conjunctions when used with pairs of `di/uo` or `in/out` modes.

Mercury divides goals into three purity categories:

pure goals have no side effects and their outputs do not depend on side effects;

semipure goals have no side effects but their outputs may depend on other side effects;

impure goals may have side effects, and may produce outputs that depend on other side effects.

Semipure and impure predicates and functions have to be declared as such, and calls to them must be prefaced with either `impure` or `semipure` (whichever is appropriate). The vast majority of Mercury code is pure, with impure and semipure code confined to a few places where it is used to implement pure interfaces. (For example, the implementations of the all-solutions predicates in the Mercury standard library use impure and semipure code.) Programmers put `promise_pure` or `promise_semipure` wrappers around a goal to promise to the compiler (and to human readers) that the goal as a whole is pure or semipure respectively, even though some of the code inside the goal may be less pure.

The compiler keeps a lot of information associated with each goal, whether atomic or not. This includes:

- the set of variables bound (or *produced*) by the goal;
- the determinism of the goal.
- the purity of the goal
- the *nonlocal set* of the goal, which means the set of variables that occur both inside the goal and outside it; and

The language reference manual [55] contains a complete description of Mercury.

2.2 Explicit parallelism in Mercury

The Mercury compiler has several backends, in this dissertation we are only concerned with the low level C backend. The low level C backend uses an abstract machine implemented using the C preprocessor. The abstract machine has 1024 general purpose registers and some special registers, such as the program counter. These registers are virtual: they are mapped onto global variables. Commonly used virtual registers are mapped to real machine registers. The number of virtual registers mapped onto real machine registers depends upon the architecture of the physical machine.

The compiler knows the determinism of every procedure. It will generate more efficient code for deterministic procedures. In particular, code with at most one solution uses the *det stack* (deterministic stack), which operates in much the same way as a stack in an imperative language. The det stack has a single pointer called the stack pointer, it points to the top of the stack. Code with one or more solutions uses the *nondet stack*: When a nondeterministic procedure produces

a solution it returns control to its caller but it leaves its stack frame on the nondet stack so that the procedure's variables are available on backtracking. Therefore, two stack pointers are used with the nondet stack, the first points to the top of the stack where new frames are allocated and the second points to the current stack frame. Mercury only supports parallel conjunctions in deterministic code, therefore the nondet stack and its two stack pointers are largely irrelevant for this dissertation. Somogyi, Henderson, and Conway [101] explains Mercury's execution algorithm in more detail.

To support parallel execution, Conway [31] grouped the variables used to implement the abstract machine's registers into a structure called an *engine*. Each engine corresponds to a POSIX Thread (pthread) [24]. Multiple engines can reside in memory at the same time, allowing each pthread to maintain its own execution state. The number of engines that a parallel Mercury program will allocate on startup is configurable by the user. Once the runtime system has been stared the number of engines in use is fixed.

In non-parallel grades a single engine is allocated statically and is known to reside at a particular address. However, in parallel grades the multiple engine's addresses are unknown at compile time. For each pthread, the runtime system tracks the thread's engine's address in a real CPU register[3], therefore there one less virtual register can be mapped to a real register.

Conway [31] also introduced a new structure called a *context*, known elsewhere as a green thread. Contexts represent computations in progress. Unlike green threads contexts cannot be preempted. An engine can either be idle, or executing a context; a context can either be running on an engine, or be suspended. Separate computations must have separate stacks, therefore each stack is associated with a context. When a computation is suspended, the registers from the engine are copied into the context, making the engine free to load a new context.

Stacks account for most of a context's memory usage. When a context finishes execution it can either be retained by the engine or have its storage released to the free context pool. This decision depends on what the engine will do next: if the engine will execute a different context or go to sleep (because there is no work to do), then the current context is released. Otherwise the engine holds into a context expecting it to be used to execute a *spark*.

Sparks represent goals that have been spawned off but whose execution has not yet been started. Wang [113] introduced sparks with the intention of enabling work stealing as described by Blumofe and Leiserson [13], Halstead [50] and Kranz, Halstead, and Mohr [68]. We did not implement work stealing until during my Ph.D. candidature and therefore it is not discussed here, in the background chapter. We discuss work stealing in Sections 3.4 and 3.6. Sparks are very light weight, being three words in size. Therefore, compared with contexts, they represent outstanding parallel work very efficiently; this is beneficial, even without work stealing. When an engine executes a spark, it will convert the spark into a context. it will use its current context if it has one (the engine's current context is always free when the engine attempts to run a spark), or if it does not have a context, it allocates a new one (from the pool of free contexts if the pool is not empty, otherwise the engine creates a new one). We will often compare our use of sparks to that of Marlow, Jones, and Singh [79] since GHC's runtime system has a similar structure. One of the more significant differences is that since Haskell is non-strict and Mercury is eager, sparks in Mercury cannot be garbage collected and must be executed. In comparison, Haskell's sparks represent speculative execution of values which may not be needed and can be safely discarded.

The only parallel construct in Mercury is parallel conjunction, denoted as $(G_1 \ \& \ \ldots \ \& \ G_n)$.

[3]If GCC global registers are unavailable, POSIX thread-local storage is used.

```
    /*
    ** A SyncTerm. One syncterm is created for each parallel conjunction.
    */
struct MR_SyncTerm_Struct {
    MR_Context              *MR_st_orig_context;
    MR_Word                 *MR_st_parent_sp;
    volatile MR_Unsigned    MR_st_count;
};

    /*
    ** A Mercury Spark.  A spark is created for a spawned off parallel
    ** conjunct.
    */
struct MR_Spark {
    MR_SyncTerm             *MR_spark_sync_term;
    MR_Code                 *MR_spark_resume;
    MR_ThreadLocalMuts      *MR_spark_thread_local_mutables;
}
```

Figure 2.5: Syncterms and sparks

All the conjuncts must be `det` or `cc_multi`, that is, they must all have exactly one solution, or commit to exactly one solution. This restriction greatly simplifies parallel execution in two ways: Firstly, it guarantees that there can never be any need to execute (G_2 & ... & G_n) multiple times, just because G_1 has succeeded multiple times. (Any local backtracking inside G_1 will not be visible to the other conjuncts; bindings made by `det` code are never retracted.) Supporting parallelisation of code that might have more than one solution requires new infrastructure for managing bindings. Any new infrastructure for managing bindings has a significant runtime cost, reducing the benefits of parallel execution. Secondly, code that might have no solutions also requires extra infrastructure. Additionally, if G_1 may fail then G_2 & ... & G_n's execution is speculative and may waste resources since its result may never be needed. Restricting parallelism to `det` and `cc_multi` code is not a significant limitation; since the design of Mercury strongly encourages deterministic code, in our experience, about 75 to 85% of all Mercury procedures are `det`. (This statistic was calculated by counting the different determinism declarations in the source code of the Mercury system.) Furthermore, we expect that most programs spend an even greater fraction of their time in `det` code (we know from profiling data that the Mercury compiler does). Existing algorithms for executing nondeterministic code in parallel have very significant overheads, generating slowdowns by integer factors. Thus we have given priority to parallelising deterministic code, which we can do with *much* lower overhead. We think that avoiding such slowdowns is a good idea, even if it does mean foregoing the parallelisation of 15 to 25% of a program.

A Mercury engine begins the execution of (G_1 & G_2 & ... & G_n) by creating a *syncterm* (synchronisation term), a data structure representing a barrier for the whole conjunction. The syncterm contains: a pointer to the context that began executing the parallel conjunction (the parent context), a pointer to the stack frame of the procedure containing this parallel conjunction, and a thread safe counter that tracks the number of conjuncts that have not yet executed their barrier. After initialising this syncterm, the engine then spawns off (G_2 & ... & G_n) as a spark and continues by executing G_1 itself. The spark contains a pointer to the syncterm, as well as

Source code	Transformed pseudo-code
	`MR_SyncTerm ST;`
	`MR_init_syncterm(&ST, 3);`
(	`spawn_off(&ST, Spawn_Label_1);`
G_1	G_1
&	`MR_join_and_continue(&ST, Cont_Label);`
	`Spawn_Label_1:`
	`spawn_off(&ST, Spawn_Label_2);`
G_2	G_2
&	`MR_join_and_continue(&ST, Cont_Label);`
	`Spawn_Label_2:`
G_3	G_3
)	`MR_join_and_continue(&ST, Cont_Label);`
	`Cont_Label:`

Figure 2.6: The implementation of a parallel conjunction

a pointer to the code it must execute, and another pointer to a set of thread local mutables.[4] Figure 2.5 shows these two data structures, the `MR_` prefixes on symbol names are used for Mercury runtime symbols, preventing their names from clashing with any symbols in C code linked with a Mercury program. Chapter 3 describes how sparks are managed.

When the engine finishes the execution of the first conjunct (G_1) it executes the barrier code `MR_join_and_continue` at the end of the conjunct. The barrier prevents the original context from executing the code that follows the parallel conjunction until each of the conjuncts has been executed. The barrier also makes several scheduling decisions, see Chapter 3.

Since (G_2 & ... & G_n) is itself a conjunction, it is handled in a similar way: the context executing it first spawns off (G_3 & ... & G_n) as a spark that points to the sync term created earlier, and then executes G_2 itself. Eventually, the spawned-off remainder of the conjunction consists only of the final conjunct, G_n, and the context just executes it. Once each conjunct synchronises using `MR_join_and_continue`, the original context will continue execution after the parallel conjunction. The introduction of the barrier at the end of the conjunction can prevent the compiler from using tail recursion optimisation. This occurs when G_n ended in a recursive call, and the whole conjunction was the last conjunction in a procedure's body. We discuss this problem in more detail and provide our solution in Chapter 5. Figure 2.6 shows an example of the code generated to execute a parallel conjunction. In this example the first conjunct creates a spark that represents the execution of the second and third conjuncts, the second conjunct then creates a spark representing just the third conjunct, which does not create any sparks.

2.3 Dependent AND-parallelism in Mercury

Mercury's mode system allows a conjunct in a sequential conjunction to consume variables that are produced by conjuncts to its left, but not to its right. However, The first parallel execution support for Mercury initially only supported independent AND-parallelism [31], Communication was not allowed between conjunctions. Wang [113], Wang and Somogyi [114] relaxed this; now parallel conjunctions have the same mode constraints as sequential conjunctions. Parallel conjunctions,

[4]Mercury has a module-local mutable feature that supports per thread mutables. This is not relevant to the dissertation.

```
map_foldl(M, F, L, Acc0, Acc) :-
    (
        L = [],
        Acc = Acc0
    ;
        L = [H | T],
        (
            M(H, MappedH),
            F(MappedH, Acc0, Acc1)
        &
            map_foldl(M, F, T, Acc1, Acc)
        )
    ).
```

Figure 2.7: Parallel `map_foldl`

```
enum MR_produced {
    MR_NOT_YET_PRODUCED,
    MR_PRODUCED
};

struct MR_Future {
    /* lock preventing concurrent accesses */
    MercuryLock             MR_fut_lock;
    /* whether this future has been signalled yet */
    enum MR_produced        MR_fut_produced;

    /* linked list of all the contexts blocked on this future */
    MR_Context              *MR_fut_suspended;
    MR_Word                 MR_fut_value;
};
```

Figure 2.8: Future data structure

such as the one in Figure 2.7, may now communicate through *shared variables* such as `Acc1`. A shared variable is a variable that is bound within the parallel conjunction and occurs in more than one conjunct. Recall that in Mercury, the code location where a variable becomes bound is known at compile time; therefore, shared variables can be identified at compile time. The compiler replaces each shared variable with a *future* [49], which is used to safely communicate the variable's value among conjuncts.

Figure 2.8 shows the future data structure, which contains room for the value of the variable, a flag indicating whether the variable has been produced yet, a queue of consumer contexts waiting for the value, and a mutex. The initial value of the future has the flag set to `MR_NOT_YET_PRODUCED`.

Consumers call `future_wait/2` when they want to retrieve a value from the future. This acquires the lock, and if `MR_fut_produced` is `MR_PRODUCED`, retrieves `MR_fut_value` before releasing the lock. If `MR_fut_produced` was `MR_NOT_YET_PRODUCED`, then `future_wait/2` will add the current context to the list of suspended contexts before unlocking the future. The engine will then suspend the context and look for other work. Once the value of the future is provided, the context will be woken up along with the others on the list of suspended contexts. `future_wait/2` contains an optimisation: it will check `MR_fut_produced` before acquiring the lock as well as after. If it is `MR_PRODUCED` before the lock is acquired, then `future_wait/2` can safely retrieve the future's value without using the lock. Note also that because Mercury's mode system ensures that variable dependencies can never form cycles, the compiler's use of futures cannot create a deadlock.

A producer will call `future_signal/2` to place a value into a future and wake up any suspended contexts. `future_signal/2` will also acquire the lock to ensure that it does not race with any `future_wait/2`. `future_signal/2` writes the value of the future to memory, before it sets `MR_fut_produced` to `MR_PRODUCED`. These operations are separated by a memory barrier to ensure that they are visible to other threads in this order. After releasing the lock, `future_signal/2` will schedule the suspended contexts. Because `future_signal/2` has no outputs and is deterministic, it must be declared as impure so that the compiler will not optimise away calls to it.

Some readers may wonder whether futures are similar to POSIX condition variables [24]. While both name their operations *wait* and *signal*, they are different in two significant ways. First, futures store a value as well as a state, POSIX condition variables store only their state. Second, when a future is signalled, all its consumers are woken, whereas only one of a POSIX condition variable's waiters is woken. A POSIX condition variable is more similar to a semaphore.

To minimise waiting, the compiler pushes `future_signal/2` operations on each future as far to the left into the producer conjunct as possible, and it pushes `future_wait/2` operations as far to the right into each of the consumer conjuncts as possible. This means not only pushing them into the bodies of predicates called by the conjunct, but also into the bodies of the predicates they call, with the intention that on each execution path each `future_signal/2` is put immediately after the primitive goal that produces the value of the variable, and each `future_wait/2` is put immediately before the leftmost primitive goal that consumes the value of the variable. Since the compiler has complete information about which goals produce and consume which variables, the only things that can stop the pushing process are module boundaries and higher order calls. This is because the compiler cannot push a `future_wait/2` or `future_signal/2` operation into the body of a predicate that it does not have access to or into a predicate that it cannot identify.

Any variable is produced exactly once along any execution path, and therefore there is exactly one `future_signal/2` operation along any path. However, a variable may be used any number of times along an execution path, therefore there can be any number of `future_wait/2` operations

along a path. The maximum number of `future_wait/2` operations along a path is equal to the number of times the future's value is used. Reading a future using `future_wait/2` is more expensive than reading a conventional variable. Therefore, we want to minimise the number of `future_wait/2` operations that can be executed along a path. The minimum number of `future_wait/2` operations along a path is either zero or one, depending on whether a variable is used on that path. This can be done either by adding an extra variable to track the value of the future, setting this variable's value from the first `future_wait/2` operation; or by introducing a new operation, `future_get/2` whose runtime cost is lower and that can safely be used for the second and subsequent `future_wait/2` operations. `future_get/2` is useful in cases where it is difficult or impossible to track the value of an extra variable.

An implementor can choose among always using a `future_wait/2` operation or trying to optimise the second and subsequent calls to `future_wait/2` or some point on a scale between these two extremes. There is a trade-off to be made here. The first option has very little analysis at compile time but has worse runtime performance, the second option can require a lot of compile time analysis and has ideal runtime performance. We have chosen to use a simple method to reduce the number of `future_wait/2` operations: When variables are replaced by futures and `future_wait/2` operations are inserted, the transformation tracks whether there is a `future_wait/2` operation on every success path leading to the current goal. If this is true and the current goal wants to access the value of the future then a `future_get/2` operation is used instead of a wait operation.

While `future_signal/2` actually makes the value of a future available, we have to consider that `future_wait/2` has the same effect since after a call to `future_wait/2` the value of a future is available and it may not have been available earlier. In other words, `future_wait/2` has the side effect of guaranteeing that the value of the future is available. `future_get/2` can only be called when the value of the future is true, therefore it is affected by `future_wait/2`'s side effect. Therefore, `future_get/2` must be semipure and `future_wait/2` must be impure. This prevents the compiler from optimising away or moving a call to `future_wait/2` that is necessary to make a call to `future_get/2` safe.

Given the `map_foldl` predicate in Figure 2.7, this synchronisation transformation generates the code in Figure 2.9.

2.4 Feedback directed automatic parallelism in Mercury

Some of the work described in this section contributed towards my honours research project, which contributed towards the Degree of Bachelor of Computer Science Honours. I have improved these contributions during my Ph.D. candidature. The following were part of my honours project: the coverage profiling transformation (Section 2.4.3), the related variable use analysis (Section 2.4.4) and the feedback framework (Section 2.4.5). Other parts of this section include: The Mercury deep profiler [32] (Section 2.4.1) and Tannier [105]'s work on automatic parallelism (Section 2.4.2).

Most compiler optimisations work on the representation of the program in the compiler's memory alone. For most optimisations this is enough. However, automatic parallelisation is sensitive to variations in the runtime cost of parallelised tasks. This sensitivity increases when dependent parallelisation is used. For example, a search operation on a small list is cheap, compared with the same operation on a large list. It may not be useful to parallelise the search on the small list against some other computation, but it will usually be useful to parallelise the search on the large list against another computation. It is important not to create too much parallelism: The

```
map_foldl(M, F, L, Acc0, Acc) :-
    (
        L = [],
        Acc = Acc0
    ;
        L = [H | T],
        future_new(FutureAcc1),
        (
            M(H, MappedH),
            F(MappedH, Acc0, Acc1),
            future_signal(FutureAcc1, Acc1)
        &
            map_foldl_par(M, F, T, FutureAcc1, Acc)
        )
    ).

map_foldl_par(M, F, L, FutureAcc0, Acc) :-
    (
        L = [],
        future_wait(FutureAcc0, Acc0),
        Acc = Acc0
    ;
        L = [H | T],
        future_new(FutureAcc1),
        (
            M(H, MappedH),
            future_wait(FutureAcc0, Acc0),
            F(MappedH, Acc0, Acc1),
            future_signal(FutureAcc1, Acc1)
        &
            map_foldl_par(M, F, T, FutureAcc1, Acc)
        )
    ).
```

Figure 2.9: `map_foldl` with synchronisation

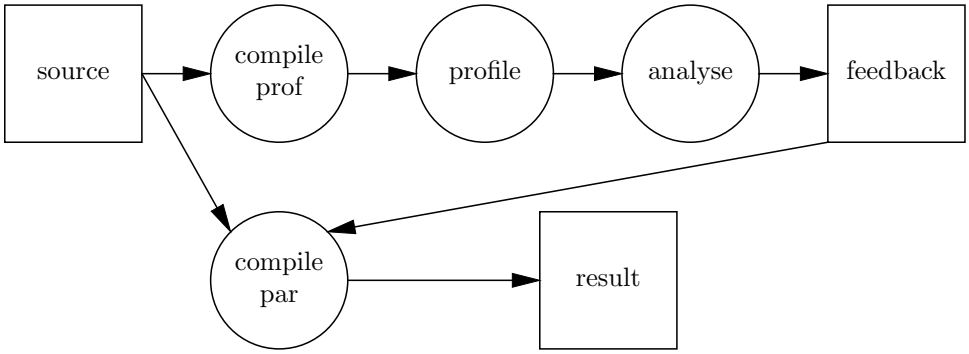


Figure 2.10: Profiler feedback loop

hardware is limited in how many parallel tasks it can execute, any more and the overheads of parallel execution will slow the program down. Therefore, it is not just sub-optimal to parallelise the search of the small list, but detrimental. Using a description of a program's typical inputs one could calculate the execution times of the program's procedures. However it is more direct, more robust and much easier to simply use a profiler to measure the typical execution times of procedures in the program while the program is executing with typical inputs, especially when we have such a powerful profiler already available (Section 2.4.1). Therefore, profiling data should be used in auto-parallelisation; it allows us to predict runtime costs for computations whose runtime is dependent on their inputs.

To use profiling data for optimisations, the usual workflow for compiling software must change. The programmer compiles the source code with profiling enabled and runs the resulting program on representative input. As the program terminates, it writes its profiling data to disk. The profiling data includes a bytecode representation of the program, similar to the compiler's representation. The analysis tool reads this file, and writes out a description of how to parallelise the program as a feedback file. The programmer then re-compiles their program, this time with auto-parallelism enabled. During compilation, the compiler reads the feedback file and introduces parallelism into the resulting program at the places indicated by the feedback file. Figure 2.10 shows this workflow.

A common criticism of profile directed optimisation is that the programmer will have to compile the program twice, and run it at least once to generate profiling data. "If I have to run the program to optimise it, then the program has already done its job, therefore there is no point continuing with the optimisation?" The answer to this is that a program's lifetime is far more than a single execution. A program will usually be used many times, and by many people. Each time the optimisation will have a benefit, this benefit will pay off the cost of the feedback directed optimisation. We expect that programmers will use feedback-directed optimisations when the they are building a release candidate of their program, after they tested the software and fixed any bugs.

Another criticism is that if the program is profiled with one set of input and used with another, then the profile will not necessarily represent the actual use of the program, and that the optimisation may not be as good as it should be. In practice parallelism is used in very few places within a program: in the Mercury Compiler itself there are only 34 (Page 120) conjuncts in 28,448[5] procedures that may be worth parallelising. This means that most differences in performance are not likely to affect the very small part of the program where parallelism may be applicable. Furthermore, as parallelisation decisions are binary, either 'do not parallelise' to 'parallelise'. a numerical

[5]The number of `ProcStatic` structures in the same profiling data that was used to provide the figures on Page 120.

difference in performance will not usually alter the binary parallelisation decision. So in most cases, auto-parallelisation based on the original input data is equivalent to auto-parallelisation based on slightly different input data.

In cases where different input data causes the program to be parallelised differently, the result is often going to be near-optimal. In these cases the main benefits of automatic parallelisation are unaffected, which are: automatic parallelisation will always be cheaper than manual parallelisation, as it requires much less time and does not require a company to hire a parallelisation expert. Therefore, automatic parallelisation will be preferred in all but the most extreme situations.

2.4.1 The deep profiler

Typical Mercury programs make heavy use of code reuse, especially through parametric polymorphism and higher order calls. This is also true for other declarative languages. For example, while a C program may have separate tables for different kinds of entities, for whose access functions the profiler would gather separate performance data, most Mercury programs would use the same polymorphic code to handle all those tables, making the task of disentangling the characteristics of the different tables infeasibly hard.

[32] solved this problem for Mercury by introducing deep profiling. Mercury's deep profiler gathers much more information about the context of each measurement than traditional profilers such as `gprof` [43] do. When it records the event of a call, a memory allocation or a profiling clock interrupt, it records with it the chain of ancestor calls, all the way from the current call to the entry point of the program (`main/2`). To make this tractable, recursive and mutually recursive calls, known as *strongly connected components* (SCCs), must be folded into a single memory structure. Therefore, the call graph of the program is a tree (it has no cycles) and SCCs are represented as single nodes in this tree.

Deep profiling allows the profiler to find and present to the user not just information such as the total number of calls to a procedure and the average cost of a call, or even information such as the total number of calls to a procedure from a particular call site and the average cost of a call from that call site. It will provide information such as the total number of calls to a procedure *from a particular call site when invoked from a particular chain of ancestor SCCs* and the average cost of a call *in that context*. We call such a context for profiling data an *ancestor context*. For example, it could tell that procedure h called procedure i ten times when h's chain of ancestors was $main \rightarrow f \rightarrow h$, while h called i only seven times when h's chain of ancestors was $main \rightarrow g \rightarrow h$, the calls from h to i took on average twice as long from the $main \rightarrow g \rightarrow h$ context as from $main \rightarrow f \rightarrow h$, so that despite making fewer calls to i, $main \rightarrow g \rightarrow h \rightarrow i$ took more time than $main \rightarrow f \rightarrow h \rightarrow i$. This is shown in Figure 2.11.

It can be difficult to locate a single goal within the body of a procedure. This is made more difficult as nested unifications are flattened into super homogeneous form (Section 2.1 on page 18), meaning that what might be a single line of code in a source file can be a conjunction of dozens of goals. Therefore the profiler uses *goal paths* to uniquely identify a sub goal within a procedure's body. A goal path is a list of goal path steps, each step describes which sub-goal to recurse into when traversing a goal structure from its root (the procedure as a whole) to either its leaves (atomic goals) or some compound goal. The goal path "c3;d2" refers to the second disjunct "d2" within the third conjunct "c3". Goal paths where first introduced to support debugging in Mercury [100]. The deep profiler was written after the debugger and was able to re-use them.

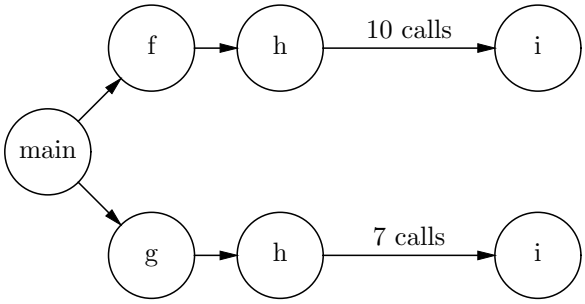


Figure 2.11: Example call graph

We use four structures during profiling.

`ProcDynamic` represents a procedure within the program's call tree. There can be multiple `ProcDynamic` structures for any given procedure: the same procedure will usually be called from a number of different contexts in the program.

`CallSiteDynamic` represents a call site in the program's call tree. As with `ProcDynamic`, there may be multiple `CallSiteDynamic` structures for any given call site. Profiling data for the call site & context is stored within its `CallSiteDynamic` structure. A `ProcDynamic` structure has an array of pointers to `CallSiteDynamic` structures representing the calls used during the execution of the procedure. A `CallSiteDynamic` structure has a pointer to the `ProcDynamic` structure of its callee. These links represent the call graph of the program.

Higher order and method calls are tracked by the `ProcDynamic` structure of the caller, the array of pointers is actually an array of arrays of pointers, multiple items in the second array represent different higher order values at the same call site. This means that a number of `CallSiteDynamic` structures can represent a single higher-order call site. This is actually rare: A call site and ancestor context are usually used with a single higher order value. Other higher order values are associated with other ancestor contexts and are therefore represented by other parts of the call graph.

`ProcStatic` represents the static data about a procedure This data includes the procedure's name, arity, source file and line number. Because multiple `ProcDynamic` structures may exist for the same procedure, this structure is used to factor out common information. Each `ProcDynamic` structure has an pointer to its `ProcStatic` structure. Each procedure in the program has a single `ProcStatic` structure. Each `ProcStatic` structure has an array of pointers to the `CallSiteStatic` structures of the calls its procedure makes.

`CallSiteStatic` represents static data for a call site. This includes the procedure the call site is found in, the line number, the goal path and a pointer to the `ProcStatic` structure of the callee (if known statically). As with `ProcStatic` structures, `CallSiteStatic` structures reduce the memory usage of `CallSiteDynamic` structures by factoring out common information. Each call site in the program has a single `CallSiteStatic` structure.

The profiling data is stored on disk using the same four structures that where used while capturing the data. When the profiling tool reads in this data, it will build the call graph and generate the list of SCCs (which Conway and Somogyi [32] call *cliques*) each SCC is used to create a `Clique` data structure. It also constructs several indexes, these indexes together with the `Clique`s make

traversing the program's profile (the other structures) more efficient. This data is used both by the interactive user interface, and automated tools.

Profilers have traditionally measured time by sampling the program counter at clock interrupts. Unfortunately, even on modern machines, the usual portable infrastructure for clock interrupts (*e.g.*, SIGPROF on Unix) supports only one frequency for such interrupts, which is usually 60 or 100Hz. This frequency is far too low for the kind of detailed measurements the Mercury deep profiler wants to make. For typical program runs of few seconds, it results in almost all calls having a recorded time of zero, with the calls recording a nonzero time (signifying a profiling interrupt during their execution) being selected almost at random.

We have therefore implemented a finer-grained measure of time that turned out to be very useful even though it is inherently approximate. This measure is *call sequence counts* (CSCs): the profiled program basically behaves as if the occurrence of a call signified the occurrence of a new kind of profiling interrupt. In imperative programs, this would be a horrible measure, since calls to different functions often have hugely different runtimes. However, in declarative languages like Mercury there are no explicit loops; what a programmer would do with a loop in an imperative language must be done by a recursive call. This means that the only thing that the program can execute between two calls is a sequence of primitive operations such as unifications and arithmetic. For any given program, there is a strict upper bound on the maximum length of such sequences, and the distribution of the length of such sequences is very strongly biased towards very short sequences of half-a-dozen to a dozen operations. In practice, we have found that the fluctuations in the lengths of different sequences can be ignored for any measurement that covers any significant number of call sequence counts, say more than a hundred. The only drawback that we have found is that on 32 bit platforms, its usability is limited to short program runs (a few seconds) by the wraparound of the global CSC counter; This has not been a concern for a number of years. On 64 bit platforms, the problem would only occur on a profiling run that lasted for years.

2.4.2 Prior auto-parallelism work

Tannier [105] describes a prior attempt at automatic parallelism in Mercury. In his approach, an analysis tool gathered parallelisation feedback data in the form of a list of the procedures with the highest costs in the program. Ancestor context independent data was used, meaning that all uses of a procedure were considered when ranking that procedure within the list. The user was able to choose between using the mean or median execution time of a procedure, measured in CSCs. Only procedures whose cost was greater than a configurable threshold were included in the feedback data. The analysis did not use a representation of the program, as at that time the profiler did not support that capability.

The compiler used the feedback data to make parallelisation decisions. Each procedure being compiled by the compiler was searched for calls that appeared in the list of top procedures and calls to these were parallelised against similar calls if they where independent or had fewer shared variables than another configurable limit.

For my honours project I attempted automatic parallelisation of Mercury [17]. My work differed from Tannier's in a number of ways: The first differences is that at the time of my work, the deep profiler was able to access a representation of the program (my work was the motivation for this feature). This allowed my analysis to use both profiling data and the representation of the program at the same time, making it possible to measure to the times at which variables are

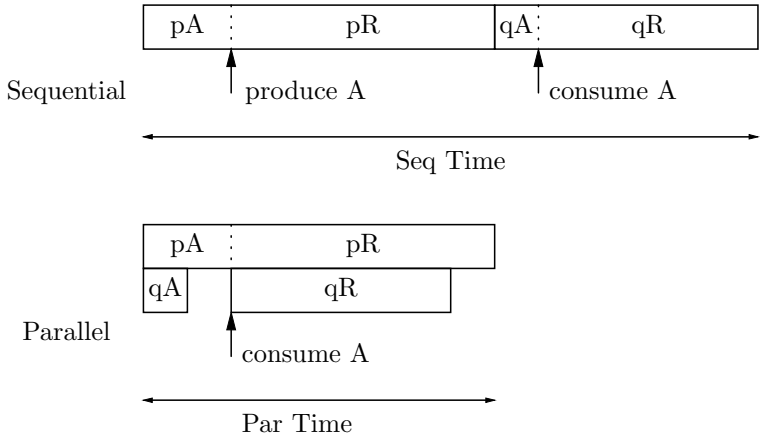


Figure 2.12: Overlap of `p` and `q`

produced and consumed within parallel conjuncts. The analysis uses this to calculate how much parallelism is available, for conjunctions with a single shared variable. Consider the timeline shown in Figure 2.12, In the sequential case the conjuncts `p` and `q` are shown side-by-side. Below this, `q` is parallelised against `p`. The future `A` is communicated from `p` to `q`, `q` will execute the `future_wait/2` operation for `A` before `p` executed `future_signal/2`, therefore `future_wait/2` will suspend `q`. Later, `p` executes `future_signal/2`, producing `A` and scheduling `q` for execution. The formula used to compute the execution times in the sequential and parallel cases, and the speedup due to parallelism is:

$$
\begin{aligned}
T_{Sequential} &= T_p + T_q \\
T_{DependentQ} &= max(T_{BeforeProduceP}, T_{BeforeConsumeQ}) + T_{AfterConsumeQ} \\
T_{Parallel} &= max(T_p, T_{DependentQ}) + T_{Overheads} \\
Speedup &= \frac{T_{Sequential}}{T_{Parallel}}
\end{aligned}
$$

This describes the speedup owing to parallelisation of two conjuncts, `p` and `q`. whose execution times are T_p and T_q. In the parallel case `q`'s execution time is $T_{DependentQ}$ since it accounts for the `q`'s dependency on `A`. Except for this small difference, the calculation of parallel execution time and speedup are the commonly used formulas for many parallel execution cost models. We will greatly expand and generalise this calculation in Chapter 4.

This cost model requires information about when a shared variable is produced by the first conjunct and when it is consumed by the second. To provide this information, we introduced a variable use time analysis (Section 2.4.4), that depends upon information about how often different goals are executed, called *coverage*. The deep profiler provides coverage information for most goals, however we need coverage information for all goals and therefore we introduced coverage profiling.

2.4.3 Coverage profiling

Code *coverage* is a term often used in the context of debugging and testing: it refers to which parts of a program get executed and which do not. It is used to determine the completeness of a test-suite. We use the term code coverage, which we often abbreviate to "coverage", to describe

how *often* a given piece of code is executed. Thus, we have extended the concept of coverage from a binary "has been executed" or "has not been executed" to a more informative "has been executed N times".

The traditional Prolog box model [25] describes each predicate as having four ports through which control flows. Control flows into a predicate through either the *call* or *redo* ports, control flows out of a predicate through either the *exit* or *fail* ports. These describe: a call being made to the predicate, re-entry to the predicate after it has exited to check for more solutions, the predicate exiting with a solution, or the predicate failing because there are no more solutions. Mercury adds an extra port, known as the *exception* port: when a predicate throws an exception it returns control via the exception port. Provided that execution has finished, control must have left a predicate the same number of times that it entered it.

$$Calls + Redos = Exits + Fails + Exceptions$$

Mercury adds additional invariants for many determinism types. For example, deterministic code may not redo or fail, therefore these port counts will be zero. For each call and its ancestor context the deep profiler will track the number of times each port is used, Port counts provide code coverage information for predicate and call site entry and exit.

Code coverage of many goals can be inferred. In the simple case of a conjunction containing deterministic conjuncts, each conjunct will have the same coverage as the one before it provided that the earlier goal does not throw an exception. There are two types of exception, those that are caught and those that terminate the program. When an exception terminates the program, we are not interested in that execution's profile, when an exception is caught and the program recovers we are interested in coverage, however this is rare. A deterministic conjunction containing a single call can have complete coverage information inferred for all of its conjuncts, provided that we are willing to loose a little accuracy if an exception is thrown. Similarly, the coverage before an *if-then-else* can be inferred as the sum of the coverage at the beginning of the *then* and *else* branches, assuming that the condition has at most one solution. The same is true for the ends of these branches and the end of the if-then-else. The coverage at the beginning of the then part is also equal to the number of times the *condition* of the if-then-else succeeds. The coverage of a switch is the sum of the branch's coverage plus the number of times the switch may fail. Many switches are known to be complete, and therefore they cannot fail. This can allow us to infer the missing coverage of a single branch provided that we know the coverage of all the other branches plus the coverage of the switch. Or to infer the coverage of the switch provided that we know the coverage of all the branches.

Based on these observations, we have implemented a coverage inference algorithm in the deep profiler. The coverage inference algorithm makes a single forward pass tracking the coverage before the current goal if known, when it reaches a call, it can read the port counts of the call site to set the coverage before and after the current goal. When it encounters a deterministic goal, it can set that goal's coverage to the coverage before the current goal. The inference algorithm will also infer the coverage of the then and else parts of an if-then-else, as described above; as well as the coverage of the last branch of a complete switch, as described above.

Unfortunately, port counts alone will not provide complete coverage information. For example, a conjunction containing a sequence of `semidet` unifications will have incomplete coverage information: we cannot know how often each individual unification fails. We introduced coverage

```
map(P, Xs0, Ys) :-
    (
        coverage_point(ProcStatic, 0);
        Xs0 = [],
        Ys = []
    ;
        Xs0 = [X | Xs],
        P(X, Y),
        map(P, Xs, Ys0),
        Ys = [Y | Ys0]
    ).
```

Figure 2.13: Coverage annotated `map/3`.

profiling as part of Mercury's deep profiler to add extra instrumentation to gather coverage data where it would otherwise be unknown. The coverage inference algorithm will look for a coverage point whenever it cannot infer the coverage from earlier goals.

We want to reduce the number of coverage points added to a procedure while still allowing inference of complete coverage information. This is because instrumentation, such as a coverage point, can slow the program down and sometimes distort the profile, [6] therefore preventing superfluous instrumentation from being added is desirable. We do this in the coverage profiling transformation. The coverage profiling transformation makes the exact same traversal through the program as the coverage inference algorithm. Instead of tracking the coverage before the current goal the coverage profiling transformation tracks a boolean that indicates whether or not coverage would be known before the current goal based on the goals before it. When this boolean becomes false, the transformation will introduce a coverage point at that place in the code and set the value to true. This has the effect of placing a coverage point at exactly the location at which the coverage inference algorithm would attempt to look up coverage from a coverage point. The coverage profiling transformation will set the value of its boolean to false after a goal with a determinism other than `det` or `cc_multi` and which does not provide coverage data of its own. It also sets the boolean to false at the beginning of a branch whose coverage cannot be inferred.

Figure 2.13 shows `map/3` annotated with a coverage point. The forward pass of the coverage inference algorithm cannot infer coverage at the beginning of this switch arm, and therefore the coverage profiling transformation in the deep profiler introduced a coverage point at that position. It is true that the coverage point could be removed and the port counts of the recursive code in this example could be used, but doing so would require multiple passes for both algorithms, making them slower. Coverage profiling adds a small cost to profiling (6% slower than normal deep profiling); it is not a high priority to optimise this further.

During my honours project I added three new fields to the `ProcStatic` structure, which we introduced in Section 2.4.1. These fields are: a constant integer representing the number of coverage points in this procedure; an array of constant structures, each describing the static details of a coverage point; and an array of integers representing the number of times each coverage point has been executed. The coverage point instrumentation code refers to the procedure's `ProcStatic` structure and increments the coverage point's counter at the appropriate index. The static coverage data, which includes the goal path of the coverage point and the type of the coverage point (branch

[6] Automatic parallelism analysis is not affected by profile distortion since it uses call sequence counts to measure time. Nevertheless, avoiding profile distortion is good as programmers may continue to use the profiler's other time measurements.

or post-goal), is written out with the profile. We were concerned that a procedure might require dozens of coverage points, each one consuming memory, therefore to avoid the risk of consuming large amounts of memory we choose to associate coverage points with `ProcStatic` structures rather than the more numerous `ProcDynamic` structures. Therefore coverage data is not associated with ancestor contexts in the way that other profiling data is. The auto-parallelisation work described in my honours report did not expect to use ancestor context specific coverage information.

The coverage point in Figure 2.13 has two arguments, the first points to the `ProcStatic` containing this procedure's coverage points, the second is the index within this procedure's coverage point arrays for this coverage point. Although the instrumentation code appears to be a call, it is implemented as a C preprocessor macro as it is very small and should be inlined into the compiled procedure.

2.4.4 Variable use time analysis

The automatic parallelism analysis used in my honours report considers conjunctions with at most two conjuncts and at most one shared variable. To predict how much parallelism is available, it needs to know when that variable is produced by the first conjunct, and when it is first consumed by the second. These times are calculated in units of call sequence counts.

During compilation of dependent conjunctions, the compiler creates a future to replace any shared variable. The compiler will attempt to push that future into the goal that produces its value, so that, the `future_signal/2` operation is after, and as close as possible to, the unification that binds the variable.

The algorithm for computing the expected production time of a shared variable looks at the form of the conjunct that produces it, this is shown in Algorithm 2.1. This algorithm is only invoked on goals that are either `det` or `cc_multi`, and that produce the variable V. For each goal type, it calculates at what point within the goal's execution the future for V will signalled. As noted above, a future cannot be pushed into certain goal types, such as higher order calls, method calls, foreign code or calls that cross module boundaries. Therefore, a conservative answer is used in these cases, the conservative answer is 'at the end of G's execution'. This is conservative not just because it is typical, but because it will avoid creating parallelisations that are not likely to speed up the program. The case for foreign calls does not measure a call sequence count for the foreign call itself, so we add one. Goals such as unifications are trivial, and always have zero cost, therefore they must produce their variable after zero call sequence counts of execution. Parallel conjunctions are converted into sequential conjunctions before profiling, they will not appear in the algorithm's input. A negation cannot produce a variable and therefore it cannot produce V, the algorithm does not need to handle negations. The algorithm is only invoked on goals that are `det` or `cc_multi`, and the only disjunction that can be `det` or `cc_multi` is one that does not produce outputs, and therefore cannot produce V, the algorithm does not need to handle disjunctions either.

The algorithm handles conjunctions by determining which conjunct G_k in the conjunction $G_1, \ldots, G_k, \ldots, G_n$ produces the variable. The production time of the variable in the conjunction is its production time in G_k plus the sum of the durations of $G_1, \ldots, G_{k-1}$.

We handle switches by invoking the algorithm recursively on each switch arm, and computing a weighted average of the results, with the weights being the arms' entry counts as determined by coverage inference. We handle if-then-elses similarly. we need the weighted average of the two possible cases: the variable being generated by the then arm versus the else branch. (It cannot

Algorithm 2.1 Variable production time analysis

$$\text{prodtime } V\,G = \begin{cases} \text{time } G & \text{if } G \text{ is not executed or } \texttt{erroneous} \\ \text{prodtime' } V\,G & \text{if } G \text{ is } \texttt{det} \text{ or } \texttt{cc\_multi} \\ \text{cannot happen} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&\text{prodtime' } V[\![X = Y]\!] && = 0 \\
&\text{prodtime' } V[\![X = f(\ldots)]\!] && = 0 \\
&\text{prodtime' } V[\![p(X_1, \ldots, X_n)]\!] && = \text{call_prodtime } V\,p\,[X_1, \ldots, X_n] \\
&\text{prodtime' } V[\![X_{n+1} = f(X_1, \ldots, X_n)]\!] && = \text{call_prodtime } V\,f\,[X_1, \ldots, X_n, X_{n+1}] \\
&\text{prodtime' } V[\![X_0(X_1, \ldots, X_n)]\!] && = \text{time_of_call} \\
&\text{prodtime' } V[\![m(X_1, \ldots, X_n)]\!] && = \text{time_of_call} \\
&\text{prodtime' } V[\![foreign(\ldots)]\!] && = \text{time_of_call} + 1 \\
&\text{prodtime' } V[\![G_1, G_2, \ldots, G_n]\!] && = \begin{cases} \text{prodtime } V\,G_1 & \text{if } G_1 \text{ binds } V \\ \left(\begin{array}{l} \text{time } G_1 + \\ \text{prodtime } V\,[\![G_2, \ldots, G_n]\!] \end{array} \right) & \text{otherwise} \end{cases} \\
&\text{prodtime' } V[\![G_1 \,\&\, \ldots \,\&\, G_n]\!] && = \text{cannot happen} \\
&\text{prodtime' } V[\![G_1 \,;\, \ldots \,;\, G_n]\!] && = \text{cannot happen} \\
&\text{prodtime' } V\left[\!\!\left[\begin{array}{l} \text{switch } X\ (f_1 : G_1\,; \\ \quad \ldots, f_n : G_n) \end{array} \right]\!\!\right] && = \sum_{1 \le i \le n} Pr_{G_i} \times \text{prodtime } V\,G_i \\
&\text{prodtime' } V\left[\!\!\left[\begin{array}{lll} \text{if } Cond & \text{then } Then \\ & \text{else } Else) \end{array} \right]\!\!\right] && = \text{time } Cond + \left(\begin{array}{l} Pr_{Then} \times \text{prodtime } V\,Then + \\ Pr_{Else} \times \text{prodtime } V\,Else \end{array} \right) \\
&\text{prodtime' } V[\![not\,G]\!] && = \text{cannot happen} \\
&\text{prodtime' } V[\![some\,[X_1, \ldots, X_n]\,G]\!] && = \text{prodtime } V\,G \\
&\text{prodtime' } V[\![promise_pure\,G]\!] && = \text{prodtime } V\,G \\
&\text{prodtime' } V[\![promise_semipure\,G]\!] && = \text{prodtime } V\,G
\end{aligned}$$

$$\text{call\_prodtime } V\,P\,Args = \begin{cases} \text{prodtime } Params\,B & \text{if } P \text{ is in the current module} \\ \text{time } B & \text{otherwise} \end{cases}$$

$$\text{where} \quad \begin{aligned} Params &= \text{arg\_to\_param } V\,Args \\ B &= \text{body } P \end{aligned}$$

time G= The duration of G's execution.

be generated by the condition: variables generated by the condition are visible only from the then branch.) To find either number, we invoke the algorithm on either the then or else branch, depending on which is applicable, and add this to the time taken by the condition. Note that this is the time taken by the condition on average (whether or not it fails), we do not have access to timing information dependent on a goal's success or failure. This is not a concern, most condition goals are inexpensive, therefore their runtimes do not vary significantly.

Using the weighted average for switches and if-then-elses is meaningful because the Mercury mode system dictates that if one branch of a switch or if-then-else generates a variable, then they *all* must do so. The sole exception is branches that are guaranteed to abort the program, whose determinism is erroneous. We use a weight of zero for erroneous branches because optimising them does not make sense. Coverage profiling was introduced to provide the weights of switch and if-then-else arms used by this algorithm.

If the goal is a quantification, then the inner goal must be `det`, in which case we invoke the algorithm recursively on it. If the inner goal were not `det`, then the outer quantification goal could be `det` only if the inner goal did not bind any variables visible from the outside. Therefore, if the analysis is invoked on a quantification, we already know that this is because the quantification produces the variable, and therefore the inner goal must be `det`.

The algorithm we use for computing the time at which a shared variable is first consumed by the second conjunct is similar, it is shown in Algorithm 2.2. The main differences are that negated goals, conditions and disjunctions are allowed to consume variables, and some arms of a switch or if-then-else may consume a variable even if other arms do not. The code that pushes wait operations into goals can be configured to include a `future_wait/2` operation on every execution path so that when the goal completes, it is guaranteed that a `future_wait/2` operation has taken place. The variable use analysis should reflect the compiler's configuration with respect to how `future_wait/2` operations are placed but it does not. It makes the conservative assumption that waits are always placed in switch arms of switches and if-then-elses that do not consume the variable. This assumption is a conservative approximation. This can prevent introducing parallelism that leads to a slow-down. To make this assumption, the algorithm checks if a goal consumes a variable, if it does not, it returns the goal's execution time as the consumption time of the variable. The other important difference is that when making any other conservative assumption such as when the algorithm finds a higher order call, the consumption time used is zero (it was the cost of the call when analysing for productions)

For example suppose the first appearance of the variable (call it X) in a conjunction $G_1, \ldots, G_n$ is in G_k, and G_k is a switch. If X is consumed by some switch arms and not others, then on some execution paths, the first consumption of the variable may be in G_k (a), on some others it may be in $G_{k+1}, \ldots, G_n$ (b), and on some others it may not be consumed at all (c). For case (a), we compute the average time of first consumption by the consuming arms, and then compute the weighted average of these times, with the weights being the probability of entry into each arm, as before. For case (b), we compute the probability of entry into arms which do *not* consume the variable, and multiply the sum of those probabilities by the weighted average of those arms' execution time *plus* the expected consumption time of the variable in $G_{k+1}, \ldots, G_n$. For case (c) we pretend X is consumed at the very end of the goal, and then handle it in the same way as (b). This is because for our overlap calculations, a goal that does not consume a variable is equivalent to a goal that consumes it at the end of its execution.

Algorithm 2.2 Variable consumption time analysis

$$\text{constime } V\,G = \begin{cases} 0 & \text{if } G \text{ is not executed or } \texttt{erroneous} \\ \text{constime' } V\,G & \text{if } G \text{ is } \texttt{det}, \texttt{semidet}, \texttt{failure}, \texttt{cc\_multi} \text{ or } \texttt{cc\_nondet} \\ \text{cannot happen} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&\text{constime' } V [\![X = Y]\!] && = 0 \\
&\text{constime' } V [\![X = f(\ldots)]\!] && = 0 \\
&\text{constime' } V [\![p(X_1, \ldots, X_n)]\!] && = \text{call_constime } V\,p\,[X_1, \ldots, X_n] \\
&\text{constime' } V [\![X_{n+1} = f(X_1, \ldots, X_n)]\!] && = \text{call_constime } V\,f\,[X_1, \ldots, X_n, X_{n+1}] \\
&\text{constime' } V [\![X_0(X_1, \ldots, X_n)]\!] && = 0 \\
&\text{constime' } V [\![m(X_1, \ldots, X_n)]\!] && = 0 \\
&\text{constime' } V [\![foreign(\ldots)]\!] && = 0 \\
&\text{constime' } V [\![G_1, G_2, \ldots, G_n]\!] && = \begin{cases} \text{constime } V\,G_1 & \text{if } G_1 \text{ consumes } V \\ \left(\begin{array}{l} \text{time } G_1 + \\ \text{constime } V\,[\![G_2, \ldots, G_n]\!] \end{array} \right) & \text{otherwise} \end{cases} \\
&\text{constime' } V [\![G_1\,\&\,\ldots\,\&\,G_n]\!] && = \text{cannot happen} \\
&\text{constime' } V [\![G_1\,;\,\ldots\,;\,G_n]\!] && = \begin{cases} \text{constime } V\,G_1 & \text{if } G_1 \text{ consumes } V \\ \left(\begin{array}{l} \text{time } G_1 + \\ \text{constime } V\,[\![G_2, \ldots, G_n]\!] \end{array} \right) & \text{otherwise} \end{cases} \\
&\text{constime' } V \left[\!\!\left[\begin{array}{c} \text{switch } X\ (f_1 : G_1\,; \\ \ldots, f_n : G_n) \end{array} \right]\!\!\right] && = \sum_{1 \le i \le n} Pr_{G_i} \times \text{constime } V\,G_i \\
&\text{constime' } V \left[\!\!\left[\begin{array}{ll} \text{if } Cond & \text{then } Then \\ & \text{else } Else) \end{array} \right]\!\!\right] && = \text{iteconstime } V\,Cond\,Then\,Else \\
&\text{constime' } V [\![not\,G]\!] && = \text{constime } V\,G \\
&\text{constime' } V [\![some\,[X_1, \ldots, X_n]\,G]\!] && = \text{constime } V\,G \\
&\text{constime' } V [\![promise_pure\,G]\!] && = \text{constime } V\,G \\
&\text{constime' } V [\![promise_semipure\,G]\!] && = \text{constime } V\,G
\end{aligned}$$

$$\text{iteconstime } V\,Cond\,Then\,Else = \begin{cases} \text{constime } V\,Cond & \text{if } Cond \text{ consumes } V \\ \text{time } Cond + Pr_{Then} \times \text{constime } V\,Then + Pr_{Else} \times \text{constime } V\,Else & \textit{otherwise} \end{cases}$$

$$\text{call\_constime } V\,P\,Args = \begin{cases} \text{constime } Params\,B & \text{if } P \text{ is in the current module} \\ 0 & \text{otherwise} \end{cases}$$

where $Params = \text{arg\_to\_param } V\,Args$, $B = \text{body } P$

time G= The duration of G's execution.

2.4.5 Feedback framework

Automatic parallelism is just one use of profiler feedback. Other optimisations such as inlining, branch hints and type specialisation might also benefit from profiling feedback.

During my honours project I designed and implemented a generic feedback framework that allows tools to create feedback information for the compiler. These tools may include the profiler, the compiler itself, or any tool that links to the feedback library code. Any Mercury value can be used as feedback information, making the feedback framework flexible. New feedback-directed optimisations may require feedback information from new analyses. We anticipate that many new feedback information types will be added to support these optimisations. We also expect that an optimisation may use more than one type of feedback information and that more than one optimisation may use the same feedback information.

The on-disc format for the feedback information is very simple: it contains a header that identifies the file format, including a version number, followed by a list of feedback information items, stored in the format that Mercury uses for reading and writing terms. When the feedback library reads the file in it will check the file format identifier and version number, then it will read the list of information items and check that no two items describe the same type of feedback information.

The API allows developers to open an existing file or create a new one, query and set information in the in-memory copy of the file, and write the file back out to disk. It is straight forward to open the file, update a specific feedback item in it, and close it, leaving the other feedback items in the file unchanged.

Chapter 3

Runtime System Improvements

Early in the project we tested two manually parallelised programs: a raytracer and a mandelbrot image generator. Both programs have a single significant loop whose iterations are independent of one another. We expect that a good automatic parallelisation system will parallelise this loop as it is the best place to introduce parallelism. When we parallelised this loop manually, we did not get the speedups that we expected. Therefore, we chose to address the performance problems before we worked on automatic parallelism. Throughout this chapter we continue to use these two benchmarks, along with a naive Fibonacci program (Page 85). These benchmarks are not diverse and they all create a lot of AND-parallelism, most of which is independent. We use these benchmarks deliberately to test that our runtime system can handle large amounts of parallelism efficiently.

In this chapter we investigate and correct these performance problems. We start with the garbage collector in Section 3.1; we analyse the collector's effects on performance and tune its parameters to improve performance. In Section 3.2 we describe how the existing runtime system schedules sparks, and provide background material for Section 3.3, which benchmarks the runtime system and describes two significant problems with spark scheduling. We address one of these problems by introducing work stealing in Section 3.4. Then in Section 3.5 we reorder conjuncts in independent parallel conjunctions to work around the second spark scheduling problem. Finally, in Section 3.6 we make further improvements to work stealing and change the data structures and algorithms used to manage idle engines, including how idle engines look for work, sleep and are woken up.

3.1 Garbage collector tweaks

One of the sources of poor parallel performance is the behaviour of the garbage collector. Like other pure declarative languages, Mercury does not allow destructive update. Therefore a call usually returns its results in newly allocated memory rather than modifying the memory of its parameters. Likewise, a call cannot modify data that may be aliased. This means that Mercury programs often have a high rate of allocation, which places significant stress on the garbage collector. Therefore, allocation and garbage collection can reduce a program's performance as we will show in this section.

Mercury uses the Boehm-Demers-Weiser conservative garbage collector (Boehm GC) [14], which is a conservative mark and sweep collector. Boehm GC supports parallel programming: it will

stop all the program's threads (*stop the world*) during its marking phase. It also supports parallel marking: it will use its own set of pthreads to do parallel marking.

For the purposes of this section we separate the program's execution time into two alternating phases: Collector time, which is when Boehm GC performs marking, and mutator time, which is when the Mercury program runs. The name 'mutator time' refers to time that mutations (changes) to memory structures are permitted. The collector may also perform some actions, such as sweeping, concurrently with the mutator.

Amdahl's law [4] describes the maximum speedup that can theoretically be achieved by parallelising a part of a program. We use Amdahl's law to predict the speedup of a program whose mutator is parallelised but whose collector runs sequentially. Consider a program with a runtime of 20 seconds which can be separated into one second of collector time and 19 seconds of mutator time. The sequential execution time of the program is $1 + 19 = 20$. If we parallelise the mutator and do not parallelise the collector then the minimum parallel execution time is $1 + 19/P$ for P processors. Using four processors the theoretical best speedup is: $(1 + 19)/(1 + 19/4) = 3.48$. The fraction of this new execution time $(1 + 19/4)$ spent in the collector (1) is 17% (it was 0.5% without parallelisation). If we use a machine with 100 processors then the speedup becomes: $(1 + 19)/(1 + 19/100) = 16.8$; with 84% of the runtime spent in the collector. As the number of processors increases, the mutator threads spend a larger proportion of their time waiting for collection to complete.

Gustafson-Barsis' law [48] is more appropriate than Amdahl's law when: there is always more data that could be processed by additional computation resources (such as in scientific computing), or when the data can be split into smaller and smaller pieces without affecting the granularity of parallel tasks. These conditions are often true for large numeric computations; however they are rarely true for symbolic computations, and Mercury is typically used for symbolic computations. Therefore we use Amdahl's law as it is more applicable.

To reduce the amount of time that mutator threads have to wait for the collector, Boehm and Weiser [14] included parallel marking support in their collector. Ideally this would remove the bottleneck described above. However, parallel garbage collection is a continuing area of research and the Boehm GC project has only modest multicore scalability goals. Therefore, we expect parallel marking to only partially reduce the bottleneck, rather than remove it completely.

Furthermore, thread safety has two significant performance costs; These types of costs prevent us from achieving the theoretical maximum speedups that Amdahl's law predicts. The first of these is that one less CPU register is available to GCC's code generator when compiling parallel Mercury programs (Section 2.2). The second is that memory allocation routines must use locking to protect shared data structures, which slows down allocation. Boehm GC's authors recognised this problem and added support for thread-local resources such as free lists. Therefore, during memory allocation, a thread uses its own free lists rather than locking a global structure. From time to time, a thread will have to retrieve new free lists from a global structure and will need to lock the structure, but the costs of locking will be amortised across several memory allocations.

To test how well Mercury and Boehm GC scale to multiple cores we used several benchmark programs with different memory allocation requirements. We wanted to determine how memory allocation rates affect performance of parallel programs. Our first benchmark is a raytracer developed for the ICFP programming contest in the year 2000. For each pixel in the image, the raytracer casts a ray into the scene to determine what colour to paint that pixel. Two nested loops build the pixels for the image: the outer loop iterates over the rows in the image and the inner

GC Markers	Sequential		Parallel w/ N Mercury Engines			
	no TS	TS	1	2	3	4
raytracer						
1	33.1 (1.13)	37.4 (1.00)	37.7 (0.99)	28.0 (1.34)	24.8 (1.51)	23.7 (1.58)
2	-	-	32.0 (1.17)	21.6 (1.73)	18.1 (2.07)	16.7 (2.24)
3	-	-	29.6 (1.26)	19.7 (1.90)	16.3 (2.29)	14.5 (2.59)
4	-	-	29.1 (1.29)	18.9 (1.98)	15.3 (2.45)	13.7 (2.73)
mandelbrot_highalloc						
1	41.0 (1.23)	50.5 (1.00)	50.6 (1.00)	33.4 (1.51)	26.6 (1.90)	23.4 (2.16)
2	-	-	46.7 (1.08)	28.6 (1.77)	22.1 (2.28)	18.8 (2.69)
3	-	-	46.3 (1.09)	27.3 (1.85)	20.9 (2.42)	17.3 (2.93)
4	-	-	45.6 (1.11)	26.8 (1.89)	20.4 (2.48)	16.7 (3.03)
mandelbrot_lowalloc						
1	15.4 (0.99)	15.2 (1.00)	15.2 (1.00)	5.1 (2.96)	7.7 (1.98)	3.9 (3.92)
2	-	-	15.3 (1.00)	7.7 (1.98)	5.1 (2.97)	3.9 (3.94)
3	-	-	14.3 (1.00)	7.7 (1.98)	5.1 (2.97)	3.9 (3.94)
4	-	-	15.3 (0.99)	7.7 (1.98)	5.1 (2.97)	3.9 (3.92)

Table 3.1: Parallelism and garbage collection

loop iterates over the pixels in each row. We manually parallelised the program by introducing a parallel conjunction into the outer loop, indicating that rows should be drawn in parallel. This parallelisation is independent. The raytracer uses many small structures to represent vectors and a pixel's colour. It is therefore memory allocation intensive. Since we suspected that garbage collection in memory allocation intensive programs was having a negative affect on performance, we developed a mandelbrot image generator. The mandelbrot image generator has a similar structure to the raytracer, but is designed so that we can test programs with different allocation rates. It draws an image using two nested loops as above. The mandelbrot image is drawn on the complex number plane. A complex number C is in the mandelbrot set if $\forall i \cdot |N_i| < 2$ where $N_0 = 0$ and $N_{i+1} = N_i^2 + C$. For each pixel in the image the program tests if the pixel's coordinates are in the set. The pixel's colour is chosen based on how many iterations i are needed before $|N_i| \geq 2$, or black if the test survived 5,000 iterations. We parallelised this program the same way as we did the raytracer: by introducing an independent parallel conjunction into the outer loop. We created two versions of this program. The first version represents coordinates and complex numbers as structures on the heap. Therefore it has a high rate of memory allocation. We call this version 'mandelbrot_highalloc'. The second version of the mandelbrot program, called 'mandelbrot_lowalloc', stores its coordinates and complex numbers on the stack and in registers. Therefore it has a lower rate of memory allocation.

We benchmarked these three programs with different numbers of Mercury engines and garbage collector threads. We show the results in Table 3.1. All benchmarks have an initial heap size of 16MB. Each result is measured in seconds and represents the mean of eight test runs. The first column is the number of garbage collector threads used. The second and third columns give sequential execution times without and with thread safety[1] enabled. In these columns the programs were compiled in such a way so that they did not execute a parallel conjunction. The remaining four columns give parallel execution times using one to four Mercury engines. The numbers in

[1] A thread safe build of a Mercury program enables thread safety in the garbage collector and runtime system. It also requires that one less CPU register is available to Mercury programs (see Section 2.2)

parentheses show the relative speedup when compared with the sequential thread safe result for the same program. We ran our benchmarks on a four-core Intel i7-2600K system with 16GB of memory, running Debian/GNU Linux 6 with a 2.6.32-5-amd64 Linux kernel and GCC 4.4.5-8. We kept frequency scaling (Speedstep and TurboBoost) disabled. The garbage collection benchmarks were gathered using a recent version of Mercury (rotd-2012-04-29). This version does not have the performance problems described in the rest of this chapter, and it does have the loop control transformation described in Chapter 5. Therefore we can observe the effects of garbage collection without interference from any other performance problems.

The raytracer program benefits from parallelism in both Mercury and the garbage collector. Using Mercury's parallelism only (four Mercury engines, and one GC thread) the program achieves a speedup of 1.58, compared to 1.29 when using the GC's parallelism only (one Mercury engine, and four GC threads). When using both Mercury and the GC's parallelism (four engines and four marker threads) it achieves a speedup of 2.73. These speedups are much lower than we might expect from such a program: either the mutator, the collector or both are not being parallelised well.

mandelbrot_lowalloc does not see any benefit from parallel marking. It achieves very good speedups from multiple Mercury engines. We know that this program has a low allocation rate but is otherwise parallelised in the same way as raytracer. Therefore, these results support the hypothesis that heavy use of garbage collection makes it difficult to achieve good speedups when parallelising programs. The more time spend in garbage collection, the worse the speedup due to parallelism.

mandelbrot_highalloc, which stores its data on the heap, sees similar trends in performance as raytracer. It is also universally slower than mandelbrot_lowalloc. mandelbrot_highalloc achieves a speedup of 2.16 when using parallelism in Mercury (four Mercury engines, and one GC thread). The corresponding figure for the raytracer is 1.58. When using the GC's parallelism (one Mercury engine, and four GC threads) mandelbrot_highalloc achieves a speedup of 1.11, compared with 1.29 for the raytracer.

We can see that mandelbrot_highalloc benefits from parallelism in Mercury more than raytracer does, conversely mandelbrot_highalloc benefits from parallelism in the garbage collector less than raytracer does. Using ThreadScope (Chapter 6), we analysed how much time these programs spend running the garbage collector or the mutator. These results are shown in Table 3.2. The table shows the elapsed time, collector time, mutator time and the number of collections for each program using one to four engines, and one to four collector threads. The times are averages taken from eight samples, using an initial heap size of 16MB. To use ThreadScope we had to compile the runtime system differently. Therefore, these results differ slightly from those in Table 3.1. Next to both the collector time and mutator time we show the percentage of elapsed time taken by the collector or mutator respectively.

As we expected, mandelbrot_lowalloc spends very little time running the collector. Typically, it runs the collector only twice during its execution. Also, total collector time was usually between 5 and 30 milliseconds. The other two programs ran the collector hundreds of times. As we increased the number of Mercury engines we noticed that these programs made fewer collections. As we varied the number of collector threads, we saw no trend in the number of collections. Any apparent variation is most likely noise in the data. All three programs see a speedup in the time spent in the mutator as the number of Mercury engines is increased. Similarly, the raytracer and mandelbrot_highalloc benefit from speedups in GC time as the number of GC threads is

GC Markers	Times & Collections	Parallel w/ N Mercury Engines 1		2		3		4	
raytracer									
1	Elapsed	37.8		28.1		24.7		22.7	
	GC	16.9	44.8%	16.7	59.4%	16.5	66.6%	15.8	69.5%
	Mutator	20.9	55.2%	11.4	40.6%	8.2	33.4%	6.9	30.5%
	# col.	384.		346.		316.		288.	
2	Elapsed	32.6		21.8		18.3		16.0	
	GC	11.0	33.8%	10.4	47.7%	10.3	56.4%	9.8	60.8%
	Mutator	21.6	66.2%	11.4	52.3&	8.0	43.6%	6.3	39.2%
	# col.	384.		346.		316.		288.	
3	Elapsed	30.9		20.0		16.4		14.1	
	GC	9.3	30.2%	8.5	42.3%	8.2	50.0%	7.8	55.4%
	Mutator	21.6	69.8%	11.5	57.7%	8.2	50.0%	6.3	44.6%
	# col.	386.		346.		316.		288.	
4	Elapsed	30.3		19.3		15.5		13.5	
	GC	8.7	28.8%	7.8	40.4%	7.4	47.8%	7.1	52.4%
	Mutator	21.6	71.2%	11.5	59.6%	8.1	52.2%	6.4	47.6%
	# col.	387.		346.		316.		288.	
mandelbrot_highalloc									
1	Elapsed	51.7		34.0		27.2		24.0	
	GC	11.0	21.4%	11.5	33.6%	11.5	42.2%	11.8	49.1%
	Mutator	40.7	78.6%	22.6	55.4%	15.5	57.8%	12.2	50.9%
	# col.	742.		692.		634.		602.	
2	Elapsed	48.5		29.3		22.3		19.1	
	GC	6.4	13.1%	6.7	22.9%	6.6	29.7%	6.9	36.1%
	Mutator	42.1	86.9%	22.6	77.1%	15.7	70.3%	12.2	63.9%
	# col.	744.		693.		633.		595.	
3	Elapsed	46.4		28.0		21.0		17.7	
	GC	5.0	10.8%	5.3	18.8%	5.3	25.0%	5.4	30.4%
	Mutator	41.4	89.2%	22.8	81.2%	15.7	75.0%	12.3	69.6%
	# col.	737.		692.		629.		600.	
4	Elapsed	46.0		27.6		20.3		16.9	
	GC	4.5	9.9%	4.7	17.0%	4.6	22.6%	4.7	27.6%
	Mutator	41.4	90.1%	22.9	83.0%	15.7	77.4%	12.3	72.4%
	# col.	740.		695.		626.		600.	
mandelbrot_lowalloc									
1	Elapsed	15.2		7.7		5.1		3.9	
	GC	0.0	0.1%	0.0	0.2%	0.0	0.7%	0.0	0.4%
	Mutator	15.2	99.9%	7.6	99.8%	5.1	99.3%	3.9	99.6%
	# col.	2.		2.		2.		2.	
2	Elapsed	15.4		7.6		5.1		3.9	
	GC	0.0	0.1%	0.0	0.1%	0.0	0.2%	0.0	0.2%
	Mutator	15.4	99.9%	7.6	99.9%	5.1	99.8%	3.9	99.8%
	# col.	2.		2.		2.		2.	
3	Elapsed	15.3		7.7		5.1		3.9	
	GC	0.0	0.1%	0.0	0.1%	0.0	0.4%	0.0	0.1%
	Mutator	15.2	99.9%	7.7	99.9%	5.1	99.6%	3.9	99.9%
	# col	2.		2.		2.		2.	
4	Elapsed	15.2		7.6		5.1		3.9	
	GC	0.0	0.1%	0.0	0.1%	0.0	0.4%	0.0	0.1%
	Mutator	15.2	99.9%	7.6	99.9%	5.1	99.6%	3.9	99.9%
	# col	2.		2.		2.		2.	

Table 3.2: Percentage of elapsed execution time used by GC/Mutator

Program	Allocations	Total alloc'd bytes	Alloc rate (M alloc/sec)
raytracer	561,431,515	9,972,697,312	26.9
mandelbrot_highalloc	3,829,971,662	29,275,209,824	94.1
mandelbrot_lowalloc	1,620,928	21,598,088	0.106

Table 3.3: Memory allocation rates

increased. In mandelbrot_highalloc, the GC time also increases slightly as Mercury engines are added. We expect that as more Mercury engines are used the garbage collector must use more inter-core communication which has additional costs. However, in the raytracer, the GC time decreases slightly as Mercury engines are added. To understand why this happens we would need to understand the collector in detail, however Boehm GC's sources are notoriously difficult to read and understand.

As Amdahl's law predicts, parallelism in one part of the program has a limited effect on the program as a whole. While using one GC thread we tested with one to four Mercury engines; the mutator time speedups for two, three and four engines were 1.83, 2.55 and 3.03 respectively. However, the elapsed time speedups for the same test were only 1.35, 1.53 and 1.66 respectively. Although there is little change in absolute time spent in the collector; there is an increase in collector time as a percentage of elapsed time. The corresponding mutator time speedups for mandelbrot_highalloc were similar, at 1.80, 2.63 and 3.34 for two, three and four Mercury engines. The elapsed time speedups on the same test for mandelbrot_highalloc were 1.52, 1.90 and 2.15. Like raytracer above, the elapsed time speedups are lower than the mutator time speedups. Similarly, when we increase the number of threads used by the collector, it improves collector time more than it does elapsed time.

When we increased both the number of threads used by the collector and the number of Mercury engines (a diagonal path through the table) both the collector and mutator time decrease. This shows that parallelism in Mercury and in Boehm GC both contribute to the elapsed time speedup we saw in the diagonal path in Table 3.1. As Table 3.2 gives us more detail, we can also see that collector time as a percentage of elapsed time increases as we add threads and engines to the collector and Mercury. This occurs for both the raytracer and mandelbrot_highalloc. It suggests that the mutator makes better use of additional Mercury engines than the collector makes use of additional threads. We can confirm this by comparing the speedup for the mutator with the speedup in the collector. In the case of raytracer using four Mercury engines and one GC thread the mutator's speedup is $20.9/6.9 = 3.03$ over the case for one Mercury engine and one GC thread. The collectors speedup with four GC threads and one Mercury engine over the case for one GC thread and one Mercury engine is $16.9/8.7 = 1.94$. The equivalent speedups for mandelbrot_highalloc are: $40.7/12.2 = 3.34$ and $11.0/4.5 = 2.44$. Similar comparisons can be made along the diagonal of the table.

Table 3.2 also shows that raytracer spends more of its elapsed time doing garbage collection than mandelbrot_highalloc does. This matches the results in Table 3.1, where mandelbrot_highalloc has better speedups because of parallelism in Mercury than raytracer does. Likewise, raytracer has better speedups because of parallelism in the collector than mandelbrot_highalloc does. This suggests that mandelbrot_highalloc is less allocation intensive than raytracer, but this is not true. Using Mercury's deep profiler we measured the number and total size of memory allocations. This is shown in Table 3.3. The rightmost column in this table gives the allocation rate measured in

Initial heap size	Sequential no TS	Sequential TS	Parallel w/ N Mercury Engines 1	2	3	4
raytracer						
1MB	32.8 (0.89)	29.3 (1.00)	29.3 (1.00)	19.0 (1.54)	15.4 (1.90)	13.6 (2.16)
16MB	33.5 (0.89)	29.7 (1.00)	29.1 (1.02)	18.9 (1.57)	15.3 (1.94)	13.7 (2.17)
32MB	34.3 (0.86)	29.5 (1.00)	29.5 (1.00)	19.5 (1.51)	15.6 (1.89)	13.6 (2.16)
64MB	33.2 (0.92)	30.4 (1.00)	30.7 (0.99)	20.3 (1.50)	16.4 (1.86)	14.5 (2.10)
128MB	22.7 (1.08)	24.5 (1.00)	24.2 (1.01)	15.0 (1.63)	11.7 (2.09)	10.2 (2.40)
256MB	18.8 (1.13)	21.4 (1.00)	21.5 (0.99)	12.6 (1.70)	9.3 (2.29)	7.7 (2.76)
384MB	17.7 (1.17)	20.8 (1.00)	20.7 (1.01)	11.7 (1.77)	8.7 (2.39)	7.2 (2.90)
512MB	17.3 (1.17)	20.3 (1.00)	20.5 (0.99)	11.5 (1.77)	8.3 (2.44)	6.8 (2.99)
mandelbrot_highalloc						
1MB	41.4 (1.10)	45.5 (1.00)	45.1 (1.01)	26.7 (1.70)	20.3 (2.24)	16.7 (2.72)
16MB	39.7 (1.14)	45.3 (1.00)	45.6 (0.99)	26.8 (1.69)	20.4 (2.23)	16.7 (2.72)
32MB	38.9 (1.16)	45.2 (1.00)	44.2 (1.02)	26.3 (1.72)	19.7 (2.29)	16.5 (2.74)
64MB	36.7 (1.21)	44.4 (1.00)	43.7 (1.02)	25.2 (1.76)	18.8 (2.36)	15.6 (2.84)
128MB	34.1 (1.25)	42.6 (1.00)	41.8 (1.02)	24.1 (1.77)	17.7 (2.41)	14.4 (2.96)
256MB	33.2 (1.24)	41.3 (1.00)	41.9 (0.99)	23.7 (1.74)	16.9 (2.45)	13.6 (3.04)
384MB	31.7 (1.29)	41.0 (1.00)	41.9 (0.98)	23.3 (1.76)	16.7 (2.46)	13.4 (3.07)
512MB	31.1 (1.32)	41.1 (1.00)	41.1 (1.00)	23.0 (1.79)	16.7 (2.46)	13.3 (3.08)

Table 3.4: Varying the initial heapsize in parallel Mercury programs.

millions of allocations per second. It is calculated by dividing the number of allocations in the second column by the average mutator time reported in Table 3.2. Therefore, it represents the allocation rate during mutator time. This is deliberate because, by definition, allocation cannot occur during collector time. mandelbrot_highalloc has a higher allocation rate than raytracer. However, measuring either absolutely or as a percentage of elapsed time, mandelbrot_highalloc spends less time doing collection than the raytracer. Garbage collectors are tuned for particular workloads. It is likely that Boehm GC handles mandelbrot_highalloc's workload more easily than raytracer's workload.

So far, we have shown that speedups due to Mercury's parallel conjunction are limited by the garbage collector. Better speedups can be achieved by using the parallel marking feature in the garbage collector. Now we will show how performance can be improved by modifying the initial heap size of the program. Table 3.4 shows the performance of the raytracer and mandelbrot_highalloc programs with various initial heap sizes. The first column shows the heap size used; we picked a number of sizes from 1MB to 512MB. The remaining columns show the average elapsed times in seconds. The first of these columns shows timing for the programs compiled for sequential execution without thread safety; this means that the collector cannot use parallel marking. The next column is for sequential execution with thread safety; the collector uses parallel marking with four threads. The next four columns give the results for the programs compiled for parallel execution, and executed with one to four Mercury engines. These results also use four marker threads. The numbers in parentheses are the ratio of elapsed time compared with the sequential thread-safe result for the same initial heap size. All the results are the averages of eight test runs. We ran these tests on raytracer and mandelbrot_highalloc. We did not use mandelbrot_lowalloc as its collection time is insignificant and would not have provided useful data.

Generally, the larger the initial heap size the better the programs performed. The Boehm GC will begin collecting if it cannot satisfy a memory allocation request. If, after a collection, it still

cannot satisfy the memory request then it will increase the size of the heap. In each collection, the collector must read all the stacks, global data, thread local data and all the in-use memory in the heap (memory that is reachable from the stacks, global data and thread local data). This causes a lot of cache misses, especially when the heap is large. The larger the heap size, the less frequently memory is exhausted and needs to be collected. Therefore, programs with larger heap sizes garbage collect less often and have fewer cache misses due to garbage collection. This explains the trend of increased performance as we increase the initial heap size of the program.

Although performance improved with larger heap sizes, there is an exception. The raytracer often ran more slowly with a heap size of 64MB than with a heap size of 32MB. 64MB is much larger than the processor's cache size (this processor's L3 cache is 8MB) and covers more page mappings than its TLBs can hold (the L2 TLB covers 2MB when using 4KB pages). The collector's structures and access patterns may be slower at this size because of these hardware limitations, and the benefits of a 64MB heap are not enough to overcome the effects of these limitations, however the benefits of a 128MB or larger heap are enough.

Above, we said that programs perform better with larger heap sizes. This measurement compares their elapsed time with one heap size with the elapsed time using a different heap size. We also found that programs *exploited parallelism more easily* with larger heap sizes: This measurement compares programs' speedups (which are themselves comparisons of elapsed time). When the raytracer uses a 16MB initial heap its speedup using four cores is 2.17 times. However, when it uses a 512MB initial heap the corresponding speedup is 2.99. Similarly, mandelbrot_highalloc has a 4-engine speedup of 2.72 using a 16MB initial heap and a speedup of 3.08 using a 512MB initial heap. Generally, larger heap sizes allow programs to exploit more parallelism. There are two reasons for this

Reason 1 In a parallel program with more than one Mercury engine, each collection must *stop-the-world*: all Mercury engines are stopped so that they do not modify the heap during the marking phase. This requires synchronisation which reduces the performance of parallel programs. The less frequently collection occurs, the less performance is affected by the synchronisation of stop-the-world events.

Reason 2 Another reason was suggested by Simon Marlow: Because Boehm GC uses its own threads for marking and not Mercury's, it cannot mark objects with the same thread that allocated the objects. Unless Mercury and Boehm GC both map and pin their threads to processors, and agree on the mapping, then marking will cause a large number of cache misses. For example, during collection, processor one (P1) marks one of processor two's (P2) objects, causing a cache miss in P1's cache and invalidating the corresponding cache line in P2's cache. Later, when collection finishes, P2's process resumes execution and incurs a cache miss for the object that it had been using. Simon Marlow made a similar observation when working with GHC's garbage collector.

We also investigated another area for increased parallel performance. Boehm GC maintains thread local free lists that allow memory to be allocated without contending for locks on the global free list. When a thread's local free list cannot satisfy a memory request, the global free list must be used. The local free lists amortise the costs of locking the global free list. We anticipate that increasing the size of local free lists will cause even less contention for global locks, allowing allocation intensive programs to have better parallel performance. We wanted to investigate if

increasing the size of the local free lists could improve performance. We contacted the Boehm GC team about this and they advised us to experiment with the `HBLKSIZE` tunable. They did not say what this tunable controls. it is undocumented and the source code is very hard to understand. Our impression is that it only indirectly tunes the sizes of the local free lists. Unfortunately this feature is experimental: adjusting `HBLKSIZE` caused our programs to crash intermittently, such that we could not determine for which values (other than the default) the programs would run reliably. Therefore we cannot evaluate how `HBLKSIZE` affects our programs. Once this feature is no longer experimental, adjusting `HBLKSIZE` to improve parallel allocation should be investigated.

3.2 Original spark scheduling algorithm

In Sections 2.2 and 2.3, we introduced parallelism in Mercury and described the runtime system in general terms. In this section we will explain how sparks were originally managed prior to 2009, when I began my Ph.D. candidature. This will provide the background for the changes we have made to the runtime system since then.

Mercury (before 2009) uses a global spark queue. The runtime system *schedules* sparks for parallel execution by placing them onto the end of the queue. In this chapter we use the word 'schedule' to mean the act of making a spark available for parallel or sequential work. An idle engine runs a spark by taking it from the beginning of the queue. The global spark queue must be protected by a lock; this prevents concurrent access from corrupting the queue. The global spark queue and its lock can easily become a bottleneck when many engines contend for access to the global queue.

Wang [113] anticipated this problem and created context local spark stacks to avoid contention on the global queue. Furthermore, the local spark stacks do not require locking. When a parallel conjunction spawns off a spark, it places the spark either at the end of the global spark queue or at the top of its local spark stack. The runtime system appends the spark to the end of the global queue if an engine is idle, and the number of contexts in use plus the number of sparks on the global queue does not exceed the maximum number of contexts permitted. If either part of the condition is false, the runtime system pushes the spark onto the top of the context's local spark stack. This algorithm has two aims. The first is to reduce contention on the global queue, especially in the common case that there is enough parallel work. The second aim is to reduce the amount of memory allocated as contexts' stacks by reducing the number of contexts allocated. Globally scheduled sparks may be converted into contexts, so they are also included in this limit. We explain this limit on the number of contexts in more detail in Section 3.3, after covering the background information in the current section. Note that sparks placed on the global queue are executed in a first-in-first-out manner; sparks placed on a context's local stack are executed in a last-in-first-out manner.

In Section 2.2 we described how parallel conjunctions are compiled (Figure 2.6 shows an example). Consider the compiled parallel conjunction in Figure 3.1. The context that executes the parallel conjunction, let us call it C_{Orig}, begins by setting up the sync term, spawning off G_2, and executing G_1 (lines 1–4). Then it executes the barrier `MR_join_and_continue`, shown in Algorithm 3.1. The algorithms throughout this chapter use a macro named `MR_ENGINE` to access the fields of the structure representing the current engine. Depending on how full the global run queue is, and how parallel tasks are interleaved, there are three important scenarios:

Algorithm 3.1 `MR_join_and_continue`— original version

```
 1  void MR_join_and_continue(MR_SyncTerm *st, MR_Code *cont_label) {
 2      MR_Context  *current_context = MR_ENGINE(MR_eng_current_context);
 3
 4      MR_acquire_lock(&MR_SyncTermLock);
 5      st->MR_st_num_outstanding--;
 5      if (st->MR_st_num_outstanding == 0) {
 6          if (st->MR_st_orig_context == current_context) {
 7              MR_release_lock(&MR_SyncTermLock);
 9              MR_GOTO(cont_label);
10          } else {
11              st->MR_st_orig_context->MR_ctxt_resume_label = cont_label;
12              MR_schedule(st->MR_st_orig_context);
13              MR_release_lock(&MR_SyncTermLock);
14              MR_GOTO(MR_idle);
15          }
16      } else {
17          MR_Spark    spark;
18          MR_bool     popped;
19
20          popped = MR_pop_spark(current_context->MR_ctxt_spark_stack, &spark);
21          if (popped && (spark.MR_spark_parent_sp == MR_parent_sp)) {
22              /*
23              ** The spark at the top of the stack is part of the same
24              ** parallel conjunction, we can exuecute it immediately.
25              */
26              MR_release_lock(&MR_SyncTermLock);
27              MR_GOTO(spark.MR_spark_code);
28          } else {
29              if (popped) {
30                  /*
31                  ** The spark is part of a different parallel conjunction,
32                  ** put it back.
33                  */
34                  MR_push_spark(current_context->MR_ctxt_spark_stack, &spark);
35              }
36              if (st->MR_st_orig_context == current_context) {
37                  MR_save_context(current_context)
38                  MR_ENGINE(MR_eng_current_context) = NULL;
39              }
40              MR_release_lock(&MR_SyncTermLock);
41              MR_GOTO(MR_idle);
42          }
43      }
44  }
```

```
1:    MR_SyncTerm ST;
2:    MR_init_syncterm(&ST, 2);
3:    spawn_off(&ST, Spawn_Label_1);
4:    G1
5:    MR_join_and_continue(&ST, Cont_Label);
6:  Spawn_Label:
7:    G2
8:    MR_join_and_continue(&ST, Cont_Label);
9:  Cont_Label:
```

Figure 3.1: Parallel conjunction implementation

Scenario one:

In this scenario C_{Orig} placed the spark for G_2 on the top of its local spark stack. Sparks placed on a context's local spark stack cannot be executed by any other context. Therefore when C_{Orig} reaches the `MR_join_and_continue` barrier (line 5 in the example, Figure 3.1), the context G_2 will be outstanding and `st->MR_st_num_outstanding` will be non-zero. C_{Orig} will execute the else branch on lines 17–42 of `MR_join_and_continue`, where it will pop the spark for G_2 off the top of the spark stack. It is not possible for some other spark to be on the top of the stack; any sparks left on the stack by G_1 would have been popped off by the `MR_join_and_continue` barriers of the conjunctions that spawned off the sparks. This invariant requires a *last-in-first-out* storage of sparks, which is why each context uses a stack rather than a queue. In Section 3.4 we explain in more detail *why* a *last-in-first-out* order is important.

The check that the spark's stack pointer is equal to the current parent stack pointer[2] will succeed (line 11 of `MR_join_and_continue`), and the context will execute the spark.

After executing G_2, C_{Orig} will execute the second call to `MR_join_and_continue`, the one on line 8 of the example. This time `st->MR_st_num_outstanding` will be zero, and C_{Orig} will execute the then branch on lines 7–15 of `MR_join_and_continue`. In the condition of the nested if-then-else, C_{Orig} will find that the current context is the original context, and therefore continue execution at line 8. This causes C_{Orig} to jump to the continuation label, line 9 in the example, completing the execution of the parallel conjunction.

Scenario two:

In this scenario C_{Orig} placed the spark for G_2 on the global spark queue, where another context, C_{Other}, picked it up and executed it in parallel. Also, as distinct from scenario three below, C_{Other} reaches the barrier on line 8 of the example (Figure 3.1) *before* C_{Orig} reaches the barrier on line 5. Even if they both seem to reach the barrier at the same time, their barrier operations are performed in sequence because of the lock protecting the barrier code.

When C_{Other} executes `MR_join_and_continue`, It will find that `st->MR_st_num_outstanding` is non-zero, and will execute the else branch on lines 17–42 of `MR_join_and_continue`. It then attempts to pop a spark off its stack, as in another scenario a spark on the stack might represent an outstanding conjunction (it cannot tell that the outstanding conjunct is G_1 executing in parallel, and not some hypothetical G_3). C_{Other} took this spark from the global queue, and was either empty or brand new before executing this spark, meaning that it had

[2]The code generator will ensure that `MR_parent_sp` is set before the parallel conjunction is executed, and that it is restored after.

no sparks on its stack before executing G_2. Therefore C_{Other} will not have any sparks of its own and `MR_pop_spark` will fail. C_{Other} will continue to line 36 in `MR_join_and_continue` whose condition will also fail since C_{Other} is not the original context (C_{Orig}). The lock will be released and `MR_join_and_continue` will determine what to do next by jumping to `MR_idle`.

Eventually C_{Orig} will execute its call to `MR_join_and_continue` (line 5 of the example), or resume execution after waiting on the barrier's lock (line 4 of `MR_join_and_continue`),. When this happens it will find that `st->MR_st_num_outstanding` is zero, and execute the then branch beginning at line 7 of `MR_join_and_continue`. C_{Orig} will test if it is the original context, which it is, and continue on line 8 of `MR_join_and_continue`. It then jumps to the continuation label on line 9 of the example, completing the parallel execution of the conjunction.

Scenario three:

As in scenario two, C_{Orig} put the spark for G_2 on the global spark queue, where another context, C_{Other}, picked it up and executed it in parallel. However, in this scenario C_{Orig} reaches the barrier on line 5 in the example (Figure 3.1) *before* C_{Other} reaches its barrier on line 8.

When C_{Orig} executes `MR_join_and_continue`, it finds that `st->MR_st_num_outstanding` is non-zero, causing it to execute the else branch on lines 17–42 of `MR_join_and_continue`. C_{Orig} will try to pop a spark of its local spark stack. However the spark for G_2 was placed on the global spark queue, the only spark it might find is one created by an outer conjunction. If a spark is found, the spark's parent stack pointer will not match the current parent stack pointer, and it will put the spark back on the stack. C_{Orig} executes the then branch (lines 37–38 of `MR_join_and_continue`), since this context is the original context. This branch will suspend C_{Orig}, and set the engine's context pointer to `NULL` before jumping to `MR_idle`.

When C_{Other} reaches the barrier on line 8, it will find that `st->MR_st_num_outstanding` is zero, and will execute the then branch of the if-then-else. Within this branch it will test to see if it is the original context, the test will fail, and the else branch of the nested if-then-else will be executed. At this point we know that C_{Orig} must be suspended because there were no outstanding conjuncts and the current context is not the original context; this can only happen if C_{Orig} is suspended. The code wakes C_{Orig} up by setting its code pointer to the continuation label, placing it on the global run queue, and then jumping to `MR_idle`.

When C_{Orig} resumes execution it executes the code on line 9, which is the continuation label.

The algorithm includes an optimisation not shown here: if the parallel conjunction has been executed by only one context, then a version of the algorithm without locking is used. We have not shown the optimisation because it is equivalent and not relevant to our discussion; we mention it only for completeness.

When an engine cannot get any local work it must search for global work. Newly created engines, except for the first, also search for global work. They do this by calling `MR_idle`, whose code is shown in Algorithm 3.2. Only one of the idle engines can execute `MR_idle` at a time. `MR_runqueue_lock` protects the context run queue and the global spark queue from concurrent access. After acquiring the lock, engines execute a loop. An engine exits the loop only when it finds some work to do or the program is exiting. Each iteration first checks if the runtime system is being shut down, if so, then this thread releases the lock, and then destroys itself. If the system

Algorithm 3.2 `MR_idle`— original version

```
void MR_idle() {
    MR_Context  *ctxt;
    MR_Spark    *spark;

    MR_acquire_lock(&MR_runqueue_lock);
    while(MR_True) {
        if (MR_exit_now) {
            MR_release_lock(MR_runqueue_lock);
            MR_destroy_thread();                          // does not return.
        }

        // Try to run a context
        ctxt = MR_get_runnable_context();
        if (ctxt != NULL) {
            MR_release_lock(&MR_runqueue_lock);
            if (MR_ENGINE(MR_eng_current_context) != NULL) {
                MR_release_context(MR_ENGINE(MR_eng_current_context));
            }
            MR_load_context(ctxt);
            MR_GOTO(ctxt->MR_ctxt_resume);
        }

        // Try to run a spark.
        spark = MR_dequeue_spark(MR_global_spark_queue);
        if (spark != NULL) {
            MR_release_lock(&MR_runqueue_lock);
            if (MR_ENGINE(MR_eng_current_context) == NULL) {
                ctxt = MR_get_free_context();
                if (ctxt == NULL) {
                    ctxt = MR_create_context();
                }
                MR_load_context(ctxt);
            }
            MR_parent_sp = spark->MR_spark_parent_sp;
            ctxt->MR_ctxt_thread_local_mutables =
                spark->MR_spark_thread_local_mutables;
            MR_GOTO(spark->MR_spark_code);
        }

        MR_wait(&MR_runqueue_cond, &MR_runqueue_lock);
    }
}
```

```
map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    P(X, Y) &
    map(P, Xs, Ys).
```

(a) Right recursive

```
map(_, [], []).
map(P, [X | Xs], [Y | Ys]) :-
    map(P, Xs, Ys) &
    P(X, Y).
```

(b) Left recursive

Figure 3.2: Right and left recursive map/3

is not being shut down, the engine will search for a runnable context. If it finds a context, it releases the run queue lock, loads the context and jumps to the resume point for the context. If it already had a context, then it first releases that context; doing so is safe because `MR_idle` has a precondition that if an engine has a context, the context must not contain a spark or represent a suspended computation. If no context was found, the engine attempts to take a spark from the global spark queue. If it finds a spark then it will need a context to execute that spark. It will try to get a context from the free list; if there is none it will create a new context. Once it has a context, it copies the context's copies of registers into the engine. It also initialises the engine's parent stack pointer register and the spark's thread local mutables (which are set by the context that created the spark) into the context. If the engine does not find any work, it will wait using a condition variable and the run queue lock. The pthread wait function is able to unlock the lock and wait on the condition variable atomically, preventing race conditions. The condition variable is used to wake up the engine if either a spark is placed on the global spark queue or a context is placed on the context run queue. When the engine wakes, it will re-execute the loop.

3.3 Original spark scheduling performance

In Section 3.1 we ran our benchmarks with a recent version of the runtime system. In the rest of this chapter we describe many of the improvements to the runtime system that led to the improved parallel performance reported in that section.

Figure 3.2 shows two alternative, parallel implementations of `map/3`. While their declarative semantics are identical, their operational semantics are very different. In Section 2.2 we explained that parallel conjunctions are implemented by spawning off the second and later conjuncts and executing the first conjunct directly. In the right recursive case (Figure 3.2(a)), the recursive call is spawned off as a spark, and in the left recursive case (Figure 3.2(b)), the recursive call is executed directly, and the loop *body* is spawned off. Declarative programmers are taught to prefer tail recursion, and therefore tend to write right recursive code.

Table 3.5 shows average elapsed time in seconds for the mandelbrot_lowalloc program from 20 test runs. We described this program above in Section 3.1. Using this program we can easily observe the speedup due to parallelism in Mercury without the effects of the garbage collector. The main loop of the renderer uses right recursion, and is similar to `map/3` (Figure 3.2(a)). This is the loop that iterates over the image's rows. The leftmost column of the table shows the maximum number of contexts that may exist at any time. This is the limit that was introduced in the previous section. The next two columns give the elapsed execution time for a sequential version of the program, that is, the program compiled without the use of the parallel conjunction operator. These two columns give results without and with thread safety. The following four columns give

Max no. of contexts	Sequential not TS	Sequential TS	Parallel w/ N Engines 1	2	3	4
4	23.2 (0.93)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
64	-	-	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
128	-	-	21.5 (1.00)	19.8 (1.09)	20.9 (1.03)	21.2 (1.01)
256	-	-	21.5 (1.00)	13.2 (1.63)	15.5 (1.38)	16.5 (1.30)
512	-	-	21.5 (1.00)	11.9 (1.81)	8.1 (2.66)	6.1 (3.55)
1024	-	-	21.5 (1.00)	11.8 (1.81)	8.0 (2.67)	6.1 (3.55)
2048	-	-	21.5 (1.00)	11.9 (1.81)	8.0 (2.67)	6.0 (3.55)

Table 3.5: Right recursion performance.

the elapsed execution times using one to four Mercury engines. Each value in the table is a mean of 20 test runs. The numbers in parentheses are the ratio between the mean time and the mean sequential thread safe time.

In general we achieve more parallelism when we use more contexts, up to a threshold of 601 contexts. The threshold is at this point because there are 600 rows in the image meaning 600 iterations of the loop plus 1 for the base case. Each iteration may consume a context. This is why the program does not benefit greatly from a high limit such as 1024 or 2048 contexts. When a spark is executed in its parent context (two iterations execute sequentially) then the program may use fewer than 600 contexts. This is possibly why a limit of 512 contexts also results in a good parallel speedup. When only 256 contexts are used, the four core version achieves a speedup of 1.30, compared to 3.55 for 512 or more contexts. Given that mandelbrot uses independent parallelism, ideally there should never be any need to suspend a context. Therefore the program should parallelise well enough when restricted to a small number of contexts (four to eight). Too many contexts are needed to execute this program at the level of parallelism that we want. We noticed the same pattern in other programs with right recursive parallelism.

To understand this problem, we must consider how parallel conjunctions are executed (see Section 2.2). We will step through the execution of `map/3` from Figure 3.2(a). The original context creates a spark for the second and later conjuncts and puts it on the global spark queue. It then executes the first conjunct `P/2`. This takes much less time to execute than the spawned off call to `map/3`. After executing `P/2` the original context will call the `MR_join_and_continue` barrier. It then attempts to execute a spark from its local spark stack, which will fail because the only spark was placed on the global spark queue. The original context cannot proceed until after the other context finishes, but it will be needed then and must then be kept in memory until then. Therefore it is suspended until all the other 599 iterations of the loop have finished. Meanwhile the spark that was placed on the global run queue is converted into a new context. This new context enters the recursive call and becomes blocked within the recursive instance of the same parallel conjunction; it must wait for the remaining 598 iterations of the loop. This process continues to repeat itself, allocating more contexts. Each context consumes a significant amount of memory, much more than one stack frame. Therefore this problem makes programs that look tail recursive *consume much more memory than* sequential programs that are not tail recursive. We implement the limit on the number of contexts to prevent pathological cases such as this one from crashing the program or the whole computer. Once the limit is reached sparks cannot be placed on the global run queue and sequential execution is used. Therefore the context limit is a trade off between memory usage and parallelism.

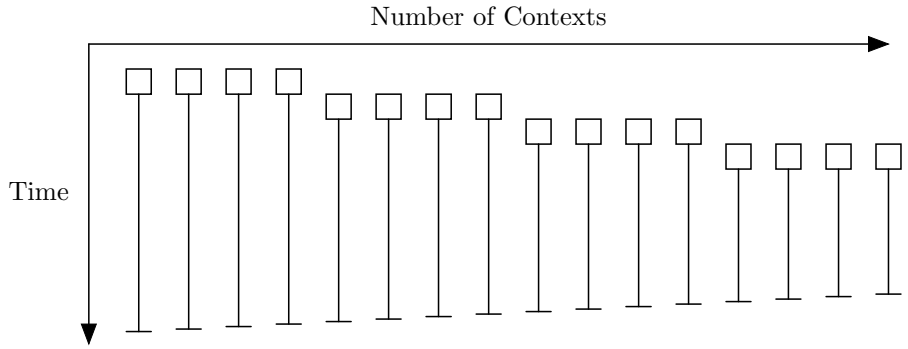


Figure 3.3: Linear context usage in right recursion

An example of context usage over time using four engines is shown in Figure 3.3. Time is shown on the Y axis and contexts are drawn so that each additional context is further along the X axis. At the top left, four contexts are created and they execute four iterations of a loop; this execution is indicated by boxes. Once each of these iterations finishes, its context stays in memory but is suspended, indicated by the long vertical lines. Another four iterations of the loop create another four contexts, and so on. Later, when all iterations of the loop have been executed, each of the blocked contexts resumes execution and immediately exits, indicated by the horizontal line at the bottom of each of the vertical lines. Later in the project we developed a solution to this problem (see Chapter 5). Before developing the solution, we developed a work around (see Section 3.5.

When using either 128 or 256 contexts, we noticed that when we used three or four Mercury engines the program ran more slowly than with two Mercury engines. A possible reason is that: the more Mercury engines there are, the more often at least one engine is idle. When creating a spark, the runtime system checks for an idle engine, if there is one then the spark may be placed on the global queue, subject to the context limit (see Section 3.2). If there is no idle engine, a spark is placed on the context's local queue regardless of the current number of contexts. When there are fewer Mercury engines being used then it is less likely that one of them is idle. Therefore more sparks are placed on the local queue and executed sequentially and the program does not hit the context limit as quickly as one using more Mercury engines. The program can exploit more parallelism as it hits the context limit later, and achieves better speedups.

Only right recursion uses one context for each iteration of a loop. A left recursive predicate (Figure 3.2(b)) has its recursive call on the left of the parallel conjunction. The context executing the conjunction sparks off the body of the loop (in the figure this is `P`), and executes the recursive call directly. By the time the recursive call finishes, the conjunct containing `P` should have also finished. The context that created the parallel conjunction is less likely to be blocked at the barrier, and the context executing the spark is never blocked since it is not the original context that executed the conjunction.

Table 3.6 shows benchmark results using left recursion. The table is broken into three sections:

1. a copy of the right recursion data from Table 3.5, which is presented again for comparison;
2. the left recursion data, which we will discuss now;
3. and left recursion data with a modified context limit, which we will discuss in a moment.

The figures in parenthesis are the standard deviation of the samples.

Max no. of contexts	Sequential not TS	Sequential TS	Parallel w/ N Engines 1	2	3	4
Right recursion						
4	23.2 (0.00)	21.5 (0.04)	21.5 (0.02)	21.5 (0.00)	21.5 (0.01)	21.5 (0.02)
64	-	-	21.5 (0.03)	21.5 (0.01)	21.5 (0.01)	21.5 (0.02)
128	-	-	21.5 (0.02)	19.8 (0.01)	20.9 (0.03)	21.2 (0.03)
256	-	-	21.5 (0.02)	13.2 (0.05)	15.5 (0.06)	16.5 (0.07)
512	-	-	21.5 (0.02)	11.9 (0.11)	8.1 (0.09)	6.1 (0.08)
1024	-	-	21.5 (0.03)	11.8 (0.11)	8.0 (0.06)	6.1 (0.08)
2048	-	-	21.5 (0.03)	11.9 (0.10)	8.0 (0.08)	6.0 (0.06)
Left recursion (Sparks included in context limit)						
4	23.2 (0.01)	21.5 (0.03)	21.5 (0.02)	21.5 (0.04)	21.5 (0.04)	21.5 (0.02)
64	-	-	21.5 (0.02)	21.5 (0.02)	21.4 (0.03)	21.5 (0.03)
128	-	-	21.5 (0.04)	21.5 (0.03)	21.5 (0.03)	21.5 (0.03)
256	-	-	21.5 (0.02)	18.3 (0.75)	18.2 (0.03)	19.6 (1.37)
512	-	-	21.5 (0.02)	17.9 (0.83)	15.5 (1.29)	16.4 (7.09)
1024	-	-	21.5 (0.03)	18.0 (0.85)	14.7 (2.18)	16.1 (5.23)
2048	-	-	21.5 (0.02)	18.0 (0.85)	15.4 (2.15)	17.8 (5.25)
Left recursion (Sparks excluded from context limit)						
N/A	23.3 (0.01)	21.5 (0.02)	21.7 (0.78)	17.9 (0.60)	15.6 (2.49)	14.2 (6.94)

Table 3.6: Right and Left recursion shown with standard deviation

The left recursive figures are underwhelming; they are worse than right recursion. Furthermore, the standard deviations for left recursion results are much higher. In particular, the more contexts and Mercury engines are used, the higher the standard deviation. Despite this, we can see that the left recursion results are much slower than the right recursive results.

Both left and right recursion are affected by the context limit, however the cause for left recursion is different. Consider a case of two Mercury engines and a context limit of eight. The first Mercury engine is executing the original context, which enters the left recursive parallel conjunction and spawns off a spark, adding it to the global spark queue. The original context then executes the recursive call and continues to spawn off sparks. As we described in Section 3.2, each spark on the global spark queue may be converted into a new context, and therefore sparks on the global queue contribute towards the context limit. With left recursion, the first engine executes its tight loop which spawns off sparks. Meanwhile the second engine takes sparks from the queue and executes them; the execution of the work that a spark represents takes more time than each recursive call executed by the first engine. Therefore the first engine will put sparks on the global queue more quickly than the second engine can remove and execute them. After eight to twelve recursions, it is very likely that the sparks on the global queue will exceed the context limit, and that new sparks are now placed on the original context's local spark stack. This happens sooner if the context limit is low.

The high variance in all the left recursive results is indicative of nondeterminism. The cause is going to be something that either varies between execution runs or does not always effect execution runs. A potential explanation is that in different runs, different numbers of sparks are executed in parallel before the context limit is reached. However this cannot be true, or at least is only partly true, as the variance is higher when the context limit is higher, including cases where 2048 contexts are permitted and we know the program needs at most 601 (usually less). Fortunately there is a better explanation. The context limit is one of the two things used to make this scheduling

decision; the other is: if there is no idle engine then the spark is always placed on the context local spark stack (Section 3.2). At the time when the parallel conjunction in `map/3` is executed, the other engines will initially be idle but will quickly become busy, and once they are busy the original context will not place sparks on the global spark queue. If the other engines are slow to respond to the first sparks placed on the global queue, then more sparks are placed on this queue and more parallel work is available. If they are quick to respond, then more sparks will be placed on the original contexts local queue, where they will be executed sequentially.

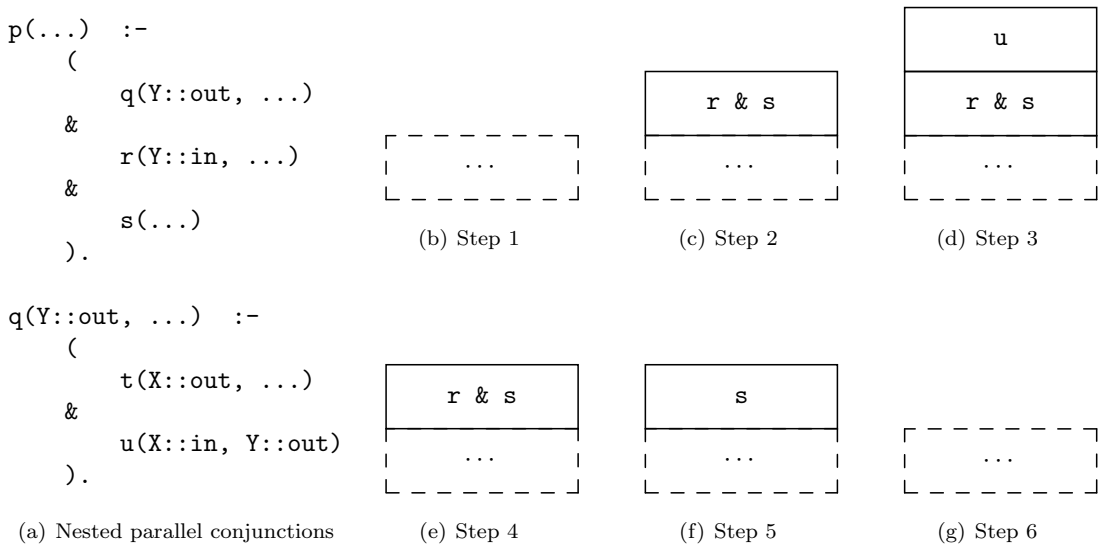
Mandelbrot uses only independent parallelism, which means no context can ever be blocked by a future. In a left recursive case, a computation that is spawned off on a spark is never blocked by a `MR_join_and_continue` barrier; only the original context executes a `MR_join_and_continue` that may block. Therefore the program never uses more than one context per Mercury engine plus the original context. For this program we can safely modify the runtime system so that globally scheduled sparks do not count towards the context limit. We did test that a low context limit prevents the majority of sparks from being placed on the global queue and being eligible for parallel execution. The final row group in Table 3.6 shows results for the left recursive test where sparks in the global queue do not count towards the context limit. As expected, we got similar results for different values of the context limit. We have therefore shown only one row of results in the table. This result is similar to those in the left recursive group with an unmodified runtime system and a sufficiently high context limit. Therefore we can say that the context limit is affecting left recursion in this way.

Both of the problems involving left recursion have the same cause: the decision to execute a spark sequentially or in parallel is made too early. This decision is made when the spark is scheduled, and placed either on the global spark queue or the context local spark stack. The data used to make the decision includes the number of contexts in use, the number of sparks on the global queue, and if there is an idle engine. These conditions will be different when the spark is scheduled compared to when it may be executed, and the conditions when it is scheduled are not a good indication of the conditions when it may be executed. We will refer to this problem as the *premature spark scheduling problem*. In the next section, we solve this problem by delaying the decision whether to execute a spark in parallel or in sequence.

3.4 Initial work stealing implementation

Work stealing addresses the premature spark scheduling problem that we described in the previous section. Wang [113] recognised this problem and proposed work stealing as its solution. In a work stealing system, sparks placed on a context's local spark stack are not committed to running in that context; they may be executed in a different context if they are stolen (we describe below why we use a stack to store sparks). This delays the decision of where to execute a spark until the moment before it is executed.

Work stealing is a popular method for managing parallel work in a shared memory multiprocessor system. Multilisp (Halstead [50]) was one of the first systems to use work stealing, which Halstead calls "an unfair scheduling policy". The term "unfair" is not an expression of the morals of stealing (work), instead it refers to the unfairness of *cooperative multitasking* when compared to something like *round-robin scheduling*. Each processor in Multilisp has a currently running task, and a stack of suspended tasks. If the current task is suspended on a future or finishes, the processor will execute the task at the top of its task stack. If there is no such task, it attempts to



(a) Nested parallel conjunctions (b) Step 1 (c) Step 2 (d) Step 3 (e) Step 4 (f) Step 5 (g) Step 6

Figure 3.4: Spark execution order

steal a task from the bottom of another processor's stack.

Halstead's "unfair scheduling" is motivated by his need to reduce the problems of embarrassing parallelism. Most significantly, he says that the system can deadlock due to running out of resources but needing resources to proceed. We would like to point out that this is not the same as our right recursion problem, even though the two have similar symptoms. Furthermore, work stealing does not fix the right recursion problem.

Work stealing's other benefits can be summarised as follows.

- It is quicker to schedule sparks because a context can place sparks on its local spark stack more quickly than it can place sparks on the global spark queue. This is because there is less contention on a context's local spark stack. The only source of contention is work stealing, which is balanced among all of the contexts. Assume each of the C contexts' spark stacks contains sparks. Contention on any one of these stacks is $1/C$ times less than it would be on a global queue.

- An idle worker can take work from its own queue quickly. Again, there is less contention on a local queue.

- In ideal circumstances, an idle worker rarely needs to steal work from another's queue. Stealing work is more costly as it requires communication between processors.

Mercury's initial work stealing implementation was built jointly by Peter Wang and myself, Wang contributed about 80% of the work and I contributed the remaining 20%. Wang's honours thesis [113] describes his proposal on which our implementation is heavily based.

Until now, each context has used a stack to manage sparks. The last item pushed onto the stack is the first to be popped off the stack. This *last-in-first-out* order is important for Mercury's parallel conjunctions. Parallel conjunctions may be nested, either in the same procedure or through procedure calls such as in Figure 3.4(a). Consider a context whose spark stack's contents are initially undefined. The spark stack is represented by Figure 3.4(b). When the context calls `p` it

creates a spark for the second and third conjuncts `r(...) & q(...)` and places the spark on its local stack (Figure 3.4(c)). It then calls `q`, where it creates another spark. This spark represents the call to `u`, and now the spark stack looks like Figure 3.4(d). Assuming that no work stealing has occurred, when the context finishes executing `t` it pops the spark from the top of its stack; the popped spark represents the call to `u` because the stack returns items in a last-in-first-out order. This is desirable because it follows the same execution order as sequential execution would. We call this left-to-right execution since by default the left operand of a conjunction is executed first. The conjunction in `q` contains a shared variable named `X`; this execution order ensures that the `future_signal/2` operation for `X`'s future occurs before the `future_wait/2` operation; the context is not blocked by the call to `future_wait/2`. (Note that the mode annotations are illustrative, they are not part of Mercury's syntax.) Once the spark for `u` is popped off the spark stack then the spark stack looks like Figure 3.4(e). After the execution of `u`, `q` returns control to `p`, which then continues the execution of its own parallel conjunction. `p` pops a spark off the spark stack; the spark popped off is the parallel conjunction `r(...) & s(...)`. This immediately creates a spark for `s` and pushes it onto the spark stack (Figure 3.4(f)) and then executes `r`. The execution of `r` at this point is also the same as what would occur if sequential conjunctions were used. This is straightforward because the spark represented the *rest* of the parallel conjunction. Finally `r` returns and `p` pops the spark for `s` off of the spark stack and executes it. At this point the context's spark stack looks like Figure 3.4(g).

If at step 3, the context executed the spark for `r(...) & s(...)` then `r` would have blocked on the future for `Y`. This would have used more contexts than necessary and more overheads than necessary. It may have also caused a deadlock: since stealing a spark may create a context which may not be permitted due to the context limit. Furthermore the original context would not be able to continue the execution in `p` without special stack handling routines; this would make the implementation more complicated than necessary. Therefore from a context's perspective its local spark storage must behave like a stack. Keeping this sequential execution order for parallel conjunctions has an additional benefit: it ensures that a popped spark's data has a good chance of being hot in the processor's cache.

Prior to work stealing, context local spark stacks returned sparks in last-in-first-out order, the global spark queue returns sparks in *first-in-first-out* order. This does not encourage a left-to-right execution order, however Wang [113] proposes that this order may be better:

> The global spark queue, however, is a queue because we assume that a spark generated earlier will have greater amounts of work underneath it than the latest generated spark, and hence is a better candidate for parallel execution.

If we translate this execution order into the work stealing system, it means that work is stolen from the bottom end of a context's local spark stack. Consider step 3 of the example above. If a thief removes the spark for `r & s` then only the spark for `u` is on the stack, and the original context can continue its execution by taking the spark to `u`. By comparison if the spark for `u` was stolen, it would force the original context to try to execute `r & s` which, for the reasons above, is less optimal. Therefore a thief steals work from the bottom of a context local stack so that a first-in-first-out order is used for parallel tasks.

Halstead [50] also uses processor-local task stacks, so that when a processor executes a task from its own stack, a last-in-first-out order is used. He chose to have thieves steal work from the top of a task stack so that a thief also executes work in the last-in-first-out order. However,

Halstead remarks that it may be better to steal work from the bottom of a stack. Halstead chose a last-in-first-out execution order for the same reasons that we did: so that when a processor runs its own task, the task is executed in the same order as it would be if the program were running sequentially. We agree with his claim that this reduces resource usage overheads. We also agree with his speculation that it results in more efficient cache usage. However, Halstead also claims that this can improve performance for embarrassingly parallel workloads. This claim is only true insofar as work stealing reduces runtime costs *in general*. No runtime method can completely eliminate the overheads of parallel execution. Furthermore, in our experience, using sparks to represent computations whose execution has not begun has had a more significant improvement in reducing runtime costs.

To preserve the last-in-first-out behaviour of context-local spark stacks, and first-in-first-out behaviour of the global queue, we chose to use double ended queues (deques) for context local spark storage. Since the deque will be used by multiple Mercury engines we must either choose a data structure that is thread safe or use a mutex to protect a non thread safe data structure. The pop and push operations are always made by the same thread,[3] therefore the stealing of work by another context is what creates the need for synchronisation. The deque described by Chase and Lev [28] supports lock free, nonblocking operation, has low overheads (especially for pop and push operations), and is dynamically resizable; all of these qualities are very important to our runtime system's implementation. The deque provides the following operations.

`void MR_push_spark(deque *d, spark *s)` The deque's owner can call this to push an item onto the top[4] or *hot end* of the deque. This can be done with only a *memory barrier* for synchronisation (see below). If necessary, `MR_push_spark()` will grow the array that is used to implement the deque.

`MR_bool MR_pop_spark(deque *d, spark *s)` The deque's owner can call this to pop an item from the top of the queue. In the case that the deque contains only one item, a compare and swap operation is used to determine if the thread lost a race to another thread attempting to steal the item from the deque (a *thief*). When this happens, the single item was stolen by the thief and the owner's call to `MR_pop_spark()` returns false, indicating that the deque was empty. In the common case the compare and swap is not used. In all cases a memory barrier is also used (see below). Internally the deque is stored as an array of sparks, not spark pointers. This is why the second argument, in which the result is returned, is not a double pointer as one might expect. This implementation detail avoids memory allocation for sparks inside the deque implementation. A caller of any of these functions temporarily stores the spark on its program stack.

`MR_ws_result MR_steal_spark(deque *d, spark *s)` A thread other than the deque's owner can steal items from the bottom of the deque. This always uses an atomic compare and swap operation as multiple thieves may call `MR_steal_spark()` on the same deque at the same time. `MR_steal_spark()` can return one of three different values: "success", "failure"

[3] Contexts move between engines by being suspended and resumed, but this always involves the context run queue's lock or other protection. Therefore the context's data structures including the spark deque are protected from concurrent access.

[4] Note when we refer to the top of the deque Chase and Lev [28] refers to the bottom and vice-versa. Most people prefer to imagine that stacks grow upwards, and since the deque is most often used as a stack we consider this end of the deque to be the top. We also call this the *hot end* as it is the busiest end of the deque.

Algorithm 3.3 `MR_push_spark()` with memory barrier

```
void MR_push_spark(MR_SparkDeque *dq, const MR_Spark *spark)
{
    int                     bot;
    int                     top;
    volatile MR_SparkArray  *arr;
    int                     size;

    bot = dq->MR_sd_bottom;
    top = dq->MR_sd_top;
    arr = dq->MR_sd_active_array;
    size = top - bot;

    if (size >= arr->MR_sa_max) {
        /* grow array omitted */
    }

    MR_sa_element(arr, top) = *spark;
    /*
    ** Make sure the spark data is stored before we store the value of
    ** bottom.
    */
    __asm__ __volatile__ ("sfence");
    dq->MR_sd_top = top + 1;
}
```

and "abort". "abort" indicates that the thief lost a race with either the owner (*victim*) or another thief.

Mercury was already using Chase and Lev [28]'s deques for spark storage, most likely because Wang had always planned to implement work stealing. We did not need to replace the data structure used for a context local spark storage, but we will now refer to it as a deque rather than a stack. We have removed the global spark queue, as work stealing does not use one. Consequently, when a context creates a spark, that spark is always placed on the context's local spark deque.

Chase and Lev [28]'s paper uses Java to describe the deque structure and algorithms. Java has a strong memory model compared to C (the implementation language of the Mercury runtime system). C's `volatile` variable storage keyword prevents the compiler from reordering operations on that variable with respect to other code and prevents generated code from caching the value of the variable. CPUs will generally reorder memory operations including executing several operations concurrently with one another and other instructions. Therefore, C's `volatile` keyword is insufficient when the order of memory operations is important. In contrast, Java's `volatile` keyword has the constraints of C's keyword plus it guarantees that memory operations become visible to other processors in the order that they appear in the program. Chase and Lev [28] use Java's `volatile` qualifier in the declaration for the `top` and `bottom` fields of the deque structure. When adding the deque algorithms to Mercury's runtime system, we translated them to C. To maintain correctness we had to introduce two *memory barriers*,[5] CPU instructions which place ordering constraints on memory operations. First, we placed a store barrier in the `MR_push_spark()`

[5]Memory barriers are also known as memory fences. In particular the x86/x86_64 instructions contain the word fence.

Algorithm 3.4 `MR_pop_spark()` with memory barrier

```
volatile MR_Spark* MR_pop_spark(MR_SparkDeque *dq)
{
    int                     bot;
    int                     top;
    int                     size;
    volatile MR_SparkArray  *arr;
    bool                    success;
    volatile MR_Spark       *spark;

    top = dq->MR_sd_top;
    arr = dq->MR_sd_active_array;

    top--;
    dq->MR_sd_top = top;

    /* top must be written before we read bottom. */
    __asm__ __volatile__ ("mfence");

    bot = dq->MR_sd_bottom;
    size = top - bot;

    if (size < 0) {
        dq->MR_sd_top = bot;
        return NULL;
    }

    spark = &MR_sa_element(arr, top);
    if (size > 0) {
        return spark;
    }

    /* size = 0 */
    success = MR_compare_and_swap_int(&dq->MR_sd_bottom, bot, bot + 1);
    dq->MR_sd_top = bot + 1;
    return success ? spark : NULL;
}
```

Algorithm 3.5 `MR_join_and_continue`— initial work stealing version

```
 1  void MR_join_and_continue(MR_SyncTerm *st, MR_Code *cont_label) {
 2      MR_bool     finished, got_spark;
 3      MR_Context  *current_context = MR_ENGINE(MR_eng_current_context);
 4      MR_Context  *orig_context = st->MR_st_orig_context;
 5      MR_Spark    spark;
 6
 7      finished = MR_atomic_dec_and_is_zero(&(st->MR_st_num_outstanding));
 8      if (finished) {
 9          if (orig_context == current_context) {
10              MR_GOTO(cont_label)
11          } else {
12              while (orig_context->MR_ctxt_resume != cont_label) {
13                  CPU_spin_loop_hint();
14              }
15              MR_schedule_context(orig_context);
16              MR_GOTO{MR_idle};
17          }
18      } else {
19          got_spark = MR_pop_spark(current_context->MR_ctxt_spark_deque, &spark);
20          if (got_spark) {
21              MR_GOTO(spark.MR_spark_code);
22          } else {
23              if (orig_context == current_context) {
24                   MR_save_context(current_context);
25                   current_context->MR_ctxt_resume = cont_label;
26                   MR_ENGINE(MR_eng_current_context) = NULL;
27              }
28              MR_GOTO(MR_idle);
29          }
30      }
31  }
```

procedure (the inline assembly[6] in Algorithm 3.3). This barrier ensures that the spark is written into the array before the pointer to the top of the array is updated, indicating that the spark is available. Second, we placed a full barrier in the `MR_pop_spark()` procedure (the inline assembly in Algorithm 3.4). This barrier requires that the procedure updates the pointer to the top of the deque before it reads the pointer to the bottom (which it uses to calculate the size). This barrier prevents a race between an owner popping a spark and two thieves stealing sparks. It ensures that each of the threads sees the correct number of items on the deque and therefore executes correctly. Both these barriers have a significant cost, in particular the full barrier in `MR_pop_spark()`. This means that the `MR_push_spark()` and `MR_pop_spark()` functions are not as efficient as they appear to be. The deque algorithms are still efficient enough; we have not needed to improve them further. For more information about C's weak memory ordering see Boehm [15]. See also Adve and Boehm [2] which discusses the Java memory model and the new C++0x[7] memory model.

A context accesses its own local spark queue in the `MR_join_and_continue` barrier introduced in Section 3.2. The introduction of work stealing has allowed us to optimise `MR_join_and_continue`. The new version of `MR_join_and_continue` is shown in Algorithm 3.5, and the previous version is

[6] In our implementation a C macro is used to abstract away the assembly code for different architectures; we have shown our figures without the macro to aid the reader.

[7] C++0x is also known as C++11

shown in Algorithm 3.1 on page 52. The first change to this algorithm is that this version is lock free. All the synchronisation is performed by atomic CPU instructions, memory write ordering and one spin loop. The number of outstanding contexts in the synchronisation term is decremented and the result is checked for zero atomically.[8] This optimisation could have been made without introducing work stealing, but it was convenient to make both changes at the same time. Note that in the previous version, a single lock was shared between all of the synchronisation terms in the system.

The next change prevents a race condition that would otherwise be possible without locking, which could occur as follows. A conjunction of two conjuncts is executed in parallel. The original context, C_{Orig}, enters the barrier, decrements the counter from two to one, and because there is another outstanding conjunct, it executes the else branch on lines 19–29. At almost the same time another context, C_{Other}, enters the barrier, decrements the counter and finds that there is no more outstanding work. C_{Other} attempts to schedule C_{Orig} on line 15. However attempting to schedule C_{Orig} before it has finished suspending would cause an inconsistent state and memory corruption. Therefore lines 12–14 wait until C_{Orig} has been suspended. The engine that is executing C_{Orig} first suspends C_{Orig} and then indicates that it has been suspended by setting C_{Orig}'s resume label (lines 24–25). The spin loop on lines 12–14 includes a use of a macro named `CPU_spin_loop_hint`, This resolves to the `pause` instruction on x86 and x86_64, which instructs the CPU to pipeline the loop differently in order to reduce memory traffic and allow the loop to exit without a pipeline stall [62]. Lines 24–25 also include memory write ordering (not shown).

The other change is around line 20: when the context pops a spark off its stack, it does not check if the spark was created by a callee's parallel conjunction. This check can be avoided because all sparks are placed on the context local spark deques and thieves never steal work from the hot end of the deque. Hence if there is an outstanding conjunct then either its spark will be at the hot end of the deque or the deque will be empty and the spark will have been stolen. Therefore any spark at the hot end of the deque cannot possibly be created by a callee's parallel conjunction.

We have also modified `MR_idle` to use spark stealing; its new code is shown in Algorithm 3.6. We have replaced the old code which dequeued a spark from the global spark queue with a call to `MR_try_steal_spark()` (see below). Before attempting to steal a spark, `MR_idle` must ensure that the context limit will not be exceeded (this was previously done when choosing where to place a new spark). This check is on lines 28–29; it is true if either the engine already has a context (which is guaranteed to be available for a new spark), or the current number of contexts is lower than the limit. When sparks are placed on a context's local deque, the run queue's condition variable is not signalled, as doing so would be wasteful. Therefore engines must wake up periodically to check if there is any parallel work available. This is done on line 48 using a call to `MR_timed_wait` with a configurable timeout (it defaults to 2ms). We discuss this timeout later in Section 3.6.

Engines that attempt to steal a spark must have access to all the spark deques. We do this with a global array of pointers to contexts' deques. When a context is created, the runtime system attempts to add the context's deque to this array by finding an empty slot, one containing a null pointer, and writing the pointer to the context's deque to that index in the array. If there is no unused slot in this array, the runtime system will resize the array. When the runtime system destroys a context, it writes `NULL` to that context's index in the deque array. To prevent concurrent access from corrupting the array, these operations are protected by `MR_spark_deques_lock`.

[8] On x86/x86_64 this is a `lock dec` instruction. We read the zero flag to determine if the decrement caused the value to become zero.

Algorithm 3.6 `MR_idle`— initial work stealing version

```
 1  void MR_idle() {
 2      MR_Context  *ctxt;
 3      MR_Context  *current_context = MR_ENGINE(MR_eng_current_context);
 4      MR_Code     *resume;
 5      MR_Spark    spark;
 6
 7      MR_acquire_lock(&MR_runqueue_lock);
 8      while(MR_True) {
 9          if (MR_exit_now) {
10              MR_release_lock(MR_runqueue_lock);
11              MR_destroy_thread();                        // does not return.
12          }
13
14          // Try to run a context
15          ctxt = MR_get_runnable_context();
16          if (ctxt != NULL) {
17              MR_release_lock(&MR_runqueue_lock);
18              if (current_context != NULL) {
19                  MR_release_context(current_context);
20              }
21              MR_load_context(ctxt);
22              resume = ctxt->MR_ctxt_resume;
23              ctxt->MR_ctxt_resume = NULL;
24              MR_GOTO(resume);
25          }
26
27          // Try to run a spark.
28          if ((current_context != NULL) ||
29                  (MR_num_outstanding_contexts < MR_max_contexts))
30          {
31              if (MR_try_steal_spark(&spark)) {
32                  MR_release_lock(&MR_runqueue_lock);
33                  if (current_context == NULL) {
34                      ctxt = MR_get_free_context();
35                      if (ctxt == NULL) {
36                          ctxt = MR_create_context();
37                      }
38                      MR_load_context(ctxt);
39                      current_context = ctxt;
40                  }
41                  MR_parent_sp = spark.MR_spark_parent_sp;
42                  current_context->MR_ctxt_thread_local_mutables =
43                      spark.MR_spark_thread_local_mutables;
44                  MR_GOTO(spark.MR_spark_code);
45              }
46          }
47
48          MR_timed_wait(&MR_runqueue_cond, &MR_runqueue_lock);
49      }
50  }
```

Algorithm 3.7 `MR_try_steal_spark()`— initial work stealing version

```
 1  MR_bool MR_try_steal_spark(MR_Spark *spark) {
 2      int         max_attempts;
 3      MR_Deque    *deque;
 4
 5      MR_acquire_lock(&MR_spark_deques_lock);
 6      max_attempts = MR_MIN(MR_max_spark_deques, MR_worksteal_max_attempts);
 7      for (int attempt = 0; attempts < max_attempts; attempt++) {
 8          MR_victim_counter++;
 9          deque = MR_spark_deques[MR_victim_counter % MR_max_spark_deques];
10          if (deque != NULL) {
11              if (MR_steal_spark(deque, spark));
12                  MR_release_lock(&MR_spark_deques_lock);
13                  return MR_true;
14              }
15          }
16      }
17      MR_release_lock(&MR_spark_deques_lock);
18      return MR_false;
19  }
```

The algorithm for `MR_try_steal_spark()` is shown in Algorithm 3.7. It must also acquire the lock before using the array. A thief may need to make several attempts before it successfully finds a deque with work it can steal. Line 6 sets the number of attempts to make, which is either the user configurable `MR_worksteal_max_attempts` or the size of the array, whichever is smaller. A loop (beginning on line 7) attempts to steal work until it succeeds or it has made `max_attempts`. We use a global variable, `MR_victim_counter`, to implement round-robin selection of the victim. On line 11 we attempt to steal work from a deque, provided that its deque pointer in the array is non-null. If the call to `MR_steal_spark()` succeeded, it will have written spark data into the memory pointed to by `spark`, then `MR_try_steal_spark()` releases the lock and returns true. Eventually `MR_try_steal_spark()` may give up (lines 17–18). If it does so it will release the lock and return false. Whether or not a thief steals work or gives up, the next thief will resume the round-robin selection where the previous thief left off. This is guaranteed because `MR_victim_counter` is protected by the lock acquired on line 5.

All sparks created by a context are placed on its local spark deque. Sparks are only removed to be executed in parallel when an idle engine executes `MR_try_steal_spark()`. Therefore the decision to execute a spark in parallel is only made once an engine is idle and able to run the spark. We expect this to correct the premature scheduling decision problem we described in Section 3.3.

We benchmarked our initial work stealing implementation with the mandelbrot program from previous sections. Table 3.7 shows the results of our benchmarks. The first and third row groups are the same results as the first and second row groups from Table 3.6 respectively. They are included to allow for easy comparison. The second and fourth row groups were generated with a newer version of Mercury that implements work stealing as described above.[9] The figures in parenthesis are speedup figures.

The first conclusion that we can draw from the results is that left recursion with work stealing

[9]The version of Mercury used is slightly modified, due to an oversight we had forgotten to include the context limit condition in `MR_idle`. The version we benchmarked includes this limit and matches the algorithms shown in this section.

Max no. of contexts	Sequmential not TS	Sequmential TS	Parallel w/ N Engines 1	2	3	4
Prior right recursion results						
4	23.2 (0.93)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
64	-	-	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
128	-	-	21.5 (1.00)	19.8 (1.09)	20.9 (1.03)	21.2 (1.01)
256	-	-	21.5 (1.00)	13.2 (1.63)	15.5 (1.38)	16.5 (1.30)
512	-	-	21.5 (1.00)	11.9 (1.81)	8.1 (2.66)	6.1 (3.55)
1024	-	-	21.5 (1.00)	11.8 (1.81)	8.0 (2.67)	6.1 (3.55)
2048	-	-	21.5 (1.00)	11.9 (1.81)	8.0 (2.67)	6.0 (3.55)
New right recursion results with work stealing						
4	23.2 (0.93)	21.6 (1.00)	21.5 (1.01)	21.7 (1.00)	21.7 (1.00)	21.5 (1.01)
64	-	-	21.7 (1.00)	21.5 (1.01)	21.5 (1.01)	21.5 (1.01)
128	-	-	21.7 (1.00)	21.6 (1.00)	21.1 (1.03)	21.2 (1.02)
256	-	-	21.7 (1.00)	19.5 (1.11)	18.1 (1.20)	18.0 (1.20)
512	-	-	21.5 (1.01)	12.9 (1.67)	9.0 (2.41)	7.9 (2.73)
1024	-	-	21.5 (1.01)	10.8 (2.00)	7.3 (2.96)	5.6 (3.87)
2048	-	-	21.5 (1.01)	10.8 (2.00)	7.3 (2.95)	5.7 (3.81)
Prior left recursion results						
4	23.2 (0.93)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
64	-	-	21.5 (1.00)	21.5 (1.00)	21.4 (1.00)	21.5 (1.00)
128	-	-	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)	21.5 (1.00)
256	-	-	21.5 (1.00)	18.3 (1.17)	18.2 (1.18)	19.6 (1.09)
512	-	-	21.5 (1.00)	17.9 (1.20)	15.5 (1.39)	16.4 (1.31)
1024	-	-	21.5 (1.00)	18.0 (1.19)	14.7 (1.46)	16.1 (1.33)
2048	-	-	21.5 (1.00)	18.0 (1.19)	15.4 (1.40)	17.8 (1.21)
New left recursion results with work stealing						
4	23.2 (0.93)	21.5 (1.00)	21.5 (1.00)	10.8 (1.99)	7.3 (2.96)	5.4 (3.95)
8	-	-	21.7 (0.99)	10.8 (1.99)	7.3 (2.94)	5.5 (3.92)
16	-	-	21.6 (0.99)	10.8 (1.99)	7.2 (2.98)	5.5 (3.92)
32	-	-	21.5 (1.00)	10.8 (1.99)	7.2 (2.98)	5.5 (3.92)

Table 3.7: Work stealing results — initial implementation

is much faster than left recursion without work stealing. We can see that work stealing has solved the premature spark scheduling problem for mandelbrot. Given that left recursive mandelbrot represented a pathological case of this problem, we are confident that the problem has generally been solved.

We can also see that the context limit no longer affects performance with left recursion. There is no significant difference between the results for the various settings of the context limit.

When we consider the case for right recursion we can see that work stealing has improved performance when there is a high context limit. This is somewhat expected since work stealing is known as a very efficient way of managing parallel work on a shared memory system. However we did not expect such a significant improvement in performance. In the non work stealing system, a spark is placed on the global spark queue if both the context limit has not been reached, and there is an idle engine. When a spark is being created there may not be an idle engine, however an engine may become idle soon and wish to execute a spark. With work stealing, all sparks are placed on local deques and an idle engine will attempt to steal work when it is ready — delaying the decision to run the spark in parallel. This suggests that the premature scheduling problem was also affecting right recursion. But the effect was minor compared to the pathological case we saw with left recursion.

There is a second observation we can make about right recursion. In the cases for 256 or 512 contexts work stealing performs worse than without work stealing. We believe that the original results were faster because more work was done in parallel. When a context spawned off a spark, it would often find that there was no free engine, and place the spark on its local queue. After completing the first conjunct of the parallel conjunction it would execute `MR_join_and_continue`. `MR_join_and_continue` would find that the context had a spark on its local stack and would execute this spark directly. When a spark is executed directly the existing context is re-used and so the context limit does not increase as quickly. The work stealing version does not do this, it always places the spark on its local stack and almost always another engine will steal that spark, and the creation of a new context will bring the system closer to the context limit. The work stealing program is more likely to reach the context limit earlier, where it will be forced into sequential execution.

3.5 Independent conjunction reordering

We have shown that right recursion suffers from a context limit problem. Right recursive programs also get smaller speedups when compared with left recursive programs. The problem with right recursion is the large number of contexts needed. This is memory intensive, and CPU intensive once we consider that the Boehm GC must scan these context's stacks to check for pointers. This can happen whenever the second or later conjunct of a parallel conjunction takes longer to execute than the first conjunct; the first conjunct becomes blocked on the later conjunct(s).

On the other hand left recursion works well: using work stealing we are able to get very good speedups (3.92 for mandelbrot when using four Mercury engines). When the conjunction is independent, Mercury's purity allows us to reorder the conjuncts of the computation and transform right recursion into left recursion. For example it would transform the procedure in Figure 3.2(a) into the one in Figure 3.2(b) (page 56).

We only reorder completely independent conjunctions. It may be possible to find independent sections of dependent conjunctions and reorder them, but we have not needed to. `reorder()` is

Algorithm 3.8 Reorder independent conjunctions

```
 1: procedure REORDER(SCC, Conjs)
 2:     if Conjs = [] then
 3:         return []
 4:     else
 5:         [Conj | ConjsTail] ← Conjs
 6:         ConjsTail ← reorder(SCC, ConjsTail)
 7:         if Conj contains a call to SCC then
 8:             return [Conj | ConjsTail]
 9:         else
10:             return try_push_conj_later(Conj, ConjsTail)
11: procedure TRY_PUSH_CONJ_LATER(Goal, Conjs)
12:     if Conjs = [] then
13:         return [Goal]
14:     else
15:         [Pivot | Rest] ← Conjs
16:         if can_swap(Goal, Pivot) then
17:             Pushed ← try_push_conj_later(Goal, Rest)
18:             return [Pivot | Pushed]
19:         else
20:             return [Goal | Conjs]
```

invoked on parallel conjunctions without shared variables; its code is shown in Algorithm 3.8. Its arguments are a reference to the current strongly connected component (SCC) and a list of the parallel conjunction's conjuncts. `reorder()` iterates over these goals in reverse order by means of the recursive call on line 6. For each goal it tests if that goal makes a call to the current SCC. If so, it leaves the goal where it is, if not `reorder()` will attempt to push the goal towards the end of the list using `try_push_conj_later()`. `try_push_conj_later()` takes the current goal (`Goal`) and the list representing the rest of the conjunction (`Conjs`). It iterates down this list attempting to swap `Goal` with the goal at the front of the list (`Pivot`). Not all swaps are legal, for example a goal cannot be swapped with an impure goal, therefore `try_push_conj_later()` may not always push a goal all the way to the end of the list.

We have benchmarked and tested this to confirm that independent right recursion now performs the same as independent left recursion. We expected these programs to perform identically because independent right recursion is transformed into independent left recursion.

3.6 Improved work stealing implementation

In this section we attempt to improve on our work stealing implementation from Section 3.4. While we made these improvements, we also found it useful to change how idle engines behave. Although these two changes are conceptually distinct, they were made together and their implementations are interlinked. Therefore we will present and benchmark them together as one set of changes. While our changes are an improvement in terms of design, they do not always yield an improvement in performance.

Sparks were originally stored on context-local deques. This has the following significant problems.

There is a dynamic number of spark deques. The number of contexts changes during a program's execution, therefore the number of spark deques also changes. This means that we

must have code to manage this changing number of deques. This code makes the runtime system more complicated than necessary, both when stealing a spark and when creating or destroying a context.

Locking is required. The management of contexts must be thread safe so that the set of spark deques is not corrupted. We store spark deques in a global array protected by a lock. It may be possible to replace the array with a lock free data structure, however it is better to remove the need for thread safety by using a constant set of deques.

A large number of contexts makes work stealing slower. In prior sections we have shown that the number of contexts in use can often be very high, much higher than the number of Mercury engines. If there are N engines and M contexts, then there can be at most N contexts running and at least $M - N$ contexts suspended (blocked and waiting to run). A context can become suspended in one of two ways: by blocking on a future's value in a call to `future_wait/2`, or by blocking on an incomplete conjunct in a call to `MR_join_and_continue`. We attempt to minimise the former case (see Chapter 4) and the latter case cannot occur if the context has a spark on its local spark deque (it would run the spark rather than block). Therefore the large majority of the suspended contexts will not have sparks on their deques, and the probability of selecting a deque at random with a spark on its stack is low. Furthermore the value of M can be very high in pathological cases. If an engine does not successfully steal a spark from a deque, it will continue by trying to steal a spark from a different deque. An engine can exhaust all its attempts, even when there is work available: a spark may be placed on a deque after the engine has already attempted to steal from it. Upon exhausting all its attempts or all the deques, the engine will sleep before making another round of attempts (see `MR_try_steal_spark()` in Algorithm 3.7 on page 69). Each round of attempts (regardless of success or failure) has a complexity of $O(M)$.

The approach we have taken to solving these problems is based on associating spark deques with engines rather than with contexts. A running Mercury program has a constant number of engines, and therefore the number of spark deques will not vary. This allows us to remove the code used to resize the deque array. This makes context creation and destruction much simpler. It also removes the need for the lock protecting the global array of spark deques. We can also remove the locking code in `MR_try_steal_spark()` (whose original version was shown in Algorithm 3.7) which was used to ensure that the array is not changed while a thief is trying to steal a spark. The cost of work stealing also becomes linear in the number of engines rather than the number of contexts.

We have made some other changes to `MR_try_steal_spark()`, whose new code is shown in Algorithm 3.9. Previously `MR_try_steal_spark()` required its caller to check that stealing a spark would not exceed the context limit; this check has been moved into `MR_try_steal_spark()` (lines 6–7). Another change is that `MR_try_steal_spark()` will now call `MR_prepare_engine_for_spark()` (line 12) which will load a context into the engine if it does not already have one (Algorithm 3.10). Finally `MR_try_steal_spark()` will return the address of the spark's code; the caller will jump to this address. If `MR_try_steal_spark()` could not find a spark it will return NULL, in this case its caller will look for other work.

In the previous version, the lock was also used to protect the victim counter; it ensured that round-robin selection of the victim was maintained. Now each engine independently performs its own round-robin selection of the victim using a new field `MR_victim_counter` in the engine

Algorithm 3.9 `MR_try_steal_spark()`— revised work stealing version

```
 1  MR_Code* MR_try_steal_spark(void) {
 2      int         victim_incex;
 3      MR_Deque    *deque;
 4      MR_Spark    spark
 5
 6      if ((MR_ENGINE(MR_eng_current_context) != NULL) ||
 7              (MR_num_outstanding_contexts < MR_max_contexts))
 8      {
 9          for (int attempt = 0; attempts < MR_num_engines; attempt++) {
10              victim_index = (MR_ENGINE(MR_eng_victim_counter) + attempt)
11                  % MR_num_engines;
12              if (victim_index == MR_ENGINE(MR_eng_id)) {
13                  continue;
14              }
15              if(MR_steal_spark(MR_spark_deques[victim_index], &spark)) {
16                  MR_ENGINE(MR_eng_victim_counter) = victim_index;
17                  MR_prepare_engine_for_spark(&spark)
18                  return spark.MR_spark_code;
19              }
20          }
21          MR_ENGINE(MR_eng_victim_counter) = victim_index;
22          return NULL;
23  }
```

Algorithm 3.10 `MR_prepare_engine_for_spark()`

```
void MR_prepare_engine_for_spark(MR_Spark *spark) {
    MR_Context *ctxt;

    ctxt = MR_ENGINE(MR_eng_current_context);
    if (ctxt == NULL) {
        ctxt = MR_get_free_context();
        if (ctxt == NULL) {
            ctxt = MR_create_context();
        }
        MR_load_context(ctxt);
    }
    MR_parent_sp = spark->MR_spark_parent_sp;
    ctxt->MR_ctxt_thread_local_mutables =
        spark->MR_spark_thread_local_mutables;
}
```

```
nested_par(...) ;-
    (
        p(..., X),
    &
        (
            q(X, ...)
        &
            r(..., Y)
        ),
        s(...)
    &
        t(Y, ...)
    ).
```

Figure 3.5: Nested dependent parallelism.

structure. If the engine finds a spark, `MR_try_steal_spark()` will save the current victim index to this field, as this deque may contain more sparks which the engine can try to steal later. We have not evaluated whether this policy is an improvement. However when compared with the improvement due to removing the lock and potentially large number of spark deques, any difference in performance due to the selection of victims will be negligible. The two other changes are minor: we have removed the configurable limit of the number of work stealing attempts per round and also the test for null array slots; both are now unnecessary.

We have also made changes to the code that idle engines execute. An idle engine with a context that is free (it can be used for any spark) or without a context will call `MR_idle` to acquire new work. `MR_idle` has changed significantly; the new version is shown in Algorithm 3.11. This algorithm uses a structure pointed to by `eng_data` to receive notifications; we will discuss this structure and how it is used to send and receive notifications later in this section. For now we will discuss how `MR_idle` looks for work in lieu of any notification.

`MR_idle` tries to find a runnable context before trying to find a local spark (lines 12–19). A context that has already been created usually represents work that is *to the left of* any work that is still a spark (Section 3.4). Executing G_1 of a parallel conjunction G_1 & G_2 can signal futures that G_2 may otherwise block on. Executing G_1 may also create sparks leading to more parallelism. Once a context is resumed and its execution finishes, then its memory can be used to execute a spark from the engine's local spark deque. Checking for runnable contexts first also helps to avoid the creation of new contexts before existing contexts' computations are finished. This partially avoids deadlocks that can occur when the context limit is exceeded: executing a waiting context rather than attempting to create a new context for a spark does not increase the number of contexts that exist.

`MR_idle` tries to run a context by attempting to take a context from the context run queue while holding the context run queue lock (lines 12–14). If it finds a context, it will execute `MR_prepare_engine_for_context()` (Algorithm 3.12), which checks for and unloads the engine's current context, and then loads the new context. It also clears the context's resume field and returns the field's previous value so that `MR_idle` can jump to the resume address. The context's resume field must be cleared here as `MR_join_and_continue` may use it later for synchronisation. Previously the run queue's lock protected all of `MR_idle`; this critical section has been made smaller, which is probably more efficient.

Algorithm 3.11 `MR_idle`— improved work stealing version

```
 1  void MR_idle(void) {
 2      int                 engine_id = MR_ENGINE(MR_eng_id);
 3      MR_EngineSleepSync  *eng_data = MR_engine_sleep_data[engine_id];
 4      MR_Context          *ctxt;
 5      MR_Context          *current_context = MR_ENGINE(MR_eng_current_context);
 6      MR_Code             *resume;
 7      MR_Spark            spark;
 8
 9      eng_data->MR_es_state = MR_LOOKING_FOR_WORK;
10
11      // Try to find a runnable context.
12      MR_acquire_lock(&MR_runqueue_lock);
13      ctxt = MR_get_runnable_context();
14      MR_release_lock(&MR_runnable_lock);
15      if (ctxt != NULL) {
16          eng_data->MR_es_state = MR_WORKING;
17          resume = MR_prepare_engine_for_context(ctxt);
18          MR_GOTO(resume);
19      }
20
21      // Try to run a spark from our local spark deque.
22      if (MR_pop_spark(MR_ENGINE(MR_eng_spark_deque), &spark)) {
23          eng_data->MR_es_state = MR_WORKING;
24          MR_prepare_engine_for_spark(&spark);
25          MR_GOTO(spark->resume);
26      }
27
28      if (!(MR_compare_and_swap(&(eng_data->MR_es_state),
29              MR_LOOKING_FOR_WORK, MR_STEALING)))
30      {
31          // Handle the notification that we have received.
32      }
33      // Try to steal a spark
34      resume = MR_try_steal_spark();
35      if (resume != NULL) {
36          eng_data->MR_es_state = MR_WORKING;
37          MR_GOTO(resume);
38      }
39      MR_GOTO(MR_sleep);
40  }
```

Algorithm 3.12 `MR_prepare_engine_for_context()`

```
MR_Code* MR_prepare_engine_for_context(MR_Context *ctxt) {
    MR_Code *resume;

    if (MR_ENGINE(MR_eng_current_context) != NULL) {
        MR_release_context(MR_ENGINE(MR_eng_current_context));
    }
    MR_load_context(ctxt);
    resume = ctxt->MR_ctxt_resume;
    ctxt->MR_ctxt_resume = NULL;
    return resume;
}
```

If `MR_idle` cannot find a context to resume then it will attempt to run a spark from its local spark deque (lines 22–26). If there is a spark to execute and the engine does not have a context, a new one needs to be created; this is handled by `MR_prepare_engine_for_spark()` above. We considered creating a context and executing the spark only if the context limit had not been reached, however this can create a problem, which we explain using an example. Consider Figure 3.5, which shows two nested parallel conjunctions and two futures that create dependencies between the parallel computations. `X` creates a dependency between `p` and `q`, and `Y` creates a dependency between `r` and `t`. When an engine, called E1, executes this procedure, it pushes the spark for the second and third conjuncts of the outer conjunction onto its local spark deque. E1 then begins the execution of `p`. A second engine, called E2, steals the spark from E1's spark deque and immediately creates two new sparks; the first is for `t` and the second for `r`. It pushes the sparks onto its deque, therefore the spark for `t` is on the bottom (cold end). E2 then begins the execution of `q`. A third engine, E3, wakes up and steals the spark for `t` from E2. Next, E2 attempts to read the value of `X` in `q`, but it has not yet been produced. Therefore E2 suspends its context and calls `MR_idle`, which cannot find a context to execute and therefore checks for and finds the local spark for `r`. E2 needs a new context to execute `r`; if the context limit is exceeded then E2 cannot proceed. Without E2's execution of `r` and production of `Y`, E3 cannot complete the execution of `t` and free up its context. The system will not deadlock, as the execution of `p` by E1 can still continue, and will eventually allow the execution of `q` to continue. However, we still wish to avoid such situations, therefore we do not check the context limit before running local contexts. Without this check, E2 is able to create a context and execute `r`. This can never create more than a handful of extra contexts: while the limit is exceeded, engines will not steal work and will therefore execute sparks in a left-to-right order; dependencies and barriers cannot cause more contexts to become blocked.

The two choices we have made, executing local sparks without checking the context limit and attempting to resume contexts before attempting to execute sparks, work together to prevent deadlocks that could be caused by the context limit and dependencies. The context limit is a work-around to prevent some kinds of worst case behaviour and should not normally affect other algorithmic choices. These algorithmic choices are valid and desirable, even if we did not need or use the context limit, as they keep the number of contexts low. This is good for performance as contexts consume significant amounts of memory and the garbage collector spends time scanning that memory.

If an engine executing `MR_idle` cannot find a local spark then it will check for any notifications (lines 28–32). If it has not been notified then it will try to steal a spark from another engine using `MR_try_steal_spark()` (above) (lines 34–38). `MR_try_steal_spark()` is a C function, allowing us to call it from multiple places in the runtime system. It returns the address of the spark's entry point to its caller, which jumps to the entry point. In this case the caller is `MR_idle`, which is C code that uses the Mercury calling convention. This convention passes all its arguments and results in the Mercury abstract machine registers, including the return address. Additionally `MR_idle` does not call any other Mercury procedures. Therefore, it does not need to create a frame on any of the C or Mercury stacks: any local C variables are stored on the C stack frame shared between all Mercury procedures (and must be saved by the caller). `MR_idle` has another difference with other Mercury procedures: it never returns control to its caller. Instead it continues execution by jumping to the code address of the next instruction to execute, such as the resume address of a context, or the entry point of a spark. If its callees, `MR_try_steal_spark()` or `MR_prepare_engine_for_context()` were to make this jump rather than returning and then

```
struct MR_engine_sleep_sync {
    volatile unsigned                   MR_es_state;
    volatile unsigned                   MR_es_action;
    union MR_engine_wake_action_data    MR_es_action_data;
    sem_t                               MR_es_sleep_sem;
    lock                                MR_es_lock;
};

union MR_engine_wake_action_data {
    MR_EngineId     MR_ewa_worksteal_engine;
    MR_Context      *MR_ewa_context;
};
```

Figure 3.6: `MR_engine_sleep_sync` structure

letting `MR_idle` jump, then this would leak the stack frame created for the C call. Therefore, `MR_try_steal_spark()` and `MR_prepare_engine_for_context()` must return the address of the next instruction to execute and allow `MR_idle` to jump to this code or jump to `MR_sleep`.

. A sleeping engine may need to wake up occasionally and handle new parallel work. There are two ways to do this:

polling: engines can wake up occasionally and *poll* for new work.

notification: engines can sleep indefinitely and be woken up by some other engine when work becomes available.

Each of these options has its own drawbacks and benefits. Polling is very simple, but it has a number of drawbacks. An engine that wakes up will often find that there is no parallel work, and it will have to go to sleep again. In first impression this uses only a very small amount of CPU time which seems harmless, however many idle processes on a system behaving in this way can add up to a significant amount of CPU time. The other problem with polling is that a parallel task will not be executed immediately as an engine's polling timer must first expire. These two problems trade off against one another: the more frequently an engine polls for work the quicker it can react but the more CPU time it will consume.

Notification has neither of these problems, and has some additional benefits. An engine making a notification performs a deliberate action. We take advantage of this by attaching additional information to the notification. We also send the notification to a particular engine, allowing us to control which of several sleeping engines gets woken up and begins the parallel work. The major drawback of a notification system is that every time parallel work is created a notification may need to be made, and this adds to the overheads of creating parallel work.

The previous version of the runtime system used polling for work stealing and notification for contexts. Engines waiting for a POSIX condition variable to be notified about free contexts would time out every two milliseconds and attempt to steal work. The system did not support selecting which engine would be woken when a context became runnable. The decision between polling and notification can be hard to make, and it may depend on the application and its work load. We have chosen to support both methods in our new work stealing code; notification is used by default when a spark is created, but the user may elect to use polling. Notification is always used when a context becomes runnable.

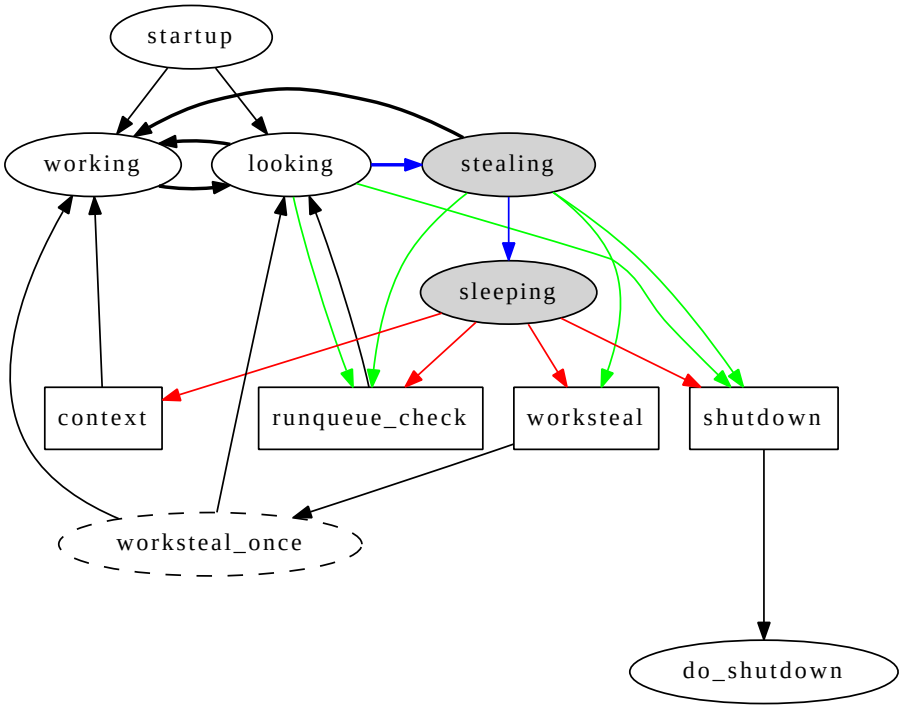


Figure 3.7: Engine states and transitions

Notifications are implemented using the `MR_engine_sleep_sync` structure shown in Figure 3.6. A global array allows engines to access one another's `MR_engine_sleep_sync` structures. Each structure contains the following fields: `MR_es_state` represents the engine's current state, and is used by both the owner and other engines to read and write the state of the engine; `MR_es_action` and `MR_es_action_data` are a closure describing the next action that the engine should perform; `MR_es_sleep_sem` is a semaphore that the owner will wait on in order to sleep and other engines will signal if they want the engine to wake up; and `MR_es_lock` is used to prevent multiple engines trying to wake the same engine at once. The `MR_idle` code above manipulates the state field in its structure, which is pointed to by `eng_data`.

`MR_WORKING`: The engine has work to do and is doing it.

`MR_LOOKING_FOR_WORK`: The engine is looking for work in the form of a runnable context or local spark. It may have work to do in the form of a local spark.

`MR_STEALING`: The engine is currently trying to steal work.

`MR_SLEEPING`: The engine is sleeping; it could not find any work and is now waiting for a notification.

`MR_BUSY`: The engine state is busy: this is a special value used for synchronisation.

`MR_NOTIFIED`: The engine has been notified of some work to do but has not started the work yet.

The states are shown in Figure 3.7 as oval-shaped nodes in the graph except for `MR_NOTIFIED`, which is a single value that represents all the notification types, depicted as rectangular nodes in the graph. The oval node with a dashed outline is illustrative only, it exists in the graph to make it

easier to visualise how work stealing may fail. An engine can move itself between any of the states connected by a black or blue edge: blue denotes a transition that is done with a compare and swap on the MR_es_state field of the MR_engine_sleep_sync structure, whilst other transitions are made with an assignment. The parallel runtime system is under the most load when there are a large number of sparks that represent small computations. When this occurs, engines spend most of their execution time in the MR_WORKING, MR_LOOKING_FOR_WORK and MR_STEALING states, or transitioning between them. Therefore these transitions are *busier* and their edges in the graph are drawn with thicker lines.

When an engine creates a spark or makes a context runnable it can notify a recipient using the recipient's MR_engine_sleep_sync structure. The sender will update the action closure and MR_es_state fields. This is modelled as a state transition of the recipient; such transitions are shown with red and green edges in the graph. Red edges denote transitions from the sleeping state: the sender acquires the MR_es_lock in the MR_engine_sleep_sync structure while updating the structure's contents and signalling the sleep semaphore. Green edges denote transitions from other states: the sender uses a compare and swap to write the MR_BUSY value to the MR_es_state field, then updates the action closure, and finally writes MR_NOTIFIED to the MR_es_state field. A sender using this latter method must not use the semaphore. Using compare and swap with the state field allows the recipient to also use compare and swap, and in some cases to simply write to the field, when changing states. This allows us to make the busy transitions more efficient. These are the transitions from MR_LOOKING_FOR_WORK to MR_STEALING and MR_WORKING. Note that in the diagram all but one of the transitions from MR_LOOKING_FOR_WORK and MR_STEALING are protected, the exceptions being transitions to MR_WORKING, which is acceptable because we allow some notifications to be ignored if the recipient has found some other work. The other transition and state not listed above occurs when the system shuts down; this is another exception as an engine could not possibly find work if the runtime were shutting down[10].

There are several different actions that a sender can specify when notifying another engine.

MR_ACTION_CONTEXT: A context has become runnable and is pointed to by the MR_es_action_data field. We attach the context to the message so that we do not have to place it in the context run queue and then retrieve it again when we know that this engine is capable of executing the context. However this notification cannot be ignored as that would cause the context to be leaked. Leaking a context means dropping or losing the context; a context that must be executed may not be leaked as the computation it represents would get lost. Therefore we only ever send MR_ACTION_CONTEXT messages to engines in the MR_SLEEPING state, the only state that does not have an owner-initiated transition from it.

MR_ACTION_RUNQUEUE_CHECK: A context has become runnable and was placed on the context run queue. Because we cannot use the MR_ACTION_CONTEXT message with non-sleeping engines, we have created this message to be used with non-sleeping engines. When a context can only be executed by a single engine, because that engine has a frame on its C stack that the context needs, then a deadlock would occur if the context were placed in the context run queue after the engine had checked the run queue, but before the engine had gone to sleep (and allowed the message above). By adding this message we can notify the engine that it should check the run queue before going to sleep.

[10] Mercury supports exceptions and it is conceivable that an exception may make the system abort.

`MR_ACTION_WORKSTEAL`: A spark has been placed on the spark deque indicated by the `MR_es_action_data` field. This message is used to notify an engine of a new spark that it might be able to steal and execute. Informing the thief which spark deque it should check is an optimisation. Before this message is sent, the sender will check that the recipient would not exceed the context limit by stealing and executing the spark.

`MR_ACTION_SHUTDOWN`: The runtime system is shutting down, this engine should exit.

The code for `MR_idle` in Algorithm 3.11 shows several of these transitions, including the transition to the `MR_STEALING` state on lines 28–29. If this compare and swap fails, it indicates that the current engine should wait for `MR_es_state` to become a value other than `MR_BUSY` and then, depending on the action closure, either re-check the run queue or exit. We have not shown the code that interprets and acts on the action closure, however it is similar to the switch statement in `MR_sleep` (Algorithm 3.13).

When an engine creates a spark, it attempts to notify other engines of the new spark. However, looking for and notifying an idle engine on every spark creation would be prohibitively expensive. Therefore we maintain a counter of the number of engines in an *idle* state. When this counter is zero, an engine creating a spark will not attempt to notify any other engine. Idle states are `MR_STEALING` and `MR_SLEEPING` and are shown in Figure 3.7 as shaded ovals. Only engines in these states are sent the `MR_ACTION_WORKSTEAL` message. We considered including the `MR_LOOKING_FOR_WORK` state as an idle state and allowing engines in this state to receive the `MR_ACTION_WORKSTEAL` message, However we think that most engines that execute `MR_idle` will find a runnable context or a spark from their own deque and therefore trying to steal work would be counter productive (and such a message would often be ignored). Sending such a message would be *too* speculative (and is still a little speculative when sent to an engine in the `MR_STEALING` state). When the user selects polling for work stealing rather notification, then an engine will never attempt to notify another engine during spark creation.

An engine that executes `MR_idle` and cannot find any work will jump to the `MR_sleep` procedure, which is shown in Algorithm 3.13. `MR_sleep` is another Mercury procedure written in C; it starts by attempting to put the engine into the sleeping state, checking for any notifications using a compare and swap as we saw in `MR_idle`. If there are no notifications it will wait on the sleep semaphore in the `MR_engine_sleep_sync` structure. When another engine signals `MR_es_sleep_sem` then the engine is woken up. The woken engine will retrieve and act on the action closure. We have shown the full switch/case statement for interpreting the values of `MR_es_action` and `MR_es_action_data` in `MR_sleep`. The implied switch/case statement in `MR_idle` is similar, except that it does not need to cover some actions. When polling is enabled the call to `MR_sem_wait` is used with a timeout. If the timeout expires, then the engine will acquire the lock and attempt to steal a spark from any other engine, if it finds a spark it moves to the `MR_WORKING` state. It holds the lock until it either decides to sleep again or sets its state so that, in the meantime, no context notification is lost, as contexts always use notification rather than polling.

When an engine receives a `MR_ACTION_WORKSTEAL` message it executes the code at lines 18–31. As mentioned in Section 3.4, a data race when attempting to steal a spark will result in `MR_steal_spark()` returning an error. In `MR_try_steal_spark()` this error is ignored and the algorithm moves to the next deque. However, when an engine has received a message telling it that there is a spark on this deque, the engine will use the loop on lines 19–23 to retry when a collision occurs. It stops once either it has been successful or the queue is empty. We made this

Algorithm 3.13 MR_sleep

```
 1  void MR_sleep(void) {
 2      int                 engine_id = MR_ENGINE(MR_eng_id);
 3      int                 victim_id;
 4      MR_EngineSleepSync  *eng_data = MR_engine_sleep_data[engine_id];
 5      MR_bool             got_spark;
 6      MR_Spark            spark;
 7      MR_Code             *resume;
 8
 9      if (!MR_compare_and_swap(&(eng_data->MR\_es\_state),
10              MR_STEALING, MR_SLEEPING)) {
11          // Handle the notification that we have received.
12      }
13      MR_sem_wait(&(eng_data->MR_es_sleep_sem));
14      switch(eng_data->MR_es_action) {
15          case MR_ACTION_SHUTDOWN:
16              MR_GOTO(MR_do_shutdown);
17          case MR_ACTION_WORKSTEAL:
18              victim_id = eng_data->MR_es_action_data.MR_ewa_worksteal_engine;
19              do {
20                  MR_spin_loop_hint();
21                  got_spark = MR_steal_spark(MR_spark_deques[victim_id],
22                      &spark);
23              } while (got_spark == BUSY);
24              if (got_spark == SUCCESS) {
25                  eng_data->MR_es_state = MR_WORKING;
26                  MR_ENGINE(MR_eng_victim_counter) = victim_id;
27                  MR_prepare_engine_for_spark(&spark);
28                  MR_GOTO(spark->MR_spark_code);
29              } else {
30                  MR_ENGINE(MR_eng_victim_counter) = victim_id + 1;
31                  MR_GOTO(MR_idle);
32              }
33          case MR_ACTION_CONTEXT:
34              eng_data->state = MR_WORKING;
35              ctxt = eng_data->MR_es_action_data.MR_ewa_context;
36              MR_GOTO(MR_prepare_engine_for_context());
37          case MR_ACTION_RUNQUEUE_CHECK:
38              MR_GOTO(MR_idle);
39      }
40  }
```

Version	Sequential		Parallel w/ N Engines			
	not TS	TS	1	2	3	4
Right recursive mandelbrot (max. 1024 contexts)						
Original	15.3 (0.0)	15.6 (1.7)	15.2 (0.1)	7.8 (0.0)	5.3 (0.0)	4.1 (0.0)
New (notify)	15.3 (0.1)	15.2 (0.0)	15.2 (0.0)	7.7 (0.0)	5.1 (0.0)	3.9 (0.0)
New (poll)	15.7 (1.9)	15.1 (0.0)	15.1 (0.0)	7.6 (0.1)	5.1 (0.0)	3.8 (0.0)
Left recursive mandelbrot						
Original	15.2 (0.0)	15.3 (0.0)	15.2 (0.1)	7.6 (0.0)	5.2 (0.6)	3.8 (0.0)
New (notify)	15.3 (0.0)	15.2 (0.1)	15.2 (0.0)	7.7 (0.0)	5.1 (0.0)	3.9 (0.0)
New (poll)	15.3 (0.1)	15.1 (0.1)	15.2 (0.3)	7.6 (0.0)	5.1 (0.0)	3.8 (0.0)

Table 3.8: Work stealing results for mandelbrot — revised implementation

decision because it is possible that many sparks may be placed on the deque at the same time, and even though our engine lost a race to another thief, there may be more sparks available. If the engine fails to steal a spark from this deque, the engine will jump to `MR_idle` where it will check the run queue and then recheck all the spark deques. The other actions performed when receiving a message are simple to understand from Figure 3.7 and Algorithm 3.13.

We have also had to make two changes to `MR_join_and_continue`, as shown in Algorithm 3.14. The first change is an optimisation on lines 13–14, when the parallel conjunction's original context is suspended and the engine's current context executes the last conjunct in the parallel conjunction, then the previous version of the algorithm would schedule the original context by placing it on the run queue and then execute `MR_idle` as its current context had finished. Since we know that the current context is finished and the original one is runnable, we now resume the original context's execution directly.

The second change is on lines 19–28. This new if statement checks whether the context that the engine holds is compatible with the spark it has just popped from the deque. A context is incompatible if it contains stack frames of a computation that is not a parent of the computation the spark represents. This means that the context is the original context for the parallel conjunction it has been working on, but the spark is not part of this parallel conjunction. When this happens we must change the context (we cannot choose a different spark without stealing and we try to minimise stealing), therefore we suspend the current context (lines 21–23). Since a context switch is mandatory, then we know that it is better to switch to a context that is already runnable, so if the run queue is non-empty we put the spark back on the deque and jump to `MR_idle` (lines 24–27); otherwise we execute the spark.

Table 3.8 shows a comparison of the previous work stealing implementation with this one using mandelbrot. There are two row groups with three rows each. The first group shows the results for right recursive mandelbrot while the second shows the results for left recursive mandelbrot. The first row in each group gives the results for the previous work stealing system and the second and third rows in each group show the results for the current system using either notifications (second row) and polling (third row). Each result is presented with its standard deviation in parentheses. We compiled the right recursive mandelbrot program with the conjunct reordering transformation from Section 3.5 disabled, otherwise it would have had the same results as the left recursive version of the same program. It is still important to test right recursion such as in this example, as the conjunct reordering transformation cannot reorder dependent parallel conjunctions so right recursion can still occur.

Algorithm 3.14 `MR_join_and_continue`— improved work stealing version

```
 1  void MR_join_and_continue(MR_SyncTerm *st, MR_Code *cont_label) {
 2      MR_bool     finished, got_spark;
 3      MR_Context  *current_context = MR_ENGINE(MR_eng_current_context);
 4      MR_Context  *orig_context = st->MR_st_orig_context;
 5      MR_Spark    spark;
 6
 7      finished = MR_atomic_dec_and_is_zero(&(st->MR_st_num_outstanding));
 8      if (finished) {
 9          if (orig_context != current_context) {
10              while (orig_context->MR_ctxt_resume != cont_label) {
11                  CPU_spin_loop_hint();
12              }
13              MR_release_context(current_context);
14              MR_load_context(orig_context);
15          }
16          MR_GOTO(cont_label);
17      } else {
18          if (MR_pop_spark(MR_ENGINE(MR_eng_spark_deque), &spark)) {
19              if ((orig_context == current_context) &&
20                      (spark.MR_spark_sync_term != st)) {
21                  MR_save_context(current_context);
22                  current_context->MR_ctxt_resume = cont_label;
23                  MR_ENGINE(MR_eng_current_context) = NULL;
24                  if (nonempty(MR_context_runqueue)) {
25                      MR_push_spark(MR_ENGINE(MR_eng_spark_deque), &spark);
26                      MR_GOTO(MR_idle);
27                  }
28              }
29              MR_prepare_engine_for_spark(&spark);
30              MR_GOTO(spark.MR_spark_code);
31          } else {
32              if (orig_context == current_context) {
33                  MR_save_context(current_context);
34                  current_context->MR_ctxt_resume = cont_label;
35                  MR_ENGINE(MR_eng_current_context) = NULL;
36              }
37              MR_GOTO(MR_idle);
38          }
39      }
40  }
```

```
:- func fibs(int) = int.

fibs(N) = F :-
    ( N < 2 ->
        F = 1
    ;
        (
            F1 = fibs(N-1)
        &
            F2 = fibs(N-2)
        ),
        F = F1 + F2
    ).
```

(a) `fibs/1+1`, no granularity control

```
:- func fibs_gc(int, int) = int.

fibsgc(N, Depth) = F :-
    ( N < 2 ->
        F = 1
    ;
        ( Depth > 0 ->
            (
                F1 = fibs_gc(N-1, Depth-1)
            &
                F2 = fibs_gc(N-2, Depth-1)
            )
        ;
            F1 = fibs_seq(N-1),
            F2 = fibs_seq(N-2)
        ),
        F = F1 + F2
    ).
```

(b) `fibs_gc/2+1`, with granularity control

Figure 3.8: Naive parallel Fibonacci with and without granularity control

The differences between the old and new results are minimal; we have shown the standard deviation for each result to make it easier to see where comparisons are valid. In both left and right recursive mandelbrot programs using one Mercury engine, the new work stealing code with notifications appears to perform slightly worse than the original work stealing code; however the figures are so close that such a comparison may not be valid. Neither of the two configurations of the new implementation is faster than the original implementation. For right recursive mandelbrot both configurations of the new implementation improved on the old implementation. In left recursive mandelbrot all three versions are comparable. The old implementation used one spark deque per context, of which there are many in right-recursive code; whereas the new implementation uses one spark deque per engine, of which there are a constant number. Therefore, we believe that this supports our decision to associate deque with engines rather than contexts, in order to make work stealing simpler.

To make it easier to see the effects of the new work stealing code, we created a new benchmark named fibs. Fibs calculates the value of the 43$^{\text{rd}}$ Fibonacci number using the naive algorithm: summing the previous two Fibonacci numbers where these are defined recursively. This calculation is shown as the function `fibs/1+1` in Figure 3.8(a). (The arity of a Mercury function is written as `N+1`. The `+1` refers to the return value, while `N` refers to the function's input arguments.) The naive fibs program is a useful micro-benchmark for work stealing. It creates many small tasks whose overheads dominate the program's execution time, allowing us easily measure the overheads of parallel execution. However as most engines will be creating sparks, engines are more likely to find sparks on their own deques and therefore will attempt to steal work less frequently. By comparison in the mandelbrot program, only one engine at a time creates sparks, and therefore all the other engines will steal work from it whenever they can. Results for this version of `fibs/1+1` are shown in the "without GC" (fifth) row group of Table 3.9.

To avoid embarrassing parallelism and create a point for comparison, we have also shown results

Version	Sequential		Parallel w/ N Engines			
	not TS	TS	1	2	3	4
Fibs program with GC (`Depth` = 10)						
Original	4.4 (0.0)	4.6 (0.0)	4.6 (0.0)	2.3 (0.0)	1.6 (0.0)	1.2 (0.0)
New (notify)	4.3 (0.0)	4.6 (0.0)	4.6 (0.0)	2.3 (0.0)	1.6 (0.0)	1.2 (0.0)
New (poll)	4.4 (0.0)	4.6 (0.0)	4.6 (0.0)	2.3 (0.0)	1.5 (0.0)	1.2 (0.0)
Fibs program with GC (`Depth` = 20)						
Original	4.4 (0.0)	4.6 (0.0)	4.7 (0.0)	2.3 (0.0)	1.6 (0.0)	1.2 (0.0)
New (notify)	4.3 (0.0)	4.6 (0.0)	4.7 (0.0)	2.3 (0.0)	1.6 (0.0)	1.2 (0.0)
New (poll)	4.4 (0.0)	4.6 (0.0)	4.7 (0.0)	2.3 (0.0)	1.6 (0.0)	1.2 (0.0)
Fibs program with GC (`Depth` = 30)						
Original	4.0 (0.0)	4.3 (0.0)	23.3 (0.4)	11.7 (0.3)	7.8 (0.2)	5.9 (0.2)
New (notify)	3.9 (0.0)	4.3 (0.0)	25.8 (0.6)	12.9 (0.3)	8.7 (0.3)	6.5 (0.0)
New (poll)	4.0 (0.0)	4.3 (0.0)	25.3 (0.1)	12.7 (0.1)	8.5 (0.0)	6.3 (0.0)
Fibs program with GC (`Depth` = 40)						
Original	3.7 (0.0)	4.1 (0.0)	33.6 (0.8)	16.7 (0.2)	11.2 (0.3)	8.4 (0.3)
New (notify)	3.7 (0.0)	4.1 (0.0)	37.6 (0.21	19.0 (0.6)	12.7 (0.5)	9.5 (0.3)
New (poll)	3.7 (0.0)	4.1 (0.0)	36.5 (0.3)	18.3 (0.1)	12.2 (0.0)	9.1 (0.0)
Fibs program without GC						
Original	4.4 (0.0)	4.6 (0.0)	32.9 (0.3)	16.5 (0.0)	11.0 (0.0)	8.3 (0.0)
New (notify)	4.3 (0.0)	4.6 (0.0)	35.8 (0.1)	18.0 (0.0)	11.9 (0.0)	8.9 (0.0)
New (poll)	4.4 (0.0)	4.6 (0.0)	35.9 (0.2)	17.8 (0.1)	11.8 (0.0)	8.9 (0.0)

Table 3.9: Work stealing results for fibs — revised implementation

for fibs using *granularity control* (GC) to create fewer but larger parallel tasks (parallelism with coarser grains). We can achieve this by introducing a new function `fibs_gc/2+1`, which is shown in Figure 3.8(b). `fibs_gc/2+1` takes an additional argument named `Depth` which is the number of recursion levels near the top of the call graph in which parallel execution should be used. Below this depth sequential execution is used. The first four rows of Table 3.9 show results using `fibs_gc/2+1`, with varying initial values for `Depth`.

Fibs has better parallel performance when `Depth`'s initial value is smaller, demonstrating how granularity control improves performance. In parallelised divide and conquer code such as this, the context limit does not affect performance as much as it does with right single recursive code. This is because both conjuncts have similar costs compared to the different costs in singly recursive code. The original context is less likely to block on the completion of the second conjunct, and if it does it will not need to block for as long as it would with right recursive code. We do not compare the context limit behaviour of divide and conquer with that of left recursion, as work stealing solved the context limit problems in left recursive code (Section 3.4). The coarse grained fibs results (smaller values for `Depth`) show little difference between the different work stealing implementations. However, in the fine grained fibs tests we can clearly see that the new work stealing system has higher overheads, both with notifications and with polling. This makes sense for the notification tests as each time an engine creates a spark it may need to notify another engine. Even when no notification is made, the test to determine if a notification is needed adds overhead. But the new results with polling are also slower than the old runtime system. In the cases with a `Depth` of 30 or 40, the new implementation with notifications performs more slowly than the new implementation with polling. This is not a big surprise as the cost of notification might be occurred any time a spark is created, and the cost can vary depending on whether a

notification needs to be made, whereas the cost of polling only affects idle engines. We cannot see this in the results as engines spend very little time idle while running fibs, and the impact on idle engines seldom has an effect on the elapsed execution time of a program (it may affect CPU utilisation slightly).

When we consider the single engine cases, there is also a clear difference between the old and new implementations, even though no work stealing or notification occurs with a single engine. Therefore this difference in performance comes from some code that is executed when only one engine is used. Work stealing itself, or the code in `MR_idle` is probably not the cause of the slow down. This leaves only `MR_join_and_continue`, whose new version contains an extra test to ensure that the engine's current context is compatible with the spark that it found on its local deque. Further tests could be performed to confirm this hypothesis.

The benefits of the new implementation appear to be mixed. Notifications make the new system slower, as does the new implementation of `MR_join_and_continue`. However, the smaller number of spark deques make work stealing faster. Overall, the benefit of the new implementation depends on the program being executed. However it is important to remember that fibs is a micro-benchmark and does not represent typical programs, especially the non-granularity controlled and under-granularity controlled versions as they create embarrassing parallelism.

There are some other interesting trends in the results. Most obviously, as expected and explained above, parallel fibs program with fine granularity (high values for `Depth` or disabled granularity control), run more slowly than those with coarse granularity. However when we consider the sequential results for varying amounts of granularity, we see some strange results. We expected to see the overheads of granularity control in these results; the results with large values for `Depth` should be slower than the ones with smaller values. We also expected the results without granularity control to be the fastest, but not significantly: in practice `fibs_gc/2+1` contributes to a tiny part of the call graphs of tests with a low `Depth` value. However the results are very different: the sequential execution times for fibs with a `Depth` of 40 are *faster* than those with a `Depth` of 10 and those without granularity control. We are not sure of the cause, however one hypothesis is that of the two procedures used in the granularity control code, `fibs_seq/1+1` and `fibs_gc/2+1`, one is faster, namely `fibs_gc/2+1`, due to some strange interaction of the processor's branch predictor and the code's alignment. This type of effect can be common with micro-benchmarks, which is one reason why their use is often discouraged. This could be tested by running several tests with randomised code layout [34].

We have noticed that in many workloads, such as a left recursive parallel loop, one engine creates all the sparks. In situations like this, work stealing is more common than executing sparks from an engine's own deque. One processor would execute its own sparks while $P-1$ processors act as thieves, and the original engine's work queue can become a bottleneck. To avoid this behaviour, a thief could steal more than one spark from a victim and put all but one of the stolen sparks onto its own deque, then it immediately executes the single spark which was not placed on the deque. With each stealing operation, sparks become distributed among more of the engines, which improves the probability that a given victim has work on its own deque. Work stealing will become less frequent as engines try to execute their own sparks before stealing other's, this should improve the efficiency of the program as a whole.

We are currently using the deque data structure and algorithms described by Chase and Lev [28] (C.L. Deque). It is a circular resizable array whose operations (Section 3.4) allow us to steal one item at a time. It is based on Arora et al. [6]. We have considered using the deque described

by Hendler and Shavit [56] (H.S. Deque), this is also a deque using a circular array based on Arora et al. [6]. It allows a thief to steal up to half the items on a victim's deque in a single operation. However, the H.S. Deque does not automatically resize, a feature that we identified as important (Section 3.4). We should investigate using a deque similar to the H.S. Deque with resizing support in the future. At that time we will also have to carefully consider spark execution order.

Another potential improvement is in the selection of a thief's victim. A thief may wish to prefer victims that are nearby in terms of memory topology, so that communication of the data relevant to the spark is cheaper. Likewise, selecting an engine to wake up when creating a spark could also consider memory topology.

Chapter 4

Overlap Analysis for Dependent AND-parallelism

Introducing parallelism into a Mercury program is easy. A programmer can use the parallel conjunction operator (an ampersand) instead of the plain conjunction operator (a comma) to tell the compiler that the conjuncts of the conjunction should be executed in parallel with one another. However, in almost all places where parallelism can be introduced, it will not be profitable as it is not worthwhile parallelising small computations. Making a task available to another CPU may take thousands of instructions, so spawning off a task that takes only a hundred instructions is clearly a loss. Even spawning off a task of a few thousand instructions is not a win; it should only be done for computations that take long enough to benefit from parallelism. It is often difficult for a programmer to determine if parallelisation of any particular computation is worthwhile. Researchers have therefore worked towards automatic parallelisation. Autoparallelising compilers have long tried to use granularity analysis to ensure that they only spawn off computations whose cost will probably exceed the spawn-off cost by a comfortable margin. However, this is not enough to yield good results, because data dependencies may *also* limit the usefulness of running computations in parallel. If a spawned off computation blocks almost immediately and can resume only after another computation has completed its work, then the cost of parallelisation again exceeds the benefit.

This chapter presents a set of algorithms for recognising places in a program where it is worthwhile to execute two or more computations in parallel, algorithms that pay attention to the second of these issues as well as the first. Our system uses profiling information to estimate the times at which a procedure call is expected to consume the values of its input arguments and the times at which it is expected to produce the values of its output arguments. Given two calls that may be executed in parallel, our system uses the estimated times of production and consumption of the variables they share to determine how much their executions are likely to overlap when run in parallel, and therefore whether executing them in parallel is a good idea or not.

We have implemented this technique for Mercury in the form of a tool that uses data from Mercury's deep profiler to generate recommendations about what to parallelise. The programmer can then execute the compiler with automatic parallelism enabled and provide the recommendations to generate a parallel version of the program. An important benefit of profile-directed parallelisation is that since programmers do not annotate the source program, it can be re-parallelised easily after

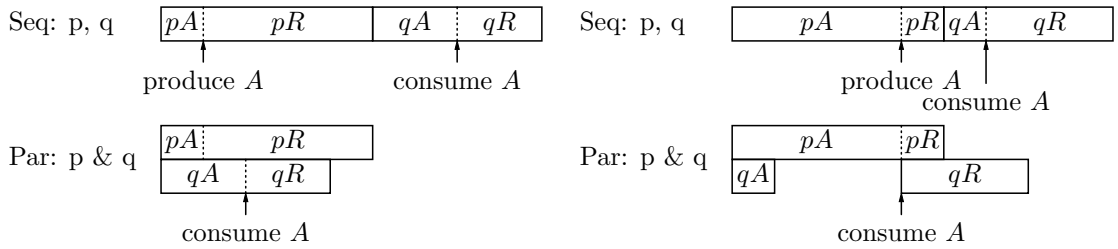


Figure 4.1: Ample vs smaller parallel overlap between `p` and `q`

a change to the program obsoletes some old parallelisation opportunities and creates others. To do this, the programmer can re-profile the program to generate fresh recommendations and recompile the program to apply those recommendations. Nevertheless, if programmers want to parallelise some conjunctions manually, they can do so: our system will not override the programmer.

We present preliminary results that show that this technique can yield useful parallelisation speedups, while requiring nothing more from the programmer than representative input data for the profiling run.

The structure of this chapter is as follows. Section 4.1 states our two aims for this chapter. Then Section 4.2 outlines our general approach including information about the call graph search for parallelisation opportunities. Section 4.3 describes how we calculate information about recursive calls missing from the profiling data. Section 4.4 describes our change to coverage profiling, which provides more accurate coverage data for the new call graph based search for parallelisation opportunities. Section 4.5 describes our algorithm for calculating the execution overlap between two or more dependent conjuncts. A conjunction with more than two conjuncts can be parallelised in several different ways; Section 4.6 shows how we choose the best way. Section 4.7 discusses some pragmatic issues. Section 4.8 evaluates how our system works in practice on some example programs, and Section 4.9 concludes with comparisons to related work.

4.1 Aims

When parallelising Mercury programs, the best parallelisation opportunities occur where two goals take a significant and roughly similar amount of time to execute. Their execution time should be as large as possible so that the relative costs of parallel execution are small, and they should be independent to minimise synchronisation costs. Unfortunately, goals expensive enough to be worth executing in parallel are rarely independent. For example, in the Mercury compiler itself, there are 69 conjunctions containing two or more expensive goals, goals with a cost above 10,000csc (call sequence counts), but in only three of those conjunctions are the expensive goals independent. This is why Mercury supports the parallel execution of dependent conjunctions through the use of futures and a compiler transformation [113, 114] (Section 2.3). If the *consumer* of the variable attempts to retrieve the variable's value before it has been produced, then its execution is blocked until the *producer* makes the variable available.

Dependent parallel conjunctions differ widely in the amount of parallelism they have available. Consider a parallel conjunction with two similarly-sized conjuncts, `p` and `q`, that share a single variable `A`. If `p` produces `A` late but `q` consumes it early, as shown on the right side of Figure 4.1, there will be little parallelism, since `q` will be blocked soon after it starts, and will be unblocked only when `p` is about to finish. Alternatively, if `p` produces `A` early and `q` consumes it late, as

```
map_foldl(_, _, [], Acc, Acc).
map_foldl(M, F, [X | Xs], Acc0, Acc) :-
    M(X, Y),
    F(Y, Acc0, Acc1),
    map_foldl(M, F, Xs, Acc1, Acc).
```

(a) Sequential `map_foldl`

```
map_foldl(_, _, [], Acc, Acc).
map_foldl(M, F, [X | Xs], Acc0, Acc) :-
    (
        M(X, Y),
        F(Y, Acc0, Acc1)
    ) &
    map_foldl(M, F, Xs, Acc1, Acc).
```

(b) Parallel `map_foldl` with overlap

Figure 4.2: Sequential and parallel `map_foldl`

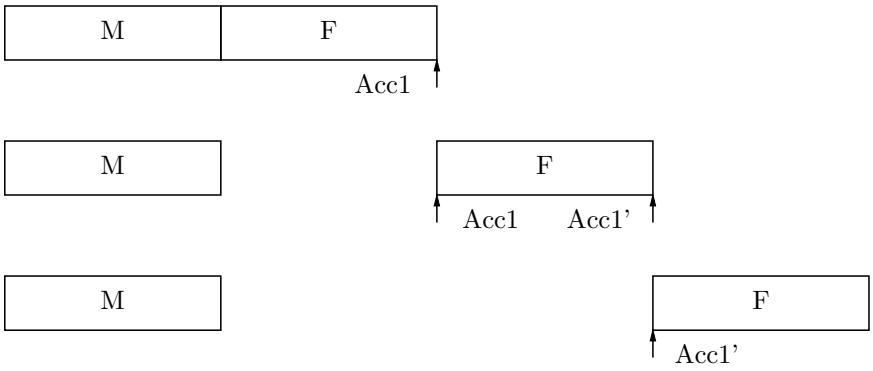


Figure 4.3: Overlap of `map_foldl` (Figure 4.2(b))

shown on the left side of in Figure 4.1, we would get much more parallelism. The top part of each scenario shows the execution of the sequential form of the conjunction.

Unfortunately, in real Mercury programs, almost all conjunctions are dependent conjunctions, and in most of them, shared variables are produced very late and consumed very early. Parallelising them would therefore yield slowdowns instead of speedups, because the overheads of parallel execution would far outweigh the benefit of the small amount of parallelism that is available. We want to parallelise only conjunctions in which any shared variables are produced early, consumed late, or (preferably) both; such computations expose more parallelism. The first purpose of this chapter is to show how one can find these conjunctions.

The second purpose of this chapter is to find the best way to parallelise these conjunctions. Consider the `map_foldl` predicate in Figure 4.2(a). The body of the recursive clause has three conjuncts. We could make each conjunct execute in parallel, or we could execute two conjuncts in sequence (either the first and second, or the second and third), and execute that sequential conjunction in parallel with the remaining conjunct. In this case, there is little point in executing the higher order calls to the `M/2` and `F/3` predicates in parallel with one another, since in virtually all cases, `M/2` will generate `Y` very late and `F/3` will consume `Y` very early. However, executing the sequential conjunction of the calls to `M/2` and `F/3` in parallel with the recursive call *will* be worthwhile if `M/2` is time-consuming, because this implies that a typical recursive call will consume its fourth argument late. The recursive call processing the second element of the list will have significant execution overlap (mainly the cost of `M/2`) with its parent processing the first element of the list even if (as is typical) the fold predicate generates `Acc1` very late. This parallel version of `map_foldl` is shown in Figure 4.2(b). A representation of the first three iterations of it is shown in Figure 4.3. (This is the kind of computation that Reform Prolog [11] was designed to parallelise.)

4.2 Our general approach

We want to find the conjunctions in the program whose parallelisation would be the most profitable. This means finding the conjunctions with conjuncts whose execution cost exceeds the spawning-off cost by the highest margin, and whose interdependencies, if any, allow their executions to overlap the most. It is better to spawn off a medium-sized computation whose execution can overlap almost completely with the execution of another medium-sized computation, than it is to spawn off a big computation whose execution can overlap only slightly with the execution of another big computation, but it is better still to spawn off a big computation whose execution can overlap almost completely with the execution of another big computation. Essentially, the more the tasks' executions can overlap with one another, the greater the margin by which the likely runtime of the parallel version of a conjunction beats the likely runtime of the sequential version (speedup), and the more beneficial parallelising that conjunction will be.

To compute this likely benefit, we need information both about the likely cost of calls and the execution overlap allowed by their dependencies. A compiler may be able to estimate some cost information from static analysis. However, this will not be accurate; static analysis cannot take into account sizes of data terms, or other values that are only available at runtime. It may be possible to provide this data by some other means, such as by requiring the programmer to provide a descriptions of the typical shapes and sizes of their program's likely input data. Programming folklore says that programmers are not good at estimating where their programs' hotspots are. Some of the reasons for this will affect a programmer's estimate of their program's likely input data, making it inaccurate. In fact, misunderstanding a program's typical input is one of the reasons why a programmer is likely to mis-estimate the location of the program's hotspots. Our argument is that an estimate, even a confident one, can only be verified by measurement, but a measurement never needs estimation to back it up. Therefore, our automatic parallelisation system uses profiler feedback information. This was introduced in Section 2.4, which also includes a description of Mercury's deep profiler. To generate the profiler feedback data, we require programmers to follow this sequence of actions after they have tested and debugged the program.

1. Compile the program with options asking for profiling for automatic parallelisation.
2. Run the program on a representative set of input data. This will generate a profiling data file.
3. Invoke our feedback tool on the profiling data file. This will generate a parallelisation feedback file.
4. Compile the program for parallel execution, specifying the feedback file. The file tells the compiler *which* sequential conjunctions to convert to parallel conjunctions, and exactly *how*. For example, `c1, c2, c3` can be converted into `c1 & (c2, c3)`, into `(c1, c2) & c3`, or into `c1 & c2 & c3`, and as the `map_foldl` example shows, the speedups you get from them can be strikingly different.

A visual representation of such a workflow is shown in Figure 2.10 on page 31. It is up to the programmer using our system to select training input for the profiling run in step 2. Obviously, programmers should pick input that is as representative as possible; but even input data that is quite different from the training input can generate useful parallelisation recommendations. Variations in our input data will change the numerical results that we use to decide whether

something should be parallelised, however they rarely change a "should parallelise" decision into a "should not parallelise" decision or vice-versa. The other source of inaccuracy comes from mis-estimating the hardware's performance on certain operations such as the cost of spawning off a new task. Such mis-estimations will have the same impact as variations in input data. The main focus of this chapter is on step 3; we give the main algorithms used by the feedback tool. However, we will also touch on steps 1 and 4. We believe that step 2 can only be addressed by the programmer, as they understand what input is representative for their program.

Our feedback tool looks for parallelisation opportunities by doing a depth-first search of the call tree of the profiling run, each node of which is an SCC (strongly connected component) of procedure calls. It explores the subtree below a node in the tree only if the per-call cost of the subtree is greater than a configurable threshold, and if the amount of parallelism it has found at and above that node is below another configurable threshold. The first test lets us avoid looking at code that would take more work to spawn off than to execute, while the second test lets us avoid creating more parallel work than the target machine can handle. Together these tests dramatically reduce the portions of a program that need analysis, reducing the time required to search for parallelisation opportunities.

For each procedure in the call tree, we search its body for conjunctions that contain two or more calls with execution times above yet another configurable threshold. This test also reduces the parts of the program that will be analysed further; it quickly rejects procedures that cannot contain any profitable parallelism. Parallelising a conjunction requires partitioning the original conjuncts into two or more groups, with the conjuncts in each group being executed sequentially but different groups being executed in parallel. Each group represents a hypothetical sequential conjunction, and the set of groups represents a hypothetical parallel conjunction. As this parallel conjunction represents a possible parallelisation of the original conjunction, we call it a *candidate parallelisation*. Most conjunctions can be partitioned into several alternative candidate parallelisations, for example, we showed above that `map_foldl` has three alternative parallelisations of its recursive branch. We use the algorithms of Section 4.5 to compute the expected parallel execution time of each parallelisation. These algorithms take into account the runtime overheads of parallel execution. Large conjunctions can have a very large number of candidate parallelisations (2^{n-1} for n conjuncts). Therefore, we use the algorithms of Section 4.6 to heuristically reduce the number of parallelisations whose expected execution time we calculate. If the best-performing parallelisation we find shows a nontrivial speedup over sequential execution, we remember that we want to perform that parallelisation on this conjunction. A procedure can contain several conjunctions with two or more goals that we consider parallelising, therefore multiple candidate parallelisations may be generated for different conjunctions in a procedure. The same procedure may also appear more than once in the call graph. Each time it occurs in the call graph its conjunctions may be parallelised differently, or not at all, therefore it is said to be *polyvariant* (having multiple forms). Currently our implementation compiles a single *monovariant* procedure. We discuss how the implementation chooses which candidate parallelisations to include in Section 4.7.

4.3 The cost of recursive calls

The Mercury deep profiler gives us the costs of all non-recursive call sites in a clique. For recursive calls, the costs of the callee are mingled together with the costs of the caller, which is either the same procedure as the callee, or is mutually recursive with it. Therefore if we want to know the

cost of a recursive call site (and we do), we have to infer this from the cost of the clique as a whole, the cost of each call site within the procedures of the clique, the structures of the bodies of those procedures, and the frequency of execution of each path through those bodies.

For now, we will restrict our attention to SCCs that contain only a single procedure and where that procedure matches one of the three recursion patterns below. These are among the most commonly used recursion patterns and the inference processes for them are also among the simplest. Later, we will discuss how partial support could be added for mutually recursive procedures.

Pattern 1: no recursion at all. This is not a very interesting pattern, but we support it completely. We do not need to compute the costs of recursive calls if a procedure is not recursive.

Pattern 2: simply recursive procedures. The first pattern consists of procedures whose bodies have just two types of execution path through them: base cases, and recursive cases containing a single recursive call site. Our example for this category is `map_foldl`, whose code is shown in Figure 4.2.

Let us say that of 100 calls to the procedure, 90 were from the recursive call site and 10 were from a call site in the parent SCC. Then we would calculate that each non-recursive call (from the parent SCC) would on average yield nine recursive calls (from within the SCC). Note that there are actually ten levels of recursion so we add one for the highest level of recursion (the call from the parent SCC). We call this the average deepest recursion:

$$AvgMaxDepth = Calls_{RecCallSites}/Calls_{ParentCallSite} + 1$$

The deepest recursive call site executes only the non-recursive path, and incurs only its costs ($CostNonRec$). We measure the costs of calls in *call sequence counts* (csc), a unit defined in Section 2.4.1. The next deepest would take the recursive path, and incur one copy of the non-recursive call costs along the recursive path ($CostNonRec$) plus the cost of the recursive call itself (1) plus the cost of the non-recursive branch ($CostNonRec$). The third last would incur two copies of the costs of the non-recursive calls along the recursive path, plus the cost of the last call. The formulas for each of these recursive call site costs is:

$$\begin{aligned} cost(0) &= CostNonRec \\ cost(1) &= CostNonRec + CostRec + 1 \\ cost(2) &= CostNonRec + 2{\times}CostRec + 2 \end{aligned}$$

By induction, the cost of a recursive call site to depth D (0 being the deepest) is:

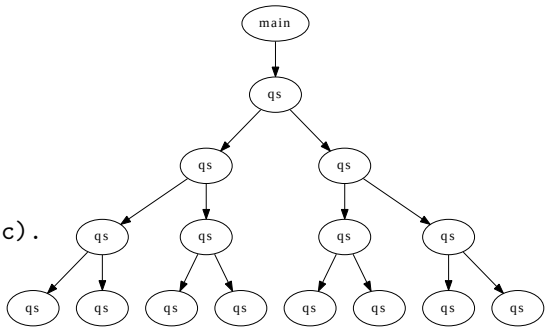
$$cost(D) = CostNonRec + D(CostRec + 1)$$

We can now calculate the average cost of a call at any level of the recursion. Simply recursive procedures have a uniform number of calls at each depth of the recursion. The depth representing the typical use of such a procedure is half of $AvgMaxDepth - 1$. We subtract 1 as the first level of recursion is not reached by a recursive call site. This allows us to calculate the typical cost of a recursive call from this call site. The typical depth of this part of the call graph is $(10 - 1)/2 - 1 = 3.5$ The extra subtraction of 1 is necessary as we start counting depth from zero. For example, if `map_foldl`'s non-recursive path cost is 10csc, its recursive path cost is 10,000csc, and its typical depth is 3.5. Then its typical cost is $10 + 3.5(10,000 + 1) = 35,013.5$ call sequence counts.

Pattern 3: Divide-and-conquer procedures. The third pattern consists of procedures

```
quicksort([], Acc, Acc).
quicksort([Pivot | Xs], Acc0, Acc) :-
    partition(Pivot, Xs, Lows, Highs),
    quicksort(Lows, Acc0, Acc1),
    quicksort(Highs, [Pivot | Acc1], Acc).
```

(a) Accumulator quicksort



(b) Call graph

Figure 4.4: Accumulator quicksort, definition and call graph

whose bodies also have just two types of execution path through them: base cases, and recursive cases containing *two* recursive call sites. Our example for this category is an accumulator version of `quicksort/3`, whose code is shown in Figure 4.4(a).

Calculating the recursion depth of `quicksort/3` can be more problematic. We know that if a good value for `Pivot` is chosen `quicksort/3` runs optimally, dividing the list in half with each recursion. Figure 4.4(b) shows the call graph for such an invocation of `quicksort/3`. There are 15 nodes in `qs`'s call tree, each node has exactly one call leading to it; therefore there is one call from outside `quicksort/3`'s SCC (the call from `main`), and 14 calls within the SCC (the calls from `qs` nodes). By inspection, there are four complete levels of recursion. There are always $\lceil \log_2(N+1) \rceil$ levels in a divide and conquer call graph of N nodes when the graph is a *complete binary tree* (we will cover non-complete binary trees later). Since there are always $N-1$ calls from within the SCC for a graph with N nodes then it follows that there are $\lceil \log_2(C+2) \rceil$ levels for a divide and conquer call graph with C recursive calls. In this example, C is 14 and therefore there are four levels of recursion as we noted above. If there were two invocations of `quicksort/3` from `main/2` then there would be two calls from outside the SCC and 28 calls from within, in this case the depth is still four. The average maximum recursion depth in terms of call counts for divide and conquer code is therefore:

$$AvgMaxDepth = \log_2 \left(\frac{Calls_{RecCallSites}}{Calls_{ParentCallSite}} + 2 \right)$$

This is the estimated depth of the tree so we omit the ceiling operator. Non-complete binary trees can be divided into two cases:

Pathologically bad trees (trees which resemble sticks) are created when consistently worst-case pivots are chosen.

These are rare and are considered performance bugs; programmers will usually want to remove such bugs in order to improve sequential execution performance before they attempt to parallelise their program.

Slightly imbalanced trees are more common, such situations fall into the same class as those where the profiling data is not quite representative of the optimised program's future. In Section 4.2 we explained that these variations are harmless.

Therefore, we assume that all divide and conquer code is, on average, evenly balanced.

As before, the deepest call executes only the non-recursive path, and incurs only its costs. The next deepest takes the recursive path, it incurs the costs of the goals along that path, plus the costs of the two calls to the base case, plus twice the base case's cost itself. The third deepest also takes the recursive path, plus the costs of the two recursive calls' executions of the recursive path, that is three times the cost of the recursive path ($3RecCost$); plus the costs of the two recursive calls and the costs of the four calls to the base case (6); plus four times the base case's cost ($4CostNonRec$).

$$\begin{aligned} cost(0) &= CostNonRec \\ cost(1) &= CostRec + 2 + 2CostNonRec \\ cost(2) &= 3CostRec + 6 + 4CostNonRec \end{aligned}$$

The cost of a recursive call in a perfectly balanced divide and conquer procedure at depth D is:

$$cost(D) = (2^D - 1)(CostRec + 2) + 2^D CostNonRec$$

Most of the execution of a divide and conquer algorithm occurs at deep recursion levels as there are many more calls made at these levels than higher levels. However, for parallelism the high recursion levels are more interesting: we know that parallelising the top of the algorithm's call graph can provide ample coarse-grained parallelism. We will show how to calculate the cost of a recursive call at the top of the call graph. First, depth is measured from zero in the equations above, so we must subtract one from AvgMaxDepth. Second, the recursive calls at the first level call the second level, to compute the cost of calls to this level we must subtract one again. Therefore we use the cost formula with $D = AvgMaxDepth - 2$.

For example, let us compute the costs of the recursive calls at the top of `quicksort/3`'s call graph. In this example, we gathered data using Mercury's deep profiler on 10 executions of `quicksort/3` sorting a list of 32,768 elements. The profiler reports that there are 655,370 calls into this SCC, 10 of which come from the parent SCC, leaving 655,360 from the two call sites within `quicksort/3`'s SCC. Using the formulas above we find that the AvgMaxDepth is $\log_2(655,360/10 + 2) \approx 16$. The total per-call cost of the call site to `partition/4` reported by the profiler is an estimated 35.5csc, it is the only other goal in either the base case or recursive case with a non-zero cost. We wish to compute the costs of the recursive calls at the 15^{th} level so D must be 14. The cost of these calls is $(2^{14} - 1)(35.5 + 2) + 2^{14} \times 0 \approx 614,363$. This is the average cost of both of the two recursive calls; assuming that the list was partitioned evenly. Since the deep profiler reports that the total per-call cost of the `quicksort/3` SCC is 1,229,106csc, and the two recursive calls cost 614,363csc each (their sum is 1,228,726csc) plus the cost of `partition/4` (35.5csc) is approximately 1,228,762. This is reasonably close to the total cost of `quicksort/3`, especially given other uncertainties.

There are two ways in which this calculation can be inaccurate. First, poorly chosen pivot values create imbalanced call graphs. We have already discussed how this can affect the calculation of the recursion depth. This can also affect the cost calculation in much the same way, and the same rebuttals apply. There is one extra concern, a pivot that is not perfect will result in two sublists of different sizes and therefore the recursive calls will have different costs rather than the same cost we computed above. It is normal to use *parallel slackness* to reduce the impact of imbalances. This means creating slightly more finely grained parallelism than is necessary in order to increase the chance that a processor can find parallel work. This works because on average the computed costs will be close enough to the real costs.

```
Line        Code                Coverage    Cost
     p(X, Y, ...)  :-             100%
         (
             X = a,               60%        0
             q(...)               60%    1,000
   5     ;
             X = b,               20%        0
             r(...)               20%    2,000
         ;
             X = c,               20%        0
  10         p(...)               20%
         ),
         (
             Y = d,               90%        0
             s(...)               90%   10,000
  15     ;
             Y = e,               10%        0
             p(...)               10%
         ).
```

Figure 4.5: Two recursive calls and six code paths.

The second cause of inaccurate results comes from our assumption about `partition/4`'s cost. We assumed that partition always has the same cost at every level of the call graph. However `partition/4`'s cost is directly proportional to the size of its input list which is itself directly proportional to the depth in `quicksort/3`'s call graph. This would seem to affect our calculations of the cost of recursive calls different levels within the call tree. However the rate at which `partition/4`'s cost increases with height in the tree is linear, while at the same time the number of calls to `partition/4` grows at a power of two with the tree's hight. Therefore as the tree becomes larger the *average* cost of calls to `partition/4` within it asymptotically approaches a constant number. So in significantly large trees the varying costs of `partition/4` do not matter.

We classify recursion types with an algorithm that walks over the structure of a procedure. As it traverses the procedure, it counts the number of recursive calls along each path, the path's cost, and the number of times the path is executed. The number of times a path is executed is generated using coverage profiling, which is described in the next section. When the algorithm finds a branching structure like an if-then-else or switch, it processes each branch independently and then merges its results at the end of the branch. If several branches have the same number of recursive calls (including zero) they can be merged. If several branches have different numbers of recursive calls they are all added to the result set. This means that the result of traversing a goal might include data for several different recursion counts. Consider the example in Figure 4.5. The example code has been annotated with coverage information (in the third column) and with cost information where it is available (fourth column). The conjunction on lines three and four does not contain a recursive call. The result of processing it is a list containing a single tuple: `[(reccalls: 0, coverage: 60%, cost: 1,000)]`. The result for the second switch arm (lines six and seven) is: `[(reccalls: 0, coverage: 20%, cost: 2,000)]`. The third conjunction in the same switch (lines nine and ten) contains a recursive call. The result of processing it is: `[(reccalls: 1, coverage: 20%, cost: 0)]`. When the algorithm is finished processing all the cases in the switch it adds them together. When adding tuples, we can add tuples together with the same

number of recursive calls by adding their coverage and adding their costs weighted by coverage (these are per-call costs). This simplifies multiple code paths with the same number of recursive calls into a single "code path". The result of processing the switch from line 2–11 is:

```
[(reccalls:  0, coverage:  80%, cost:  1,250),
 (reccalls:  1, coverage:  20%, cost:      0)].
```

In this way, the result of processing a goal represents all the possible code paths through that goal. In this case there are three code paths through the switch, and the result has two entries, one represents the two base case code paths, the other represents the single recursive case.

The result of processing the other switch in the example, lines 12–18, is:

```
[(reccalls:  0, coverage:  90%, cost:  10,000),
 (reccalls:  1, coverage:  10%, cost:       0)].
```

In order to compute the result for the whole procedure, we must compute the product of these two results; this computes all the possible paths through the two switches. We do this by constructing pairs of tuples from the two lists. Since each list has two entries there are four pairs:

```
[ (rc:  0, cvg:  80%, cost:   1,250)  ×  (rc:  0, cvg:  90%, cost:  10,000),
  (rc:  0, cvg:  80%, cost:   1,250)  ×  (rc:  1, cvg:  10%, cost:       0),
  (rc:  1, cvg:  20%, cost:       0)  ×  (rc:  0, cvg:  90%, cost:  10,000),
  (rc:  1, cvg:  20%, cost:       0)  ×  (rc:  1, cvg:  10%, cost:       0)]
```

For each pair we compute a new tuple by adding the number of recursive calls, averaging the coverage counts, and adding the costs.

```
[ (rc:  0, cvg:  72%, cost:  11,250),
  (rc:  1, cvg:   8%, cost:   1,250),
  (rc:  1, cvg:  18%, cost:  10,000),
  (rc:  2, cvg:   2%, cost:       0),
```

Again, we merge the cases with the same numbers of recursive calls.

```
[ (rc:  0, cvg:  72%, cost:  11,250),
  (rc:  1, cvg:  26%, cost:   7,308),
  (rc:  2, cvg:   2%, cost:       0)]
```

There are six paths through this procedure, and two recursive calls. The number of tuples needed is linear in the number of path types; in this case we represent all six paths using three tuples. This allows us to conveniently handle procedures of different forms as rather simple recursion types such as "simple recursion" we saw above: a procedure with two recursive paths with one call each can be handled as if it has just one recursive path and one base case.

We can determine the type of recursion for any list of recursion path information. If there is a single path entry with zero recursive calls then the procedure is not recursive. If there is an entry for a path with zero recursive calls, and a path with one recursive call then the procedure is "simply recursive". Finally if there are two paths, one with zero recursive calls and one with two recursive calls, then we know that the procedure uses "divide and conquer" recursion. It is possible to generalise further, if a procedure has two entries, one with zero recursive calls and the other with some N recursive calls. Then the recursion pattern is similar to divide and conquer except that the base in the formulas shown above is N rather than 2. It has not been necessary to handle these cases.

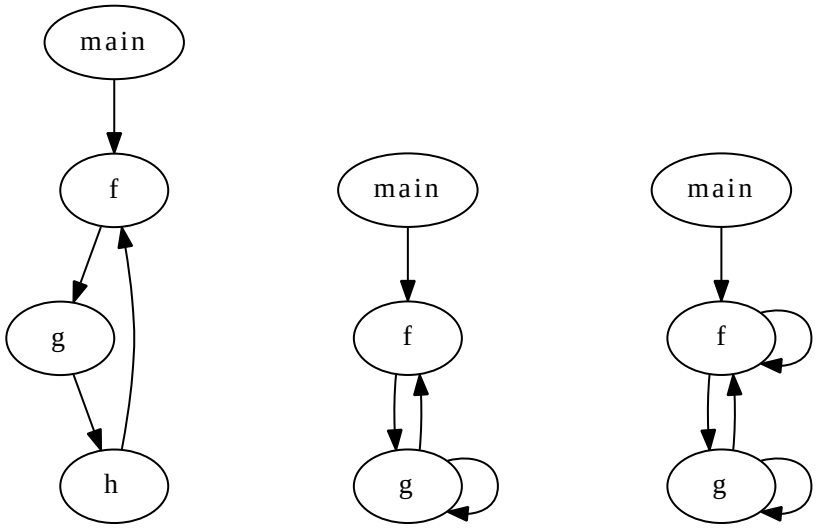
Recursion Type	No. of SCCs	Percent	Total cost
Not recursive	292,893	78.07%	2,320,270,385
Simple recursion	48,458	12.92%	402,430,967
Mutual recursion: totals	19,293	5.14%	198,326,577
2 procs	4,066	1.08%	27,099,504
3 procs	9,393	2.50%	14,846,076
4 procs	1,092	0.29%	12,542,308
5 procs	1,035	0.28%	3,863,295
6+ procs	3,707	0.99%	139,975.394
Unknown	11,803	3.05%	8,838,655
Divide and conquer	1,917	0.51%	5,337,293
Multiple recursive paths: totals	1,089	0.28%	4,678,467
rec-branches: 1, 2	44	0.01%	580,994
rec-branches: 2, 3	281	0.07%	189,377
rec-branches: 2, 3, 4	564	0.15%	3,902,188
other	200	0.05%	5,908

Table 4.1: Survey of recursion types in an execution of the Mercury compiler

There are many possible types of recursion, and we wanted to limit our development effort to just those recursion types that would occur often enough to be important. Therefore, we ran our analysis across all the SCCs in the Mercury compiler, the largest open source Mercury program, to determine which types of recursion are most common. Table 4.1 summarises the results. We can see that most SCCs are not recursive, and the next biggest group is the simply recursive SCCs, accounting for nearly 13% of the profile's SCCs. If our analysis finds an SCC with more than one procedure, it counts the number of procedures and marks the whole SCC as mutually recursive and does no further analysis. It may be possible to perform further analysis on mutually recursive SCCs, but we have not found it important to do this yet. Mutual recursion as a whole accounts for a larger proportion of procedures than divide and conquer. The "Multiple recursive paths" recursion types refer to cases where there are multiple recursive paths through the SCC's procedure with different numbers of recursive calls plus a base case. The table row labelled "Unknown" refers to cases that our algorithm could not or does not handle. This can include builtin code, foreign language code and procedures that may backtrack because they are either `nondet` or `multi`. Note that some procedures such as `semidet` procedures which we cannot parallelise are still interesting: such a procedure may be involved in the production or consumption of a variable that is an argument to the procedure; understanding this procedure may provide information needed to decide if a parallelisation in the procedure's caller is profitable or not.

The profiler represents each procedure in the program's source code as a `ProcStatic` structure (Section 2.4.1). Each procedure may be used multiple times and therefore have multiple `ProcDynamic` structures appearing in different SCCs. Each SCC may have a different recursion type depending on how it was called. For example, calling `length/2` on an empty list will not execute the base case and therefore this *use* of `length/2` is non-recursive.

Some of the "Multiple recursive paths" recursion type cases are due to the implementation of the `map` ADT and code in Mercury's standard library, which uses a 2-3-4 tree implementation, and hence almost all of the cases with 2, 3 or 4 recursive calls on a path are 2-3-4 tree traversals. These traversals also account for some of the other multi recursive path cases, such as those with 2 or 3 recursive calls on a path. In many cases these are the same code running on a tree without any 4-nodes.



(a) One loop (b) Two loops (c) Three loops

Figure 4.6: Mutual recursion

It may be possible to handle simple cases of mutual recursion by a process of hypothetical inlining. Consider the call graph in Figure 4.6(a). In this example f, g and h represent an SCC, f calls g, which calls h, which calls f. These calls are recursive, they create a loop between all three procedures. Prior to doing the recursion path analysis we could inline each of these procedure's representations inside one another as follows: inline h into g and then inline g into f. This creates a new pseudo-procedure that is equivalent to the loop between the three procedures. We can then run the recursion path analysis on this procedure and apply the results to the three original procedures. We have not yet needed to implement call site cost analysis for mutually recursive procedures; therefore this discussion is a thought experiment and not part of our analysis tool.

We believe that we can handle some other forms of mutually recursive code. One example of this is the graph in Figure 4.6(b), which has two loops, one mutually recursive loop and one loop within g. We cannot inline g into f because g has a recursive call whose entry point would disappear after inlining. But we can duplicate f, creating a new $f\prime$, and re-write g's call to f as a call to $f\prime$. This allows us to inline $f\prime$ into g without an issue, and f's call to g is no longer recursive.

The graph in Figure 4.6(c) is more complicated; it has three loops. In this case we cannot inline g into f because of the loop within g, and we cannot inline f into g because of the loop within f. Simple duplication of either f or g does not help us.

Our current implementation handles non-recursive and simply recursive loops within a single procedure. We have shown how to calculate the cost of recursive calls in divide and conquer code. However, generating efficient parallel divide and conquer code is slightly more complicated, as Granularity Control is needed to prevent embarrassingly parallel workloads (Section 3.6). This is less true with simply recursive code as a non-recursive call that is parallelised against will usually have a non-trivial cost in all levels of the recursion.

Line	Coverage		Percall Cost	Code
	Static	Deep		
1	7,851	2,689	3,898,634	`foldl3(P, Xs0, !Acc1, ...) :-`
2	-	-	-	`(`
3	2,142	1	0	`Xs0 = []`
4	-	-	-	`;`
5	5,709	2,688	0	`Xs0 = [X | Xs],`
6	5,709	2,688	1,449	`P(X, !Acc1, ...),`
7	5,709	2,688	3,897,182	`foldl3(P, Xs, !Acc1, ...)`
8	-	-	-	`).`

Figure 4.7: Static and deep coverage data for `foldl3/8`

4.4 Deep coverage information

In Section 2.4.1 we introduced the deep profiler, a profiler that can gather *ancestor context* specific profiling data. We also introduced coverage profiling in Section 2.4.3, which can gather the *coverage* of any program point. Before we started this project the coverage data gathered was not ancestor context specific. Coverage data was collected and stored in the profiler's `ProcStatic` structures where it was not associated with a specific ancestor context of a procedure. Throughout this section we will use the term *deep* to describe data that is ancestor context specific, and we will use *static* to describe data that is not ancestor context specific. We want to use deep profiling data for automatic parallelisation, which requires using analyses that in turn require coverage data such as the calculation of recursive call sites' costs (Section 4.3), and variable use time analysis (Section 2.4.4). Therefore we have changed the way coverage data is collected and stored; this section describes our changes.

Our automatic parallelisation tool uses deep profiling data throughout. However, using deep profiling data and static coverage data creates inconsistencies that can result in erroneous results. Figure 4.7 shows `list.foldl3/8` from Mercury's standard library on the right hand side. On the immediate left (the fourth column) of each line of code, we have shown deep profiling data from a profiling run of the Mercury compiler. The first cell in this column shows the per-call cost of the call site in the parent SCC (the total cost of `foldl3/8`'s call graph). The other cells in this column show the per-call costs of `foldl3/8`'s call sites; the recursive calls cost is the cost of the first recursive call (the call on the top-most recursion level) as calculated by the algorithms in the previous section. This recursion level was chosen as so that the sum of the costs of calls in the body of the procedure is approximately equal to the cost of the procedure from its parent call site. The second and third columns in this table report static and deep coverage data respectively. The deep profiling and coverage data was collected from a single SCC of `foldl3/8`[1] whose ancestor context is part of the compiler's mode checker; the last procedure in the call chain, excluding `foldl3/8`, is `modecheck_to_fixpoint/8`.

Using the static coverage data we can see that `foldl3/8` is called 7,851 times (line one) throughout the execution of the compiler; this includes recursive calls. There are 5,709 directly recursive calls (the coverage on line seven). Using these figures we can calculate that the average deepest recursion is 2.66 levels. We can conclude that on average across the whole program, `foldl3/8`

[1] We do not mean that `foldl3/8` has more than one SCC, and we only picked one; we mean that `foldl3/8` is called many times during the compiler's execution, each time this creates a new SCC and we have picked one of these.

does not make very deep recursions (it is called on short lists). This information is correct when generalised across the program, but it is incorrect in specific ancestor contexts: if we tried to analyse this loop in a specific context it would cause problems. For example, if we use this information with the deep profiling information in the fourth column we would calculate an incorrect value of 4,464csc for the recursive call's cost at this top-most level of the call tree. This is obviously incorrect: if we add it to the cost of the higher order call, 1,449csc then the sum is 5,913csc, which is a far cry from the measured cost of the call to this SCC (3,898,634csc). It is easy to understand how this discrepancy could affect a later analysis such as the variable use time analysis.

If instead we use coverage information specific to this ancestor context, then we see that the base case is executed only once and the recursive case is executed 2,688 times. This means that the recursion is much deeper, with a depth of 2,689 levels. Using this value, we calculate that the cost of the recursive call at the top-most level of the recursion is 3,897,182csc.

We said above that coverage data was stored in `ProcStatic` structures where it was not associated with an ancestor context. The data was represented using two arrays and an integer describing the arrays' lengths. Each corresponding pair of slots in the arrays referred to a single coverage point in the procedure. The first array gives static information such as the type of coverage point and its location in the compiler's representation of the procedure. The second array contains the current value of the coverage point, which is incremented each time execution reaches the program point. Each procedure in the program is associated with a single `ProcStatic` structure and any number of `ProcDynamic` structures. Each `ProcDynamic` structure represents a use of a procedure in the program's call graph (modulo recursive calls). We moved the array containing the coverage points' execution counts into the `ProcDynamic` structure. Since there are more `ProcDynamic` structures than `ProcStatic` structures in a program's profile, this will make the profile larger, consuming more memory and potentially affecting performance. It is important to minimise a profiler's impact on performance. Poor performance both affects the user's experience and increases the risk of distorting the program's profile. However, we do not have to worry about distortion of time measured in call sequence counts. Time measured in seconds may be distorted slightly, but this is minimal.

Normally during profiling we disable optimisations that can transform a program in ways that would make it hard for a programmer to recognise their program's profile. For example inlining one procedure into another might cause the programmer to see one of their procedures making calls that they did not write. However automatic parallelisation, like other optimisations, is used to speedup a program; a programmer will typically use other optimisations in conjunction with automatic parallelism in order to achieve more greater speedups. The compiler performs parallelisation after most other optimisations, and therefore it operates on a version of the program that has already had these optimisations applied. So that we can generate feedback information that can easily be applied to the optimised program, the feedback tool must also operate on an optimised version of the program. Therefore we must enable optimisations when compiling the program for profiling for automatic parallelism.

Table 4.2 shows the overheads of profiling the Mercury compiler, including coverage profiling. We compiled the Mercury compiler with seven different sets of options for profiling. The two row groups of the table show results with optimisations both disabled and enabled. The rows of the table give results for no profiling support[2], profiling support without coverage profiling, static

[2]The non-profiling version of the compiler was not compiled or tested with optimisations disabled as it ignores the '`--profile-optimised`' flag when profiling is disabled.

Profiling type	Time	.text	.(ro)data	.bss	Heap growth	Profile size
Without optimisations						
no coverage	55.4s	30,585K	32,040K	5,646K	419,360K	35,066K
static coverage	56.0s	34,097K	33,547K	6,267K	419,380K	36,151K
deep coverage	57.6s	34,327K	33,306K	5,646K	476,880K	42,896K
With optimisations						
no profiling	10.6s	12,003K	3,203K	5,645K	186,752K	-
no coverage	35.9s	29,599K	31,218K	5,646K	319,184K	23,465K
static coverage	37.1s	33,391K	32,856K	6,302K	319,188K	24,640K
deep coverage	37.6s	33,628K	32,630K	5,646K	370,288K	30,901K

Table 4.2: Coverage profiling overheads

coverage profiling support, and dynamic coverage profiling support. The second column in the table shows the user time, an average of 20 executions while compiling the eight largest modules of the compiler itself. The next three columns show the sizes of different sections in the executable file: the size of the executable's '`.text`' section (the compiled executable code); the size of the '`.data`' and '`.rodata`' sections (static data); the size of the '`.bss`' section (static data that is initially zero). The sixth column shows the *heap growth*, the amount by which the program break[3] moved during the execution. The final column shows the size of the resulting profiling file. All sizes are shown in kilobytes (KiB) (1024 bytes).

We can see that enabling profiling slows the program down by at least a factor of three, or a factor of five if optimisations are disabled. The difference between the results with and without optimisations is unsurprising. It is well established that simple optimisations such as inlining can make a significant difference to performance. As optimisations make the program simpler, the call graph also becomes simpler. For example, inlining reduces the number of procedures in the program, which means that fewer profiling data structures are needed to represent them both in the program's memory and in the data file. This is why the results show that the optimised programs use less memory and generate smaller profiling data files. Optimisations can also improve performance in another way: inlining reduces the number of procedures which reduces the amount of instrumentation placed in the code by the profiler.

Enabling static coverage profiling does not significantly impact the heap usage. The `ProcStatic` structures and their coverage data are stored in the program's static data, the '`.data`', '`.rodata`' and '`.bss`' sections of the executable; in particular, the coverage point values themselves are in the '`.bss`' section. Coverage profiling also creates a small increase in the size of the executable code, as the code now contains some additional instrumentation. Likewise this instrumentation affects performance very slightly. The effect is well within our goal of minimising coverage profiling overheads. Associating coverage data with `ProcDynamic` structures rather than `ProcStatic` structures significantly affects heap usage as `ProcDynamic` structures and the coverage points therein are all stored on the heap. This additional amount of heap usage (56MB or 50MB, without and with optimisations respectively) is acceptable. Conversely we see that the amount of statically allocated memory decreases when using deep coverage profiling. The size of the '`.text`' section and the program's execution time increase only slightly when using deep coverage information. This additional cost of collecting deep coverage data is very small compared to the benefit that this data provides, especially when optimisations are enabled during a profiling build.

[3]The program break marks the end of the heap area. See the `sbrk(2)` Unix system call.

Algorithm 4.1 Dependent parallel conjunction algorithm, for exactly two conjuncts

```
1:  procedure OVERLAP_SIMPLE(SeqTime_q, VarUsesList_q, ProdTimeMap_p)
2:      CurSeqTime ← 0
3:      CurParTime ← 0
4:      sort VarUsesList_q on ConsTime_{q,i}
5:      for (Var_i, ConsTime_{q,i}) ∈ VarUsesList_q do
6:          Duration_{q,i} ← ConsTime_{q,i} − CurSeqTime
7:          CurSeqTime ← CurSeqTime + Duration_{q,i}
8:          ParWantTime_{q,i} ← CurParTime + Duration_{q,i}
9:          CurParTime ← max(ParWantTime_{q,i}, ProdTimeMap_p[Var_i])
10:     DurationRest_q ← SeqTime_q − CurSeqTime
11:     ParTime_q ← CurParTime + DurationRest_q
12:     return ParTime_q
```

4.5 Calculating the overlap between dependent conjuncts

The previous two sections provide the methods necessary for estimating a candidate parallelisation's performance, which is the topic of this section. As stated earlier, dependencies between conjuncts affect the amount of parallelism available. Our goal is to determine if a dependent parallel conjunction exposes enough parallelism to be profitable. We show this as the amount that the boxes overlap in diagrams such as Figure 4.1; this *overlap* specifically represents the amount of elapsed time that can be saved by parallel execution. Estimating this overlap in the parallel executions of two dependent conjuncts requires knowing, for each of the variables they share, when that variable is produced by the first conjunct and when it is first consumed by the second conjunct. The two algorithms for calculating estimates of when variables are produced and consumed (variable use times) are described in Section 2.4.4. The implementations of both algorithms have been updated since my honours project; the algorithms now make use of the deep coverage information described in the previous section.

Suppose a candidate parallel conjunction has two conjuncts p and q, and their execution times in the original, sequential conjunction (p, q), are $SeqTime_p$ and $SeqTime_q$, which are both provided by the profiler. The parallel execution time of p in $p\&q$, namely $ParTime_p$, is the same as its sequential execution time; if we ignore overheads, which we do for now but will come back to them later. Whilst the parallel execution time of q, namely $ParTime_q$, may be different from its sequential execution time as q may block on futures that p may not have signalled yet. In Section 2.4.2 we showed how to calculate $ParTime_q$ in cases where there is a single shared variable between p and q. Then we use $ParTime_p$ and $ParTime_q$ to calculate the speedup due to parallelism.

When there are multiple shared variables we must use a different algorithm such as the one shown in Algorithm 4.1, which is a part of the full algorithm, shown in Algorithm 4.3. It calculates $ParTime_q$ from its arguments $SeqTime_q$, $VarUsesList_q$ and $ProdTimeMap_p$. $VarUsesList_q$ is a list of tuples, with each tuple being a shared variable and the time at which q would consume the variable during sequential execution. $ProdTimeMap_p$ is a map (sometimes called a dictionary) that maps each shared variable to the time at which p would produce the variable during parallel execution.

The algorithm works by dividing $SeqTime_q$ into chunks and processing them in order; it keeps track of the sequential and parallel execution times of the chunks so far. The end of each chunk, except the last, is defined by q's consumption of a shared variable; the last chunk ends at the end of the q's execution. This way each of the first i chunks represents the time before q is likely to

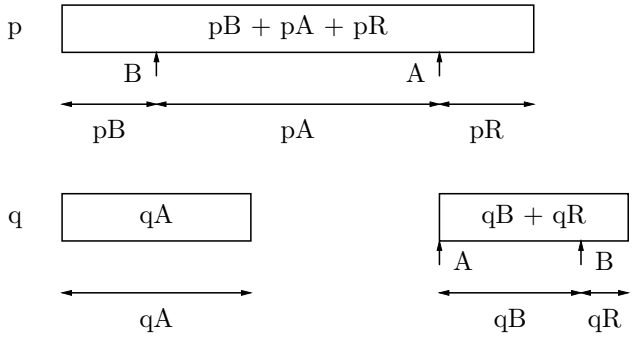


Figure 4.8: Overlap with multiple variables

consume the i^{th} shared variable, these chunks' durations are $Duration_{q,i}$ (line 6). The last $(i+1^{\text{th}})$ chunk represents the time after producing the last shared variable but before q finishes: q may use this time to produce any variables that are not shared with p but are consumed by code after the parallel conjunction. We call this chunk's duration is $DurationRest_q$ (line 10).

Figure 4.8 shows the overlap of a parallel conjunction (p & q) with two shared variables `A` and `B`. The figure is to scale and the sequential execution times of p and q are 5 and 4 respectively. q is broken up into the chunks qA, qB and qR. The algorithm keeps track of the sequential and parallel execution times of q up to the consumption of the current shared variable. During sequential execution, each chunk can execute immediately after the previous chunk, since the values of the shared variables are all available when q starts. During parallel execution, p is producing the shared variables while q is running. If q tries to use a variable (by calling `future_wait/2` on the variable's future) and the variable has not been produced yet then q will block. We can see this for `A`: $ParWantTime_{q,0}$ is 2 but $ProdTimeMap_p$[`A`] is 4, so during the first execution of Algorithm 4.1's loop, line 9 sets $CurParTime$ to 4, which is when q may resume execution. It is also possible that p will produce the variable before it is needed, as is the case for `B`. In cases such as this, q will not be suspended. In `B`'s case, when the loop starts its second iteration $CurSeqTime$ is 2, $CurParTime$ is 4, $ConsTime_{q,1}$ is 3.5 and $ProdTimeMap_q$[`B`] is 1; the algorithm calculates the duration of this second chunk as 1.5, and $ParWantTime_{q,1}$ as 5.5. Line 9 sets $CurParTime$ to 5.5 which is the value of $ParWantTime_{q,1}$ as `B`'s value was already available. After the loop terminates the algorithm calculates the length of the final chunk ($DurationRest_q$) as 0.5, and adds it to $CurParTime$ to get the parallel execution time of the conjunction: 6.

When there are more than two conjuncts in the parallel conjunction our algorithm requires an outer loop. An example of such parallel conjunction is shown in Figure 4.9. We can see that the third conjunct's execution depends on the second's, which depends upon the first's. Dependencies can only exist in the left to right direction: Mercury requires all dependencies to be left to right; it will reorder right to left dependencies and report errors for cyclic dependencies. The new algorithm therefore iterates over the conjuncts from left to right, processing each conjunct with a loop similar to the one in Algorithm 4.1.

Algorithm 4.2 shows a more complete version of our algorithm; this version handles multiple conjuncts and variables, but does not account for overheads. The input of the algorithm is $Conjs$, the conjuncts themselves. The algorithm returns the parallel execution time of the conjunction as a whole. The algorithm processes the conjuncts in an outer loop on lines 4–20. Within each iteration of this loop, it processes the first i chunks of each conjunct in the loop on lines 9–17.

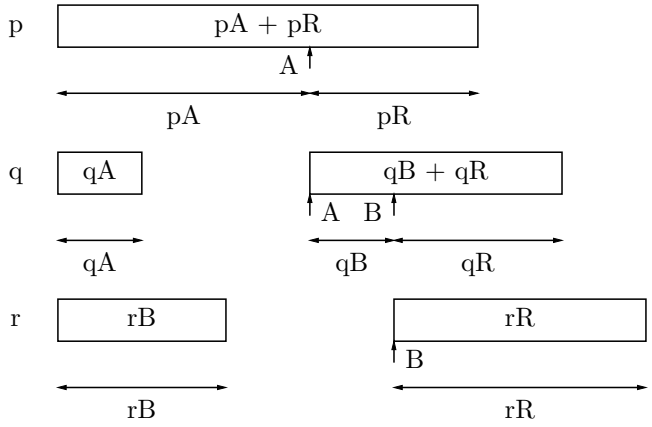


Figure 4.9: Overlap of more than two conjuncts

This inner loop is based on the previous algorithm, the difference is that it must now also act on variables being produced by the current conjunct. $VarUsesList$ now contains both variable consumptions and productions, so the inner loop creates chunks from variable productions as well. The if-then-else on lines 12–17 handles the different producing and consuming cases. When the current conjunct produces a variable, we simply record when this happens in $ProdTimeMap$, which is shared by all conjuncts for variable productions. The map must be built in this way, at the same time as we iterate over the conjuncts' chunks, so that it reflects how a delay in the execution of one task will usually affect when variables may be consumed by another task. We can see this in Figure 4.9: r blocks for longer on `B` than it normally would because q blocks on `A`. Note that $Var_{i,j}$ will always be in $ProdTimeMap$ when we look for it, because it must have been produced by an earlier conjunct and therefore processed by an earlier iteration of the loop over the conjuncts. The only new code outside the inner loop is the use of the $TotalParTime$ variable, the total execution time of the parallel conjunction; it is calculated as the maximum of all the conjuncts' parallel execution times.

In many cases, the conjunction given to Algorithm 4.2 will contain a recursive call. In these cases, the algorithm uses the recursive call's cost at its average recursion depth in the sequential execution data gathered by the profiler. This is naive because it assumes that the recursive call calls the *original, sequential* version of the procedure, however the call is recursive and so the parallelised procedure calls itself, the *transformed parallel* procedure whose cost at its average recursion depth is going to be different from the sequential version's. When the recursive call calls the parallelised version, there may be a similar saving (absolute time, not ratio) on *every* recursive invocation, provided that there are enough free CPUs. How this affects the expected speedup of the top level call depends on the structure of the recursion.

It should be possible to estimate the parallel execution time of the top level call into the recursive procedure, including the parallelism created at each level of the recursion, provided that the recursion pattern is one that is understood by the algorithms in Section 4.3. Before we implemented this it was more practical to improve the efficiency of recursive code (Chapter 5). We have not yet returned to this problem, see Section 7.1. Nevertheless, our current approach handles non-recursive cases correctly, which are the majority (78%) of all cases; it handles a further 13% of cases (single recursion) reasonably well (Section 4.3). Note that even better results for singly recursive procedures can be achieved because of the work in Chapter 5.

Algorithm 4.2 Dependent parallel conjunction algorithm

```
1:  procedure FIND_PAR_TIME(Conjs)
2:      ProdTimeMap ← empty
3:      TotalParTime ← 0
4:      for Conj_i ∈ Conjs do
5:          CurSeqTime_i ← 0
6:          CurParTime_i ← 0
7:          VarUsesList_i ← get_variable_uses(Conj_i)
8:          sort VarUsesList_i by time
9:          for (Var_{i,j}, Time_{i,j}) ∈ VarUsesList_i do
10:             Duration_{i,j} ← Time_{i,j} − CurSeqTime_i
11:             CurSeqTime_i ← CurSeqTime_i + Duration_{i,j}
12:             if Conj_i produces Var_{i,j} then
13:                 CurParTime_i ← CurParTime_i + Duration_{i,j}
14:                 ProdTimeMap[Var_{i,j}] ← CurParTime_i
15:             else                                  ▷ Conj_i must consume Var_{i,j}
16:                 ParWantTime_{i,j} ← CurParTime_i + Duration_{i,j}
17:                 CurParTime_i ← max(ParWantTime_{i,j}, ProdTimeMap[Var_{i,j}])
18:         DurationRest_i ← SeqTime_i − CurSeqTime_i
19:         CurParTime_i ← CurParTime_i + DurationRest_i
20:         TotalParTime ← max(TotalParTime, CurParTime_i)
21:     return TotalParTime
```

	Cost	**Local use of `Acc1`**	
`M`	1,625,050		none
`F`	3	production	3
`map_foldl`	1,625,054	consumption	1,625,051

Table 4.3: Profiling data for `map_foldl`

So far, we have assumed an unlimited number of CPUs, which of course is unrealistic. If the machine has e.g. four CPUs, then the prediction of any speedup higher than four is obviously invalid. Less obviously, even a predicted overall speedup of less than four may depend on more than four conjuncts executing all at once at *some* point. We have not found this to be a problem yet. If and when we do, we intend to extend our algorithm to keep track of the number of active conjuncts in all time periods. Then if a chunk of a conjunct wants to run in a time period when all CPUs are predicted to be already busy, we assume that the start of that chunk is delayed until a CPU becomes free.

To see how the algorithm works on realistic data, consider the `map_foldl` example from Figure 4.2. Table 4.3 gives the costs, rounded to integers, of the calls in the recursive clause of `map_foldl` when used in a Mandelbrot image generator (as in Section 4.8). Each call to `M` draws a row, while `F` appends the new row onto the cord of the rows already drawn. The cord is an instance of a data structure that prescribes a sequence (like a list) but whose append operation runs in constant time. The table also shows when `F` produces `Acc1` and when the recursive call consumes `Acc1`. The costs were collected from a real execution using Mercury's deep profiler and then rounded to make mental arithmetic easier.

Figure 4.2(b) shows the best parallelisation of `map_foldl`. When evaluating the speedup for this parallelisation, the production time for `Acc1` in the first conjunct (`M(X, Y), F(Y, Acc0, Acc1)`) is $1,625,050 + 3 = 1,625,053$, and the consumption time for `Acc1` in the recursive call, `map_foldl(M,`

`F, Xs, Acc1, Acc)`, is $1,625,051$. In this example we are assuming that the recursive call calls a sequential version of the code and that the recursive case will be executed, and in turn the recursive case calls the base case; we revisit this assumption later. The first iteration of Algorithm 4.2's outer loop processes the first conjunct, breaking it into two chunks separated by the production of `Acc1`, which is added to $ProdTimeMap$. During the second iteration the recursive call is also broken into two chunks, which are separated by the consumption of `Acc1`. However, inside the recursive call `Acc1` is called `Acc0`, and so it is the consumption of `Acc0` within the recursive call that is actually considered. This chunk is a long chunk (1,625,051csc). $ParWantTime$ will be 1,625,051 (above) and $ProdTimeMap$[`Acc1`] will be 1,625,053 (above). The execution of the recursive call is likely to block but only very briefly, $CurParTime_i$ will be set to 1,625,053 and the second chunk's duration will be very small, only $1,625,054 - 1,625,051 = 3$. This is added to $CurParTime_i$ to determine the parallel execution time of the recursive call (1,625,056), which is also the maximum of either of the conjuncts' total parallel execution times, making it the parallel execution time of the conjunction as a whole.

If there are many conjuncts in the parallel conjunction or if the parallel conjunction contains a recursive call, we can create more parallelism than the machine can handle. If the machine has e.g. four CPUs, then we do not actually want to spawn off hundreds of iterations for parallel execution, since parallel execution has several forms of overhead. We classify each form of overhead into one of two groups, costs and delays. Costs are time spent *doing* something, e.g. the current context must do something such as spawn off another computation or read a future. Delays represent time spent *waiting* for something; during this time the context is suspended (or being suspended or being woken up).

SparkCost is the cost of creating a spark and adding it to the local spark stack. In a parallel conjunction, every conjunct that is not the last conjunct incurs this cost to create the spark for the rest of the conjunction.

SparkDelay is the estimated length of time between the creation of a spark and the beginning of its execution on another engine. Every parallel conjunct that is not the first incurs this delay before it starts running.

SignalCost is the cost of signalling a future.

WaitCost is the cost of waiting on a future.

ContextWakeupDelay is the estimated time that it takes for a context to resume execution after being placed on the runnable queue, assuming that the queue is empty and there is an idle engine. This can occur in two places: either at the end of a parallel conjunction the original context may need to be resumed to continue its execution, and if a context is blocked on a future it will need to be resumed once that future is signalled.

BarrierCost is the cost of executing the operation that synchronises all the conjuncts at the barrier at the end of the conjunction.

Our runtime system is now quite efficient so that these costs are kept low (Chapter 3). This does not eliminate the overheads, and cannot ever hope to, instead we wish to parallelise only those conjunctions that are still profitable despite overheads. This is why our system actually uses Algorithm 4.3, a version of Algorithm 4.2 that accounts for overheads.

Algorithm 4.3 Dependent parallel conjunction algorithm with overheads

```
1:  procedure FIND_PAR_TIME(Conjs, SeqTimes)
2:      N ← length(Conjs)
3:      ProdTimeMap ← empty
4:      FirstConjTime ← 0
5:      TotalParTime ← 0
6:      for i ← 0 to N − 1 do
7:          Conj_i ← Conjs[i]
8:          CurSeqTime_i ← 0
9:          CurParTime_i ← (SparkCost + SparkDelay) × i
10:         VarUsersList_i ← get_variable_uses(Conj_i)
11:         sort VarUsesList_i by time
12:         if i ≠ N then
13:             CurParTime_i ← CurParTime_i + SparkCost
14:         for (Var_{i,j}, Time_{i,j}) ∈ VarUsesList_i do
15:             Duration_{i,j} ← Time_{i,j} − CurSeqTime_i
16:             CurSeqTime_i ← CurSeqTime_i + Duration_{i,j}
17:             if Conj_i produces Var_{i,j} then
18:                 CurParTime_i ← CurParTime_i + Duration_{i,j} + SignalCost
19:                 ProdTimeMap[Var_{i,j}]← CurParTime_i
20:             else                                  ▷ Conj_i must consume Var_{i,j}
21:                 ParWantTime_{i,j} ← CurParTime_i + Duration_{i,j}
22:                 CurParTime_i ← max(ParWantTime_{i,j}, ProdTimeMap[Var_{i,j}]) + WaitCost
23:                 if ParWantTime_{i,j} < ProdTimeMap[Var_{i,j}] then
24:                     CurParTime_i ← CurParTime_i + ContextWakeupDelay
25:         DurationRest_i ← SeqTime_i − CurSeqTime_i
26:         CurParTime_i ← CurParTime_i + DurationRest_i + BarrierCost
27:         if i = 0 then
28:             FirstConjTime = CurParTime_i
29:         TotalParTime ← max(TotalParTime, CurParTime_i)
30:     if TotalParTime > FirstConjTime then
31:         TotalParTime ← TotalParTime + ContextWakeupDelay
32:     return TotalParTime
```

Algorithm 4.3 handles each of the overheads listed above. The first two overheads that this algorithm accounts for are *SparkCost* and *SparkDelay* on lines 9 and 12–13. Each conjunct is started by a spark created by the previous conjunct, except the first which is executed directly and therefore has no delay. In a conjunction G_0 & G_1 & G_2, the first conjunct (G_0) is executed directly and spends *SparkCost* time creating the spark for G_1 & G_2. This spark is not executed until a further *SparkDelay* time has passed. Therefore on line 9, when we process G_1 and $i = 1$ we add the time $SparkCost + SparkDelay$ to the parallel execution time so far. G_1 will the create the spark for G_2 which costs G_1 a further *SparkCost* time (lines 12–13). The third conjunct (G_2) must wait for the first conjunct to create the second and the second to create it and then *SparkDelay*, so line 9 adds $2 \times (SparkDelay + SparkCost)$ to the current parallel execution time. G_2 does not need to create any other sparks, so the cost on lines 12–13 is not applied.

The next overhead is on line 18, the cost of signalling a future. It is applied each time a future is signalled and the result of applying it is factored into $ProdTimeMap$ along with all other costs, so that its indirect effects on later conjuncts are accounted for. A related pair of overheads are accounted for on lines 22–24: $WaitCost$ is the cost of a `future_wait/2` operation and is paid any time any context waits on a future. $ContextWakeupDelay$ is also used here; if a context is likely to suspend because of a future then it will take some time between getting signalled and waking up. Line 26 of the algorithm accounts for the *BarrierCost* overhead, which is paid at the end of every parallel conjunct.

Throughout Chapter 3 we saw the problems that right recursion can create. These problems were caused because the leftmost conjunct of a parallel conjunction will be executed by the context that started the execution of the conjunction (the original context), and if the same context does not execute the other conjuncts and those conjuncts take longer to execute than the leftmost one, the original context must be suspended. When this context is resumed, we must wait $ContextWakeupDelay$ before it can resume the execution of the code that follows the parallel conjunction. We account for this overhead on lines 30–31. This requires knowing the cost of the first conjunct $FirstConjTime$, which is computed by code on lines 4 and 27–28.

There are two other important pieces of information that we compute, although neither are shown in the algorithm. The first is "CPU utilisation"; it is the sum of all the chunks' lenghts which include the costs but not the delays. The second is the "dead time": it is the sum of the time spent waiting on each future plus the time that the original context spends waiting at the barrier for the conjunction to finish. This represents the amount of time that contexts consume memory without the contexts being actively used. We do not currently use either of these metrics to make decisions about parallelisations, but they could be used to break ties between alternative parallelisations with similar parallel speedups.

4.6 Choosing how to parallelise a conjunction

A conjunction with more than two conjuncts can be converted into several different parallel conjunctions. Converting all the commas into ampersands (e.g. G_1, G_2, G_3 into G_1 & G_2 & G_3) yields the most parallelism. Unfortunately, this will often be *too* much parallelism, because in practice many conjuncts are unifications and arithmetic operations whose execution takes very few instructions. Executing such conjuncts in their own threads costs far more in overheads than can be gained from their parallel execution. To see just how big conjunctions can be, let us consider the quadratic equation example from Section 2.1, which we have repeated here. The quadratic

equation is often written as a single conjunct:

```
X = (-B + sqrt(pow(B, 2) - 4*A*C)) / (2*A)
```

Which is decomposed by the compiler into 12 conjuncts:

```
V_1  = 2,              V_2  = pow(B, V_1),     V_3 = 4,
V_4  = V_3 * A,        V_5  = V_4 * C,         V_6 = V_2 - V_5,
V_7  = sqrt(V_6),      V_8  = -1,              V_9 = V_8 * B,
V_10 = V_9 + V_7,      V_11 = V_1 * A,         X   = V_10 / V_11
```

If the quadratic equation were involved in a parallel conjunction, we would not want to create 12 separate parallel tasks for it. Therefore in most cases, we want to transform sequential conjunctions with n conjuncts into parallel conjunctions with k conjuncts where $k < n$. Each conjunct should consist of a contiguous sequence of one or more of the original sequential conjuncts, effectively partitioning the original conjuncts into groups.

Deciding how to parallelise something in this way can be thought of as choosing which of the $n-1$ conjunction operators (‘`,`’) in an n-ary conjunction to turn into parallel conjunction operators (‘`&`’). For example, $G_1,\ G_2,\ G_3$ can be converted into any of:

G_1 & ($G_2,\ G_3$);
($G_1,\ G_2$) & G_3; or
G_1 & G_2 & G_3.

Our goal is to find the best parallelisation of a conjunction from among the various possible parallel conjunctions. The solution space is as large as 2^{n-1}, and as the quadratic equation example demonstrates that n can be quite large even for simple expressions. The largest conjunction we have seen and tried to parallelise contains about 150 conjuncts. This is in the Mercury compiler itself.

The prerequisite for a profitable parallel conjunction is two or more expensive conjuncts (conjuncts whose per-call cost is above a certain threshold) (Section 4.2). In such conjunctions we create a list of goals from the first expensive conjunct to the last, which we dub the “middle goals”. There are (possibly empty) lists of cheap goals before and after the list of middle goals. Our initial search assumes that the set of conjuncts in the parallel conjunction we want to create is exactly the set of conjuncts in the middle. A post-processing step then removes that assumption.

The initial search space is explored by Algorithm 4.4, which processes the middle goals. The algorithm starts with an empty list as $InitPartition$, zero as $InitTime$, and the list of middle conjuncts as $LaterConjs$. $InitPartition$ expresses a partition of an initial sub-sequence of the middle goals into parallel conjuncts whose estimated execution time is $InitTime$, and considers whether it is better to add the next middle goal to the last existing parallel conjunct ($Extend$) (line 6), or to put it into a new parallel conjunct ($AddNew$) (line 7). The calls to `find_par_time` (lines 8 and 9) evaluate these two alternatives by estimating their parallel overlap using Algorithm 4.3. If $Extend$ is more profitable than $AddNew$, then the recursive call in line 11 searches for the best parallel conjunction beginning with $Extend$; otherwise similar code is executed on line 26.

It is desirable to explore the full $O(2^n)$ solution space however doing so for large conjunctions is infeasible. Therefore we compromise by using an algorithm that is initially complete, but can switch into a greedy (linear) mode if the solution space is too large. Before the algorithm begins we set the global variable $NumEvals$ to zero. It counts how many overlap calculations have been performed and if this reaches the threshold ($PreferLinearEvals$) then our algorithm explores

Algorithm 4.4 Search for the best parallelisation

```
 1: procedure FIND_BEST_PARTITION(InitPartition, InitTime, LaterConjs)
 2:     if empty(LaterConjs) then
 3:         return (InitTime, InitPartition)
 4:     else
 5:         (Head, Tail) ← deconstruct(LaterConjs)
 6:         Extend ← all_but_last(InitPartition) ++ [last(InitPartition) ++ [Head]]
 7:         AddNew ← InitPartition ++ [Head]
 8:         ExtendTime ← find_par_time(Extend)
 9:         AddNewTime ← find_par_time(AddNew)
10:         NumEvals ← NumEvals + 2
11:         if ExtendTime < AddNewTime then
12:             BestExtendSoln ← find_best_partition(Extend, ExtendTime, Tail)
13:             (BestExTime, BestExPartSet) ← BestExtendSoln
14:             if NumEvals < PreferLinearEvals then
15:                 BestAddNewSoln ← find_best_partition(AddNew, AddNewTime, Tail)
16:                 (BestANTime, BestANPartSet) ← BestAddNewSoln
17:                 if BestExTime < BestANTime then
18:                     return BestExtendSoln
19:                 else if BestExTime = BestANTime then
20:                     return (BextExTime, choose(BestExPartSet, BestANPartSet)
21:                 else
22:                     return BestAddNewSoln
23:             else
24:                 return BestExtendSoln
25:         else
26:             symmetric with the then case
27:
28: NumEvals ← 0
29: BestPar ← find_best_partition([], 0, MiddleGoals)
```

only the most profitable of the two alternatives at every choice point. If the threshold has not been reached then the code on lines 15–22 is executed which explores the extensions of the less profitable alternative (*AddNew*) as well. Symmetric code which explores *Extend* when it is the least profitable belongs on line 26 and the lines omitted after it. The algorithm returns the best of the two solutions, or chooses one of the solutions when they are of equal value on line 20. Currently this choice is arbitrary, but it could be based on other metrics such as CPU utilisation. If the threshold has been reached the least profitable alternative is not explored (line 24).

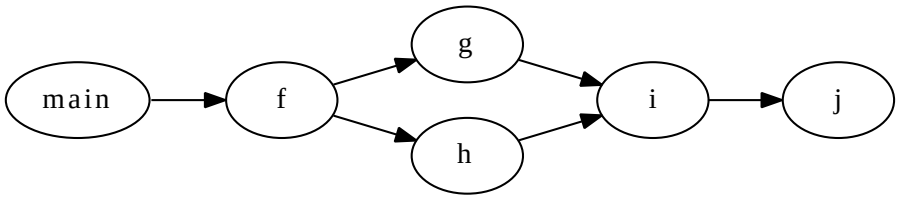
The actual algorithm in the Mercury system is more complex than Algorithm 4.4: our algorithm will also test each partial parallelisation against the best solution found so far. If the expected execution time for the candidate currently being considered is already greater than the fastest existing complete parallelisation, we can stop exploring that branch; it cannot lead to a better solution. This is a simple branch-and-bound algorithm.

There are some simple ways to improve this algorithm.

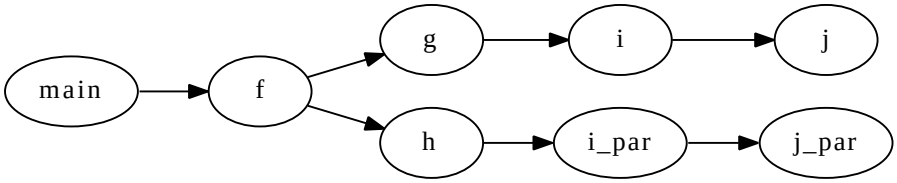
- Most invocations of `find_par_time` specify a partition that is an extension of a partition processed in the recent past. In such cases, `find_par_time` should do its task incrementally, not from scratch.
- Sometimes consecutive conjuncts do things that are obviously a bad idea to do in parallel, such as building a ground term. The algorithm should treat these as a single conjunct.
- Take other metrics such as total CPU utilisation, dead time, or GC pressure into account, at least by using it to break ties on parallel execution time.
- Also, the current implementation does not make an estimate of the minimum cost of the work that is yet to be scheduled after the current point. This affects the amount of pruning that the branch-and-bound code is able to achieve.

At the completion of the search, we select one of the equal best parallelisations, and post-process it to adjust both edges. Suppose the best parallel form of the middle goals is P_1 & ... & P_p, where each P_i is a sequential conjunction. We compare the execution time of P_1 & ... & P_p with that of P_1, (P_2 & ... & P_p). If the former is slower, which can happen if P_1 produces its outputs at its very end and the other P_i consume those outputs at their start, then we conceptually move P_1 out of the parallel conjunction (from the "middle" part of the conjunction to the "before" part). We keep doing this for P_2, P_3 et cetera, until either we find a goal worth keeping in the parallel conjunction, or we run out of conjuncts. We also do the same thing at the other end of the middle part. This process can shrink the middle part.

In cases where we do not shrink an edge, we can consider expanding that edge. Normally, we want to keep cheap goals out of parallel conjunctions, since more conjuncts tends to mean more shared variables and thus more synchronisation overhead, but sometimes this consideration is overruled by others. Suppose the goals before P_1 & ... & P_p in the original conjunction were B_1, ..., B_b and the goals after it A_1, ..., A_a, and consider A_1 after P_p. If P_p finishes before the other parallel conjuncts, then executing A_1 just after P_p in P_p's context may be effectively free: the last context could still arrive at the barrier at the same time, but this way, A_1 would have been done by then. Now consider B_b before P_1. If P_1 finishes before the other parallel conjuncts, *and* if none of the other conjuncts wait for variables produced by P_1, then executing B_b in the same context as P_1 can be similarly free.



(a) j is called from two different ancestor contexts



(b) j's parallel version is a specialisation, which requires specialising i

Figure 4.10: Parallel specialisation

We loop from $i = b$ down towards $i = 1$, and check whether including $B_i, \ \ldots, \ B_b$ at the start of P_1 is improvement. If not, we stop; if it is, we keep going. We do the same from the other end. The stopping points of the loops of the contraction and expansion phases dictate our preferred parallel form of the conjunction, which (if we shrunk the middle at the left edge and expanded it at the right) will look something like $B_1, \ldots, B_b, P_1, \ldots P_k, (P_{k+1}$ & $\ldots$ & P_{p-1} & $(P_p, A_1, \ldots, A_j)), A_{j+1}, \ldots, A_a$. If this preferred parallelisation is better than the original sequential version of the conjunction by at least 1% (a configurable threshold), then we include a recommendation for its conversion to this form in the feedback file we create for the compiler.

4.7 Pragmatic issues

In this section we discuss several pragmatic issues with our implementation. We also discuss several directions for further research.

4.7.1 Merging parallelisations from different ancestor contexts

We search the program's call graph for parallelisation opportunities (Section 4.2). This means that we may visit the same procedure multiple times from different ancestor contexts. In a different context a procedure may have a different optimal parallelisation, or no profitable parallelisation. Me must somehow resolve cases where there are multiple candidate parallelisations for the same procedure.

At the moment, for any procedure and conjunction within that procedure which our analysis indicates is worth parallelising in any context, we pick one particular parallelisation (usually there is only one anyway), and transform the procedure accordingly. This gets the benefit of parallelisation when it is worthwhile, but incurs its costs even in contexts when it is not.

In the future, we plan to use multi-version specialisation. For every procedure with different parallelisation recommendations in different ancestor contexts, we intend to create a specialised version for each recommendation, leaving the original sequential version. This will of course require

the creation of specialised versions of its parent, grandparent etc procedures, until we get to an ancestor procedure that can be used to separate the specialised version, or versions, from the original versions. This is shown in Figure 4.10, in the figure j is being parallelised when it is called from the $main \rightarrow f \rightarrow h \rightarrow i \rightarrow j$ ancestor context, but not when it is called from the $main \rightarrow f \rightarrow g \rightarrow i \rightarrow j$ context. Therefore by specialising both j and i into new versions j_par and i_par respectively and calling i_par from h rather than i we can parallelise j only when it is called from the correct ancestor context.

Each candidate parallel conjunction sent to the compiler includes a goal path and procedure; this points to the procedure and conjunction within the procedure that we wish to parallelise. When different conjunctions in the same procedure have parallelisation advice, we save all this advice into the feedback file. When the compiler acts on this advice, it applies the parallelisations deeper within compound goals first. This ensures that as we apply advice, the goal path leading to each successive conjunction that we parallelise, still points to the correct conjunction. To ensure that the compiler applies advice correctly, it attempts to identify the parallel conjunction in the procedure's body that the parallelisation advice refers to. It does this by comparing the goal structure, and the calls within the procedure body to the goal structure and calls in the candidate conjunction in the feedback data. It compares calls using the callee's name and the names of some of the variables in the argument list. Not all variables are used as many are named automatically by other compiler transformations, the compiler knows which ones are automatically introduced by other transformations. If any conjunction cannot be found, then the match fails and the compiler skips that piece of advice and issues a warning.

4.7.2 Parallelising children vs ancestors

What happens when we decide that a conjunction that should be parallelised has an ancestor that we decided should also be parallelised? This can happen both with an ancestor in the same procedure (a compound goal in a parallel conjunction that also contains a parallel conjunction) or a call from a parallel conjunction to the current procedure. In either case our options are:

1. parallelise neither,
2. parallelise only the ancestor,
3. parallelise only this conjunction, or
4. parallelise both

The first alternative (parallelise neither) has already been rejected twice, since we concluded that (2) was better than (1) when we decided to parallelise the ancestor, and we concluded that (3) was better than (1) when we decided to parallelise this conjunction.

Currently our system will parallelise both the ancestor and the current conjunction. Our system will not explore parts of the call graph (and therefore conjunctions) when it thinks there is enough parallelism in the procedure's ancestors to occupy all the CPUs. In the future we could choose among the three reasonable alternatives: we could evaluate the speedup from each of them, and just pick the best. This is simple to do when both candidates are in the same procedure. When one of the candidates is in an ancestor call we must also take into account the fact that for each invocation of the ancestor conjunction, we will invoke the current conjunction many times. Therefore we will

incur both the overheads and the benefits of parallelising the current conjunction many times. We will be able to determine the actual number of invocations from the profile.

4.7.3 Parallelising branched goals

Many programs have code that looks like this:

```
( if ... then
    ...,
    expensive_call_1(...),
    ...
else
    ...,
    cheap_call(...),
    ...
),
expensive_call_2(...)
```

If the condition of the if-then-else succeeds only rarely, then the average cost of the if-then-else may be below the threshold of what we consider to be an expensive goal. We therefore would not consider parallelising the top-level conjunction (the conjunction of the if-then-else and `expensive_call_2`); this is correct as its overheads would probably outweigh its benefits.

In these cases we remember the expensive goal within the if-then-else so that when we see `expensive_call_2` we virtually *push* `expensive_call_2` into both branches of the if-then-else and test if it creates profitable parallelism in the branch next to `expensive_call_1`. If we estimate that this will create profitable parallelism then when we give feedback to the compiler we tell the compiler to push `expensive_call_2` just as we have. Then during execution parallelism is only used when the then branch of this if-then-else is executed.

```
( if ... then
    ...,
    expensive_call_1(...),
    ...,
    expensive_call_2(...)
else
    ...,
    cheap_call(...),
    ...,
    expensive_call_2(...)
)
```

This transformation is only applied when it does not involve reordering goals. For example, we do not push `expensive_call_2` past `other_call` in the following code:

```
( if ... then
    ...,
    ( if ... then
        expensive_call_1(...),
```

```
    else
        ...
    ),
    other_call(...),
else
    ...,
    cheap_call(...),
    ...
),
expensive_call_2(...)
```

4.7.4 Garbage collection and estimated parallelism

We described the garbage collector's effects on parallel performance in detail in Section 3.1. In this section we will describe how it relates to automatic parallelism in particular.

Throughout this chapter we have measured the costs of computations as call sequence counts, however this metric does not take into account memory allocations or collection. There is a fixed (and usually small) limit on the number of instructions that the program can execute between increments of the call sequence count (though the limit is program-dependent). There is no such limit on the collector, which can be a problem. Since a construction unification does not involve a call, our profiler considers its cost (in CSCs) to be zero. Yet if the memory allocation required by a construction triggers a collection, then this nominally zero-cost action can actually take as much time as many thousands of CSCs.

The normal way to view the time taken by a collection is to simply distribute it among the allocations, so that one CSC represents the average time taken by the mutator between two calls plus the average amortised cost of the collections triggered by the unifications between those calls. For a sequential program, this view works very well. For a parallel program, it works less well, because the performance of the mutator and the collector scale differently (Section 3.1).

We could take allocations into account in the algorithms above in a couple of different ways. We could create a new unit for time that combines call sequence counts and allocations and use this to determine how much parallelism we can achieve. At first glance allocations take time and parallelising them against one another may be beneficial. Therefore we may wish to arrange parallel conjunctions so that memory allocation is done in parallel. However this may lead to slowdowns when there is too much parallel allocation. Memory can be allocated from the shared memory pool or from thread local pools. Threads will try to allocate memory from their own local memory pool first. This amortises the accesses to the shared memory pool across a number of allocations. Therefore parallelising memory allocation will increase contention on the shared memory pool, slowing the computation down. High allocation rates also correspond to high memory bandwidth usage. Parallel computations with high memory bandwidth demands may exhaust the available bandwidth, which will reduce the amount that parallelism can improve performance. Therefore, if we use memory allocation information when estimating the benefit of parallelisation we must weigh up these factors carefully.

There is also the consideration that with the Boehm-Demers-Weiser [14] a collection stops the world, and the overheads of this stopping scale with the number of CPUs being used. The overheads of stopping include not just the direct costs of the interruption, but also indirect costs,

Version	Sequential		Parallel w/ N Engines			
	not TS	TS	1	2	3	4
Raytracer						
indep	14.7	16.9	16.9 (1.00)	16.9 (1.00)	17.0 (0.99)	17.0 (1.00)
naive	-	-	17.5 (0.97)	13.4 (1.26)	10.4 (1.62)	8.9 (1.90)
overlap	-	-	17.4 (0.97)	13.2 (1.28)	10.5 (1.61)	9.0 (1.88)
Raytracer w/ 16-row chunks						
indep	14.9	17.8	17.9 (1.00)	18.0 (0.99)	18.0 (0.99)	18.1 (0.98)
naive	-	-	17.4 (1.02)	10.2 (1.75)	7.9 (2.26)	6.5 (2.73)
overlap	-	-	17.5 (1.02)	10.2 (1.75)	7.9 (2.27)	6.5 (2.73)
Right recursive dependent Mandelbrot						
indep	15.2	15.1	15.1 (1.01)	15.2 (1.01)	15.2 (1.01)	15.2 (1.01)
naive	-	-	15.2 (1.01)	9.2 (1.66)	5.7 (2.69)	4.1 (3.76)
overlap	-	-	15.2 (1.01)	9.8 (1.56)	5.7 (2.66)	4.0 (3.85)
Matrix multiplication						
indep	5.10	7.69	7.69 (1.00)	3.87 (1.99)	2.60 (2.96)	1.97 (3.90)
naive	-	-	7.69 (1.00)	3.87 (1.98)	2.60 (2.96)	1.97 (3.90)
overlap	-	-	7.69 (1.00)	3.87 (1.99)	2.60 (2.96)	1.97 (3.90)

Table 4.4: Automatic parallelism performance results

such as having to refill the cache after the collector trashes it.

4.8 Performance results

We tested our system on four benchmark programs: the first three are modified versions of the raytracer and mandelbrot program from Chapter 3, the fourth is a matrix multiplication program. The two versions of the raytracer and the mandelbrot program only have dependent parallelism available. Mandelbrot uses the `map_foldl` predicate in Figure 4.2. `map_foldl` is used to iterate over the rows of pixels in the image. Raytracer does not use `map_foldl`, but does use a similar code structure to perform a similar task. This similar structure is not an accident: *many* predicates use this kind of code structure, partly because programmers in declarative languages often use accumulators to make their loops tail recursive. The second raytracer program renders *chunks* of 16 rows at a time in each iteration of its render loop. This is not normally how a programmer would write such a program; we have done this deliberately to aid a discussion below. Matrixmult has abundant independent AND-parallelism.

We ran all four programs with one set of input parameters to collect profiling data, and with a *different* set of input parameters to produce the timing results in the following table. All tests were run on the same system as the benchmarks on Chapter 3, (the system is described on page 46). Each test was executed 40 times, we give the reason for the high number of samples below.

Table 4.4 presents these results. Each group of three rows reports the results for one benchmark. We auto-parallelised each program three different ways: executing expensive goals in parallel only when they are independent ("indep"); even if they are dependent, regardless of how dependencies affect the estimated available parallelism ("naive"); and even if they are dependent, but only if they have good overlap ("overlap"). The next two columns give sequential execution times of each program, the first of these is the runtime of the program when compiled for sequential execution, the second is its runtime when compiled for parallel execution but without enabling

auto-parallelisation. This shows the overhead of support for parallel execution when it does not have any benefits. The last four columns give the runtime in seconds of each of these versions of the program using one to four Mercury engines, with speedups compared to the sequential thread safe version.

Compiling for parallelism but not using it often yields a slowdown, sometimes as high as 50% (for matrixmult). (We observe such slowdowns for other programs as well.) There are two reasons for this. The first reason is that the runtime system, and in particular the garbage collector, must be thread safe. We tested how the garbage collector affected parallel performance extensively in Section 3.1. The results shown in Table 4.4 were collected using one marker thread in the garbage collector. The second reason for a slowdown in thread safe grades comes from how we compile Mercury code. Mercury uses four of the x86_64's six callee-save registers [81] to implement four of the Mercury abstract machine registers[4] (Section 2.2). The four abstract machine registers that use real registers are: the stack pointer, the success instruction pointer (continuation pointer), and the first two general purpose registers. The parallel version of the Mercury system needs to use a real machine register to point to thread-specific data, such as each engine's other abstract machine registers (Section 2.2). On x86_64, this means that only the first abstract general purpose register is implemented using a real machine register.

The parallelised versions running on one Mercury engine get only this slowdown, plus the (small) additional overheads of all the parallel conjunctions which cannot get any parallelism. However, when we move to two, three or four engines, many of the auto-parallelised programs do get speedups.

In mandelbrot and both raytracers, all the parallelism is dependent, which is why indep gets no speedup for them as it refuses to create dependent parallel conjunctions. The non-chunking raytracer achieves speedups of 1.90 and 1.88 with four Mercury engines. We have found two reasons for these poor results. The first reason is that, as we discussed in Section 3.1, the raytracer's performance is limited by how much time it spends in the garbage collector. The second reason is the "right recursion problem" (see below). For mandelbrot, overlap gets a speedup that is as good as one can reasonably expect: 3.85 on four engines. At first glance there appears to be a difference in the performance of the naive and overlap versions of the mandelbrot program. However the feedback data given to the compiler is the same in both cases, so the generated code will be the same. Any apparent difference is due to the high degree of variability in the results, which is why we executed each benchmark 40 times. This variability does not exist in the indep case, so we believe it comes from how dependencies in parallel conjunctions are handled. The performance is probably affected by Both the implementation of futures, and the "right recursion problem" (Section 3.3), for which we introduce a work around in Section 3.5. The work around reorders the conjuncts of a parallel conjunction to transform right recursion into left recursion. However this cannot be done to dependent conjunctions as it would create mode incorrect code. If non-chunking raytracer and mandelbrot do not reach the context limit, the garbage collector needs to scan the large number of stacks created by right recursion. This means that parallel versions of these programs put more demands on the garbage collector than their sequential equivalents. Additionally, the more Mercury engines that are used, the more sparks are stolen and therefore more contexts are created. Dependent parallel right recursive code will not scale to large numbers of processors well. We fix the right recursion problem in Chapter 5, and provide better benchmark

[4]We only use four real registers because we may use only callee-save registers, and we do not want to over-constrain the C compiler's register allocator.

results for explicitly-parallel versions of the non-chunking raytracer, mandelbrot and a dependent version of matrixmult.

The version of the Mercury runtime system that we used in these tests does not behave exactly as described in Section 3.6. In some cases the context limit is not used to reduce the amount of memory consumed by contexts' stacks. This means that right recursive dependent parallel conjunctions can consume more memory (reducing performance), but they can also create more parallel tasks (usually improving performance). This does not affect the analysis of how well automatic parallelism works, it merely makes it difficult to compare these results with results in the other chapters of the dissertation.

We verified the right recursion problem's impact in garbage collection by testing the chunking version of raytracer. This version renders 16 rows of the image on each iteration and therefore creates fewer larger contexts. We can see that chunking raytracer performs much better than the non-chunking raytracer when either naive or overlap parallelism is used. However it performs worse without parallelism, including indep auto-parallelism (which cannot find any candidate parallel conjunctions). We are not sure why it performs worse without parallelism; one idea is that the garbage collector is affected somehow by how and where memory is allocated. These versions spawn many fewer contexts, thus putting much less load on the GC. We know that this speed up is due to the interaction of the right recursion problem and the GC because mandelbrot, which creates just as many contexts but has a much lower allocation rate, already runs very optimally. This shows that program transformations that cause more work to be done in each context are likely to be optimisations, this includes the work in Chapter 5.

The parallelism in the main predicate of matrixmult is independent. Regardless of which parallel cost calculation we used (naive, indep or overlap), the same feedback was sent to the compiler. All versions of matrixmult were parallelised the same way and performed the same. This is normal for such a small example with simple parallelism. (We compare this to a more complicated example of the Mercury compiler below.) In an earlier version of our system, the version that was used in Bone, Somogyi, and Schachte [19], the naive strategy parallelised matrixmult's main loop differently, it included an extra goal inside the parallel conjunct thereby creating a dependent parallel conjunction even though the parallelisation was independent. Therefore in Bone, Somogyi, and Schachte [19], naive performed worse than either indep or overlap.

Most small programs like these benchmarks have only one loop that dominates their runtime. In all four of these benchmarks, and in many others, the naive and overlap methods will parallelise the same loops, and usually the same way; they tend to differ only in how they parallelise code that executes much less often (typically only once) whose effect is lost in the noise.

To see the difference between naive and overlap, we need to look at larger programs. Our standard large test program is the Mercury compiler, which contains 69 conjunctions with two or more expensive goals (goals with a per call cost of at least 10,000csc). Of these, 66 are dependent, and only 34 have an overlap that leads to a predicted local speedup of more than 2%. Our algorithms can thus prevent the unproductive parallelisation of $69 - 34 = 35$ (51%) of these conjunctions. Unfortunately, programs that are large and complex enough to show a performance effect from this saving also tend to have large components that cannot be profitably parallelised with existing techniques, which means that (due to Amdahl's law) our autoparallelisation system cannot yield overall speedups for them yet.

On the bright side, our feedback tool can process the profiles of small programs like these benchmarks in less than a second and in only a minute or two even for much larger profiles. The

extra time taken by the Mercury compiler when it follows the recommendations in feedback files is so small that it is not noticeable.

4.9 Related work

Mercury's strong mode and determinism systems greatly simplify the parallel execution of logic programs. As we discussed in Sections 1.1.2, 2.2 and 2.3, they make it easy to implement dependent AND-parallelism efficiently. The mode system provides complete information allowing us to identify shared variables at compile time.

Most research in parallel logic programming so far has focused on trying to solve the 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 parallelisation has focused on granularity control: reducing the number of parallel tasks while increasing their size [76], and properly accounting for the overheads of parallelism itself [99]. Most of the rest has tried to find opportunities to exploit independent AND-parallelism during the execution of otherwise-dependent conjunctions [26, 87].

Our experience with our feedback tool shows that for Mercury programs, this is far from enough. For most programs, it finds enough conjunctions with two or more expensive conjuncts, but almost all are dependent, and, as we mention in Section 4.8, many of these have too little overlap to be worth parallelising.

We know of only four attempts to estimate the overlap between parallel computations, and two of these are in Mercury.

Tannier [105] attempted to automatically parallelise Mercury programs. Tannier's approach was to use the number of shared variables in a parallel conjunction as an analogue for how dependent the conjunction was. While two conjuncts are indeed less likely to have useful parallel overlap if they have more shared variables, we have found this heuristic too inaccurate to be useful. Conjunctions with only a single variable can significantly vary in their amount of overlap,which we have shown with the examples in this chapter. Analysing overlap properly is important as most parallel conjunctions have very little overlap, making their parallelisation wasteful.

After Tannier's work, we attempted to estimate overlap more accurately in our prior work (Bone [17] [5]). Compared with the work in this chapter, our earlier work performed a much simpler analysis of the parallelism available in dependent parallel conjunctions. It could handle only conjunctions with two conjuncts and a single shared variable. It also did not use the ancestor context specific information provided by the deep profiler or perform the call graph traversal. The call graph traversal is important as it allows us to avoid parallelising a callee when the caller already provides enough parallelism. Most critically our earlier work did not attempt to calculate the costs of recursive calls, and therefore failed to find any parallelism in any of the test programs we used, including the raytracer that we use in this chapter.

Another dependency aware auto-parallelisation effort was in the context of speculative execution in imperative programs. Given two successive blocks of instructions, von Praun, Ceze, and Caşcaval [112] decide whether the second block should be executed speculatively based on the difference between the addresses of two instructions, one that writes a value to a register and one that reads

[5] This earlier work was done as part of my honours project and contributed to the degree of Bachelor of Computer Science Honours at The University of Melbourne.

from that register. This only works when instructions take a bounded time to execute, but in the presence of call instructions this heuristic will be inaccurate.

The most closely related work to ours is that of Pontelli, Gupta, Pulvirenti, and Ferro [92]. They generated parallelism annotations for the ACE AND/OR-parallel system. They recognised the varying amounts of parallelism that may be available in dependent AND-parallel code. This system used, much as we do, estimates of the costs of calls and of the times at which variables are produced and consumed. It produced its estimates through static analysis of the program. This can work for small programs, where the call trees of the relevant calls can be quite small and regular. However, in large programs, the call trees of the expensive calls are almost certain to be both tall and wide, with a huge gulf between best-case and worst-case behaviour. Pontelli et al.'s analysis is a whole program analysis, which can also be troublesome for large programs. Our whole-program analysis covers only the parts of the call graph that are deemed costly enough to be worth exploring, which is another benefit of profiler feedback analysis. Pontelli et al.'s analysis of the variable use times is incomplete when analysing branching code. We recognised the impact that diverging code may have and created coverage profiling so that we could gather information and analyse diverging code accurately [17]. Using profiling data is the only way for an automatic parallelisation system to find out what the *typical* cost and variable use times are. Finally, Pontelli et al.'s overlap calculation is pairwise: it considers the overlap of only two conjuncts at a time. We have found that one must consider the overlap of the whole parallel conjunction, as a delay in the production of a variable in one conjunct can affect the variable production times of other conjuncts. This is also why we attempt to perform a complete search for the most optimal parallelisation. Changes anywhere within a conjunction's execution can affect other parts of the conjunction. For example, the addition of an extra conjunct at the end of the conjunction can create a new shared variable which increases the costs of earlier conjuncts.

Our system's predictions of the likely speedup from parallelising a conjunction are also fallible, since they currently ignore several relevant issues, including cache effects and the effects of bottlenecks such as CPU-memory buses and stop-the-world garbage collection. However, our system seems to be a sound basis for further refinements like these. In the future, we plan to support parallelisation as a specialisation: applying a specific parallelisation only when a predicate is called from a specific parent, grandparent or other ancestor. We also plan to modify our feedback tool to accept several profiling data files, with a priority or (weighted) voting scheme to resolve any conflicts. There is also potential further work in the handling of loops and recursion. We will talk about this in Chapter 7 after we improve the efficiency of some loops in the next chapter.

This chapter is an extended and revised version of Bone et al. [19].

Chapter 5

Loop Control

In the previous chapter we described our system that uses profiling data to automatically parallelise Mercury programs by finding conjunctions with expensive conjuncts that can run in parallel with minimal synchronisation delays. This worked very well in some programs but not as well as we had hoped for others, including the raytracer. This is because the way Mercury must execute dependent conjunctions and the way programmers typically write logic programs are at odds. We introduced this as "the right recursion problem" in Section 3.3.

In this chapter we present a novel program transformation that eliminates this problem in all situations. The transformation has several benefits: First, it reduces peak memory consumption by putting a limit on how many stacks a conjunction will need to have in memory at the same time. Second, it reduces the number of synchronisation barriers needed from one per loop iteration to one per loop. Third, it allows recursive calls inside parallel conjunctions to take advantage of tail recursion optimisation. Finally, it obsoletes the conjunct reordering transformation. Our benchmark results show that our new transformation greatly increases the speedups we can get from parallelising Mercury programs; in one case, it changes no speedup into almost perfect speedup on four cores.

We have written about the problem elsewhere in the dissertation, however we have found that this problem is sometimes difficult to understand. Therefore the introduction section (Section 5.1) briefly describes the problem, providing only the details necessary to understand and evaluate the rest of this chapter. For more details about the problem see Sections 3.2 and 3.3, see also Bone et al. [20], the paper on which this chapter is based. The rest of the chapter is organised as follows. Section 5.2 describes the program transformation we have developed to control memory consumption by loops. Section 5.3 evaluates how our system works in practice on some benchmarks. Section 5.4 describes potential further work, and Section 5.5 concludes with discussion of related work.

5.1 Introduction

The implementation of a parallel conjunction has to execute the first conjunct after spawning off the later conjuncts. For dependent conjunctions, it cannot be done the other way around, because only the first conjunct is guaranteed to be immediately executable: later conjuncts may need to wait for data to be generated by earlier conjuncts. This poses a problem when the last conjunct contains a recursive call:

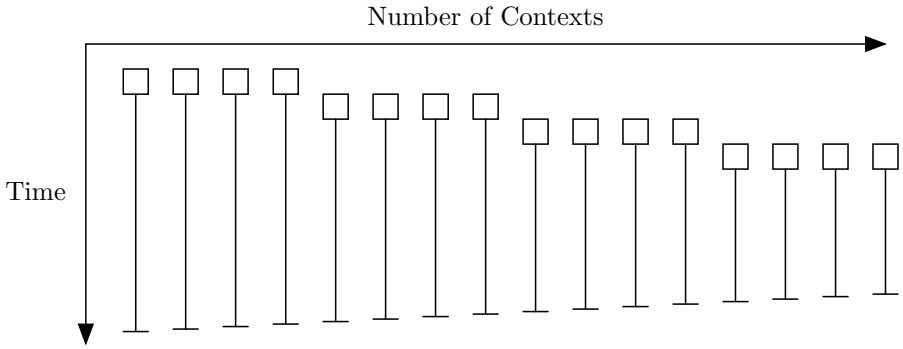


Figure 5.1: Linear context usage in right recursion

- the state of the computation up to this iteration of the loop is stored in the stack used by the original computation's context, whereas
- the state of the computation after this iteration of the loop is stored in the stack used by the spawned off context.

We can continue the computation after the parallel conjunction only if we have both the original stack and the results computed on the spawned-off stack. This means the original stack must be kept in memory until the recursive call is done. However, there is no way to distinguish the first iteration of a loop from the second, third etc, so we must preserve the original stack on *every* iteration. This problem is very common, since in logic programming languages, tail recursive code has long been the preferred way to write a loop. In independent code, we can workaround the issue by reordering the conjuncts in a conjunction (see Section 3.5), but in dependent code this is not possible.

We call this the "right recursion problem" because its effects are at their worst when the recursive call is on the right hand side of a parallel conjunction operator. Unfortunately, this is a natural way to (manually or automatically) parallelise programs that were originally written with tail recursion in mind. Thus, parallelisation often transforms tail recursive sequential computations, which run in constant stack space, into parallel computations that allocate a complete stack for each recursive call and do not free them until the recursive call returns. This means that each iteration effectively requires memory to store an entire stack, not just a stack frame.

Figure 5.1 shows a visualisation of this stack usage. At the top left, four contexts are created to execute four iterations of a loop, as indicated by boxes. Once each of these iterations finishes, its context stays in memory but is suspended, indicated by the long vertical lines. Another four iterations of the loop create another four contexts, and so on. Later, when all iterations of the loop have been executed, each of the blocked contexts resumes execution and immediately exits, indicated by the horizontal line at the bottom of each of the vertical lines.

If we allow the number of contexts, and therefore their stacks, to grow unbounded, then the program will very quickly run out of memory, often bringing the operating system to its knees. This is why we introduced the context limit work around (described in Section 3.3), which can prevent a program from crashing, but which limits the amount of parallel execution.

Our transformation explicitly limits the number of stacks allocated to recursive calls to a small multiple of the number of available processors in the system. This transformation can also be asked to remove the dependency of a parallel loop iteration on the parent stack frame from

```
map_foldl_par(M, F, L, FutureAcc0, Acc) :-
    lc_create_loop_control(LC),
    map_foldl_par_lc(LC, M, F, L, FutureAcc0, Acc).

map_foldl_par_lc(LC, M, F, L, FutureAcc0, Acc) :-
    (
        L = [],
        % The base case.
        wait_future(FutureAcc0, Acc0),
        Acc = Acc0,
        lc_finish(LC)
    ;
        L = [H | T],
        new_future(FutureAcc1),
        lc_wait_free_slot(LC, LCslot),
        lc_spawn_off(LC, LCslot, (
            M(H, MappedH),
            wait_future(FutureAcc0, Acc0),
            F(MappedH, Acc0, Acc1),
            signal_future(FutureAcc1, Acc1),
            lc_join_and_terminate(LCslot, LC)
        )),
        map_foldl_par_lc(LC, M, F, T,
            FutureAcc1, Acc)
    ).
```

Figure 5.2: `map_foldl_par` after the loop control transformation

which it was spawned, allowing the parent frame to be reclaimed before the completion of the recursive call. This allows parallel tail recursive computations to run in constant stack space. The transformation is applied after the automatic parallelisation transformation, so it benefits both manually and automatically parallelised Mercury code.

5.2 The loop control transformation

The main aim of loop control is to set an upper bound on the number of contexts that a loop may use, regardless of how many iterations of the loop may be executed, without limiting the amount of parallelism available. The loops we are concerned about are procedures that we call right recursive: procedures in which the recursive execution path ends in a parallel conjunction whose last conjunct contains the recursive call. A right recursive procedure may be tail recursive, or it may not be: the recursive call could be followed by other code either within the last conjunct, or after the whole parallel conjunction. Programmers have long tended to write loops whose last call is recursive in order to benefit from tail recursion. Therefore, right recursion is very common; most parallel conjunctions in recursive procedures are right recursive.

To guarantee the imposition of an upper bound on the number of contexts created during one of these loops, we associate with each loop a data structure that has a fixed number of slots, and require each iteration of the loop that would spawn off a goal to reserve a slot for the context of each spawned-off computation. This slot is marked as in-use until that spawned-off computation finishes, at which time it becomes available for use by another iteration.

```
typedef struct MR_LoopControl_Struct        MR_LoopControl;
typedef struct MR_LoopControlSlot_Struct    MR_LoopControlSlot;

struct MR_LoopControlSlot_Struct
{
    MR_Context                *MR_lcs_context;
    MR_bool                   MR_lcs_is_free;
};

struct MR_LoopControl_Struct
{
    volatile MR_Integer       MR_lc_outstanding_workers;

    MR_Context* volatile      MR_lc_master_context;
    volatile MR_Lock          MR_lc_master_context_lock;

    volatile MR_bool          MR_lc_finished;

    /*
    ** MR_lc_slots MUST be the last field, since in practice, we treat
    ** the array as having as many slots as we need, adding the size of
    ** all the elements except the first to sizeof(MR_LoopControl) when
    ** we allocate memory for the structure.
    */
    unsigned                  MR_lc_num_slots;
    MR_LoopControlSlot        MR_lc_slots[1];
};
```

Figure 5.3: Loop control structure

This scheme requires us to use two separate predicates: the first sets up the data structure (which we call the *loop control* structure) and the second actually performs the loop. The rest of the program knows only about the first predicate; the second predicate is only ever called from the first predicate and from itself. Figure 5.2 shows what these predicates look like. In Section 5.2.1, we describe the loop control structure and the operations on it; in Section 5.2.2, we give the algorithm that does the transformation; in Section 5.2.3, we discuss its interaction with tail recursion optimisation.

5.2.1 Loop control structures

The loop control structure, shown in Figure 5.3, contains the following fields:

`MR_lc_slots` is an array of slots, each of which contains a boolean and a pointer. The boolean says whether the slot is free, and if it is not, the pointer points to the context that is currently occupying it. When the occupying context finishes, the slot is marked as free again, but the pointer remains in the slot to make it easier (and faster) for the next computation that uses that slot to find a free context to reuse. Therefore we cannot encode the boolean in the pointer being `NULL` or non-`NULL`. Although this is the most significant field in the structure it is last so that the array can be stored with the data structure, and an extra memory dereference can be avoided. This is the last field in the structure even though it is the most significant field.

`MR_lc_num_slots` stores the number of slots in the array.

`MR_lc_outstanding_workers` is the count of the number of slots that are currently in use.

`MR_lc_master_context` is a possibly null pointer to the *master* context, the context that created this structure, and the context that will spawn off all of the iterations. This slot will point to the master context whenever it is sleeping, and will be `NULL` at all other times.

`MR_lc_master_context_lock` a mutex that is used to protect access to MR_lc_master_context. The other fields are protected using atomic instructions; we will describe them when we show the code for the loop control procedures.

`MR_lc_finished` A boolean flag that says whether the loop has finished or not. It is initialised to false, and is set to true as the first step of the `lc_finish` operation.

The finished flag is not strictly needed for the correctness of the following operations, but it can help the loop control code cleanup at the end of a loop more quickly. In the following description of the primitive operations on the loop control structure, `LC` is a reference to the whole of a loop control structure, while `LCslot` is an index into the array of slots stored within `LC`.

`LC = lc_create_loop_control()` This operation creates a new loop control structure, and initialises its fields. The number of slots in the array in the structure will be a small multiple of the number of cores in the system. The multiplier is configurable by setting an environment variable when the program is run.

`LCslot = lc_wait_free_slot(LC)` This operation tests whether `LC` has any free slots. If it does not, the operation suspends until a slot becomes available. When some slots are available, either immediately or after a wait, the operation chooses one of the free slots, marks it in use, fills in its context pointer and returns its index. It can get the context to point to from the last previous user of the slot, from a global list of free contexts, (in both cases it gets contexts which have been used previously by computations that have terminated earlier), or by allocating a new context (which typically happens only soon after startup).

`lc_spawn_off(LC, LCslot, CodeLabel)` This operation sets up the context in the loop control slot, and then puts it on the global runqueue, where any engine looking for work can find it. Setup of the context consists of initialising the context's parent stack pointer to point to its master's stack frame, and the context's resume instruction pointer to the value of `CodeLabel`.

`lc_join_and_terminate(LC, LCslot)` This operation marks the slot named by `LCslot` in `LC` as available again. It then terminates the context executing it, allowing the engine that was running it to look for other work.

`lc_finish(LC)` This operation is executed by the master context when we know that this loop will not spawn off any more work packages. It suspends its executing context until all the slots in `LC` become free. This will happen only when all the goals spawned off by the loop have terminated. This is necessary to ensure that all variables produced by the recursive call that are *not* signalled via futures have in fact had values generated for them. A variable generated by a parallel conjunct that is consumed by a later parallel conjunct will be signalled via a future, but if the variable is consumed only by code after the parallel conjunction, then it is made available by writing its value directly in its stack slot. Therefore such variables can

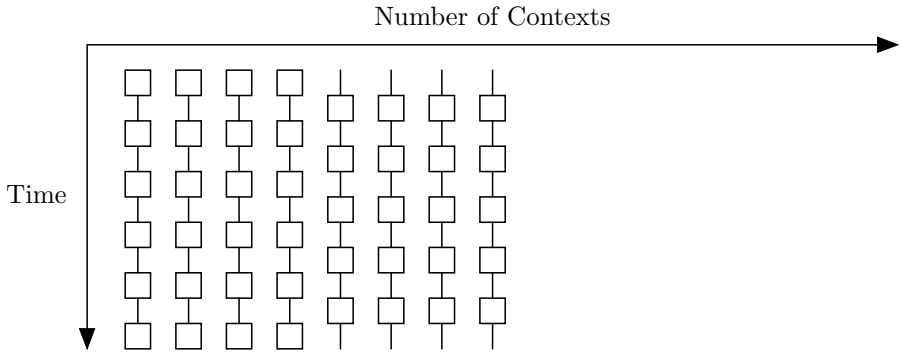


Figure 5.4: Loop control context usage

exist only if the original predicate had code after the parallel conjunction; for example, map over lists must perform a construction after the recursive call. This barrier is the only barrier in the loop and it is executed just once; in comparison, the normal parallel conjunction execution mechanism executes one barrier in each iteration of the loop.

See Figure 5.2 for an example of how we use these operations. Note in particular that in this transformed version of `map_foldl`, the spawned-off computation contains the calls to `M` and `F`, with the main thread of execution making the recursive call. This is the first step in preserving tail recursion optimisation.

Figure 5.4 shows a visual representation of context usage when using loop control; it should be compared with Figure 5.1. As before, this is how contexts are likely to be used on a four processor system when using a multiplier of two so that eight slots are used; minor differences in the execution times of each task and similar variables will mean that no execution will look as regular as in the figures. In the Figure 5.4, we can see that a total of eight contexts are created and four are in use at a time. When the loop begins, the master thread performs a few iterations, executing `lc_wait_free_slot` and `lc_spawn_off`. This creates all the contexts and adds them to the runqueue, but only the first four contexts can be executed as there are only four processors. Once those contexts finish their work, they execute `lc_join_and_terminate`. Each call to `lc_join_and_terminate` marks the relevant slot in the loop control structure as free, allowing the master context to spawn off more work using the free slot. Meanwhile, the other four contexts are now able to execute their work. This continues until the loop is finished, at which point `lc_finish` releases all the contexts.

5.2.2 The loop control transformation

Our algorithm for transforming procedures to use loop control is shown in Algorithms 5.1, 5.2 and 5.3.

Algorithm 5.1 shows the top level of the algorithm, which is mainly concerned with testing whether the loop control transformation is applicable to a given procedure, and creating the interface procedure if it is.

We impose conditions (1) and (2) because we need to ensure that every loop we start for `OrigProc` is finished exactly once, by the call to `lc_finish` we insert into its base cases. If `OrigProc` is mutually recursive with some other procedure, then the recursion may terminate in a base case of the other procedure, which our algorithm does not transform. Additionally, if

Algorithm 5.1 The top level of the transformation algorithm

procedure LOOP_CONTROL_TRANSFORM(*OrigProc*)
  OrigGoal ← body(*OrigProc*)
  RecParConjs ← set of parallel conjunctions in *OrigGoal* that contain recursive calls
  if
    OrigProc is directly but not mutually recursive (HO calls are assumed not to create recursion), and ▷ (1)
    OrigGoal has at most one recursive call on all possible execution paths, and ▷ (2)
    OrigGoal has determinism `det`, and ▷ (3)
    no recursive call is within a disjunction, a scope that changes the determinism of a goal, a negation, or the condition of a if-then-else, and ▷ (4)
    no member of *RecParConjs* is nested within another parallel conjunction, and ▷ (5)
    every recursive call is inside the last conjunct of a member of *RecParConjs*, and ▷ (6)
    every execution path through one of these last conjuncts makes exactly one recursive call ▷ (7)
  then
    LC ← create_new_variable()
    LCGoal ← the call 'lc_create_loop_control(*LC*)'
    LoopProcName ← a new unique predicate name
    OrigArgs ← arg_list(*OrigProc*)
    LoopArgs ← [*LC*] ++ *OrigArgs*
    CallLoopGoal ← the call 'LoopProcName(*LoopArgs*)'
    NewProcBody ← the conjunction '*LCGoal*, *CallLoopGoal*'
    NewProc ← *OrigProc* with its body replaced by *NewProcBody*
    LoopGoal ← create_loop_goal(*OrigGoal*, *OrigProcName*, *LoopProcName*, *RecParConjs*, *LC*)
    LoopProc ← new_procedure $\left(\begin{array}{c} \mathit{LoopProcName}\texttt{(}\mathit{LoopArgs}\texttt{) :-} \\ \mathit{LoopGoal}\texttt{.} \end{array}\right)$
    NewProcs ← [*NewProc*, *LoopProc*]
  else
    NewProcs ← [*OrigProc*]
  return *NewProcs*

Algorithm 5.2 Algorithm for transforming the recursive cases

procedure CREATE_LOOP_GOAL(*OrigGoal*, *OrigProcName*, *LoopProcName*, *RecParConjs*, *LC*)
  LoopGoal ← *OrigGoal*
  for *RecParConj* ∈ *RecParConjs* **do**
    RecParConj has the form '$Conjunct_1$ & ... & $Conjunct_n$' for some n
    for $i \leftarrow 1$ to $n - 1$ **do** ▷ This does not visit the last goal in *RecParConj*
      $LCSlot_i$ ← create_new_variable()
      $WaitGoal_i$ ← the call 'lc_wait_free_slot(*LC*, $LCSlot_i$)'
      $JoinGoal_i$ ← the call 'lc_join_and_terminate(*LC*, $LCSlot_i$)'
      $SpawnGoal_i$ ← a goal that spawns off the sequential conjunction
          '$Conjunct_i, JoinGoal_i$' as a work package
      $Conjunct_i$ ← the sequential conjunction '$WaitGoal_i, SpawnGoal_i$'
    $Conjunct'_n \leftarrow Conjunct_n$
    for each recursive call *RecCall* in $Conjunct'_n$ **do**:
      RecCall has the form '*OrigProcName*(*Args*)'
      RecCall′ ← the call '*LoopProcName*([*LC*] ++ *Args*)'
      replace *RecCall* with *RecCall′* in $Conjunct'_n$
    Replacement ← the flattened form of the sequential conjunction
        '$Conjunct'_1, \ldots, Conjunct'_n$'
    replace *RecParConj* in *LoopGoal* with *Replacement*
  LoopGoal′ ← put_barriers_in_base_cases(*LoopGoal*, *RecParConjs*, *LoopProcName*, *LC*)
  return *LoopGoal′*

`OrigProc` has some execution path on which it calls itself twice, then the second call may continue executing loop iterations after a base case reached through the first call has finished the loop.

We impose conditions (3) and (4) because the Mercury implementation does not support the parallel execution of code that is not deterministic. We do not want a recursive call to be called twice because some code between the entry point of `OrigProc` and the recursive call succeeded twice, and we do not want a recursive call to be backtracked into because some code between the recursive call and the exit point of `OrigProc` has failed. These conditions prevent both of those situations.

We impose condition (5) because we do not want another instance of loop control, or an instance of the normal parallel conjunction execution mechanism, to interfere with this instance of loop control.

We impose condition (6) for two reasons. First, the structure of our transformation requires right recursive code: we could not terminate the loop in base case code if the call that lead to that code was followed by any part of an earlier loop iteration. Second, allowing recursion to sometimes occur outside the parallel conjunctions we are trying to optimise would unnecessarily complicate the algorithm. (We do believe that it should be possible to extend our algorithm to handle recursive calls made outside of parallel conjunctions.)

We impose condition (7) to ensure that our algorithm for transforming base cases (Algorithm 5.3) does not have to process goals that have already been processed by our algorithm for transforming recursive calls (Algorithm 5.2).

If the transformation is applicable, we apply it. The transformed original procedure has only one purpose: to initialise the loop control structure. Once that is done, it passes a reference to that structure to `LoopProc`, the procedure that does the actual work.

The argument list of `LoopProc` is the argument list of `OrigProc` plus the `LC` variable that holds

the reference to the loop control structure. The code of `LoopProc` is derived from the code of `OrigProc`. Some execution paths in this code include a recursive call; some do not. The execution paths that contain a recursive call are transformed by Algorithm 5.2; the execution paths that do not are transformed by Algorithm 5.3.

We start with Algorithm 5.2. Due to condition (6), every recursive call in `OrigGoal` will be inside the last conjunct a parallel conjunction, and the main task of `create_loop_goal` is to iterate over and transform these parallel conjunctions. (It is possible that some parallel conjunctions do not contain recursive calls; `create_loop_goal` will leave these untouched.)

The main aim of the loop control transformation is to limit the number of work packages spawned off by the loop at any one time, in order to limit memory consumption. The goals we want to spawn off are all the conjuncts before the final recursive conjunct. (Without loop control, we would spawn off all the *later* conjuncts.) The first half of the main loop in `create_loop_goal` therefore generates code that creates and makes available each work package only after it obtains a slot for it in the loop control structure, waiting for a slot to become available if necessary. We make the spawned-off computation free that slot when it finishes.

To implement the spawning off process, we extended the internal representation of Mercury goals with a new kind of scope. The only one shown in the abstract syntax in Figure 2.1 was the existential quantification scope (*some* $[X_1, \ldots, X_n]$ G), but the Mercury implementation had several other kinds of scopes already, though none of those are relevant for this dissertation. We call the new kind of scope the spawn-off scope, and we make $SpawnGoal_i$ be a scope goal of this kind. When the code generator processes such scopes, it

- generates code for the goal inside the scope (which will end with a call to `lc_join_and_terminate`),
- allocates a new label,
- puts the new label in front of that code,
- puts this labelled code aside so that later it can be added to the end of the current procedure's code, and
- inserts into the instruction stream a call to `lc_spawn_off` that specifies that the spawned-off computation should start execution at the label of the set-aside code. The other arguments of `lc_spawn_off` come from the scope kind.

Since we allocate a loop slot `LCSlot` just before we spawn off this computation, waiting for a slot to become available if needed, and free the slot once this computation has finished executing, the number of computations that have been spawned-off by this loop and which have not yet been terminated cannot exceed the number of slots in the loop control structure.

The second half of the main loop in `create_loop_goal` transforms the last conjunct in the parallel conjunction by locating all the recursive calls inside it and modifying them in two ways. The first change is to make the call actually call the loop procedure, not the original procedure, which after the transformation is non-recursive; the second is to make the list of actual parameters match the loop procedure's formal parameters by adding the variable referring to the loop control structure to the argument list. Due to condition (6), there can be no recursive call in `OrigGoal` that is left untransformed when the main loop of `create_loop_goal` finishes.

In some cases, the last conjunct may simply *be* a recursive call. In some other cases, the last conjunct may be a sequential conjunction consisting of some unifications and/or some non-recursive

Algorithm 5.3 Algorithm for transforming the base cases

```
procedure PUT_BARRIERS_IN_BASE_CASES(LoopGoal, RecParConjs, LoopProcName, LC)
    if LoopGoal is a parallel conjunction in RecParConjs then                ▷ case 1
        LoopGoal' ← LoopGoal
    else if there no call to LoopProcName in LoopGoal then                   ▷ case 2
        FinishGoal ← the call 'lc_finish(LC)'
        LoopGoal' ← the sequential conjunction 'LoopGoal, FinishGoal'
    else                                                                     ▷ case 3
        switch goal_type(LoopGoal) do
            case 'ite(C, T, E)'
                T' ← put_barriers_in_base_cases(T, RecParConjs, LoopProcName, LC)
                E' ← put_barriers_in_base_cases(E, RecParConjs, LoopProcName, LC)
                LoopGoal' ← 'ite(C, T', E')'
            case 'switch(V, [Case_1, ..., Case_N])'
                for i ← 1toN do
                    Case_i ← 'case(FunctionSymbol_i, Goal_i)'
                    Goal'_i ← put_barriers_in_base_cases(Goal_i, RecParConjs, LoopProcName,
                              LC)
                    Case'_i ← 'case(FunctionSymbol_i, Goal'_i)'
                LoopGoal' ← 'switch(V, [Case'_1, ..., Case'_N])'
            case 'Conj_1, ... Conj_N'                                        ▷ Sequential conjunction
                i ← 1
                while Conj_i does not contain a call to LoopProcName do
                    i ← i + 1
                end while
                Conj'_i ← put_barriers_in_base_cases(Conj_i, RecParConjs, LoopProcName, LC)
                LoopGoal' ← LoopGoal with Conj_i replaced with Conj'_i
            case 'some(Vars, SubGoal)'                                       ▷ Existential quantification
                SubGoal' ← put_barriers_in_base_cases(SubGoal, RecParConjs,
                           LoopProcName, LC)
                LoopGoal' ← 'some(Vars, SubGoal')'
            case a call 'ProcName(Args)'
                if ProcName = OrigProcName then
                    LoopGoal' ← the call 'LoopProcName([LC] ++ Args)'
                else
                    LoopGoal' ← LoopGoal
    return LoopGoal'
```

calls as well as a recursive call, with the unifications and non-recursive calls usually constructing and computing some of the arguments of the recursive call. And in yet other cases, the last conjunct may be an if-then-else or a switch, possibly with other if-then-elses and/or switches nested inside them. In all these cases, due to condition (7), the last parallel conjunct will execute exactly one recursive call on all its possible execution paths.

The last task of `create_loop_goal` is to invoke the `put_barriers_in_base_cases` function that is shown in Algorithm 5.3 to transform the base cases of the goal that will later become the body of `LoopProc`. This function recurses on the structure of `LoopGoal`, as updated by the main loop in Algorithm 5.2.

When `put_barriers_in_base_cases` is called, its caller knows that `LoopGoal` may contain the already processed parallel conjunctions (those containing recursive calls), it may contain base cases, or it may contain both. The main if-then-else in `put_barriers_in_base_cases` handles each of these situations in turn.

If `LoopGoal` is a parallel conjunction that is in `RecParConjs`, then the main loop of `create_loop_goal` has already processed it, and due to condition (7), this function does not need to touch it. Our objective in imposing condition (7) was to make this possible.

If, on the other hand, `LoopGoal` contains no call to `LoopProc`, then it did not have any recursive calls in the first place, since (due to condition (6)) they would all have been turned into calls to `LoopProc` by the main loop of `create_loop_goal`. Therefore this goal either *is* a base case of `LoopProc`, or it is part of a base case. In either case, we add a call to `lc_finish(`LC`)` after it. In the middle of the correctness argument below, we will discuss why this is the right thing to do.

If both those conditions fail, then `LoopGoal` definitely contains some execution paths that execute a recursive call, and may also contain some execution paths that do not. What we do in that case (case 3) depends on what kind of goal `LoopGoal` is.

If `LoopGoal` is an if-then-else, then we know from condition (4) that any recursive calls in it must be in the then part or the else part, and by definition the last part of any base case code in the if-then-else must be in one of those two places as well. We therefore recursively process both the then part and the else part. Likewise, if `LoopGoal` is a switch, some arms of the switch may execute a recursive call and some may not, and we therefore recursively process all the arms. For both if-then-elses and switches, if the possible execution paths inside them do not involve conjunctions, then the recursive invocations of `put_barriers_in_base_cases` will add a call to `lc_finish` at the end of each execution path that does not make recursive calls.

What if those execution paths do involve conjunctions? If `LoopGoal` is a conjunction, then we recursively transform the first conjunct that makes recursive calls, and leave the conjuncts both before and after it (if any) untouched. There is guaranteed to be at least one conjunct that makes a recursive call, because if there were not, the second condition would have succeeded, and we would never get to the switch on the goal type. We also know at most one conjunct makes a recursive call. If more than one did, then there would be an execution path through those conjuncts that would make more than one recursive call, then condition (2) would have failed, and the loop control transformation would not be applicable.

Correctness argument. One can view the procedure body, or indeed any goal, as a set of execution paths that diverge from each other in if-then-elses and switches (on entry to the then or else parts and the switch arms respectively) and then converge again (when execution continues after the if-then-else or switch). Our algorithm inserts calls to `lc_finish` into the procedure body at all the places needed to ensure that every non-recursive execution path executes such a call

exactly once, and does so after the last goal in the non-recursive execution path that is not shared with a recursive execution path. These places are the ends of non-recursive then parts whose corresponding else parts are recursive, the ends of non-recursive else parts whose corresponding then parts are recursive, and the ends of non-recursive switch arms where at least one other switch arm is recursive. Condition (4) tests for recursive calls in the conditions of if-then-elses (which are rare in any case) specifically to make this correctness argument possible.

Note that for most kinds of goals, execution cannot reach case 3 in Algorithm 5.3. Unifications are not parallel conjunctions and cannot contain calls, so if `LoopGoal` is a unification, we will execute case 2. If `LoopGoal` is a first order call, we will also execute case 2, due to condition (6) in Algorithm 5.1, all recursive calls are inside parallel conjunctions; since case 1 does not recurse, we never get to those recursive calls. `LoopGoal` might be a higher order call. Although a higher order call might create mutual recursion, any call to `OrigProc` will create a new loop control object and execute its loop independently. This may not be optimal, but it will not cause the transformation to create invalid code or create code that performs worse than Mercury's normal parallel execution mechanics. Therefore we treat higher order calls the same way that we treat plain calls to procedures other than `OrigProc`. If `LoopGoal` is a parallel conjunction, then it is either in `RecParConj`, in which case we execute case 1, or (due to condition (5)) it does not contain any recursive calls, in which case we execute case 2. Condition (4) also guarantees that we will execute case 2 if `LoopGoal` is a disjunction, negation, or a quantification that changes the determinism of a goal by cutting away (indistinguishable) solutions. The only other goal type for which execution may get to case 3 are quantification scopes that have no effect on the subgoal they wrap, whose handling is trivial.

We can view the execution of a procedure body that satisfies condition (2) and therefore has at most one recursive call on every execution path as a descent from a top level invocation from another procedure to a base case, followed by ascent back to the top. During the descent, each invocation of the procedure executes the part of a recursive execution path up to the recursive call; during the ascent, after each return we execute the part of the chosen recursive execution path after the recursive call. At the bottom, we execute exactly one of the non-recursive execution paths.

In our case, conditions (5) and (6) guarantee that all the goals we spawn off will be spawned off during the descent phase. When we get to the bottom and commit to a non-recursive execution path through the procedure body, we know that we will not spawn off any more goals, which is why we can invoke `lc_finish` at that point. We can call `lc_finish` at any point in `LoopGoal` that is after the point where we have committed to a non-recursive execution path, and before the point where that non-recursive execution path joins back up with some recursive execution paths.

The code at case 2 puts the call to `lc_finish` at the last allowed point, not the first, or a point somewhere in the middle. We chose to do this because after the code executing `LoopProc` has spawned off one or more goals one level above the base case, we expect that other processors will be busy executing those spawned off goals for rather longer than it takes this processor to execute the base case. By making this core do as much useful work as possible before it must suspend to wait for the spawned-off goals to finish, we expect to reduce the amount of work remaining to be done after the call to `lc_finish` by a small but possibly useful amount. `lc_finish` returns *after* all the spawned-off goals have finished, so any code placed after it (such as if `lc_finish` were placed at the first valid point) would be executed sequentially after the loop; where it would definitely add to the overall runtime. Therefore, we prefer to place `lc_finish` as late as possible, so that

this code occurs before `lc_finish` and is executed in parallel with the rest of the loop, where it may have no effect on the overall runtime of the program; it will just put what would otherwise be dead time to good use.

We must of course be sure that every loop, and therefore every execution of any base case of `LoopGoal`, will call `lc_finish` exactly once: no more, no less. (It should be clear that our transformation never puts that call on an execution path that includes a recursive call.) Now any non-recursive execution path through `LoopGoal` will share a (possibly empty) initial part and a (possibly empty) final part with some recursive execution paths. On any non-recursive execution path, `put_barriers_in_base_cases` will put the call to `lc_finish` just before the first point where that path rejoins a recursive execution path. Since `LoopProc` is `det` (condition (3)), all recursive execution paths must consist entirely of `det` goals and the conditions of if-then-elses, and (due to condition (4)) cannot go through disjunctions. The difference between a non-recursive execution path and the recursive path it rejoins must be either that one takes the then part of an if-then-else and the other takes the else part, or that they take different arms of a switch. Such an if-then-else or switch must be `det`: if it were `semidet`, `LoopProc` would be too, and if it were `nondet` or `multi`, then its extra solutions could be thrown away only by an existential quantification that quantifies away all the output variables of the goal inside it. However, by condition (4), the part of the recursive execution path that distinguishes it from a non-recursive path, the recursive call itself cannot appear inside such scopes. This guarantees that the middle part of the non-recursive execution path, which is not part of either a prefix or a suffix shared with some recursive paths, must also be `det` overall, though it may have nondeterminism inside it. Any code put after the second of these three parts of the execution path (shared prefix, middle, shared suffix), all three of which are `det`, is guaranteed to be executed exactly once.

5.2.3 Loop control and tail recursion

When a parallel conjunction spawns off a conjunct as a work package that other cores can pick up, the code that executes that conjunct has to know where it should pick up its inputs, where it should put its outputs, and where it should store its local variables. All the inputs come from the stack frame of the procedure that executes the parallel conjunction, and all the outputs go there as well, so the simplest solution, and the one used by the Mercury system, is for the spawned-off conjunct to do all its work in the exact same stack frame. Normally, Mercury code accesses stack slots via offsets from the standard stack pointer. Spawned-off code accesses stack slots using a special Mercury abstract machine register called the parent stack pointer, which the code that spawns off goals sets up to point to the stack frame of the procedure doing the spawning. That same spawning-off code sets up the normal stack pointer to point to the start of the stack in the context executing the work package, so any calls made by the spawned-off goal will allocate their stack frames in that stack, but the spawned-off conjunct will use the original frame in the stack of the parent context.

This approach works, and is simple to implement: the code generator generates code for spawned-off conjuncts normally, and then just substitutes the base pointer in all references to stack slots. However, it does have an obvious drawback: until the spawned-off computation finishes execution, it may make references to the stack frame of the parallel conjunction, whose space therefore cannot be reused until then. This means that even if a recursive call in the last conjunct of the parallel conjunction happens to be a tail call, it cannot have the usual tail call optimisation

applied to it.

Before this work, this did not matter, because the barrier synchronisation needed at the end of the parallel conjunction (`MR_join_and_continue`), which had to be executed at every level of recursion except the base case, prevented tail recursion optimisation anyway. However, the loop control transformation eliminates that barrier, replacing it with the single call to `lc_finish` in the base case. So now this limitation *does* matter in cases where all of the recursive calls in the last conjunct of a parallel conjunction are tail recursive.

If at least one call is not tail recursive, then it prevents the reuse of the original stack frame, so our system will still follow the scheme described above. However, if they all are, then our system can now be asked to follow a different approach. The code that spawns off a conjunct will allocate a frame at the start of the stack in the child context, and will copy the input variables of the spawned-off conjunct into it. The local variables of the spawned-off goal will also be stored in this stack frame. The question of where its output variables are stored is moot: there cannot *be* any output variables whose stack slots would need to be assigned to.

The reason this is true has to do with the way the Mercury compiler handles synchronisation between parallel conjuncts. Any variable whose value is generated by one parallel conjunct and consumed by one or more other conjuncts in that conjunction will have a future created for it (Section 2.3). The generating conjunct, once it has computed the value of the variable, will execute `future_signal/2` on the variable's future to wake up any consumers that may be waiting for the value of this variable. Those consumers will get the value of the original variable from the future, and will store that value in a variable that is local to each consumer. Since futures are always stored on the heap, the communication of bindings from one parallel conjunct to another does *not* go through the stack frame.

A variable whose value is generated by a parallel conjunct and is consumed by code after the parallel conjunction does need to have its value put into its stack slot, so that the code after the parallel conjunction can find it. However, if all the recursive calls in the last conjunct are in fact tail calls, then by definition there can be no code after the parallel conjunction. Since neither code later *in* the parallel conjunction, nor code *after* the parallel conjunction, requires the values of variables generated by a conjunct to be stored in the original stack frame, storing it in the spawned-off goal's child stack frame is good enough.

In our current system, the stack frame used by the spawned-off goal has exactly the same layout as its parent, the spawning-off goal. This means that in general, both the parent and child stack frames will have some unused slots, slots used only in the *other* stack frame. This is trivial to implement, and we have not found the wasted space to be a problem. This may be because we have mostly been working with automatically parallelised programs, and our automatic parallelisation tools put much effort into granularity control (Chapter 4): the rarer spawning-off a goal is, the less the effect of any wasted space. However, if we ever find this to be an issue, squeezing the unused stack slots out of each stack frame would not be difficult.

5.3 Performance evaluation

We have benchmarked our system with four different programs, three of which were used in earlier chapters. All these programs use explicit parallelism.

mandelbrot uses dependent parallelism using `map_foldl` from Figure 2.7.

matrixmult multiplies two large matrices. It computes the rows of the result in parallel.

raytracer uses dependent parallelism and is tail recursive. Like mandelbrot, it renders the rows of the generated image in parallel, but it does not use `map_foldl`.

spectralnorm was donated by Chris King[1]. It computes the eigenvalue of a large matrix using the power method. It has two parallel loops, both of which are executed multiple times. Therefore, the parallelism available in spectralnorm is very fine-grained. Chris' original version uses dependent parallelism.

All these benchmarks have dependent AND-parallelism, but for two of the benchmarks, matrixmult and spectralnorm, we have created versions that use independent AND-parallelism as well. The difference between the dependent and independent versions is just the location of a unification that constructs a cell from the results of two parallel conjuncts: the unification is outside the parallel conjunction in the independent versions, while it is in the last parallel conjunct in the dependent versions. The sequential versions of mandelbrot and raytracer can both use tail call optimisation to run in constant stack space. This is not possible in the other programs without the last call modulo constructor (LCMC) optimisation [94]. LCMC is not supported with tail recursion so we have not used it.

We found that the independent spectralnorm benchmark performed poorly due to its fine granularity and the overhead of notification. Therefore, we compiled the runtime system in such a way so that notifications were not used to communicate the availability of sparks (Section 3.6). Instead Mercury engines were configured to poll each other to find sparks to execute. This change only affects the non-loop control tests. We ran each test of each program twenty times.

Tables 5.1 and 5.2 presents our memory consumption and timing results respectively. In both tables, the columns list the benchmark programs, while the rows show the different ways the programs can be compiled and executed.

In Table 5.1, each box has two numbers. The first reports the maximum number of contexts alive at the same time, while the second reports the maximum number of megabytes ever used to store the stacks of these contexts. In Table 5.2, each box has three numbers. The first is the execution time of that benchmark in seconds when it is compiled and executed in the manner prescribed by the row. The second and third numbers (the ones in parentheses) show respectively the speedup this time represents over the sequential version of the benchmark (the first row), and over the base parallel version (the second row). Some of the numbers are affected by rounding.

In both tables, the first row compiles the program without using any parallelism at all, asking the compiler to automatically convert all parallel conjunctions into sequential conjunctions. Obviously, the resulting program will execute on one core.

The second row compiles the program in a way that prepares it for parallel execution, but it still asks the compiler to automatically convert all parallel conjunctions into sequential conjunctions. The resulting executables will differ from the versions in the first row in two main ways. First, they will incur some overheads that the versions in the first row do not, overheads that are needed to support the possibility of parallel execution. The most important of these overheads is that, as described in Chapters 3 and 4, the runtime reserves a hardware register to point to thread specific data and compiling the garbage collector and the rest of the runtime system for thread safety. These overheads lead to slowdowns in most cases, even when using a single core for user code.

[1]Chris' version was published at `http://adventuresinmercury.blogspot.com/search/label/parallelization`.

	mandelbrot		mmult-depi		mmult-indep		raytracer		spectral-dep		spectral-indep	
seq	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62
par, no &	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62	1	0.62
par, &, 1c, nolc, c128	1	0.62	1	0.62	1	0.62	1	0.62	1	1.12	1	1.12
par, &, 1c, nolc, c512	1	0.62	1	0.62	1	0.62	1	0.62	1	1.12	1	1.12
par, &, 1c, lc1	2	1.25	2	1.25	n/a	n/a	2	1.25	2	1.75	n/a	n/a
par, &, 1c, lc2	3	1.88	3	1.88	n/a	n/a	3	1.88	3	2.38	n/a	n/a
par, &, 1c, lc4	5	3.12	5	3.12	n/a	n/a	5	3.12	5	3.62	n/a	n/a
par, &, 2c, nolc, c128	256	160.00	256	160.00	2	1.25	256	160.00	256	160.50	2	1.75
par, &, 2c, nolc, c512	506	316.78	1024	640.00	2	1.25	1024	640.03	1024	640.50	2	1.75
par, &, 2c, lc1	3	1.88	3	1.88	n/a	n/a	4	2.50	3	2.38	n/a	n/a
par, &, 2c, lc2	5	3.12	5	3.12	n/a	n/a	6	3.75	5	3.62	n/a	n/a
par, &, 2c, lc4	9	5.62	9	5.62	n/a	n/a	10	6.25	9	6.12	n/a	n/a
par, &, 3c, nolc, c128	384	240.00	384	240.00	3	1.88	384	240.00	384	240.50	3	2.38
par, &, 3c, nolc, c512	518	323.94	1183	739.53	3	1.88	1201	750.62	1536	960.50	3	2.38
par, &, 3c, lc1	4	2.50	4	2.50	n/a	n/a	5	3.12	4	3.00	n/a	n/a
par, &, 3c, lc2	7	4.38	7	4.38	n/a	n/a	8	5.00	7	4.88	n/a	n/a
par, &, 3c, lc4	13	8.12	13	8.12	n/a	n/a	14	8.75	13	8.62	n/a	n/a
par, &, 4c, nolc, c128	511	319.38	512	320.00	4	2.50	512	320.00	512	320.50	4	3.00
par, &, 4c, nolc, c512	529	330.78	1201	750.62	4	2.50	1201	750.62	2048	1280.50	4	3.00
par, &, 4c, lc1	5	3.12	5	3.12	n/a	n/a	6	3.75	5	3.62	n/a	n/a
par, &, 4c, lc2	9	5.62	9	5.62	n/a	n/a	10	6.25	9	6.12	n/a	n/a
par, &, 4c, lc4	17	10.62	17	10.62	n/a	n/a	18	11.25	17	11.12	n/a	n/a

Table 5.1: Peak number of contexts used, and peak memory usage for stacks, measured in megabytes

	mandelbrot	mmult-dep	mmult-indep	raytracer	spectral-dep	spectral-indep
seq	15.25 (1.00, 0.99)	5.12 (1.00, 1.50)	5.12 (1.00, 1.50)	17.93 (1.00, 1.19)	12.76 (1.00, 1.12)	12.76 (1.00, 1.12)
par, no &	15.17 (1.01, 1.00)	7.70 (0.66, 1.00)	7.70 (0.66, 1.00)	21.27 (0.84, 1.00)	14.33 (0.89, 1.00)	14.33 (0.89, 1.00)
par, &, 1c, nolc, c128	15.24 (1.00, 1.00)	7.71 (0.66, 1.00)	7.70 (0.66, 1.00)	21.20 (0.85, 1.00)	14.51 (0.88, 0.99)	14.38 (0.89, 1.00)
par, &, 1c, nolc, c512	15.22 (1.00, 1.00)	7.71 (0.66, 1.00)	7.70 (0.66, 1.00)	21.23 (0.84, 1.00)	14.50 (0.88, 0.99)	14.38 (0.89, 1.00)
par, &, 1c, lc1	15.20 (1.00, 1.00)	7.71 (0.66, 1.00)	n/a	21.49 (0.83, 0.99)	14.49 (0.88, 0.99)	n/a
par, &, 1c, lc1, tr	15.20 (1.00, 1.00)	n/a	n/a	21.68 (0.83, 0.98)	n/a	n/a
par, &, 1c, lc2	15.73 (0.97, 0.96)	7.70 (0.66, 1.00)	n/a	21.56 (0.83, 0.99)	14.52 (0.88, 0.99)	n/a
par, &, 1c, lc2, tr	15.20 (1.00, 1.00)	n/a	n/a	21.44 (0.84, 0.99)	n/a	n/a
par, &, 1c, lc4	15.19 (1.00, 1.00)	7.70 (0.66, 1.00)	n/a	21.37 (0.84, 1.00)	14.50 (0.88, 0.99)	n/a
par, &, 1c, lc4, tr	16.70 (0.91, 0.91)	n/a	n/a	21.61 (0.83, 0.98)	n/a	n/a
par, &, 2c, nolc, c128	12.80 (1.19, 1.18)	6.92 (0.74, 1.11)	3.88 (1.32, 1.99)	23.66 (0.76, 0.90)	14.35 (0.89, 1.00)	7.28 (1.75, 1.97)
par, &, 2c, nolc, c512	7.70 (1.98, 1.97)	4.70 (1.09, 1.64)	3.91 (1.31, 1.97)	17.65 (1.02, 1.21)	14.12 (0.90, 1.01)	7.26 (1.76, 1.97)
par, &, 2c, lc1	7.63 (2.00, 1.99)	4.02 (1.27, 1.92)	n/a	12.63 (1.42, 1.68)	7.31 (1.74, 1.96)	n/a
par, &, 2c, lc1, tr	7.63 (2.00, 1.99)	n/a	n/a	12.75 (1.41, 1.67)	n/a	n/a
par, &, 2c, lc2	7.87 (1.94, 1.93)	3.86 (1.33, 2.00)	n/a	13.94 (1.29, 1.53)	7.27 (1.76, 1.97)	n/a
par, &, 2c, lc2, tr	7.95 (1.92, 1.91)	n/a	n/a	13.89 (1.29, 1.53)	n/a	n/a
par, &, 2c, lc4	7.71 (1.98, 1.97)	3.85 (1.33, 2.00)	n/a	12.63 (1.42, 1.68)	7.27 (1.75, 1.97)	n/a
par, &, 2c, lc4, tr	8.03 (1.90, 1.89)	n/a	n/a	12.75 (1.41, 1.67)	n/a	n/a
par, &, 3c, nolc, c128	9.06 (1.68, 1.68)	6.12 (0.84, 1.26)	2.69 (1.90, 2.86)	23.89 (0.75, 0.89)	14.26 (0.90, 1.01)	4.87 (2.62, 2.95)
par, &, 3c, nolc, c512	5.16 (2.96, 2.94)	2.96 (1.73, 2.61)	2.63 (1.95, 2.93)	12.88 (1.39, 1.65)	13.94 (0.92, 1.03)	4.91 (2.60, 2.92)
par, &, 3c, lc1	5.26 (2.90, 2.88)	2.60 (1.97, 2.96)	n/a	9.50 (1.89, 2.24)	4.89 (2.61, 2.93)	n/a
par, &, 3c, lc1, tr	5.18 (2.95, 2.93)	n/a	n/a	9.58 (1.87, 2.22)	n/a	n/a
par, &, 3c, lc2	5.26 (2.90, 2.88)	2.59 (1.98, 2.97)	n/a	9.45 (1.90, 2.25)	4.87 (2.62, 2.94)	n/a
par, &, 3c, lc2, tr	5.18 (2.94, 2.93)	n/a	n/a	9.56 (1.88, 2.23)	n/a	n/a
par, &, 3c, lc4	5.11 (2.99, 2.97)	2.60 (1.97, 2.97)	n/a	9.48 (1.89, 2.24)	4.87 (2.62, 2.94)	n/a
par, &, 3c, lc4, tr	5.47 (2.79, 2.78)	n/a	n/a	9.57 (1.87, 2.22)	n/a	n/a
par, &, 4c, nolc, c128	4.52 (3.38, 3.36)	5.36 (0.96, 1.44)	1.98 (2.58, 3.88)	24.32 (0.74, 0.87)	14.14 (0.90, 1.01)	3.71 (3.44, 3.87)
par, &, 4c, nolc, c512	3.90 (3.91, 3.89)	2.33 (2.20, 3.31)	2.00 (2.56, 3.85)	11.47 (1.56, 1.85)	13.81 (0.92, 1.04)	3.71 (3.44, 3.87)
par, &, 4c, lc1	3.84 (3.97, 3.95)	1.97 (2.60, 3.91)	n/a	8.06 (2.22, 2.64)	3.72 (3.43, 3.85)	n/a
par, &, 4c, lc1, tr	3.94 (3.88, 3.86)	n/a	n/a	8.08 (2.22, 2.63)	n/a	n/a
par, &, 4c, lc2	3.84 (3.97, 3.95)	1.96 (2.61, 3.92)	n/a	8.03 (2.23, 2.65)	3.68 (3.47, 3.90)	n/a
par, &, 4c, lc2, tr	3.89 (3.93, 3.90)	n/a	n/a	8.07 (2.22, 2.63)	n/a	n/a
par, &, 4c, lc4	3.85 (3.97, 3.94)	1.96 (2.61, 3.93)	n/a	8.04 (2.23, 2.65)	3.77 (3.38, 3.80)	n/a
par, &, 4c, lc4, tr	3.88 (3.93, 3.91)	n/a	n/a	8.09 (2.22, 2.63)	n/a	n/a

Table 5.2: Execution times measured in seconds, and speedups

(The garbage collector uses one core in all of our tests.) However, the mandelbrot program speeds up when thread safety is enabled; it does not do very much memory allocation and is therefore affected less by the overheads of thread safety in the garbage collector. Its slight speedup may be due to its different code and data layouts interacting with the cache system differently.

All the later rows compile the program for parallel execution, and leave the parallel conjunctions in the program intact. They execute the program on 1 to 4 cores ('1c' to '4c'). The versions that execute on the same number of cores differ from each other mainly in how they handle loops. The rows marked 'nolc' are the controls. They do not use the loop control mechanism described in this chapter; instead, they rely on the context limit. The actual limit is the number of engines multiplied by a specified parameter, which we have set to 128 in 'c128' rows and to 512 in 'c512' rows. Although our code uses thread-safe mechanisms for counting the number of contexts in use, when it checks this value against the limit it does not use a thread-safe mechanism. Therefore, whilst the number of contexts in use can never be corrupted, it can be exceed the limit by about one or two contexts. Since different contexts can have different sized stacks, the limit is only an approximate control over memory consumption anyway, so this is an acceptable price to pay for reduced synchronisation overhead.

The rows marked 'lc$N$' do use our loop control mechanism, with the value of N indicating the value of another parameter we specify when the program is run. When the `lc_create_loop_control` instruction creates a loop control structure, it computes the number of slots to create in it, by multiplying the configured number of Mercury engines (each of which can execute on its own core) with this parameter. We show memory consumption and timing results for $N = 1$, 2 and 4. The timing results for $N = 1$ and $N = 4$ are almost identical to those for $N = 2$. It seems that as long as we put a reasonably small limit on the number of contexts a loop control structure can use, speed is not much affected by the precise value of the limit.

The rows in Table 5.2 marked 'lc$N$, tr' are like the corresponding 'lc$N$' rows, but they also switch on tail recursion preservation in the two benchmarks (mandelbrot and raytracer) whose parallel loops are naturally tail recursive. The implementation of parallelism without loop control destroys this tail recursion, and so does loop control unless we ask it to preserve tail recursion. That means that mandelbrot and raytracer use tail recursion in all the test setups except for the parallel, non-loop control ones, and loop control ones without tail recursion. Since the other benchmarks are not naturally tail recursive, they will not be tail recursive however they are compiled. There are no such rows in Table 5.1 since the results in each 'lc$N$, tr' row would be identical to the corresponding 'lc$N$' row.

There are several things to note in Table 5.1. The most important is that when the programs are run on more than one core, switching on loop control yields a dramatic reduction in the maximum number of contexts used at any one time, and therefore also in the maximum amount of memory used by stacks. (The total amount of memory used by these benchmarks is approximately the maximum of this number and the configured initial size of the heap.) This shows that we have achieved our main objective. Without loop control, the execution of three of our four dependent benchmarks (mandelbrot, matrixmult and raytracer) require the simultaneous existence of a context for every parallel task that the program can spawn off. For example, mandelbrot generates an image with 600 rows, so the original context can never spawn off more than 600 other contexts.

On one core, the 'nolc' versions spawn off sparks, but since there is no other engine to pick them up, the one engine eventually picks them up itself, and executes them in the original context. By contrast, the 'lc$N$' versions directly spawn off new contexts, not sparks. This avoids the overhead

of converting a spark to a context, but we can do this only because we know we will not create too many contexts.

When executing on two or more cores, mandelbrot and raytracer use one more context that one would expect. Before the compiler applies the loop control transformation, it adds the synchronisation operations needed by dependent parallel conjunctions. As shown by Figure 2.9, this duplicates the original procedure. Only the inner procedure is recursive, so the compiler performs the loop control transformation only on it. The extra context is the conjunct spawned off by the parallel conjunction in the outer procedure.

There are several things to note in Table 5.2 as well. The first is that in the absence of loop control, increasing the per-engine context limit from 128 to 512 yields significant speedups for three out of four the dependent benchmarks. Nevertheless, the versions with loop control significantly outperform the versions without, even 'c512', for all these benchmarks except for mandelbrot. On mandelbrot, 'c512' already gets a near-perfect speedup, yet loop control still gets a small improvement. Thus on all our dependent benchmarks, switching on loop control yields a speedup while greatly reducing memory consumption.

Overall, the versions with loop control get excellent speedups on three of the benchmarks: speedups of 3.95, 3.92 and 3.90 on four CPUs for mandelbrot, matrixmult and spectralnorm, respectively. The one apparent exception, raytracer, is very memory-allocation-intensive. In Section 3.1 we showed that this can significantly reduce the benefit of parallel execution of Mercury code, especially when the collector does not use parallelism itself. In we have observed nearly 45% of a the raytracer's runtime being used in garbage collection (Table 3.2): This means that parallel execution can only speed up the remaining 55% of the program. Therefore the best speedup we can hope for is $(4 \times 0.55 + 0.45)/(0.55 + 0.45) \approx 2.65$, which our result of 2.65 reaches. It is unusual to see results this close to their theoretical maximums; in this case the parallelisation of the runtime may affect the garbage collector's performance, which can affect the figures that we have used with Amdahl's law.

Note second that loop control is crucial for getting this kind of speedup, without wasting lots of memory. On four cores, loop control raises the speedup compared to c128 from 3.36 to 3.95 for mandelbrot, from 1.44 to 3.92 for matrixmult, from 0.87 to 2.65 for raytracer, and from 1.01 to 3.90 for spectralnorm.

Third, for the benchmarks that have versions using independent parallelism, the independent versions are faster than the dependent versions without loop control, while there is no significant difference between the speeds of the independent versions and the dependent loop control versions. For matrix multiplication, the loop control dependent version is faster, but the difference is small. While for spectral-norm, the independent version result falls within the range of loop control results. This shows that on these benchmarks, loop control completely avoids the problems described at the beginning of this chapter.

Fourth, preserving tail recursion has a mixed effect on speed: of the six relevant cases (mandelbrot and raytracer on 2, 3 and 4 cores), one case gets a slight speedup, while the others get slight slowdowns. Due to the extra copying required, this tilt towards slowdowns is to be expected. However, the effect is very small: usually within 2%. The possibility of such slight slowdowns is an acceptable price to pay for allowing parallel code to recurse arbitrarily deeply while using constant stack space.

5.4 Further work

There are a number of small improvements that could be made to improve loop control. As we mentioned in Section 5.2.3, the allocation of stack slots in tail recursive code is rather naive. We could perform better stack slot allocation for both the parent stack frame and the first stack frame on the stack slot's context. We could also change how stack slots are allocated in non-recursive code. By allocating the slots that are shared between the parent and child computations into separate parts of the same stack frame, we may be able to reduce *false sharing*. False sharing normally occurs when two parallel tasks contend for access to a single cache-line sized area of memory *without* actually using the data in that memory to communicate; this creates extra cache misses that can be avoided by allocating memory differently.

Loop control allows the same small number of contexts to be used to execute a parallel loop. However, futures are still allocated dynamically and are not re-used. We could achieve further performance improvements by controlling the allocation and use of futures. This could reduce the number of allocations of futures from $num\_futures\_per\_level \times num\_red\_levels$ to just a single allocation, depending on how futures where used in the loop.

These changes are minor optimisations, and might only increase performance by a small fraction; there are other areas of further work that could provide more benefit. For example, applying a similar transformation to other parallel code patterns, such as right recursion or divide and conquer, could improve performance of code using those patterns. In some patterns this may require novel transformations such as we have shown here, in others it may simply be the application of loop control transformations such as those of Shen et al. [99]

5.5 Conclusion

Ever since the first parallel implementations of declarative languages in the 1980s, researchers have known that getting more parallelism out of a program than the hardware could use can be a major problem, because the excess parallelism brings no benefits, only overhead, and these overheads could swamp the speedups the system would otherwise have gotten. Accordingly, they have devised systems to throttle parallelism, keeping it at a reasonable level.

However, most throttling mechanisms we know of have been general in nature, such as granularity control systems [66, 76, 99]. These have similar objectives, but use totally different methods: restricting the set of places in a program *where* they choose to exploit parallelism, not changing *how* they choose to exploit it.

We know of one system that tries to preserve tail recursion even when the tail comes from a parallel conjunction. The ACE system [46] normally generates one parcall frame for each parallel conjunction, but it will flatten two or more nested parcall frames into one if runtime determinacy tests indicate it is safe to do so. While these tests usually succeed for loops, they can also succeed for other code, and (unlike our system) the ACE compiler does not identify in advance the places where the optimisation may apply. The other main difference from our system is the motivation: the main motivation of this mechanism in the ACE system is neither throttling nor the ability to handling unbounded input in constant stack space, but reducing the overheads of backtracking. This is totally irrelevant for us, since our restrictions prevent any interaction between AND-parallel code and code that can backtrack.

The only work on specially loop-oriented parallelism in logic languages that we are aware of is

Reform Prolog [11]. This system was not designed for throttling either, but it is more general than ours in one sense (it can handle recursion in the middle of a clause) and less general in other senses (it cannot handle parallelism in any form other than loops, and it cannot execute one parallel loop inside another). It also has significantly higher overheads than our system: it traverses the whole spine of the data structure being iterated over (typically a list) *before* starting parallel execution; in some cases it synchronises computations by busy waiting; and it requires variables stored on the heap to have a timestamp. To avoid even higher overheads, it imposes the same restriction we do: it parallelises only deterministic code (though the definition of "deterministic" it uses is a bit different). Reform Prolog does not handle any other forms of parallelism, whereas Mercury handles parallelism in various situations, and the loop control transformation simply optimises one form of parallelism.

The only work on loop-oriented parallelism in functional languages we know of is Sisal [38]. It shares two of Reform Prolog's limits: no parallelism anywhere except loops, and no nesting of parallel computations inside one another. Since it was designed for number crunching on supercomputers, it had to have lower overheads than Reform Prolog, but it achieved those low overheads primarily by limiting the use of parallelism to loops whose iterations are *independent* of each other, which makes the problem much easier. Similarly, while ACE Prolog supports both AND- and OR-parallelism, the only form of AND-parallelism it supports is independent.

Our system is designed to throttle loops with dependent iterations, and it seems to be quite effective. By placing a hard bound on the number of contexts that may be needed to handle a single loop, our transformation allows parallel Mercury programs to do their work in a reasonable amount of memory, and since it does so without adding significant overhead, it permits them to live up to their full potential. For one of our benchmarks, loop control makes a huge difference: on four cores, it turns a speedup of 1.01 into a speedup of 3.90. It significantly improves speedups on two other benchmarks, and it even helps the fourth and last benchmark, even though that was already close to the maximum possible speedup.

The other main advantage of our system is that it allows procedures to keep exploiting tail recursion optimisation (TRO). If TRO is applicable to the sequential version of a procedure, then it will stay applicable to its parallel version. Many programs cannot handle large inputs without TRO, so they cannot be parallelised at all without this capability. The previous advantage may be specific to systems that resemble the Mercury implementation, but this should apply to the implementation of every eager declarative language.

This chapter is an extended and revised version of Bone et al. [20].

Chapter 6

Visualisation

The behaviour of parallel programs is even harder to understand than the behaviour of sequential programs. Parallel programs may suffer from any of the performance problems afflicting sequential programs, as well as from several problems unique to parallel systems. Many of these problems are quite hard (or even practically impossible) to diagnose without help from specialised tools. In this chapter we describe a proposal for a tool for profiling the parallel execution of Mercury programs, a proposal whose implementation we have already started. This tool is an adaptation and extension of the ThreadScope profiler that was first built to help programmers visualise the execution of parallel Haskell programs.

The structure of this chapter is as follows. Section 6.1 describes three different groups of people, Mercury programmers, runtime system implementors and automatic parallelism researchers. For each group we describe how a visual profiler based on ThreadScope would benefit them. Section 6.2 gives a description of the ThreadScope tool, how it is used in Haskell, and how we can use it for Mercury without any modifications. Section 6.3 describes how we modify and extended the ThreadScope system to collect the kinds of data needed to more completely describe the parallel execution of Mercury programs, while Section 6.4 describes how one can analyse that data to yield insights that can be useful to our three audiences: application programmers, runtime system implementors, and the implementors of auto-parallelisation tools. Section 6.5 shows some examples of our work so far. Finally, Section 6.6 concludes with a discussion of related work.

6.1 Introduction

When programmers need to improve the performance of their program, they must first understand it. The standard way to do this is to use a profiler. Profilers record performance data from executions of the program, and give this data to programmers to help them understand how their program behaves. This enhanced understanding then makes it easier for programmers to speed up the program.

Profilers are needed because the actual behaviour of a program often differs from the behaviour assumed by the programmer. This is true even for sequential programs. For parallel programs, the gulf between human expectations and machine realities is usually even wider. This is because every cause of unexpected behaviour for sequential programs is present in parallel programs as well, while parallel programs also have several causes of their own.

Consider a program containing a loop with many iterations in which the iterations do not

depend on one another and can thus be done in parallel. Actually executing every iteration in parallel may generate an overwhelming number of parallel tasks. It may be that the average amount of computation done by one of these tasks is less than the amount of work it takes to spawn one of them off. In that case, executing all the iterations in parallel will increase overheads to the point of actually slowing the program down, perhaps quite substantially. The usual way to correct this is to use granularity control to make each spawned-off task execute not one but several iterations of the loop, thus making fewer but larger parallel tasks. However, if this is done too aggressively, performance may once again suffer. For example, there may be fewer tasks created than the computer has processors, causing some processors to be idle when they could be used. More commonly, the iterations of the loop and thus the tasks may each need a different amount of CPU time. If there are eight processors and eight unevenly sized tasks, then some processors will finish their work early, and be idle while waiting for the others.

All these problems can arise in programs with independent parallelism: programs in which parallel tasks do not need to communicate. Communication between tasks makes this situation even more complicated, especially when a task may block waiting for information from another task. A task that blocks may in turn further delay other tasks that depend on data *it* produces. Chains of tasks that produce and consume values from one another are common, Programmers need tools to help them identify and understand performance problems, whether they result from such dependencies or other causes.

Profiling tools that can help application programmers can also be very useful for the implementors of runtime systems. While optimising Mercury's parallel runtime system (Chapter 3), we have needed to measure the costs of certain operations and the frequency of certain behaviours. Some examples are: when a piece of work is made available, how quickly can a sleeping worker-thread respond to this request? When a task is made runnable after being blocked, how often will it be executed on the same CPU that was previously executing it, so that its cache has a chance to be warm? Such information from profiles of typical parallel programs can be used to improve the runtime system, which can help improve the performance of *all* parallel programs.

A third category of people who can use profiling tools for parallel programs are researchers working on automatic parallelisation tools (Chapter 4). Doing a good job of automatically parallelising a program requires a cost-benefit analysis for each parallelism opportunity, which requires estimates of both the cost and the benefit of each opportunity. Profiling tools can help researchers calibrate the algorithms they use to generate estimates of both costs and benefits. We did not use the work in this chapter to help us with these problems in Chapter 4, so using ThreadScope to tune the work of Chapter 4 represents some potential further work.

As researchers working on the parallel implementation of Mercury, including automatic parallelism, we fall into all three of the above categories. We have long needed a tool to help us understand the behaviour of parallel Mercury programs, of the parallel Mercury runtime system and of our auto-parallelisation tool (Chapter 4), but the cost of building such a tool seemed daunting. We therefore looked around for alternative approaches. The one we selected was to adapt to Mercury an existing profiler for another parallel system.

The profiler we chose was ThreadScope [63], a profiler built by the team behind the Glasgow Haskell Compiler (GHC) for their parallel implementation of Haskell. We chose ThreadScope because the runtime system it was designed to record information from has substantial similarities to the Mercury runtime system, because ThreadScope was *designed* to be extensible, and, like many visualisation tools, it is extremely useful for finding subtle issues that cause significant problems.

6.2 Existing ThreadScope events

ThreadScope was originally built to help programmers visualise the parallel execution of Haskell programs compiled with the dominant implementation of Haskell, the Glasgow Haskell Compiler (GHC). The idea is that during the execution of a parallel Haskell program, the Haskell runtime system writes time-stamped reports about significant events to a log file. The ThreadScope tool later reads this log file, and shows the user graphically what each CPU was doing over time. The diagrams it displays reveal to the programmer the places where the program is getting the expected amount of parallelism, as well as the places where it is not.

We were able to adapt the ThreadScope system to Mercury for two main reasons. The first is that the parallel implementations of Haskell and Mercury in GHC and mmc (the Melbourne Mercury Compiler) are quite similar in several important respects, even though they use different terms for (slightly different implementations of) the same concepts. For example, what Mercury calls an engine GHC calls a *capability*, and what Mercury calls a context GHC calls a *thread*. (Except where noted, we will continue to use Mercury terminology as we have done so far in the dissertation.) Mercury and GHC both use sparks, although GHC's sparks do not need to be evaluated, and may be garbage collected. The second reason that we were able to adapt ThreadScope to Mercury is that the ThreadScope log file format was designed to be extensible. Each log file starts with a description of each kind of event that may occur in it. This description includes three fields:

- the event's id
- a string that describes the event to users and implementors,
- and the total size of the event, which may be variable.

The fields within the event and their sizes and meanings are established by convention. This means that when the event has a variable size, only one of the fields may have a variable size, unless self-describing fields or extra fields that specify the sizes of other fields are used. This description is precise enough that tools can skip events that they do not understand, and process the rest.

Mercury is able to make use of a number of event types already supported by ThreadScope and GHC, and in other cases we are able to add support for Mercury-specific event types to ThreadScope. Here we introduce the events in the first category; the next section describes those in the second category. Events of all types have a 64bit timestamp that records the time of their occurrence, measured in nanoseconds. This unit illustrates the level of precision ThreadScope aims for, although of course there is no guarantee that the system clock is capable of achieving it. Implementors are free to choose which clock they use to timestamp events. The GHC implementors use the `gettimeofday(2)` system call, and we use either `gettimeofday(2)` or the processors' time stamp counters (TSCs).

Most events in the log file are associated with an engine. To avoid having to include an engine id with every event in the log file, the log file format groups sequences of events that are all from the same engine into a block, and writes out the engine id just once, in a block header pseudo-event. Since tools reading the log file can trivially remember the engine id in the last block header they have read, this makes the current engine id (the id of the engine in whose block an event appears) an implicit parameter of most event types.

The event types supported by the original version of ThreadScope that are relevant to our work are listed here.

STARTUP marks the beginning of the execution of the program, and records the number of engines that the program will use.

SHUTDOWN marks the end of the execution of the program.

CREATE_THREAD records the act of the current engine creating a context, and gives the id of the context being created. (What Mercury calls a context Haskell calls a thread; the name of the event uses Haskell terminology.) Context ids are allocated sequentially, but the id cannot be omitted, since the runtime system may reuse the storage of a context after the termination of the computation that has previously used it, and therefore by specifying the id we can determine when a thread is reused.

RUN_THREAD records the scheduling event of the current engine switching to execute a context, and gives the id of the context being switched to.

STOP_THREAD records the scheduling event of the current engine switching away from executing a context. Gives the id of the context being switched from, as well as the reason for the switch. The possible reasons include: (a) the heap is full, and so the engine must invoke the garbage collector; (b) the context has blocked, and (c) the context has finished. GHC uses many more reason values, but these are the only reasons that are applicable to our work. The context id could be derived implicitly, but the convention that was established by the GHC developers requires us to provide it.

THREAD_RUNNABLE records that the current engine has made a blocked context runnable, and gives the id of the newly-runnable context.

RUN_SPARK records that the current engine is starting to run a spark that it retrieved from its own local spark queue. Gives the id of the context that will execute the spark, although, like `STOP_THREAD`, this can be inferred by context. However the inference required is slightly different between GHC and Mercury.

STEAL_SPARK records that the current engine will run a spark that it stole from the spark queue of another engine. Gives the id of the context that will execute the spark, although again, this can be inferred by context. Also gives the id of the engine that the spark was stolen from.

CREATE_SPARK_THREAD is slightly different in GHC and Mercury. In GHC this records the creation of a worker context that runs a special routine that processes sparks in a loop; this can be thought of as a worker context for executing sparks. In Mercury this event is recorded when an engine wishes to execute a particular spark and needs a new context to do it; the context may be used to execute subsequent sparks. The event gives the id of the context. The context may be an existing context that is reused, but the event does not say whether the context is new or reused. In most cases, that information is simply not needed, but if it is, it can be inferred from context.

GC_START is used when the current engine has initiated garbage collection; control of the engine has been taken away from the mutator. The garbage collection events in GHC are much more complex. However, since Mercury uses the Boehm-Demers-Weiser conservative garbage collector (Boehm GC) [14], we have much less insight into the garbage collector's (GC) behaviour, and thus fewer GC related events and simpler semantics.

GC_END indicates when garbage collection has finished on the current engine; control of the engine has been returned to the mutator.

The longer a parallel program being profiled runs, the more events it will generate. Long running programs can generate enormous log files. To keep the sizes of log files down as much as possible, events include only the information they have to. If some information about an event can be inferred from information recorded for other events, then the design principles of ThreadScope say that information should be inferred at analysis time rather than recorded at profiling runtime. (The presence of context ids in `RUN_SPARK` and `STEAL_SPARK` events is considered a mistake [102], since they are guaranteed to be the same as the id of the context already running on that engine or the context id in the following `CREATE_SPARK_THREAD` event, but their removal is prevented by backwards compatibility concerns.) In most cases, the required inference algorithm is quite simple. For example, answering the question of whether the context mentioned by a `CREATE_SPARK_THREAD` event is new or reused merely requires searching the part of the log up to that event looking for mentions of the same context id. We have already seen how the engine id parameter missing from most of the above events can be deduced from block headers.

6.3 New events

In order to support the profiling of parallel Mercury programs, we had to add new arguments to two of the existing ThreadScope events. Originally, the `RUN_SPARK` and `STEAL_SPARK` events did *not* specify the identity of the spark being run or stolen. This is not a problem for the Haskell version of ThreadScope, since the implementors did not care about sparks' identities [102]. However, we do, since sparks correspond to conjuncts in parallel conjunctions, and we want to give to the user not just general information about the behaviour of the program as a whole, but also about the behaviour of individual parallel conjunctions, and of the conjuncts in them. We have therefore added an id that uniquely identifies each spark as an additional argument of the `RUN_SPARK` and `STEAL_SPARK` events. Note that `CREATE_SPARK_THREAD` does not need a spark id, since the only two ways it can get the spark it converts into a context is by getting it from its own queue or from the queue of another engine. A `CREATE_SPARK_THREAD` event will therefore always be preceded by either a `RUN_SPARK` event or a `STEAL_SPARK` event, and the id of the spark in that event will be the id of the spark being converted.

We have extended the set of ThreadScope event types to include several new types of events. Most of these record information about constructs that do not exist in Haskell: parallel conjunctions, conjuncts in those conjunctions, and futures. Some provide information about the behaviour of Mercury engines that ThreadScope for Haskell does not need, either because that information is of no interest, or because the information is of interest but it can be deduced from other events. Even though the Haskell and Mercury runtime systems generate many of the same events, the stream of these common events they generate do not necessarily obey the same invariants.

However, most of the work we have done towards adapting ThreadScope to Mercury has been in the modification of the parallel Mercury runtime system to generate all of the existing, modified and new events when called for.

In the rest of this section, we report on the event types we have added to ThreadScope. In the next section, Section 6.4, we will show how the old and new events can be used together to infer interesting and useful information about the behaviour of parallel Mercury programs.

STRING records an association between a text string and an integer so that a string occurring many times within a log file can be replaced with an integer. Making the log files shorter and therefore not flushing them to disk as often.

START_PAR_CONJUNCTION records the fact that the current engine is about to start executing a parallel conjunction. It identifies the parallel conjunction in two different ways. The static id identifies the location of the conjunction in the source code of the program. The dynamic id is the address of the barrier structure that all the conjuncts in the conjunction will synchronise on when they finish. Note that barrier structures' storage may be reused for other objects, including other barriers, by later code, so these dynamic ids are not unique across time, but they *do* uniquely identify a parallel conjunction at any given moment in time. See the end of this section for a discussion of why we chose this design.

END_PAR_CONJUNCTION records the end of a parallel conjunction. It gives the dynamic id of the finishing conjunction. Its static id can be looked up in the matching `START_PAR_CONJUNCTION` event.

CREATE_SPARK records the creation of a spark for a parallel conjunct by the current engine. It gives the id of the spark itself, and the dynamic id of the parallel conjunction containing the conjunct. To keep the log file small, it does *not* give the position of the conjunct in the conjunction. However, since in every parallel conjunction the spark for conjunct $n+1$ will be created by the context executing conjunct n, and unless $n = 1$, that context will itself have been created from the spark for conjunct n. (The first conjunct of a parallel conjunction is always executed directly, without ever being represented by a spark.) This means that the sparks for the non-first conjuncts will always be created in order, which makes it quite easy to determine which spark represents which conjunct.

The id of a spark is an integer consisting of two parts: the first part is the id of the engine that created the spark, and the second is an engine-specific sequence number, so that in successive sparks created by the same engine, the second part will be 1, 2, 3 etc. This design allows the runtime system to allocate globally unique spark ids without synchronisation.

END_PAR_CONJUNCT records that the current engine has finished the execution of a conjunct in a parallel conjunction. It also gives the dynamic id of the parallel conjunction. Note that there is no `START_PAR_CONJUNCT` event to mark the start of execution of any conjunct. For the first conjunct, its execution will start as soon as the engine has finished recording the `START_PAR_CONJUNCTION` event. The first thing the conjunct will do is create a spark representing the second and later conjuncts (which above we informally referred to as the spark for the second conjunct). The `CREATE_SPARK` event records the id of the spark, then, either the `RUN_SPARK` event or `STEAL_SPARK` event records which engine and context executes the spark. If the engine does not yet have a context a `CREATE_SPARK_THREAD` event is posted that identifies the newly created context. `RUN_THREAD` and `STOP_THREAD` events also tell us when that context is executing, and on which engine. The first thing the second conjunct does is spawn off a spark representing the third and later conjuncts. By following these links, a single forward traversal of the log file can find out, for each engine, which conjunct of which dynamic parallel conjunction it is running at any given time (if in fact it is running any: an engine can be idle, or it may be running code that is outside of any parallel conjunction).

When an engine records an `END_PAR_CONJUNCT` event, it can only be for the conjunct it is currently executing.

FUTURE_CREATE records the creation of a future by the current engine. It gives the name of the variable the future is for, as well as the dynamic id of the future. It does not give the dynamic id of the parallel conjunction whose conjuncts the future is intended to synchronise, but that can be determined quite easily: just scan forward for the next `START_PAR_CONJUNCTION` event. The reason why this inference works is that the code that creates futures is only ever put into programs by the Mercury compiler, which never puts that code anywhere except just before the start of a parallel conjunction. If a parallel conjunction uses n futures, then every one of its `START_PAR_CONJUNCTION` events will be preceded by n `FUTURE_CREATE` events, one for each future.

The dynamic id of the future is its address in memory. This has the same reuse caveat as the dynamic ids of parallel conjunctions.

FUTURE_SIGNAL records that code running on the current engine has signalled the availability of the value of the variable protected by a given future. The event provides the id of the future. If another conjunct is already waiting for the value stored in this future, then its context is now unblocked.

FUTURE_WAIT_NO_SUSPEND records that the current engine retrieved the value of the future without blocking. Gives the id of the future. The future's value was already available when the engine tried to wait for the value of the future, so the context running on the current engine was not blocked.

FUTURE_WAIT_SUSPEND records that the current engine tried to retrieve the value of a future, but was blocked. Gives the id of the future. Since the future's value was not already available, the current context has been be suspended until it is.

We have added a group of four event types that record the actions of engines that cannot continue to work on what they were working before, either because the conjunct they were executing finished, or because it suspended.

TRY_GET_RUNNABLE_CONTEXT records when the engine is checking the global run queue for a context to execute.

TRY_GET_LOCAL_SPARK records that this engine is attempting to get a spark from its own deque.

TRY_STEAL_SPARK records that this engine is attempting to steal a spark from another engine.

ENGINE_WILL_SLEEP records that the engine is about to sleep. The next event from this engine will be from when it is next awake.

The idea is that an idle engine can look for work in three different places: (a) the global runnable context queue, (b) its own local spark queue, and (c) another engine's spark queue. An idle engine will try these in some order, the actual order depending on the scheduling algorithm. The engine will post the try event for a queue before it actually looks for work in that queue. If one of the

tests is successful, the engine will post a START_THREAD event, a RUN_SPARK event or a STEAL_SPARK event respectively. If one of the tests fails, the engine will go on the next. If it does not find work in any of the queues, it will go to sleep after posing the ENGINE_WILL_SLEEP event.

This design uses two events to record the successful search for work in a queue: TRY_GET_RUNNABLE_CONTEXT and START_THREAD, TRY_GET_LOCAL_SPARK and RUN_SPARK, or TRY_STEAL_SPARK and STEAL_SPARK. This may seem wasteful compared to using one event, but this design enables us to measure several interesting things:

- how quickly a sleeping engine can wake up and try to find work when work is made available by a CREATE_SPARK and or CONTEXT_RUNNABLE event;
- how often an engine is successful at finding work;
- when it is successful, how long it takes an engine to find work and begin its execution;
- when it is unsuccessful, how long it takes to try to find work from other sources, fail, and then to go to sleep.

As we mentioned above, using the address of the barrier structure as the dynamic id of the parallel conjunction whose end the barrier represents and using the address of the future structure as the dynamic id of the future are both design choices. The obvious alternative would be to give them both sequence numbers using either global or engine-specific counters. However, this would require adding a field to both structures to hold the id, which costs memory, and filling in the field, which costs time. Both these costs would be incurred by the program execution whose performance we are trying to measure, interfering with the measurement itself. While such interference cannot be avoided (writing events out to the log file also takes time), we nevertheless want to minimise it if at all possible. In this case, it is possible: the addresses of the structures are trivially available, and require no extra time or space during the profiled run.

The tradeoff is that if an analysis requires globally unique dynamic ids for parallel conjunctions or for futures, and most analyses do, then it needs to ensure that a pre-pass has been run over the log file. This pre-pass would maintain a map from future ids to a pair containing an active/inactive flag, and a replacement id, and another map from dynamic conjunction ids to a tuple containing an active/inactive flag, a replacement id, and a list of future ids. When the pre-pass sees a dynamic conjunction id or future id in an event, it looks it up in the relevant table. If the flag says the id is active, it replaces the id with the replacement. If the flag says the id is inactive, it gets a new replacement id (e.g. by incrementing a global counter), and sets the flag to active. If the event is a FUTURE_CREATE event, the pre-pass adds the future's original id to a list. If the event is a START_PAR_CONJUNCTION event, it copies this list to the conjunction's entry, and then clears the list. If the event is an END_PAR_CONJUNCTION event, which ends the lifetime not only of the parallel conjunction but also of all futures created for that conjunction, the pre-pass sets to inactive that flags of both the conjunction itself and the futures listed in its entry.

This algorithm consistently renames apart the different instances of the same id, replacing them with globally unique values. Its complexity can be close to linear, provided the maps are implemented using a suitable data structure, such as a hash table. This transformation does not even need an extra traversal of the log file data. The ThreadScope tool must traverse the log file anyway when it gets ready to display its contents, and this transformation can be done as part of that traversal. The extra cost is therefore quite small.

6.4 Deriving metrics from events

ThreadScope is primarily used to visualise the execution of parallel programs. However, we can also use it, along with our new events, to calculate and to present to the user a number of different metrics about the execution of parallel Mercury programs. The GHC developers have independently[1] recognised this need, and have created alternative views within the ThreadScope GUI to present textual information such as reports of these metrics. In our rudimentary prototype of how these metrics could be calculated we used a command line driven console interface. (Some parts of ThreadScope are implemented as a library, allowing the development of different independent tools.) Some of the metrics are of interest only to application programmers, or only to runtime system implementors, or only to auto-parallelisation tool implementors, but several are useful to two or even all three of those groups. Also, metrics can be organised by their *subject*, whether that is:

- the whole program,
- an engine,
- a parallel conjunction,
- a parallel conjunct,
- a future.

We discuss our proposed metrics in that order; we then discuss how we intend to present them to users.

6.4.1 Whole program metrics

CPUs over time is the number of CPUs being used by the program at any given time. It is trivial to scan all the events in the trace, keeping track of the number of engines currently being used, each of which corresponds to a CPU. The resulting curve tells programmers which parts of their program's execution is already sufficiently parallelised, which parts are parallelised but not yet enough to use all available CPUs, and which parts still have no parallelism at all. They can then focus their efforts on the latter. The visualisation of this curve is already implemented in ThreadScope [63].

GC stats is the number of garbage collections, and the average, minimum, maximum and variance of the elapsed time taken by each collection. Boehm GC supports parallel marking using GC-specific helper threads rather than the OS threads that run Mercury engines. Therefore, even when parallel marking is used, we can only calculate the elapsed time used by garbage collection, and not the *CPU time* used by garbage collection. The elapsed time of each garbage collection is the interval between pairs of `GC_START` and `GC_END` events.

Mutator vs GC time is the fraction of the program's runtime used by the garbage collector. We calculate this by summing the times between pairs of `GC_START` and `GC_END` events, and dividing the result by the program's total runtime. (Both sides of the division refer to elapsed time, not CPU time.) Due to Amdahl's law [4], the fraction of time spent in garbage collection limits the best possible speedup we can get for the program as a whole by parallelising the mutator. For example,

[1] The GHC developers added the alternative views to ThreadScope after we published the paper on which this chapter is based [18].

if the program spends one third of its time doing GC (which unfortunately actually happens for some parallel programs), then no parallelisation of the program can yield a speedup of more than three, *regardless* of the number of CPUs available.

Nanosecs per call is the average number of nanoseconds between successive procedure calls. The Mercury deep profiler (for sequential programs) [32] (Section 2.4.1), which has nothing to do with ThreadScope, measures time in call sequence counts (CSCs); the call sequence counter is incremented at every procedure call. It does this because there is no portable, or even semi-portable way to access any real-time clocks that may exist on the machine, and even the non-portable method on x86 machines (the RDTSC instruction) is too expensive for it. (The Mercury sequential profiler needs to look up the time at every call, which in typical programs will happen every couple of dozen instructions or so. The Mercury runtime support for ThreadScope needs to look up the time only at each event, which occur *much* less frequently.)

The final value of the call sequence counter at the end of a sequential execution of a program gives its length in CSCs; say n CSCs. When a parallelised version of the same program is executed on the same data under ThreadScope, we can compute the total amount of *user time* taken by the program on all CPUs; say m nanoseconds. The ratio n/m gives the average number of nanoseconds in the parallel execution of the program per CSC in its sequential execution. This is useful information, because our automatic parallelisation system uses CSCs, as measured by the Mercury sequential profiler, as its unit of measurement of the execution time of both program components and system overheads. Using this scale, we can convert predictions about time made by the tool from being expressed in CSCs to being expressed in nanoseconds, which is an essential first step in comparing them to reality.

Contexts allocated is the number of contexts allocated at any given time; it includes contexts that are running, runnable and blocked.

Contexts runnable is the number of contexts that are on the global context run queue at any given time.

6.4.2 Per engine metrics

Sparks available is the number of sparks available for execution at any given time. Since the time of publishing Bone and Somogyi [18] the GHC developers added similar metrics to ThreadScope. Their metrics are more complicated as their spark implementation is more complicated. Their implementation introduces a new event that provides summary data just for this purpose; by comparison our spark events provide more detail as we know more accurately when each spark event occurred. We wish to measure the number of sparks available on each Mercury engine's deque at any given time. This can be done by maintaining a counter for each engine, incrementing it each time that engine uses the `CREATE_SPARK` event, and decrementing it whenever the engine uses a `RUN_SPARK` event or another engine uses a `STEAL_SPARK` event for a spark on this engine's deque.

6.4.3 Conjunction specific metrics

Parconj time is the time taken by a given parallel conjunction. For each dynamic parallel conjunction id that occurs in the trace, it is easy to compute the difference between the times at which that parallel conjunction starts and ends, and it is just as trivial to associate these time intervals with the conjunction's static id. From this, we can compute, for each parallel conjunction

in the program that was actually executed, both the average time its execution took, and the variance in that time. This information can then be compared, either by programmers or by automatic tools, with the sequential execution time of that conjunction recorded by Mercury's deep profiler, to see whether executing the conjunction in parallel was a good idea or not, *provided* that the two measurements are done in the same units. At the moment, they are not, but as we discussed above, CSCs can be converted to nanoseconds using the NANOSECS PER CALL metric.

PARCONJ RUNNABLE SELF is the number of CPUs that can be used by a given parallel conjunction. For each dynamic parallel conjunction id that occurs in the trace, we can divide the time between its start and end events into blocks, with the blocks bounded by the `CREATE_SPARK_THREAD` and `STOP_THREAD` events for its conjuncts, and the `FUTURE_WAIT_SUSPEND` and `FUTURE_SIGNAL` events of the futures used by those conjuncts. We can then compute, for each of these blocks, the number of runnable tasks that these conjuncts represent. From this we can compute the history of the number of runnable tasks made available by this dynamic conjunction. We can derive the maximum and the time-weighted average of this number, and we can summarise those across all dynamic instances of a given static conjunction.

Consider a conjunction with n conjuncts. If it has a maximum number of runnable tasks significantly less than n, this is a sign that the parallelism the programmer aimed for could not be achieved. If instead the average but not the maximum number of runnable tasks is significantly less than n, this suggests that the parallelism the programmer aimed for was achieved, but only briefly. The number of runnable tasks can drop due to either a wait operation that suspends or the completion of a conjunct. Both dependencies among conjuncts and differences in the execution times of the conjuncts limit the amount of parallelism available in the conjunction. The impact of the individual drops on the time-weighted average shows which effects are the most limiting in any given parallel conjunction.

PARCONJ RUNNABLE SELF AND DESCENDANTS is the number of CPUs that can be used by a given parallel conjunction and its descendants. This metric is almost the same as PARCONJ RUNNABLE SELF, but it operates not just on a given dynamic parallel conjunction, but also on the parallel conjunctions spawned by the call-trees of its conjuncts as well. It takes their events into account when it divides time into blocks, and it counts the number of runnable tasks they represent.

The two metrics, PARCONJ RUNNABLE SELF and PARCONJ RUNNABLE SELF AND DESCENDANTS, can be used together to see whether the amount of parallelism that can be exploited by the two parallel conjunctions together is substantially greater than the amount of parallelism that can be exploited by just the outer parallel conjunction alone. If it is, then executing the inner conjunction in parallel is a good idea; if it is not, then it is a bad idea.

If the outer and inner conjunction in the dynamic execution come from different places in the source code, then acting on such conclusions is relatively straightforward. If they represent different invocations of the same conjunction in the program, which can happen if one of the conjuncts contains a recursive call, then acting on such conclusions will typically require the application of some form of runtime granularity control.

PARCONJ RUNNING SELF is the number of CPUs that are actually used by a given parallel conjunction. This metric is computed similarly to PARCONJ RUNNABLE SELF, but it does not count a runnable conjunct until gets to use a CPU, and stops counting a conjunct when it stops using the CPU (when it blocks on a future, and when it finishes).

Obviously, PARCONJ RUNNING SELF can never exceed PARCONJ RUNNABLE SELF for any parallel

conjunction at any given point of time. However, one important difference between the two metrics is that PARCONJ RUNNING SELF can never exceed the number of CPUs on the system either. If the maximum value of PARCONJ RUNNABLE SELF for a parallel conjunction does not exceed the number of CPUs, then its PARCONJ RUNNING SELF metric can have the same value as its PARCONJ RUNNABLE SELF metric, barring competition for CPUs by other parallel conjunctions (see later) or by the garbage collector.

On the other hand, some conjunctions do generate more runnable tasks than there are CPUs. In such cases, we want the extra parallelism that the system hardware cannot accommodate in the period of peak demand for the CPU to "fill in" later valleys, periods of time when the conjunction demands less than the available number of CPUs. This will happen only to the extent that such valleys do not occur after a barrier at the end of the period of time where there is an abundance of parallelism, or that the delayed execution of tasks that lost the competition for the CPU does not lead to further delays in later conjuncts through variable dependencies.

The best way to measure this effect is to visually compare the PARCONJ RUNNING SELF curves for the conjunction taken from two different systems, e.g. one with four CPUs and one with eight. However, given measurements taken from e.g. a four CPU system, it should also be possible to predict with *some* certainty what the curve would look like on an eight CPU system, by using the times and dependencies recorded in the four-CPU trace to simulate how the scheduler would handle the conjunction on an eight CPU machine. Unfortunately, the simulation cannot be exact unless it correctly accounts for *everything*, including cache effects and the effects on the GC system.

The most obvious use of the PARCONJ RUNNING SELF curve of a conjunction is to tell programmers whether and to what extent that parallel conjunction can exploit the available CPUs.

PARCONJ RUNNING SELF AND DESCENDANTS is the number of CPUs that are actually used by a given parallel conjunction and its descendants. This metric has the same relationship to PARCONJ RUNNING SELF as PARCONJ RUNNABLE SELF AND DESCENDANTS has to PARCONJ RUNNABLE SELF. Its main use is similar to the main use of PARCONJ RUNNING SELF: to tell programmers whether and to what extent that parallel conjunction and its descendants can exploit the available CPUs. This is important because a conjunction and its descendants may have enough parallelism even if the top-level conjunction by itself does not, and in such cases the programmer can stop looking for more parallelism, at least in that part of the program's execution timeline.

PARCONJ AVAILABLE CPUS is the number of CPUs available to a given parallel conjunction. Scanning through the entire trace, we can compute and record the number of CPUs being used at any given time during the execution of the program. For any given parallel conjunction, we can also compute PARCONJ RUNNING SELF AND DESCENDANTS, the number of CPUs used by that conjunction and its descendants. By taking the difference between the curves of those two numbers, we can compute the curve of the number of CPUs that execute some task *outside* that conjunction, Subtracting that difference curve from the constant number of the available CPUs gives the number of CPUs available for use by this conjunction.

This number's maximum, time-weighted average and the shape of the curve of its value over time, averaged over the different dynamic occurrences of a given static parallel conjunction, tell programmers the level of parallelism they should aim for. Generating parallelism in a conjunction that consistently exceeds the number of CPUs available for that conjunction is more likely to lead to slowdowns from overheads than to speedups from parallelism.

PARCONJ CONTINUE ON BEFORE is the frequency, for any given parallel conjunction, that the code after the conjunction will continue executing on the same CPU as the code before the

conjunction. If the parallel conjunction as a whole took only a small amount of time, then CPUs other than the original CPU will still have relatively cold caches, even if they ran some part of the parallel conjunction. We want to keep using the warm cache of the original CPU. The better the scheduling strategy is at ensuring this, the more effectively the system as a whole will exploit the cache system, and the better overall system performance will be.

Parconj continue on last is the frequency, for any given parallel conjunction, that the code after the conjunction will continue executing on the same CPU as the last conjunct to finish. If the parallel conjunction as a whole took a large amount of time, then how warm the cache of a CPU will be for the code after the conjunction depends on how recently that CPU executed either a conjunct of this conjunction or the code before the conjunction. Obviously, all the conjuncts execute after the code before the conjunction, so if the conjunction takes long enough for most of the data accessed before the conjunction to be evicted from the cache, then only the CPUs executing the conjuncts will have useful data in their caches. In the absence of specific information about the code after the conjunct preferentially accessing data that was also accessed (read or written) by specific conjuncts, the best guess is that the CPU with the warmest cache will be the one that last executed a conjunct of this conjunction. The more often the scheduling strategy executes the code after the conjunction on that CPU, the more effectively the system as a whole will exploit the cache system, and the better overall system performance will be.

6.4.4 Conjunct specific metrics

There are several simple times whose maximums, minimums, averages and variances can be computed for each static parallel conjunct.

Conjunct time as spark is the time between the conjunct's creation (which will be as a spark) and the start of its execution (when the spark will be converted into a context).

Conjunct time as context is the time between the start of the conjunct's execution and its end.

Conjunct time blocked is the total amount of time between the start of the conjunct's execution and its end that the conjunct spends blocked waiting for a future.

Conjunct time runnable: is the total amount of time between the start of the conjunct's execution and its end that the conjunct spends runnable but not running.

Conjunct time running is the total amount of time between the start of the conjunct's execution and its end that the conjunct spends actually running on a CPU.

Since every context is always either running, runnable or blocked, the last three numbers must sum up to the second (in absolute terms and on average, not in e.g. maximum or variance).

Programmers may wish to look at conjuncts that spend a large percentage of their time blocked to see whether the dependencies that cause those blocks can be eliminated.

If such a dependency cannot be eliminated, it may still be possible to improve at least the memory impact of the conjunct by converting some of the blocked time into spark time. The scheduler should definitely prefer executing an existing runnable context over taking a conjunct that is still a spark, converting the spark into a context and running that context. When there are no existing runnable contexts and it must convert a spark into a context, the scheduler should try to choose a spark whose consumed variables (the variables it will wait for) are all currently available. Of course, such a scheduling algorithm is not possible without information about which variables conjuncts consume, information that schedulers do not typically have access to, but which

it is easy to give them.

If all the sparks consume at least one variable that is not currently available, the scheduler should prefer to execute the one whose consumed variables are the *closest* to being available. This requires knowledge of the expected behaviour of the program, to wit, the expected running times of the conjuncts generating those variables. In some cases, that information may nevertheless be available, derived from measured previous runs of the program, although of course it can only ever be an approximation.

Note also this is only one of several considerations that an ideal scheduler should take into account. For example, schedulers should also prefer to execute the conjunct (whether it is a context or a spark) that is currently next on the critical path to the end of the conjunction. Knowledge of the critical path is also knowledge about the future, and similarly must also be an approximation. Ideally, the scheduler should take into account all these different considerations before coming to a decision based on balancing their relevance in the current situation.

Conjunct time after is the time, for each parallel conjunct, between the end of its execution and the end of the conjunction as a whole. It is easy to compute this for every dynamic conjunct, and to summarise it as a minimum, maximum, average and variance for any given static conjunct. If the average Conjunct time after metrics for different conjuncts in a given parallel conjunction are relatively stable (have low variance) but differ substantially compared to the runtime of the conjunction as a whole, then the speedup from the parallel execution of the conjunction will be significantly limited by overheads. In such cases, the programmer may wish to take two conjuncts that are now executed in parallel and execute them in sequence. Provided the combined conjunct is not the last-ending conjunct, and provided that delaying the execution of one of the original conjuncts does not unduly delay any other conjuncts that consume the data it generates, the resulting small loss of parallelism may be more than compensated for by the reduction in parallelism overhead. It is of course much harder to select the two conjuncts to execute in sequence if the times after for at least some of the conjuncts are *not* stable (i.e. they have high variance).

Out of order frequency: given two parallel conjuncts A and B, with A coming logically before B either by being to the left of B in some conjunction, or by some ancestor of A being to the left of an ancestor of B in some conjunction, how much time does the system spend with B running while A is runnable but not running, compared to the time it spends with either A or B running? Ideally, when A and B are both runnable but the system has only one available CPU, it should choose to run A. Since it comes logically earlier, the consumers that depend on its outputs are also likely to come logically earlier. Delaying the running of A has a substantial risk of delaying them, thus delaying the tasks depending on *their* outputs, and so on. Any efficient scheduling algorithm must take this effect into consideration.

Out of order scheduling blocks another context: when tasks are executed out of order, as described above, how much of the time is another context C blocked on a future produced by A? This metric describes how often out of order execution has a direct impact on another task.

Out of order scheduling blocks another context tc: when tasks are executed out of order, as described above, how much of the time do other contexts $D, E, F \ldots$ block waiting on a future signalled by $C, D, E, \ldots$ which, eventually, depend on A? This metric considers the transitive closure of the dependency chain measured by the previous metric.

6.4.5 Future specific metrics

Future suspend time first waits is, for each shared variable in each parallel conjunction, the minimum, average, maximum and variance of the time of the first wait event on a future for this variable by each consuming conjunct minus the time of the corresponding signal event. If this value is consistently positive, then contexts never or rarely suspend waiting for this future; if this value is consistently negative, then contexts often suspend on this future, and do so for a long time. Programmers should look at shared variables that fit into the second category and try to eliminate the dependencies they represent. If that is not possible, they may nevertheless try to reduce them by computing the variable earlier or pushing the point of first consumption later.

Future suspend time all waits is, for each future in each dynamic parallel conjunction, the minimum, average, maximum and variance of the time of all wait events minus the time of the corresponding signal event, and the count of all such wait events. This metric helps programmers understand how hard delaying the point of first consumption of a shared variable in a conjunct is likely to be. Delaying the first consumption is easiest when the variable is consumed in only a few places, and the first consumption occurs a long time before later consumptions. If the variable is consumed in many few places, many of these points of consumption are just slightly after the original point of first consumption, then significantly delaying the point of first consumption requires eliminating the consumption of the shared variable in *many* places in the code, or at least significantly delaying the execution of those many pieces of code.

Future suspend frequency first wait is, for the first wait event of each future, the number of times the signal event occurs before the wait event versus the number of times the wait event occurs before the signal event.

Future suspend frequency all waits for each wait event of each future, the number of times the signal event occurs before the wait event versus the number of times the wait event occurs before the signal event. Like Future suspend time all waits and Future suspend time first wait, these two metrics can help programmers find out how often contexts are suspended waiting for futures.

Waited future signal to conjunct end time the average, maximum and variance of the time between the signalling of a future on which another context is blocked and the end of the parallel conjunct that signalled the future. This should be computed for every parallel conjunct that signals at least one future. If the average is below the time that it normally takes for a context made runnable to begin executing on an engine, and the variance is low, then it suggests that it is better to execute the context made runnable by the signal on the same engine immediately after the current conjunct finishes.

Future signal to conjunct end time the average, maximum and variance of the time between the signalling of a future and the end of the parallel conjunct that signalled that future. As above, this should be computed for every parallel conjunct that signals at least one future. This value can be used to determine how often the optimisation described above will not be useful.

6.4.6 Presenting metrics to users

The ThreadScope tool is a graphical program. Its main display screen shows a continuous timeline of the program's execution on the horizontal axis, while along the vertical axis, the display is divided into a discrete number of rows, with one row per CPU (per engine in Mercury, per capability in Haskell). The colour of a display point for CPU n at time t shows what CPU n was doing at time

t: whether it was idle, doing GC, or executing the program. The time axis can be scaled to see an overview of the execution as a whole or to focus on a specific period during the program run. If the current scale allows for it, the display also shows the individual events that the on-screen picture is derived from. Above the per-CPU rows ThreadScope shows a plot that is virtually identical to the curve of the CPUs OVER TIME metric we defined above, and now it can do so for Mercury programs as well as for Haskell programs.

The ThreadScope GUI includes different views, including a view that can provide reports to the user, including simple scalar data about the profiled program run. This data already includes GHC's equivalent of the GC STATS and MUTATOR VS GC TIME metrics; we could easily use this to report our metrics. It is also trivial to add a new view that allows the user to input the number of call sequence counts (CSCs) executed by a sequence version of the same program on the same data, which would allow the tool to compute and display the value of the NANOSECS PER CALL metric.

We plan to modify the ThreadScope GUI so that if the user hovers the pointer over the representation of an event connected with a parallel conjunction as a whole (such as a `START_PAR_CONJUNCTION` or `END_PAR_CONJUNCTION` event), they get a menu allowing them to select one of the conjunction-specific metrics listed above. The system should then compute and print the selected metric. Similarly if the user hovers the pointer over the representation of an event connected with a specific parallel conjunct, they should be able to ask for and get the value of one of the conjunct-specific metrics.

We plan to add to ThreadScope a view that lists all the parallel conjunctions executed by the program. (The information needed for that list is available in the `STRING` events that record strings representing the static ids of the executed parallel conjunctions.) By selecting items from this list, the tool should be able to print summaries from all the dynamic occurrences of the conjunction for any of the conjunction-specific metrics. It should also be able to take users to all those occurrences in the main display.

We also plan to add to ThreadScope a mechanism that allows the user to request the generation of a plain text file in a machine-readable format containing all the metrics that may be of interest to our automatic parallelisation tool. We want our tool to be able to compare its predictions of auto-parallelised programs with metrics derived from actual measurements of such programs, so that we can help tune its performance to reduce the resulting discrepancies.

6.5 Preliminary examples

Some of our early examples use only the ThreadScope features introduced by the GHC developers. We modified Mercury's runtime system to write out compatible log files using only the events in Section 6.2. Using this system we were able to see the ThreadScope profiles of the raytracer (Figure 6.1) and the mandelbrot image generator (Figure 6.2), two of the example programs from previous chapters. These profiles were generated on an eight processor machine, using eight Mercury engines. In both figures, green areas in the eight lower rows indicate when each of the eight Mercury engines was executing a context. Orange areas indicate when that engine initiated garbage collection. The top-most row shows, at each point in time, how many engines are executing contexts.

In Figure 6.1 we can see the effect on runtime of the garbage collector (GC). The mutator is interrupted many times for short periods of time while the GC runs. This profile was generated

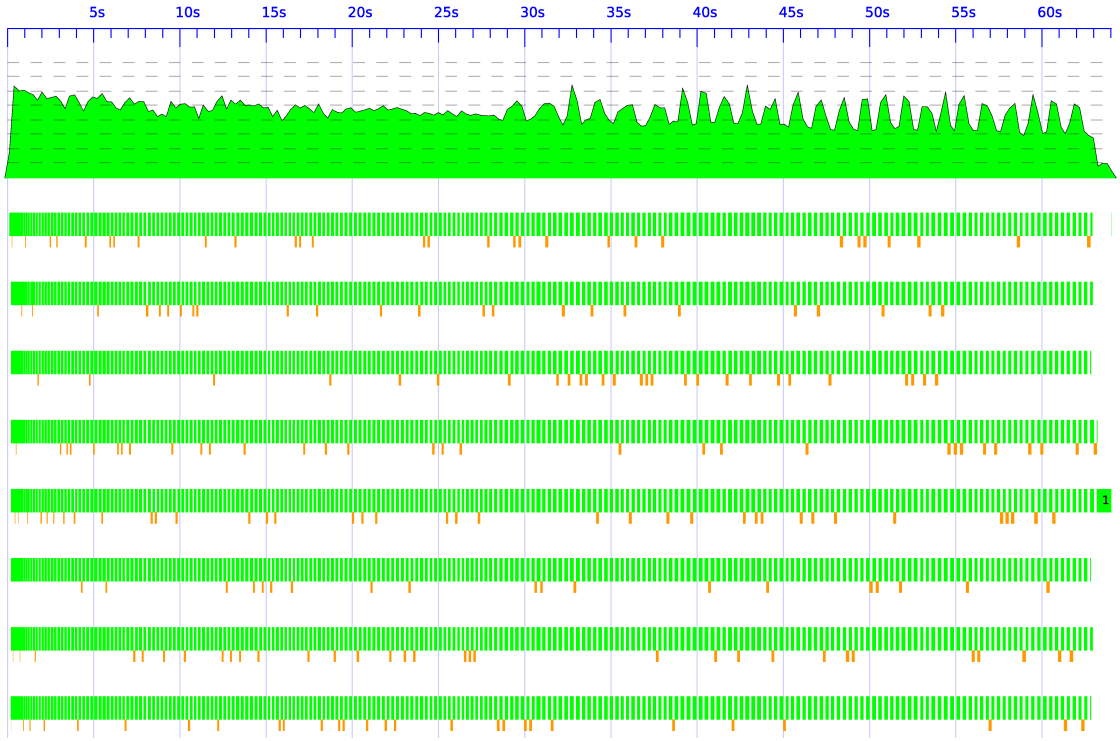


Figure 6.1: A profile of the raytracer

with the default initial heap size, therefore GC events occur quite frequently.

The mandelbrot program has a much lower memory allocation rate. One collection event can be seen roughly in the middle of the execution profile (Figure 6.2), and there is another one that cannot be seen at the beginning of the execution[2]. After the visible GC event, we can see that the third and fifth engines do not resume their work immediately; this is a potential area of optimisation that runtime system programmers could improve.

In Section 3.1 we spoke about the impact of garbage collection in detail. The data for Table 3.2 in that section was collected using ThreadScope. In particular we used the GC STATS and MUTATOR VS GC TIME metrics. Figure 6.3 shows this data for the raytracer using initial heap sizes of 16MB and 512MB. They report data averaged over eight executions, as our command line tool supports merging ThreadScope reports. We can see that the number of collections, the average time of a collection, and the total time spent performing collection vary dramatically, which we explored in detail in Section 3.1.

We have not yet implemented any more advanced metrics. However, even our simplest metrics have already been useful.

6.6 Related work and conclusion

Many parallel implementations of programming languages come with visualisation tools since (as we argued in the introduction) the people writing parallel programs need such tools to understand the actual behaviour of their programs, and the people implementing those languages find them useful too. Such tools have been created for all kinds of programming languages: imperative (such

[2]The ThreadScope tool allows us to zoom into executions, and small events such as very short garbage collections can be seen with higher magnification.

Figure 6.2: A profile of the mandelbrot program

```
Basic Statistics
    Number of engines:       4
    Total elapsed time:     13.46s
    Startup time:            1.89ms    0.0%
    Running time:           13.45s   100.0%
    Number of collections: 288
    Total GC time:           7.05s    52.4%
    Total Mut time:          6.40s    47.6%
    Average GC time:        24.48ms
```

(a) 16MB initial heap size

```
Basic Statistics
    Number of engines:       4
    Total elapsed time:      6.96s
    Startup time:            1.89ms    0.0%
    Running time:            6.95s   100.0%
    Number of collections:  26
    Total GC time:           1.25s    17.9%
    Total Mut time:          5.71s    82.1%
    Average GC time:        47.96ms
```

(b) 512MB initial heap size

Figure 6.3: Basic metrics for the raytracer

as in Visual Studio), functional (Berthold and Loogen [10], Loidl [75], Runciman and Wakeling [95] among others) and logic (Foster [39], Vaupel et al. [111] and the systems cited by Gupta et al. [47]).

Many of these visualisers share common ideas and features. For example, many systems (including ThreadScope) use a horizontal bar with different parts coloured differently to display the behaviour of one CPU, while many other systems use trees that display the structure of the computation. (The latter approach is more common among visualisers for sequential languages.) Nevertheless, each visualiser is necessarily oriented towards giving its users what they want, and the users of different systems want different information, since they are using different languages based on different concepts. For example, functional languages have no concept of parallel conjunctions, and their concept of data flow between computations is quite different from Mercury's, and users of concurrent logic programming languages such as Concurrent Prolog, Parlog and Guarded Horn Clauses have quite different concerns from users of Mercury. A concurrent logic programmer's concerns might include where to disable parallelism; a Mercury programmer might instead be concerned with where to introduce parallelism.

We know of only one visualiser for a not-inherently-concurrent logic programming language that does nevertheless support dependent AND-parallelism: VACE [111]. However, the ACE system supports OR-parallelism and independent AND-parallelism as well as dependent AND-parallelism, and its features do not seem designed to help programmers exploit dependent AND-parallelism better. As far as we know, no-one has built a profiler for a dependent AND-parallel logic programming language with the capabilities that we propose. We haven't yet built one either, but we are working on it.

Chapter 7

Conclusion

In this chapter we summarise the contributions of the previous four chapters, emphasising on how all the chapters fit together.

Chapter 3 investigated a number of problems that affected the performance of parallel Mercury programs, because before we could work on automatic parallelism, we had to fix them. These problems included the negative performance effects of garbage collection (GC), the issues with the execution of sparks, and the behaviour of contexts in right-recursive code. While in Chapter 3 we did not fix the negative performance effects of garbage collection or the behaviour of contexts in right-recursive code, we did find workarounds that reduced the impact of both these problems. We also made general improvements, such as an efficient implementation of work stealing, which fixed a spark scheduling problem. Without work stealing the runtime system would commit to either the parallel or sequential execution of a spark too early, which made a frequently incorrect assumption about whether an idle engine would be available to execute the spark. By using work stealing, the scheduling decision is made when the spark is either executed by its owner, or stolen by a thief, which is the first point in time that it is known if there is an idle engine (the thief).

Chapter 4 describes a unique way to use profiling data to find profitable dependent parallelism. We made several novel contributions. Most significantly, to the best of our knowledge, our analysis is the first that estimates how dependencies affect the parallelism in parallel conjunctions of any size and with any number of dependencies. As the analysis algorithm executes, it mimics the same execution steps that our runtime system would do, as described in Chapter 2, the background, and Chapter 3, our modifications to the runtime system. Chapter 4's other contributions include an algorithm that selects the best parallelisation of any conjunction from the 2^{N-1} possibilities. This algorithm will not explore some parts of the search space that will not provide a solution better than the best solution so far (branch and bound). It will also switch to a greedy algorithm that runs in $O(N)$ time if the search is taking too long. For small values of N a complete search is used and for larger values of N a greedy (linear) search is sued.

In Chapter 5 we describe and fix a pathological performance problem that occurs in most singly recursive code. Our solution is a novel program transformation that fixes this problem by restricting the number of contexts that such code can use. Our transformation also allows us to take advantage of tail recursion inside parallel conjunctions, if the original sequential code supports tail recursion. This is important as it allows parallel Mercury programs to operate on data with unbounded size. Programs that work on large data structures are usually also those that programmers wish to parallelise, therefore this transformation is critical for practical parallel

programming in Mercury. While the transformation in this chapter supersedes the workarounds in Chapter 3, the context limit work around is still useful when the transformation presented in this chapter cannot be applied, such as in divide and conquer code.

In Chapter 6 we proposed modifications to both Mercury and ThreadScope that allow programmers to visualise the parallel profile of their Mercury program. We have enough of this system implemented to know that it is feasible and that the completed system will be able to provide some very powerful analyses and visualisations of Mercury programs. We described many different analyses and how each could be implemented. We believe that these will be useful for programmers trying to parallelise their Mercury programs, as well as implementors working on the runtime system and/or automatic parallelisation tools. Someone working on the runtime system can use Mercury's support for ThreadScope to profile the parallel runtime system and tune it for better performance, while someone working on automatic parallelism can use data gained through ThreadScope to spot performance problems that need attention, and to adjust the values that represent the costs of parallel execution overheads in the cost model.

7.1 Further work

Throughout this dissertation we have discussed further work that may apply to each contribution. In this section we wish to describe further work that may apply to the whole system.

In Chapter 4, we said that the estimation of parallelism in dependent conjunctions (overlap analysis) should be more aware of recursive code, such as loops. This should include an understanding of how the loop control transformation affects the execution of loops, including the different runtime costs that apply to loop controlled code. The overlap analysis could also be improved by taking account of how many processors are available to execute the iterations of the loop. This information, the number of available processors and the cost of each iteration, can be used to apply other loop throttling methods, such as chunking.

Both loop control, and analysis of how recursion affects parallelism, should be applied to code of other recursion types such as divide and conquer code. Provided that the two recursive calls have parallel overlap, one can easily introduce granularity control near the top of the call graph of a divide and conquer predicate.

In Chapter 6 we did not introduce any events for the loop control work, or discuss loop control at all. This is also an area for further work: we must design events and metrics that can measure loop controlled code. It would also be interesting to compare the profiles of loop controlled code with the profiles of the same code without loop control.

We briefly discussed (above and in Chapter 6) that ThreadScope can be used to gather data that could be used as input for the auto-parallelism analysis. This can inform the auto-parallelism analysis in several ways. First, it will be able to convert between real time (in nanoseconds) and call sequence counts. Second, it can analyse the profile to determine the actual costs (in either unit) of the parallelisation overheads, and then use those metrics in re-application of the auto-parallelism analysis. This may generate more accurate estimates of the benefits of parallelism, and could result in better parallelisations. This is likely to be useful as different machines and environments will have different costs for many overheads. Making it easy to determine the costs of overheads will make the auto parallelisation tool easier to use. Finally, the auto parallelism analysis will be able to determine if parallelising a particular computation was actually a good idea. It could then provide a report about each parallel conjunction, saying how much that conjunction *actually* benefited from

parallelism compared to the analysis tool's estimate. Such reports would be extremely valuable when extending the overlap analysis to account for more of the many factors affecting performance.

7.2 Final words

We have created a very powerful parallel programming environment, with good runtime behaviours (Chapters 3 and 5), a very good automatic parallelisation system (Chapter 4) and the basis of a useful visual profiler (Chapter 6). We have shown that this system is useful for parallelising some small and medium sized programs, getting almost perfect linear speedups on several benchmarks. Our system provides a solid basis for further research into automatic parallelism. We believe that further development along these lines will produce a system that is capable of automatically parallelising large and complex programs. We look forward to a future where parallelisation is just another optimisation, and programmers think no more of it than they currently think of inlining.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, second edition, 1984.

[2] Sarita V. Adve and Hans-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[3] Khayri A. M. Ali and Roland Karlsson. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the 1967 AFIPS Spring Joint Computer Conference*, pages 483–485, Atlantic City, New Jersey, 1967. ACM.

[5] Joe Armstrong. *Concurrent Programming in ERLANG*. Prentice Hall, second edition, 1996.

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, Puerto Vallarta, Mexico, 1998.

[7] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the (v, g)-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, 1989.

[8] Uri Baron, Jacques Cassin de Kergommeaux, Max Hailperin, Micheal Ratcliffe, Phillippe Robert, Jean-Claude Syre, and Harald Westpal. The parallel ECRC Prolog system PEPSys: an overview and evaluation results. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[9] Geoff Barrett. occam 3 reference manual. online `http://www.wotug.org/occam/documentation/oc3refman.pdf` retrieved 22/11/2012, March 1992. Draft.

[10] Jost Berthold and Rita Loogen. Visualizing parallel functional program runs: Case studies with the Eden trace viewer. In *Proceedings of the International Conference ParCo 2007*, Parallel Computing: Architectures, Algoritthms and Applications, Jülich, Germany, 2007.

[11] Johan Bevemyr, Thomas Lindgren, and Hkan Millroth. Reform Prolog: the language and its implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298, Budapest, Hungary, 1993.

[12] Guy Blelloch. NESL: A nested data-parallel language. (version 3.1),. Technical Report CMU-CS-95-170, Carnegie-Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, September 1995.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 356–368, Washington, District of Columbia, 1994. IEEE Computer Society.

[14] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18:807–820, 1988.

[15] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Notices*, 40(6): 261–268, June 2005.

[16] Mark T. Bohr, Robert S. Chau, Tahir Chani, and Kaizad Mistry. The high-k solution. *IEEE Spectrum*, 44(10):29–35, October 2007.

[17] Paul Bone. Calculating likely parallelism within dependant conjunctions for logic programs. Honours thesis, University of Melbourne, Melbourne, Australia, October 2008.

[18] Paul Bone and Zoltan Somogyi. Profiling parallel Mercury programs with ThreadScope. In *Proceedings of the 21st Workshop on Logic-based methods in Programming Environments (WLPE 2011)*, pages 32–46, July 2011.

[19] Paul Bone, Zoltan Somogyi, and Peter Schachte. Estimating the overlap between dependent computations for automatic parallelization. *Theory and Practice of Logic Programming*, 11 (4–5):575–591, 2011.

[20] Paul Bone, Zoltan Somogyi, and Peter Schachte. Controlling loops in parallel Mercury code. In *Proceedings of the International Conference on Declarative Aspects and Applications of Multicore Programming*, Philadelphia, Pennsylvania, January 2012.

[21] T. Brus, M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer, and H.P. Barendregt. CLEAN - a language for functional graph rewriting. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes on Computer Science*, pages 364–384, Portland, Oregon, 1987. Springer-Verlag.

[22] G. L. Burn. Overview of a parallel reduction machine project II. In *Proceedings of Parallel Architectures and Languages Europe 1989*, volume 365 of *Lecture Notes in Computer Science*, pages 385–396, Eindhoven, The Netherlands, June 1989. Springer Berlin Heidelberg.

[23] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 136–143. ACM, 1980.

[24] D.R. Butenhof. *Programming With Posix Threads*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997. ISBN 9780201633924.

[25] Lawrence Byrd. Prolog debugging facilities. Technical report, Department of Artificial Intelligence, Edinburgh University, 1980.

[26] 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.

[27] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, Nice, France, 2007.

[28] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM Symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, Las Vegas, Nevada, 2005. ACM.

[29] Keith Clark and Steve Gregory. Notes on systems programming in PARLOG. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 299–306. ICOT, 1984.

[30] Keith Clark and Steve Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

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

[32] 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.

[33] Microsoft Corporation. Processes and threads. `http://msdn.microsoft.com/en-us/library/ms684841`, April 2012.

[34] Charlie Curtsinger and Emery D. Berger. Stabilizer: Enabling statistically rigorous performance evaluation. Technical report, University of Massachusetts, Amherst, 2012.

[35] L. Dagum and P. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, Janurary–March 1998.

[36] Saumya K. Debray, Nai-Wei Lin, and Manuel Hermnegildo. Task granularity analysis in logic programs. *SIGPLAN Notices*, 25(6):174–188, June 1990.

[37] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, September 1965.

[38] John T. Feo, David C. Cann, and Rodney R. Oldehoeeft. A report on the Sisal language project. *Journal of Parallel and Disributed Computing*, 10:349–366, 1990.

[39] Ian Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[40] Ian Foster and Stephen Taylor. Flat Parlog: A basis for comparison. *International Journal of Parallel Programming*, 16(2):87–125, April 1987.

[41] A. Geist. *PVM: Parallel Virtual Machine :a Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Press, 1994. ISBN 9780262571081.

[42] Google. The Go programming language specification, 2012. retrived from `http://golang.org/ref/spec` on 2012-12-05.

[43] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massachusetts, June 1982.

[44] 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.

[45] Gopal Gupta and Manuel Hermenegildo. ACE: And/Or-parallel copying-based execution of logic programs. *Parallel Execution of Logic Programs*, 569:146–156, 1991.

[46] Gopal Gupta and p Enrico Pontelli. Optimization schemas for parallel implementation of non-deterministic languages and systems. *Software: Practice and Experience*, 31(12):1143–1181, 2001.

[47] Gopal Gupta, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: A survey. *ACM Transactions on Programming Languages and Systems*, 23:2001, 1995.

[48] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[49] Robert H. Halstead. Implementation of MultiLisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on List and Functional Programming*, pages 9–17, Austin, Texas, 1984.

[50] Robert H. Halstead, Jr. MultiLisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[51] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990. MIT Press.

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

[53] Tim Harris, Simon Marlow, and Simon Peyton-Jones. Haskell on a shared memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, Tallinn, Estonia, September 2005. ACM.

[54] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[55] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, Ralph Becket, and Mark Brown. The Mercury language reference manual, version 10.04. Available from http://www.mercurylang.org/information/documentation.html, 2010.

[56] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual Symposium on the Principles of Distributed Computing*, PODC '02, pages 280–289, Monterey, California, 2002.

[57] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. Lopez, and G. Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85, 1999.

[58] M. V. Hermenegildo and K. J. Greene. The &-Prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9(3–4):233–257, 1991.

[59] Manuel V. Hermenegildo and Francesca Rossi. Strict and nonstrict independent AND-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[60] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[61] Charles Antony Richard Hoare. Communicating sequential processes. *Communincations of the ACM*, 21(8):666–677, August 1978.

[62] Intel Corporation, editor. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2B: Instruction Set Reference, N-Z, page 4:63. Intel Corporation, March 2009.

[63] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 81–92, Edinburgh, Scotland, 2009.

[64] Simon Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, fourth edition, 2003.

[65] Brian W. Kerninghan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[66] Andy King, Kish Shen, and Florence Benoy. Lower-bound time-complexity analysis of logic programs. In *Proceedings of the 1997 International Symposium on logic Programming*, ILPS '97, pages 261–275, Port Washington, New York, 1997. MIT Press.

[67] Robert Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP Congress*, pages 569–574. North Holland Publishing Company, 1974.

[68] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a high-performance parallel Lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 81–90, Portland, Oregon, 1989. ACM.

[69] John Launchbury and Simon Peyton-Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[70] Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[71] Xavier Leroy. The effectiveness of type-based unboxing. In *Proceedings of the Workshop on Types in Compilation*, 1997.

[72] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[73] INMOS Limited. *OCCAM Programming Manual*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1984. ISBN 0-13-629296-8.

[74] Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, June 2009.

[75] Hans Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computer Science, University of Glasgow, March 1998.

[76] 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.

[77] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora OR-parallel Prolog system. *New Generation Computing*, 7(2–3):243–271, 1990.

[78] Luc Maranget and Fabrice Le Fessant. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998.

[79] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. *SIGPLAN Notices*, 44(9):65–78, 2009.

[80] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Notices*, 46(12):71–82, September 2011.

[81] Michael. Matz, Jan. Hubička, Andreas. Jaeger, and Mark. Mitchell. System V application binary interface, AMD64 architecture processor supplement. `http://www.x86-64.org/documentation/abi.pdf`, 2009. Draft version 0.99.

[82] Message Passing Interface Forum. MPI: A message-passing interface standard version 2.2, September 2009.

[83] Leon Mika. Software transactional memory in Mercury. Technical report, Department of Computer Science and Software Engineering, Melbourne University, Melbourne, Australia, October 2007.

[84] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[85] Robin Milner. The polyadic π-calculus: a tutorial. Technical report, Laboratory for Foundations of Computer Science, Computer Science Deptment, University of Edinburgh, The King's Buildings, Edinburgh, Scotland, October 1991.

[86] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[87] 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.

[88] Oracle. Concurrency. `http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`, April 2012.

[89] Simon Peyton-Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In *Proceedings of Parallel Architectures and Languages Europe 1989*, volume 365 of *Lecture Notes in Computer Science*, pages 193–206, Eindhoven, The Netherlands, June 1989. Springer Berlin Heidelberg.

[90] Simon Peyton Jones, Roman Leshchinskly, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, page 138, Bangalore, India, 2008. Springer-Verlag, Berlin, Heidelberg.

[91] Enrico Pontelli and Gopal Gupta. Non-determinate dependent and-parallelism revisited. Technical report, Laboratory for Logic, Databases, and Advanced Programming; New Mexico State University, Las Cruces, New Mexico, February 1996.

[92] Enrico Pontelli, Gopal Gupta, Francesco Pulvirenti, and Alfredo Ferro. Automatic compile-time parallelization of prolog programs for dependent AND-parallelism. In *Proceedings of the 14th International Conference on Logic Programming*, pages 108–122, Leuven, Belgium, 1997.

[93] John Rose and Guy Steele Jr. C*: An extended (c) language for data parallel programming. Technical Report PL87-5, Thinking Machines Corporation, April 1987.

[94] Peter Ross, David Overton, and Zoltan Somogyi. Making Mercury programs tail recursive. In *Proceedings of the Ninth International Workshop on Logic-based Program Synthesis and Transformation*, Venice, Italy, September 1999.

[95] Colin Runciman and David Wakeling. Profiling parallel functional computations (without parallel machines). In *Proceedings of the Glasgow Workshop on Functional Programming*. Springer, July 1993.

[96] Vijay A. Saraswat. The concurrent logic programming language CP: Definition and operational semantics. Carnegie-Mellon University, September 1986.

[97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2): 99–116, 1997.

[98] Kish Shen. Overview of DASWAM: Exploitation of dependent AND-parallelism. *The Journal of Logic Programming*, 29(1–3):245–293, October–December 1996.

[99] Kish Shen, Vítor Santos Costa, and Andy King. Distance: a new metric for controlling granularity for parallel execution. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 85–99, Manchester, United Kingdom, 1998. MIT Press.

[100] Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, pages 256 – 275, 2000.

[101] 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.

[102] ThreadScope summit attendees. First ThreadScope summit. personal correspondance, July 2011.

[103] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr Dobb's Journel*, 30(3), March 2005.

[104] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[105] Jéröme Tannier. Parallel Mercury. Master's thesis, Institut d'informatique, Facultés Universitaires Notre-Dame de la Paix, 21, rue Grandgagnage, B-5000 Namur, Belgium, 2007.

[106] Stephen Taylor, Shmuel Safra, and Ehud Shapiro. A parallel implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming*, 15(3):245–275, June 1986.

[107] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, Pennsylvania, May 1996.

[108] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[109] Kazunori Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.

[110] Kazunori Ueda and Masao Morita. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

[111] R. Vaupel, E. Pontelli, and G. Gupta. Visualization of And/Or-parallel execution of logic programs. In *Proceedings of the 14th International Conference on Logic Programming*, pages 271–285, Leuven, Belgium, July 1997.

[112] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming*, pages 79–89, San Jose, California, 2007.

[113] Peter Wang. Parallel Mercury. Honours thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, October 2006.

[114] 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.

[115] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.