Parallel Mercury

Peter Wang
Supervisor: Zoltan Somogyi

October 2006

Abstract

Mercury is a purely declarative programming language that is highly suited to parallel execution. We present two main contributions towards parallel execution of Mercury programs. The first is a source-to-source transformation that allows goals with dependencies to execute in parallel, with no negative impact on code which doesn't make use of it. The second is the design and implementation of a new parallel execution mechanism that allows parallel code to execute far more efficiently than in the existing Mercury implementation. We present experimental results that show the effectiveness of our changes.

Contents

1 Introduction

With symmetric multiprocessing (SMP) machines becoming increasingly common at the consumer level, a question naturally arises as to how we can take advantage of such hardware. The most common way of achieving a speedup on parallel hardware is writing programs that use explicit threads of control in imperative programming languages. Unfortunately, writing and reasoning about threaded programs is notoriously difficult, see e.g. [31]. It is hard to be sure of one's own multithreaded code, as bugs in multithreaded programs often cannot be reproduced reliably. Data parallelism is another, complementary, approach to task-based parallelism, but can only be exploited when a single operation is performed on multiple data. As such, it usually only benefits number-crunching programs.

It has long been recognised that declarative programming languages, including logic and functional programming languages, are potentially better suited to parallel programming [11, 28]. Declarative programming has a clearer separation of "what" the program computes from the details of "how" the computation will take place. The language implementor has greater flexibility to execute the program in parallel without changing the semantics of the program, and the emphasis on referential transparency and computation without mutating state neatly sidesteps many pitfalls of parallel programming in imperative languages.

Even in the space of declarative programming languages, we feel that Mercury is particularly suited to parallelism. Many functional and logic languages are impure, for example the Lisp, ML and Prolog families, and all real programs in those languages venture outside their pure subsets. Impurity, i.e. side effects, makes it very difficult for a compiler to automatically infer whether one part of the program depends on another part having been executed previously. Pure functional languages, such as Haskell, often use a lazy evaluation strategy. Laziness introduces considerable overheads even before parallelism, and dealing with contention for shared thunks (suspended computations) by multiple threads incurs further overheads [15].

Mercury programs can already get very good performance running on uniprocessor hardware. The aim of this project is to get the same out of parallel hardware. We want to make it practical, even easy, to write and modify existing Mercury programs that show real performance benefits on consumer-level symmetric multiprocessing machines.

The rest of this report is organised as follows. We will assume the reader is familiar with the general ideas of logic programming. We begin with a brief introduction to Mercury and describe the parallelism support already present. Next, we describe some related work for other programming languages. Section 3 describes how we extended the Mercury implementation to allow goals with dependencies to execute in parallel. Parallel execution is more complex than sequential execution, and the associated overheads always threaten to swamp the potential speedup that can be realised. Section 4 discusses some overheads in the Mercury implementation and the improvements we made to reduce those overheads. Section 5 discusses some additional issues uncovered during the project. In section 6 we experimentally evaluate the performance of parallel Mercury programs after our modifications. We conclude with a summary of the contributions of this project and discuss potential further work in this area.

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.

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

Figure 1: append/3 in Mercury

2 Background

2.1 Mercury

Mercury is a functional/logic programming language developed at the University of Melbourne, with an emphasis on reliability and efficiency. It was designed for the development of large and maintainable software systems. The following is a brief overview of the language; a full description can be found in [16].

Mercury's syntax is derived from Prolog. It has a strong type system which is broadly similar to the Hindley-Milner type systems found in the ML languages and Haskell.

Mercury has a strong *mode* system. A mode of a predicate specifies the initial and final instantiation states of arguments in a call to that predicate. A predicate can have multiple modes; each is a *procedure*. For example, one mode of the standard `append/3` predicate is `append(in, in, out)`. A call to that procedure must have the first two arguments instantiated before the call ("in"). The mode also guarantees that the variable in the last argument position will be bound to the result when the procedure returns ("out"). The modes in which a predicate can be used must be declared or inferred. A called predicate imposes a set of constraints on the instantiation states of arguments in the call: input variables must be ground at the time of the call, and output variables will be ground after the call. The compiler will attempt to rearrange goals in a procedure so that all mode constraints are satisfied. If that is not possible then the program will be rejected by the compiler. The end result is that we will always know, at compile time, which mode of a predicate is called at each call site, and which variables are instantiated when.

Mercury has a strong *determinism* system. The determinism of a procedure says how many solutions it can produce for a given set of inputs. Most procedures produce exactly one solution, and therefore have the determinism `det` ("deterministic"). Procedures which may produce no solutions or one solution are `semidet` ("semi-determinisitic"). Procedures which produce one or more solutions are `multi`. Procedures which produce zero or more solutions are `nondet`. The determinism of goals is inferred from the determinism of its subgoals. The inferred determinism of a procedure must match any declared determinism, otherwise the program is rejected.

A Mercury implementation of `append/3` is given in figure 1. The `pred` line declares the predicate and the types of its arguments (all three are lists whose elements are of the same type T). The `mode` lines declare the modes of the predicate and their associated determinism categories.

Mercury predicates are pure by default. The set of solutions produced by a call to a pure predicate depends only on the values of its input arguments. A variable can only be bound once, and values are immutable. Even predicates that perform I/O are pure, since they must take a pair of arguments that represent the state of the world before and after a call to that predicate. I/O states are guaranteed to be unique by the mode system.

Mercury has an impurity system which allows impure predicates to be written. However its use is highly discouraged. We will discuss the implication of the impurity system on parallelism in section 5.3 but otherwise we will assume that the code we deal with is pure.

Everything we have said above about predicates applies to functions as well, which are only different to predicates in syntax. Together, type, mode and determinism checking ensure that many of the more trivial programming errors are caught at compile time rather than manifesting at run-time. They also allow the compiler to generate very efficient code. Purity can sometimes be onerous on the program writer but makes reading and maintaining code much easier, as all dependencies are explicitly written out. Finding where dependencies exist also happens to be very important for parallelism, both for humans and the Mercury compiler.

2.2 Previous work on parallelism in Mercury

It is useful at this stage to distinguish between *parallelism* and *concurrency*, as in [21] and elsewhere. A parallel program may use multiple processors to gain performance, but parallelism does not change the meaning of the program. A concurrent program explicitly deals with multiple things happening simultaneously; that is essential to the program. The two concepts are orthogonal: a concurrent server program can handle multiple client requests "at once" while running on a single processor, whereas a parallel program to compute the value of π converges on the same value no matter how many processors it utilises.

Existing support for parallelism in Mercury was designed and implemented by Thomas Conway [10]. Two aspects of his work are directly relevant to this report. Firstly, he added the parallel conjunction operator ("`&`") at the Mercury language level. The syntax "`A & B`" is an indication from the programmer to the compiler that the two goals `A` and `B` would benefit from parallel execution. In Conway's implementation, `A` and `B` must be independent of one another and must both be deterministic (produce one solution). Due to the availability of purity information and Mercury's mode system, we can check the independence of a parallel conjunction "`A & B`", at compile-time, by ensuring that the set of variables bound by `A` and `B` are disjoint, and that `A` and `B` do not depend on variables bound by the other. Mercury does not allow the unification of two free variables so we do not have to consider variable aliasing when detecting independence. Nor do we have to consider partially instantiated data structures as they are not supported by the Mercury implementation yet. Section 3 describes our relaxation of the independence constraint.

Conway also described the design and implementation of the parts of the Mercury compiler and runtime system that allowed parallel execution of code. Although a good foundation, it was not suitable for running "realistic" parallel programs. Section 4 describes some improvements we made to the existing runtime system to support parallel execution.

2.3 Parallelism in Prolog and related languages

As noted earlier, Mercury inherits syntax and the logic programming paradigm from Prolog. Here we give an extremely brief introduction to the efforts behind parallelising logic programming languages, especially Prolog, so that we can contrast the design decisions made in the parallelism work on Mercury so far. Most of the information here is due to [11] and [14].

There are two main sources of parallelism that researchers try to exploit in logic programming languages. AND-parallelism attempts to execute goals in a conjunction in parallel to obtain a speedup. OR-parallelism attempts to search the branches of a disjunction in parallel, rather than waiting for one disjunct to fail before trying the next alternative. Both present their own challenges for efficient execution but we will only consider AND-parallelism here, as that is our interest. Most Mercury programs make very little use of nondeterminism (backtracking) so OR-parallelism would be much less useful.

In Prolog, multiple workers executing different goals in parallel may try to bind the same logic variable to conflicting values. Some systems only support parallel execution of independent goals (goals which do not share free variables) to avoid the overheads required to deal with this possibility. The problem is then in the detection of independence, which can be done at compile-time, at run-time, or partly at compile-time and partly at run-time. Detection at run-time needs to be done very efficiently (consider a large partially instantiated data structure) but there is always an overhead. On the other hand, compile-time analyses are not complete. Due to the impurity of the language and the lack of mode information, a Prolog compiler must work very hard to detect independence at compile time (see for example [6] which performs abstract interpretation-based dataflow analyses over various domains to detect independence). Systems which use a combined approach try to reduce the number of run-time independence tests and make the run-time tests as simple as possible. &-Prolog [20] is an example of a system which can generate Conditional Graph Expressions (CGEs) which are simple tests evaluated at run-time to decide if a goal should be executed in parallel or not. CGEs can be simplified to reduce runtime overhead, but are nevertheless expensive and may fail to detect the independence of some goals.

Other AND-parallel systems do allow parallel goals to share free variables, which act as communication channels; this is called dependent AND-parallelism. Goals are then viewed as being either producers or consumers of each variable they share with other goals. A consumer must block its execution if the shared variable it needs is not available yet. Some systems use mode declarations to classify goals as producers or consumers (e.g. Parlog [9]). In DDAS-based schemes [29], the leftmost subgoal (which may change dynamically) that has access to a shared variable is designated the producer. Detecting whether a goal is the leftmost subgoal of course incurs overheads.

Backtracking and dependent AND-parallelism do not mix well. As an example, a parallel conjunct which fails would require the killing of all the active parallel conjuncts to its right, and induce backtracking of the parallel conjunct to its left. Accounting for these possibilities requires a lot of overhead. Most dependent AND-parallel systems therefore don't implement Prolog but rather "committed-choice" or "concurrent logic" languages (e.g. Concurrent Prolog, Parlog, GHC). *Guard* expressions are introduced between the clause head and body, and are executed when the head of a predicate clause *matches* a call (not full unification). Execution commits to one of the clauses for which the guards succeeds; all other alternatives are discarded. No choicepoints (records of alternatives to be tried) are ever created and there is no backtracking, so these languages do not have to contend with nondeterminism in parallel conjunctions. For efficiency reasons, "flat" versions of these languages restricted the guards to certain predefined operations.

Mercury's mode system forbids one of the most troublesome features for AND-parallelism, namely the unification of two free variables. Coupled with the purity of Mercury code, this makes completely static detection of independent goals almost trivial. For dependent goals, the mode system provides us with complete information about producer/consumer relationships so we don't need any complicated schemes to determine that information at run-time. On the other hand, the problems with nondeterminism and parallel execution do not magically disappear in Mercury. Our solution is pragmatic: only allow parallel conjuncts that succeed exactly once. Of course, nondeterministic code can still be used within a parallel conjunct, but the nondeterminism must be encapsulated (e.g. by using an "all-solutions" predicate) so as not to propagate all the way up to the top of the parallel conjunct.

3 Dependent parallel conjunction

As described earlier, the Mercury implementation previously supported an independent parallel conjunction operator. For simplicity, goals in a parallel conjunction were restricted to be deterministic (succeed exactly once), which was assured at compile-time due to the strong determinism system.

In this section we describe a relaxation of the independence criteria for parallel conjuncts. We will allow dependencies between conjuncts of a parallel conjunction if the parallel conjunction would be well-moded when treated as a sequential conjunction. Another way to say it is, we will allow any sequential conjunction of deterministic conjuncts to be converted into a parallel conjunction by replacing the "`,`" connectives by "`&`" connectives. Therefore the following well-modedness criteria for parallel conjunctions follow from the well-modedness criteria of sequential conjunctions.

Consider a single parallel conjunction $C_1 \ \& \ \cdots \ \& \ C_n$. Let us call variables which are bound within one parallel conjunct and used in one or more additional parallel conjuncts *shared* variables. Every shared variable must be bound by exactly one parallel conjunct, the producer, and all other conjuncts must be consumers of that variable or else ignore it. In addition, there must exist a single left-to-right ordering of the parallel conjuncts such that, for all shared variables in the conjunction, a conjunct which produces a shared variable comes before all the conjuncts which consume that shared variable. (This last condition prevents circular dataflows.)

Operationally, if a consumer of a shared variable is executed before the variable has been bound by a producer, then execution of the consumer must suspend until the variable's value is ready. In practice, for parallel execution to be worthwhile, a consuming goal must have at least some prior work to do that doesn't require the value of the shared variable.

3.1 Implementation strategy

A consumer of a shared variable needs to know if the shared variable has been produced yet and what value it was bound to. A *future* (also known as a promise) is a proxy for a value which is not yet available. We introduce future objects into our run-time system, in which we can record whether the variable proxied by a future has been bound yet, and if so what value it has been bound to. We also record in a future the list of which consumers are suspended, waiting for the variable to be bound. The operations relating to futures are:

- `new_future(Future)` allocates a new future object on the heap and binds the address to the variable `Future`.
- `signal(Future, Value)` says that the variable represented by the future has now been instantiated with the value `Value`. We say that the future has been signalled or *resolved*. Any call to `wait` which was suspended on the future is resumed. It is an error to signal a future more than once.
- `wait(Future, Value)` causes the caller to suspend if the future is unresolved. Once the future is resolved, execution resumes and the variable `Value` is bound to the value that the future was signalled with.
- `lookup(Future, Value)` is equivalent to `wait(Future, Value)` but assumes the future is resolved.

As with futures themselves, these operations are implementation details which are invisible to the Mercury programmer.

Our implementation of dependent AND-parallelism works by performing a source-to-source transformation of the parallel Mercury program that replaces references to shared variables with references to futures. Instead of passing a value to a consumer directly (the value may not be available yet) the consumer must ask for the value from a future by calling the built-in operation `wait`. Producers of shared variables are transformed to call the built-in `signal` operation on the future when the variable that it represents is bound. `wait` suspends execution of its caller if the future was not previously signalled. `signal` will record the variable's value and cause consumers that are suspended in `wait` calls to resume execution. Our mode restrictions ensure that a future will always be signalled exactly once, so conflicting bindings are not possible, and all suspended `wait` calls will resume eventually.

By signalling as soon as possible and delaying waits as late as possible, we try to minimise the chance that a future is still unresolved when it is actually needed, and so try to extract the maximum amount of parallelism out of the code as given.

3.2 Source-to-source transformation algorithm

The source-to-source transformation is implemented as a pass in the Mercury compiler that works with the high-level representation of the module being compiled, called the high-level data structure (HLDS). At this stage the module has been checked for type-, mode- and determinism correctness. Mode checking will have reordered goals according to the criteria described earlier. The HLDS stores parallel conjunctions as flat lists, i.e. "`A & (B & C)`" is treated as a single list `[A, B, C]`. Parallel conjunctions can still be nested if there is a different goal type between an outer and inner parallel conjunction, e.g. "`(A & (B, (C & D)))`"—the sequential conjunction prevents the two parallel conjunctions from being treated as one. Internally, Mercury predicates are transformed into "superhomogeneous form". Most importantly, for us, it means all arguments in procedure calls will be variables.

The transformation algorithm is given in figure 2, which will be explained below. The principle, however, is more important than the exact details.

First, we analyse a parallel conjunction to find the set of shared variables, i.e. variables appearing in multiple conjuncts but not instantiated before the parallel conjunction. The HLDS records the mapping of variables to their initial instantiation states (free or ground) at the start of each procedure, in a structure called an "instmap". Each goal carries with

1. For each parallel conjunction (from inner parallel conjunctions outwards):
   (a) Find the set of shared variables, V.
   (b) Find the producer conjunct and the consumer conjuncts for each $v \in V$.
   (c) Before the parallel conjunction, insert a call $\mathit{new\_future}(F_v)$ for each $v \in V$.
   (d) In each conjunct that produces a shared variable v, insert a call $\mathit{signal}(F_v, v)$ directly after the leftmost goal that contains v. In branching goals, insert a copy of the call into each branch.
   (e) In each conjunct that consumes a shared variable v, invent a new variable v' and insert a call $\mathit{wait}(F_v, v')$ directly before the leftmost goal that contains v. In branching goals, insert a copy of the call into each branch. Replace occurrences of v in the branch with v'.
2. In each sequential conjunction:
   (a) Find contiguous sequences of zero or more *wait* calls, followed by a call C to a procedure P, followed by zero or more *signal* calls, where there is at least one *wait* or *signal* call in the sequence, and P is a procedure of which we have the source code.
   (b) Let $\mathrm{vars}(C)$ be the set of variables in the call goal C. Remove all the $\mathit{wait}(F_v, v)$ and $\mathit{signal}(F_v, v)$ calls in the matching sequence for which $v \in \mathrm{vars}(C)$. Let ϕ be a substitution of variables to futures for the corresponding pairs of variables and futures in the removed *wait* and *signal* calls.
   (c) Replace the call $P(a_1, a_2, \cdots, a_n)$ with a call $P'(\phi(a_1), \phi(a_2), \cdots, \phi(a_n))$ where a_i are the arguments in the call C and P' is a *parallel variant* of P for the set of argument positions which have been replaced by futures (the *calling pattern*). If the calling pattern has not been previously seen for the procedure P, then note down on a global queue that the parallel variant P' needs to be produced.
   (d) For each $\mathit{wait}(F_v, v)$ or $\mathit{signal}(F_v, v)$ call removed, insert a $\mathit{lookup}(F_v, v)$ call after the call to P'.

Figure 2: The dependent parallel conjunction transformation algorithm for a single procedure.

it the change in instantiation states of variables across that goal; these changes are called "instmap deltas". One of the parallel conjuncts referring to a shared variable must be the producer of that variable, and the rest of the conjuncts that refer to the variable must be consumers. We can discover which variables are shared variables and which conjunct produces and which conjuncts consume each shared variable by looking at the instmap deltas.

For each shared variable we insert a call to `new_future` before the parallel conjunction to allocate a future object that will be passed around in place of that variable. In the producer conjunct of a shared variable, we insert a `signal` call as soon as possible after the variable is instantiated, i.e. just after the leftmost (producing) occurrence of the variable in that conjunct. In conjuncts that consume the shared variable, we insert a `wait` call as late as possible, i.e. just before the first use (leftmost occurrence) of the variable in those conjuncts.[1] For branching goals such as *if-then-else* constructs, we insert a `wait` call before, or a `signal` call after, the leftmost occurrence of the variable in each branch. Shared variables in parallel conjuncts are renamed apart to prevent the inconsistency of a single variable being bound multiple times, firstly by the producer, then by each of the `wait` calls for that variable. We allow the conjunct which produces a shared variable to keep that variable's name, so that references to that variable following the parallel conjunction do not need to be updated.

Parallel conjuncts may contain procedure calls and these call goals may contain shared variables as procedure arguments. To maximise our chance of achieving any parallelism, `signal` and `wait` calls need to happen not just before and after the procedure calls that take shared variables as arguments, but *within* those procedures. Our aim is to push waits as late as possible into the call graph of a consuming goal, and to push signals as early as possible in the call graph of a producing goal. Doing so maximises the amount of parallelism we can exploit.

Therefore the next step in the transformation is to search for procedure calls flanked on the left by any number of `waits` and on the right by any number of `signals`. If there is at least one `wait` or `signal` pertaining to shared variables that the procedure takes as arguments, then we replace the entire sequence of relevant `waits`, the call, and the relevant `signals` by a single call to a *parallel variant* of the procedure. A parallel variant is a new version of a procedure in which some arguments have been replaced by futures instead. For example, the sequence:

```
..., wait(FutureU, U),  foo(U, V),  signal(FutureV, V), ...
```

can be replaced by a call to a parallel variant of `foo/2`.

```
..., 'Parallel__foo__[1,2]'(FutureU, FutureV), ...
```

We name it `'Parallel__foo__[1,2]'` to indicate that this is the parallel variant of `foo/2` where the first and second arguments are replaced by futures. There can be other variants created as needed by other call sites.

For each `signal` or `wait` call that we removed, we must add a corresponding `lookup` call after the call to the parallel variant. This is simply because there may be further references to the shared variables in the code following the call, so we need the `lookup` calls to make sure those variables are bound. We could put in effort to avoid generating unnecessary `lookup` calls, but in practice they can be removed by a later simplification pass which is run anyway.

[1] It would be possible to reorder goals to change when the leftmost instance of a variable appears, which could improve parallelism of some programs. We haven't implemented it.

Making the parallel variants themselves is quite straightforward. The types of the arguments which are replaced by futures become `future(T)` where T is the original argument type. The modes at those argument positions are changed to "in" if they were previously "out". The body of a parallel variant procedure can be considered as though it were a conjunct of an enclosing parallel conjunction, and transformed accordingly. If one of its formal parameters is replaced by a future, then whether it is a producer or a consumer of that variable depends on the original mode of the parameter ("in" or "out"). One subtlety is that variables which have been replaced by futures already should not have new futures allocated for them. Also, when producing a parallel variant we should start from the version of the procedure which has had no dependent parallel conjunction transformation applied to it, otherwise, for example, the calls to `new_future`, `signal`, `wait` and `lookup` that have already been inserted will need to be taken into consideration.

The process of creating new parallel variants repeats until all the parallel variants which are needed have been created. In our implementation we create parallel variants in two stages. When we reach a call site that requires an unseen parallel variant, we create an initial procedure in which the body has not yet been transformed and insert that procedure onto a queue. We can immediately generate call goals to the new parallel variant procedure, although the body has not yet been transformed. Once we make a complete pass over the module's procedures, we start producing the parallel variants.

A minor technical detail is that the `signal` predicate takes two inputs and produces no output. To prevent the Mercury compiler optimising away `signal` calls, we actually need to mark them as `impure`. To prevent the impurity from propagating up the call graph, we wrap the parallel conjunction with a `promise_pure` annotation. A similar thing happens with parallel variants in which all output arguments are replaced by futures.

3.3 Example

Let us now demonstrate the dependent parallel conjunction transformation on a simple predicate, and its utility. Imagine we have a predicate that maps one list of elements to another list of elements, then performs a fold over the second list to combine the elements into a single result. Let us say that the original sequential code was written so that the fold step was intermingled with the map step, perhaps to save memory by not producing the intermediate list. We can write a parallel version of this predicate making use of dependent AND-parallelism like this:

```
:- pred mapfold(list(T), U,  U).
:- mode mapfold(in,      in, out) is det.

mapfold([],       A,  A).
mapfold([X | Xs], A0, A) :-
    (
        transform(X, Y),
        combine(Y, A0, A1)
    &
        mapfold(Xs, A1, A)
    ).
```

The first argument to `mapfold` is the list of elements to transform. The second argument is an accumulator containing the combined result so far. When the final result is calculated it is bound to the third argument.

On a non-empty list, `mapfold` calls `transform` to map the head of the list `X` to `Y`. `Y` is then passed to `combine` along with the accumulator `A0` to get the result `A1`. In parallel, `mapfold` is run on the rest of the list and given the new accumulator `A1`. Clearly this predicate makes use of dependent AND-parallelism since `A1` is shared. The `transform` calls are independent of one another so some parallelism should be achievable.

In this predicate `A1` is the only shared variable. It is produced by the left parallel conjunct and consumed by the right parallel conjunct. We allocate a future to act as its proxy, and insert a `signal` call just after the goal that binds it in the producer, and insert a `wait` call just before the first use of the variable in the consumer. We also need to rename occurrences of `A1` in the consumer to a new variable as the `wait` call will be binding that variable.

```
mapfold([],       A,  A).
mapfold([X | Xs], A0, A) :-
    new_future(FutureA1),
    (
        transform(X, Y),
        combine(Y, A0, A1),
        signal(FutureA1, A1)
    &
        wait(FutureA1, A2),
        mapfold(Xs, A2, Acc)
    ).
```

Now there are two sequences of code we want to replace by calls to parallel variant procedures. In the producer conjunct the `combine` call is directly followed by a `signal` call involving the shared variable that it binds. In the consumer conjunct the `mapfold` call is directly preceded by a `wait` call involving the shared variable that it uses. We will produce a parallel variant of `combine` in which the third argument is replaced by a future, and a parallel variant of `mapfold` in which the second argument is replaced by a future. We have omitted `lookup` calls, which are unnecessary in this predicate.

```
mapfold([],       A,  A).
mapfold([X | Xs], A0, A) :-
    new_future(FutureA1),
    (
        transform(X, Y),
        'Parallel__combine__[3]'(Y, A0, FutureA1)
    &
        'Parallel__mapfold__[2]'(Xs, FutureA1, A)
    ).
```

Now consider the code of the parallel variant `Parallel__mapfold__[2]`. We take the original code of `mapfold` and insert waits into the body for the future in the second argument, since it represents a consumed variable. In the second clause there is again the shared variable A1 so we create another future to proxy it and insert signals and waits. We also rename shared variables in parallel conjuncts apart.

```
:- pred 'Parallel__mapfold__[2]'(list(T), future(U),  U).
:- mode 'Parallel__mapfold__[2]'(in,      in,         out) is det.

'Parallel__mapfold__[2]'([], FutureA, A) :-
    wait(FutureA, A).
'Parallel__mapfold__[2]'([X | Xs], FutureA0, A) :-
    new_future(FutureA1),
    (
        transform(X, Y),
        wait(FutureA0, A0),
        combine(Y, A0, A1),
        signal(FutureA1, A1)
    &
        wait(FutureA1, A2),
        mapfold(Xs, A2, A)
    ).
```

When we replace sequences of code by calls to parallel variants, we find that we need a new variant `Parallel__combine__[2,3]` in which both the second and third arguments are futures. We also generate a call to the parallel variant `Parallel__mapfold__[2]` which, of course, already exists.

```
'Parallel__mapfold__[2]'([], FutureA, A) :-
    wait(FutureA, A).
'Parallel__mapfold__[2]'([X | Xs], FutureA0, A) :-
    new_future(FutureA1),
    (
        transform(X, Y),
        'Parallel__combine__[2,3]'(Y, FutureA0, FutureA1)
    &
        'Parallel__mapfold__[2]'(Xs, FutureA1, A)
    ).
```

`Parallel__combine__[3]` and `Parallel__combine__[2,3]` are created similarly.

The code above shows that each element of the input list can be transformed in parallel, just as we would like. The `wait` calls that would have blocked the recursive calls to `mapfold` are now delayed until the variables they wait for are actually needed in `combine`. The `combine` steps will be run in sequential order, as required by the dataflow.

3.4 Discussion

For best effect our transformation clearly requires the source code of procedures called in parallel conjuncts to be available. Not being able to push signals and waits deeper into the call graph can limit the parallelism we can exploit. Source code might not be available if the called procedure is in another module, or if the procedure is written in a foreign language.

In some cases, even if the called procedure is in another module, we can use a repeated recompilation technique to alleviate the problem of not having the source code available. For example, if module A requires a parallel variant of a procedure defined in module B then during the compilation of module A, the compiler can record a "request" that the parallel variant be produced when B is compiled. Compiling module B then recompiling module A would allow the signal and waits to be pushed deeper. This idea has not yet been implemented in the Mercury compiler, however the intermodule analysis infrastructure needed to support it, based on [7], is already present. If there are many intermodule parallel variant requests then many recompilations would be required, but we do not think it will be too much of a problem in practice. The transformation algorithm presented is simple and runs quickly.

Higher order calls also pose a problem to pushing signals and waits. Higher order specialisation can help in some cases, by turning higher order calls into first order calls, which our transformation can then handle. Intermodule inlining should also help. Higher order predicates are often short pieces of code defined in library modules that capture a common pattern (e.g. `mapfold`). With intermodule inlining, the code of short predicates can be pulled into a calling module, which would then allow higher order specialisation to convert higher order calls into first order calls.

We feel our implementation strategy is a useful way of handling dependent AND-parallelism. In particular, it has no impact on code which does not use parallel conjunctions, nor even on code which only uses independent parallel conjunctions—the "no distributed fat" principle is very important in the Mercury project. We are not aware of any other logic programming language implementation which uses a similar method to avoid the inefficiencies of supporting dependent AND-parallelism.

The idea of using futures goes back to the languages MultiLisp [22] and Mul-T [23]. In a dynamically typed language, the type of a variable is not known until run-time, and the usual way to implement this is by using type tags on each value, i.e. reserving a few bits of each machine word to record its type. When a primitive function is called, it must check that it was passed values of the right types. The type tests can be extended to handle futures as well. Thus the synchronisation needed for parallel evaluation of values can be added with almost no performance loss, above that already incurred by the run-time tests. This is the approach taken by MultiLisp, Mul-T, Mozart-Oz [24], to name a few. Prolog systems, of course, have run-time representations of unbound variables.

In lazy functional programming languages, *thunks* represent values yet to be computed. It is not hard to see the similarities between thunks and futures, and thunks can be reused for parallel evaluation. Even a language which uses strict evaluation by default, if it has support for lazy evaluation then the scheme used to handle thunks can be reused for futures. For example, Alice [27] is an extension of Standard ML with four types of futures, two of them being "concurrent futures" and "lazy futures". These are represented in the underlying virtual machine by tagged pointers [5].

On the other hand, Mercury is statically typed and has no built-in support for lazy evaluation. For efficiency and interoperability with C, values are represented without type tags and there are no run-time type tests. Thus there was no existing mechanism which we could extend to support the synchronisation of values being evaluated in parallel. We also had the goal of not making any changes that would be detrimental the performance of code which doesn't use parallelism, which is by far the majority. A source-to-source transformation was the best solution to achieve our aims.

4 Parallel execution model

This section is about the low-level details of how parallel code is executed and managed in the Mercury implementation. We start with a brief overview of the original mechanisms as presented in [10] then present changes to improve the efficiency of parallel execution and the reasons for the changes.

The Melbourne Mercury compiler can generate code for various targets using various backends. The most mature and commonly used backends are the original low-level C backend [30] and the newer high-level C backend [17]. The low-level C backend outputs instructions for a Mercury abstract machine, written using C syntax and many layers of C preprocessor macros. This is compiled by a C compiler, most often gcc, to produce object code. The low-level backend treats C as a "portable assembly language" — it does its own register allocation (for the abstract machine level), uses its own stacks, has its own calling conventions, uses gotos for transferring control, and so on.

In contrast, the high-level C backend generates more conventional C code that uses high-level language constructs such as loops and standard C function calls.

Presently, parallel conjunctions are only supported by the low-level backend. All other backends treat parallel conjunctions as sequential conjunctions. The main reason for this is that the high-level C backend was not added until after Thomas Conway started his work on parallelism in Mercury. Also, the finer control that the low-level backend allows makes some things much easier, e.g. suspending a thread of computation. We consider only the low-level backend in the rest of this section.

4.1 Overview of the execution model

A running Mercury program consists of at least one *engine*, each of which runs on a separate operating system thread. The number of engines is chosen by the user via an environment variable setting. The intention is for one engine to be started for each idle processor on the system. We assume the operating system maps engine threads to processors appropriately.

Engines are the "workers" which run Mercury code. We consider the registers of the thread that the engine runs on to be part of the engine itself. Additional state specific to each engine is stored in a `MercuryEngine` data structure. Although the most frequently used abstract machine registers can be mapped to real machine registers, additional abstract machine registers (fake registers) are mapped to fields in the `MercuryEngine` structure.

In sequential Mercury programs there is only one engine and so there is one `MercuryEngine` data structure, located at a fixed address. In parallel programs `MercuryEngine`s are dynamically allocated at startup, and each engine thread must somehow keep the address of its `MercuryEngine`, the engine base address. In the best case we can reserve a hardware register solely to hold the engine base address, which allows accesses to the contents of the `MercuryEngine` to be very fast.

A *context* is a data structure holding the state of a computation of a particular goal.[2]

[2]Contexts are called *tasks* in [10] but have always been called contexts in the implementation.

The main components of a context are the *det stack* and the *nondet stack*, which are used for executing deterministic or nondeterministic code respectively. A Mercury context also contains backing store for registers, used for suspending computations. A context must be "loaded" into an engine before the engine can begin executing a goal. This involves restoring some register values from the backing store into the engine's registers and setting the engine's stack pointers to point to the context's stack, etc. During execution the engine's registers (real or fake) are used. To suspend a computation, the engine's register values are copied back to the context's backing store, after which the engine can be idle or else load another context and do something else. Contexts can generally be migrated transparently between engines. The exceptions are due to calling Mercury code from within C code, which we will not consider here.

4.2 Original parallel execution mechanism

We now consider how a parallel conjunction is treated in Tom Conway's original design. When a parallel conjunction is reached, we save all variables which are currently instantiated and referred to in any of the parallel conjuncts into the current procedure's stack frame. These are the inputs needed for the parallel conjuncts. The current value of the stack pointer register `sp` is also saved in a stack slot that is known to all the parallel conjuncts.

Next, we allocate a *sync term* object on the heap, which keeps track of the state of a parallel conjunction. It consists of a mutex to protect concurrent accesses to the rest of the fields in the sync term[3], a count of the number of conjuncts yet to complete, and a single slot to point to a suspended context. The address of the sync term is also saved to a stack slot, which is known to all the parallel conjuncts.

At the start of each conjunct a `fork_new_context(ParentLabel, ChildLabel, NumSlots)` Mercury abstract machine instruction will be executed. This instruction creates a new context (the child context) and initialises it to begin execution at the address `ChildLabel`. The top `NumSlots` stack slots are copied from the current stack frame to the child context's stack. Thus code executing in the child context will have the same initial view of the stack frame. This is how input variable bindings are made visible in new contexts. The caller of the fork instruction (the parent) then jumps to the address given by `ParentLabel`. In a conjunct "C_1 & C_2", `ChildLabel` would point to the code for C_1 and `ParentLabel` would point to the code for C_2. We treat parallel conjunctions of more than two conjuncts like "(C_1 & (C_2 & (C_3 & C_n)))"—the left conjunct at each step is forked off in a new context, and finally the rightmost conjunct is executed directly in the current context. Since the conjuncts were always independent in Conway's implementation, it did not matter which order conjuncts were executed in.

At the end of each parallel conjunct, the following things happen. New non-local variable bindings produced in child contexts must be reflected in the original context, since the variables will be needed in later code. We make the new bindings visible in the parent context by special copy-back code added to the end of all parallel conjuncts by the compiler. Stack slots for the new variable bindings would have already been reserved in the original context's stack frame. The child contexts have access to the stack frame in the parent context because the stack pointer was saved in a stack slot before the contexts were forked, and hence later get copied to child stacks.

Figure 3 contains the source code of an example predicate using parallel conjunction,

[3] The mutex is unnecessary if atomic instructions are available and used to operate on the sync term counter.

```
:- pred p(list(int), list(int), list(int)).
:- mode p(in, in, out) is det.

p(As, Bs, Es) :-
    (
        append(As, Bs, Cs)
    &
        append(Bs, As, Ds)
    ),
    append(Cs, Ds, Es).
```

Figure 3: An example of a parallel conjunction

and figure 4 contains the code generated for that predicate. Both are from [10]. Briefly, lines 4–5 flush input variables to the stack. Line 6 saves the current stack pointer to a stack slot which will be visible in child contexts. Lines 7–9 allocate and initialise a new sync term object. Line 10 forks a new context, copying the first six stack slots of the current stack frame. Lines 12–20 contain the code of the left parallel conjunct. Line 19 copies back the binding of `Cs` to the stack frame in the parent original context, using the stack pointer saved in line 6. Lines 22–30 contain similar code for the second conjunct. Lines 32–39 set up the final tail-call to `append/3` that follows the parallel conjunction.

Scheduling in the original model is very simple. There is a global queue of contexts which are ready to execute, protected by a single mutex. `fork_new_context` creates a new context and adds it to the end of the queue. It then sends a signal (using a POSIX threads condition variable) to all the Mercury engines. Idle engines wake up and attempt to execute the first context on the queue that they can.

4.3 Revised parallel execution mechanism

The original design suffered from several problems.

Contexts were allocated and initialised for every parallel conjunct. However, contexts are large objects (due mainly to the det stack, which is on the order of megabytes in size). In order to reduce the overhead of allocating contexts, previously allocated but currently unused contexts were put on a free list. This further compounded the problem. A simple recursive procedure containing a parallel conjunction would be quite capable of bringing a machine to its knees, due to the excessive memory consumption of contexts. Thus programmers would have to be very aware of the underlying execution mechanism of parallel conjunctions in order to use them. Only very coarse-grained parallelism could be safely expressed, otherwise the program would consume all the memory on the machine and perform very badly.

Keeping allocated contexts around (and not freeing them) has another cost. All real-sized Mercury programs make use of the Boehm-Demers-Weiser conservative garbage collector library [4] to reclaim memory. During the mark phase, the garbage collector scans all allocated memory regions, looking for bit patterns that look like references to objects. Any objects which are referenced in this way are considered live; the rest are definitely garbage and can be reclaimed. Each allocated context increases the amount of memory that needs to be scanned (wasting time), and unless we clear a context's contents before

```
/* code for predicate 'p'/3 in mode 0 */
p_3_0:                              /* procedure entry point */
        incr_sp(7);                 /* allocate stack frame */
        stackvar(7) = succip;       /* save success instruction pointer */
        stackvar(1) = r1;           /* flush As, Bs to the stack */
        stackvar(2) = r2;
        stackvar(6) = sp;           /* make stack pointer visible to child contexts */
        incr_hp(r4, 8);             /* allocate sync term (size is 8 words) */
        init_sync_term(r4, 2);      /* initialise sync term (2 conjuncts) */
        stackvar(5) = r4;           /* save sync term address */
        fork_new_context(p_3_0_i2, p_3_0_i3, 6);
          /* fork context for first conjunct, jump to second conjunct */

 p_3_0_i2: /* begin first conjunct */
        r1 = &mercury_data___type_ctor_info_int_0;
        r2 = stackvar(1);                                 /* load As */
        r3 = stackvar(2);                                 /* load Bs */
        call_localret(list__append_3_1, p_3_0_i4);       /* call append/3 */
 p_3_0_i4:
        stackvar(3) = r1;                                 /* save Cs */
        field(mktag(0), stackvar(6), -3) = stackvar(3); /* copy back Cs */
        join_and_terminate(stackvar(5));                 /* stackvar(5) is sync term */

 p_3_0_i3: /* begin second conjunct */
        r1 = &mercury_data___type_ctor_info_int_0;
        r2 = stackvar(2);                                 /* load Bs */
        r3 = stackvar(1);                                 /* load As */
        call_localret(list__append_3_1, p_3_0_i7);       /* call append/3 */
 p_3_0_i7:
        stackvar(4) = r1;                                 /* save Ds */
        field(mktag(0), stackvar(6), -4) = stackvar(4); /* copy back Ds */
        join_and_continue(stackvar(5), p_3_0_i8);

 p_3_0_i8: /* after parallel conjunction */
        r1 = &mercury_data___type_ctor_info_int_0;
        r2 = stackvar(3);           /* load Cs */
        r3 = stackvar(4);           /* load Ds */
        succip = stackvar(7);       /* restore success instruction pointer */
        decr_sp_pop_msg(7);         /* deallocate stack frame */
        tailcall(list__append_3_1); /* tail-call to append/3 */
```

Figure 4: The generated code for the parallel conjunction in figure 3 in the original parallel execution model.

putting it back on the free list, it increases the chance of accidental memory retention (wasting space).

Apart from the memory consumption, creating a new context for each parallel conjunct would often be unnecessary. If the number of processors on the system is low (as in consumer-level machines) then many programs will have far more potential parallelism expressed in the source code than can be exploited. If the engines are kept busy then parallel conjuncts ought to be executed in a manner close to sequential conjuncts to reduce the overheads. Copying stack frames is also an unnecessarily inefficient way of passing variable bindings to child contexts.

The revised model is an improvement in the following ways:

- Variables bindings are no longer made visible by stack copying. Parallel conjuncts now access variable bindings in the original stack frame directly.
- We don't allocate a context for every conjunct. We introduce a new structure type called a "spark" that contains just enough information to begin execution of a single conjunct and no more. Thus sparks perform part of the functionality of contexts using far less space.
- Not all parallel conjuncts are executed in separate contexts. Once the previous parallel conjunct is finished executing, the next parallel conjunct can execute in the same context. Only parallel conjuncts that actually are executed in parallel require separate contexts.
- Since we now support dependent AND-parallelism, the order in which parallel conjuncts are executed matters. We now issue parallel conjuncts "left-to-right", as in sequential conjunction, since left conjuncts can produce values for right conjuncts but not vice versa. This reduces the need to suspend contexts which are waiting for values, and the overhead associated with suspending and resuming contexts.

As described earlier, parallel conjuncts were executed in separate contexts and input variables were available in their copies of the procedure stack frame via the stack slot register `sp`. The revised parallel execution mechanism requires a new abstract machine register, `parent_sp`, which is set up to point to the stack frame of the procedure containing the parallel conjunction. The compiler generates code for parallel conjuncts such that all references they make to stack slots are relative to `parent_sp` instead of `sp`.[4] For conjuncts executing in the same context, `sp` and `parent_sp` will be equal, so this makes no difference. But for a conjunct executing in a different context (e.g. in parallel) then in that child context `sp` would be pointing into its own det stack, but `parent_sp` will still point to the stack frame of the procedure that originated that parallel conjunct (see figure 5). Therefore no matter which context a parallel conjunct is executed in, it has access to the stack frame containing the variable bindings that it needs. A conjunct running in a child context is also able to post new variable bindings back into the parent stack frame. To support nested parallel conjunctions, the value of `parent_sp` needs to be saved before and restored after each parallel conjunction.

Sparks are small structures allocated on the heap. They are only three machine words long, containing (1) the code address of the next parallel conjunct to execute, (2) the address that the `parent_sp` register needs to point to when the spark is executed and

[4] To be clear, here we are only talking about code directly in a parallel conjunct. If a parallel conjunct makes a procedure call, the called procedure would create a new stack frame for itself on the child context's stack and refer to its own stack slots through `sp`.

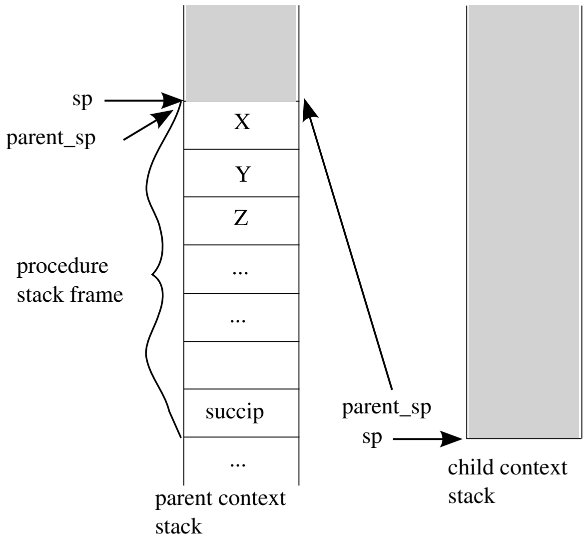

Figure 5: `parent_sp` points to the stack frame of the procedure that originated the parallel conjunct currently executing in that context.

(3) a next pointer so they can be linked in a stack or queue. We allocate and initialise a new spark for the following parallel conjunct for each parallel conjunct reached. If we determine that not enough parallel work is available (see later), the spark is placed on the end of a global *spark queue*. If there is plently of work available then the spark is instead placed on a *spark stack* that is local to the current context.

Engines thus have three sources of work. Picking up work from the local spark stack is preferred when it is available. Otherwise an idle engine can resume a ready context at the head of the global context queue, or pick up a spark from the global spark queue, load the spark into a fresh context and begin execution from there. Resuming existing contexts is preferred as they represent already started computations, whereas sparks are unstarted computations. Completing already started computations will typically reduce the overall number of contexts that need to be allocated.

On the other hand, context-local spark stacks record the parallel conjuncts which we decide should *not* be executed in parallel. Those parallel conjuncts will be executed in the same context as that which originated the spark, as only the context which owns the spark stack can pop sparks off it. We lose potential parallelism by placing a spark on a local spark stack, but don't incur the overhead of creating and switching contexts. Also, as each spark stack is only accessed by a single Mercury engine at a time, there is no need for synchronisation. Access to the global queues is slower as they are protected by a mutex.

Notice that because of the FIFO scheduling policy of contexts in the original model, parallel goals in the call graph were executed in an order closer to breadth-first order than depth-first (the last parallel conjunct would always executed in the original context so it is not exactly breadth-first). This would exacerbate the problem of memory consumption due to the number of contexts on the global queue. A depth-first traversal allows contexts to be recycled sooner. In the revised model, sparks stacks emulate the depth-first traversal order of sequential conjunction. 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.

Figure 6 contains the generated code for the predicate in figure 3 for the new parallel execution mechanism. The code is not drastically different from the code generated for the older mechanism (figure 4); the most significant changes are hidden in the definitions of the instructions. Some things to note in the generated code:

- The code of parallel conjuncts refer to "parent_stackvars" instead of "stackvars". These are stack slots referenced relative to `parent_sp` instead of `sp`.
- While a parallel conjunct is executing, the `parent_sp` register needs to point at the stack frame of the procedure which originated that parallel conjunct.
- There is no copy-back code. Outputs of parallel conjuncts still must be flushed to the stack frame of the procedure that originated the parallel conjunction but this can be done directly, instead of only at the end of a parallel conjunct. The stack slot allocator has to take into account the possibility that parallel conjuncts could be executing simultaneously while sharing a single stack frame.

The new parallelism-related Mercury abstract machine instructions are:

fork_new_child(Label) Allocate and initialise a new spark structure to begin execution of the parallel conjunct beginning at the given `Label`. The current value of `parent_sp` is stored in the spark, so that if the spark ends up executing in a different context, the value of the `parent_sp` register can be set appropriately for that context. The new spark is either put on the global spark queue or the context-local spark stack (see below). The caller falls through to the next instruction.

par_conj_done(SyncTermSlot, JoinLabel) This instruction is put at the end of every parallel conjunct. It causes the engine that executes it to do "something else", preferably working on later conjuncts of the same parallel conjunction in the same context. Figure 7 contains pseudo-code for the instruction.

`SyncTermSlot` is a pointer to the sync term of the parallel conjunction that the current parallel conjunct is part of. `JoinLabel` is the code address of the instruction that follows the parallel conjunction. Only the context which originated the parallel conjunction (the "parent" context) can jump to the `JoinLabel`, as it will be the context containing the stack frames needed by the code following the parallel conjunction.

We'll call the engine which executes the `par_conj_done` instruction the "calling engine", and the context it is currently loaded with the "calling context".

`par_conj_done` first decrements the sync term's count of the number of conjuncts still outstanding. If the count drops to zero, then all the parallel conjuncts have completed. Now, if the calling context is the originator of the parallel conjunction, the parallel conjunction is done and we jump immediately to `JoinLabel`. On the other hand, if the calling context actually picked up the execution of a parallel conjunct which was originated by some other context, it schedules the originating context (which would be suspended at this point) to resume execution at the `JoinLabel`. The calling engine can then look for other work, perhaps reusing the current context.

Let's consider now what happens if not all of the conjuncts of the parallel conjunction have been completed. If the spark at the top of the calling context's spark stack came from the same parallel conjunction as the conjunct that has just completed, we can immediately begin execution of that spark in the current context. This mimics the execution of sequential conjuncts. We can tell if the spark came from the same parallel conjunction

```
p_3_0:                                /* procedure entry point */
        incr_sp(7);                   /* allocate stack frame */
        stackvar(7) = succip;         /* save success instruction pointer */
        /* end of procedure prologue */
        stackvar(5) = parent_sp;      /* save old parent_sp and */
        parent_sp   = sp;             /* set it for the upcoming parallel conjunction */
        stackvar(1) = r1;             /* flush As, Bs to the stack */
        stackvar(2) = r2;
        {
        Word tempr1;
        alloc_heap(tempr1, 8);        /* allocate sync term (size is 8 words) */
        init_sync_term(tempr1, 2);    /* initialise sync term (2 conjuncts to execute) */
        stackvar(6) = tempr1;         /* save sync term address */
        }
        fork_new_child(p_3_0_i4);     /* create spark for second conjunct */

        /* begin first conjunct */
        r1 = INT_CTOR_ADDR;
        r2 = parent_stackvar(1);                        /* load As */
        r3 = parent_stackvar(2);                        /* load Bs */
        call_localret(append_3_1, p_3_0_i3);            /* call append/3 */
p_3_0_i3:
        parent_stackvar(3) = r1;                        /* save Cs */
        par_conj_done(parent_stackvar(6), p_3_0_i2);
        /* parent_stackvar(6) is sync term */

p_3_0_i4: /* begin second conjunct */
        r1 = INT_CTOR_ADDR;
        r2 = parent_stackvar(2);                        /* load Bs */
        r3 = parent_stackvar(1);                        /* load As */
        call_localret(append_3_1, p_3_0_i5);            /* call append/3 */
p_3_0_i5:
        parent_stackvar(4) = r1;                        /* save Ds */
        par_conj_done(parent_stackvar(6), p_3_0_i2);

p_3_0_i2: /* after parallel conjunction */
        parent_sp = stackvar(5);      /* restore old parent_sp */
        r1 = INT_CTOR_ADDR;
        r2 = stackvar(3);             /* load Cs */
        r3 = stackvar(4);             /* load Ds */
        succip = stackvar(7);         /* restore success instruction pointer */
        decr_sp(7);                   /* deallocate stack frame */
        tailcall(append_3_1);         /* tail-call to append/3 */
```

Figure 6: The new generated code for the parallel conjunction in figure 3. Some prefixes and macros were stripped away to improve readability.

`par_conj_done(SyncTerm, JoinLabel)`:

1. Decrement the sync term's count of the number of conjuncts of the parallel conjunction not yet completed.
2. If the count drops to zero, then the parallel conjunction is finished.
   (a) If the calling context is the context that originated the parallel conjunction, jump to the JoinLabel and we are done.
   (b) Otherwise the originating context must be suspended. Schedule the originating context to resume. Making the calling engine look for other work, discarding or reusing the current context.
3. If the count is non-zero, there must be parallel conjuncts still outstanding.
   (a) If we have a spark at the top of the spark stack that came from the same parallel conjunction as the conjunct that has just finished, begin execution of that spark in the current context.
   (b) Otherwise, if the current context originated a parallel conjunction then it must wait until all the conjuncts have completed, so suspend execution of the current context and wait for another context to schedule it. Make the calling engine find another source of work.

Figure 7: Pseudo-code for the par_conj_done instruction.

by comparing the value of `parent_sp` saved in the spark with the current value of the `parent_sp` register.

Otherwise, we can do nothing more with the calling context right now. If the calling context originated a parallel conjunction, we cannot discard it yet as it will have more to do once the parallel conjunction is finished. In that case we suspend it and wait for another context to schedule it when the last parallel conjunct is completed. Finally, the calling engine can look for other work to do, from the global context or spark queues.

We currently use a simple heuristic to decide whether new sparks should be placed on the global spark queue or on a context-local spark stack. We keep a count of the number of contexts and sparks already on global queues. At the time that a new spark is created, if the number of sparks and contexts on global queues is below a certain threshold, then we allow the new spark to be put on the global queue, otherwise the spark is put on a local spark stack. The threshold can be chosen by the user via an environment variable.

The heuristic was designed to limit the number of contexts allocated in total, to keep memory usage reasonable. However, as we found later, a simple threshold artificially limits the amount of parallelism that can be achieved in some types of programs (e.g. those not using a divide-and-conquer strategy) so the heuristic will need to be revised in the future.

4.4 Smaller contexts

Although in the revised model we try to reduce the number of allocated contexts in order to reduce memory consumption (and subsequent scanning by the garbage collector), some programs employing parallel conjunctions can still require many contexts to be present in order to exploit parallelism. For example, a whole queue of contexts might be waiting for a value to be produced by the previous context, but meanwhile there might be more work available. If the number of contexts is artifically restricted then idle engines will have to remain idle instead of picking up new work. This problem was realised only late in this project, when trying to use fine-grained parallelism on more realistic programs, so only a partial solution could be implemented.

We do not know ahead of time how much stack space will be required by the whole program so stacks are created big to "play it safe". But in many cases, parallel conjuncts contain only relatively shallow computations. Instead of restricting the number of contexts, we can reduce memory consumption by reducing the size of each context. We divide contexts into two sizes, regular and small, where the names refer to the size of the stacks they are given. A regular-sized context is used for the initial context that is used for the entire lifetime of the program, whereas small-sized contexts are used for executing parallel goals. The sizes can be chosen via an environment variable.

Clearly this is only a partial solution. If the chosen stack sizes are too small it will lead to a crash at runtime. There is no guarantee that any particular stack sizes are large enough for a sequential Mercury program either, but then there is only one context so we can afford to make stacks very large. In future, probably the best solution would be to create *all* contexts with small stacks and grow them as necessary. Perhaps the best examples of this are the implementations of Erlang, which can handle thousands of "processes" partly because each process starts with extremely small stacks which grow as necessary [1, 25].

Stack growing presents some difficulties for our revised parallel execution model, however. Let us assume that before a stack pointer is incremented, we check whether the stack pointer would go past the end of the region allocated to the stack. If so, the region is resized and, if necessary, moved to a different base address. Unfortunately, if the stack is moves then, in our model, the `parent_sp` pointers in *other* contexts would need to be updated. Worse, old `parent_sp` pointers saved in stack slots would need to be updated as well. Apart from being a headache to implement, preparing for these possibilities represents an additional overhead on parallel execution compared to sequential execution.

We propose an alternative to stack resizing using disjoint stack segments that wouldn't require updating any pointers in other contexts. This is inspired by the "stacklets" approach in [13]. Once the current stack segment is exhausted, instead of growing the segment and potentially moving it, we can simply allocate another region of memory as the logical continuation of the stack. Thus when incrementing the current context's stack pointer, it may jump to an unrelated part the program's address space, but no other pointers need to be updated.

When the stack pointer is later decremented it must handle the possibility of falling off the end of the current segment and so point back to the previous segment instead. Again, [13] provides a way to avoid the overhead of an explicit stack pointer check by using "stacklet stubs". When the stack pointer is incremented at the start of the procedure prologue, if it causes a new stack segment to be allocated then we can insert a "stub" stack frame at the bottom of the new stack segment. When the current procedure returns, instead of returning directly to its caller, it would jump into a stub handler. The stub handler would reset the stack pointer to point into the previous stack segment before returning to the actual caller. Thus no explicit check is needed for the stack pointer decrement before a procedure return.

Going further, if we guarantee that a certain number of words at the top of a stack will always be available, then leaf procedures (procedures which don't call other procedures) would not need explicit overflow checks when incrementing the stack pointer either. This optimisation was used in the HiPE Erlang compiler [25], for example.

4.5 Work stealing

Choosing whether to put a spark on the global spark queue or on the context-local spark stack at the time the `fork_new_child` instruction is executed means that we are making a guess ahead of time as to whether that particular conjunct should be executed in parallel or not. This is a form of *load-based inlining*. It is impossible to know whether there will be an idle engine available to pick up a piece of work in the near future. An "incorrect" decision can mean a loss of parallelism or unnecessary overhead.

An alternative strategy employed by some systems is work stealing [3, 22, 23]. The idea is that the decision whether to execute a piece of code in parallel is not made until a worker thread is idle. By default work is placed in a local pool, usually a stack. An idle worker thread becomes a *thief* which tries to steal work from a *victim* thread's work stack. The choice of whether to steal work from the cold or hot end of stacks varies. In general the assumption is that work should be stolen from as near the top of the program's call graph as possible, which corresponds to the cold end of a work stack. Since work may be removed from either end, the work stacks become double-ended queues, or deques.

A work-stealing deque is a specialised kind of deque in which one thread, the owner, treats it as a stack, pushing and popping work on one end, and all other threads only steal work from the other end. These restrictions allow a work-stealing deque to be implemented more efficiently than generic deques. In particular, the push and pop operations performed by the owner thread need to be as efficient as possible, as they will be done very often. Efficiency of the actual work stealing operation is less important as it is not as common; most worker threads should be busy with their own work.

We can readily extend our parallel execution mechanism to work stealing by making context-local spark stacks into work-stealing deques. An idle engine would turn into a thief and try to steal a spark from a victim context's spark deque. If successful, execution of the spark would begin in a new allocated context. This is exactly the same as if the spark was put on the global spark queue.

There are many algorithms for work-stealing deques, e.g. [2, 18, 19, 26]. Currently we have an incomplete implementation of the work-stealing deque algorithm in [8]. This algorithm is simple and has the advantage that the deques do not have a fixed limit with the potential for overflowing. A problem with work-stealing is finding work to steal efficiently. Usually this is done by assuming that a program will have far more parallelism available than worker threads, so an idle worker can simply choose a victim by a (uniformly) random process. This is currently where our implementation is at: we need a means of finding contexts with non-empty spark stacks quickly, but without maintaining complicated data structures to facilitate that. It would also be interesting in the future to see if there is any advantage in mixing a work stealing strategy with load-based inlining.

5 Other aspects

5.1 Garbage collector tweaks

The Mercury implementations uses the popular Boehm-Demers-Weiser conservative garbage collector library. Previously, ineffective use of the garbage collector significantly impacted on efficiency, causing CPUs to be idle for much more time than expected in parallel programs.

- We were not enabling *thread-local allocation* support in the garbage collector. In the default configuration, every memory allocation required acquiring and releasing a global lock. Most Mercury programs allocate memory extensively, so in a parallel program there was a lot of contention for that global lock and performance suffered severely. The thread-local allocation feature causes each thread to satisfy small memory block requests from its own thread-local free lists whenever possible. The global lock only needs to be acquired to ask the operating system for more memory when there are no more memory blocks of the appropriate size on the thread-local free lists. This reduces contention on the global lock considerably and allows much more efficient use of multiple processors.
- The Boehm garbage collector uses a mark and sweep algorithm. The marking phase, that is tracing known reachable objects to find more reachable objects, takes up the majority of the garbage collection time. All program threads are suspended while marking is in progress (this is called "stopping the world").[5] This is relatively more costly, in terms of CPU time wasted, for a parallel program than for a sequential program.

  Fortunately, on a few platforms, the Boehm garbage collector supports a feature to run the mark phase in parallel. It still needs to stop all the Mercury engines when the program runs out of memory, but the additional CPUs can help to perform the marking more quickly, and so reduce the absolute time spent with the world stopped.

5.2 Thread-local engine base address

As discussed in section 4, a parallel Mercury program consists of multiple engines running on separate POSIX threads, each with its own `MercuryEngine` data structure. Each thread therefore needs a way to refer to its own `MercuryEngine`, the engine base address. The engine base address is accessed extremely often. For example, all references to fake Mercury abstract machine registers (i.e. those that can't be mapped to real hardware registers) require a lookup of the engine base address as the fake registers are actually fields in the `MercuryEngine` structure.

In the best case we make use of the gcc "global registers" extension to C, which allows us to reserve some machine registers so that gcc won't generate code which uses those registers. Then we can reserve one register to hold the engine base address for that thread and lookups of the engine base address are very fast.

When the gcc extension is not available or is disabled by the user, the engine base address must be stored in some form of thread-local storage. The POSIX threads API

[5] The Boehm garbage collector has support for incremental garbage collection, in which the world needs to be stopped only for a short while. However, in past experiments on sequential programs, it was found that the overheads of using incremental garbage collection were greater than the benefits.

provides library functions for this purpose (`pthread_getspecific`, etc.) and some C compilers have facilities that allow a global variable to take different values for different threads (e.g. gcc has a `__thread` keyword). Since the engine base address is accessed so often, using either of these methods is much slower than keeping the address in a register.

As the engine base address doesn't change at any time after an engine is created, our solution to this problem is to use one of the thread-local storage methods above, but make a copy of the engine base address into a C local variable at the start of each basic block. The rest of the basic block then refers to the C local variable whenever it needs the engine base address, so the C compiler can generate more efficient code if the engine base address is accessed more than once in the basic block.

On an x86-64 machine, the run time of a Fibonacci benchmark went from 3.1 seconds down to 2.3 seconds (25% faster) when this optimisation was enabled. Reserving a hardware register for the engine base address took the run time down to 0.98 seconds (57% faster again).

5.3 Interaction with impurity

Mercury has an impurity system in which three levels of purity are identified. Pure predicates are the default. Impure predicates can perform side-effects or modify state which can affect the semantics of other code. The declarative semantics of semipure predicates may be affected by previous calls to impure predicates, but semipure predicates themselves cannot affect other code. Calls to semipure and impure predicates must be explicitly marked as "`semipure`" and "`impure`" respectively; this makes them easy to spot. Impurity levels propagate up the call graph until promised away by the programmer. Generally impurity is only used at very low levels to implement pure interfaces [12].

One property that we desire of parallel conjunctions is that they have exactly the same declarative semantics as normal conjunctions. A deterministic parallel conjunction should produce exactly the solution as a deterministic sequential conjunction. However, when impurities are introduced into a parallel conjunction, this is no longer true. Consider the following:

```
impure set_global(1),
(
   impure set_global(2)
&
   semipure get_global(X)
)
```

Assume `set_global` is a predicate that destructively modifies the value of a hidden global variable, and `get_global` is a predicate that retrieves the current value of that variable. If the parallel conjunction were replaced by a sequential conjunction, then we would read the code from left-to-right and `X` would be bound to 2. But, as a parallel conjunction, `X` might be bound to 1—it depends on whether the left conjunct changed the global variable before the right conjunct read it. This introduces concurrency issues which did not exist in the parallel conjunctions which we considered up till now.

One way to avoid these concurrency issues is to automatically introduce synchronisation that will delay any semipure predicate call in a parallel conjunct until the last impure predicate call in the previous parallel conjunct is complete, and to delay any impure predicate call until the last semipure predicate call in the previous parallel conjunct is complete.

However, that is very heavyweight for a feature that is unlikely to be used. Instead, we simply disallow impure parallel conjuncts (the compiler will report an error), although semipure parallel conjuncts are allowed.

The programmer can still call impure predicates in parallel conjunctions and promise away the impurity if he is *extremely* careful. The following code may be reasonable in a sequential program:

```
foo(Params, Result) :-
    promise_pure (
        impure   reset_internal_state(Params),
        impure   compute_result,
        semipure get_result(Result)
    ).
```

However, assuming there really is only one internal state, the `promise_pure` annotation is a lie in a parallel program and the compiler will get its revenge. The following goal would have a race condition:

```
foo(Params1, Result1) & foo(Params2, Result2)
```

The solution would be to make each call to `foo` work with a different copy of the internal state.

5.4 Exception handling

Mercury features an exception system to deal with unexpected runtime conditions. However, the exception handling implementation has not been updated to support parallel execution yet. Throwing an exception from inside a parallel conjunct to an exception handler outside of the parallel conjunction does not work. (The technical reason is that the stack unwinding code does not know to continue unwinding on the parent context's stack if it doesn't find an exception handler on a child context's stack. Once found, the exception handler must be executed in the correct context.)

Exceptions introduce complications into the dependent parallel conjunction transformation from section 3. If a producing conjunct throws an exception before a `signal` call is made then consuming conjuncts would suspend forever. However, the problem is not insurmountable. We can use information from the Mercury compiler's exception analysis pass, which detects which goals may or won't throw exceptions. If a producer is guaranteed not to throw an exception before the `signal` call then our transformation works as presented. Otherwise, for example, we can set an exception handler around the parts of the code that may throw exceptions, which would send an abort message to any consumers suspended on the future at that time. Killing all sibling conjuncts when a parallel conjunct throws an exception past the parallel conjunction would also work.

6 Experimental evaluation

In this section we detail some experiments performed with the Mercury system, first parallelising some toy benchmarks, then a medium sized program, then part of the Mercury compiler itself. The two machines we performed the tests on are:

- saturn — a PC with a 64-bit Pentium D 3.0 GHz, dual core CPU with 4 GB RAM, running Linux 2.6.12.5, glibc 2.3.5 and gcc 3.3.5. Each core has a 12 kB trace cache, 16 kB D-cache and a 1 MB L2 cache.
- mundara — a 32-bit, Sun Fire T2000 machine with 16 GB RAM, running Sun Solaris 2.10 and gcc 3.4.3. It has an UltraSparc T1 CPU consisting of 8 cores, each 4-way multithreaded with a 16 kB I-cache and a 8 kB D-cache, backed by a 3 MB unified secondary cache. It has a single floating point unit.

The main development machine for the work in this report was saturn. Very little development work was done on mundara, i.e. we didn't try to optimise for it. We tried to run the tests when the machines were lightly loaded, but this was not always possible.

In the results tables that follow, "asm_fast.gc" indicates versions of the program which were compiled using the low-level C backend, without any support for parallelism. "asm_fast.par.gc" adds parallelism support, and enables thread-local allocation and parallel marking in the garbage collector. The number in brackets indicates the number of engines started for that trial. "hlc.gc" indicates the programs were compiled using the high level C backend. All programs were compiled with `mmc -O5 --intermodule-optimisation` (basically the best settings when optimising for time). All times given are average wall-clock times in seconds. Each program was run six times consecutively, with the best and worst run times discarded and an average taken of the remaining times. Values in the "speedup" columns were calculated as the speedup relative to the run time for asm_fast.gc: $T_{\text{asm\_fast.gc}} \big/ T_{\text{other}}$.

We limited the parallel marking feature to use the same number of parallel marking threads as the number of Mercury engines we had running. For example, if we were checking how well a program performs with 4 engines, we would only allow parallel marking to use 4 threads.

We haven't compared directly our parallel execution mechanism changes with the original parallel execution mechanism because (1) most of the benchmark programs would not run well with the old system without a lot of "tuning", which would not make for a meaningful comparison, and (2) many changes unrelated to parallelism have been made to the Mercury implementation while this project was in progress, so it would not be fair just to compare with an old version of the Mercury system. Backporting our changes to an old version of the compiler would also be infeasible.

We can say, however, that the benchmark programs which would run well with the old parallel execution mechanism would *not* see much benefit with the new parallel execution mechanism. That is because they could only be exploiting coarse-grained parallelism, when the inefficient aspects of the old system have little impact. Only when programs attempt to exploit parallelism at a finer level do the advantages of the new system show up, with the biggest difference being that programs run to completion in reasonable time instead of swapping heavily.

6.1 Toy benchmarks

Four of the programs here are from the Ciao Prolog web site; along with some others they were used for the paper [6]. We picked these particular programs as they were known to be suitable for AND-parallelism and required few changes to translate into Mercury. Benchmark parameters were updated for current hardware.

fib calculates Fibonacci numbers by the doubly recursive method. tak is a triply recursive function. hanoiapp solves the Towers of Hanoi puzzle by a recursive method and returns the moves of the solution appended in a big list. mmatrix performs multiplication of two square, integer matrices.

The qsortapp benchmark was written by myself. It generates a list of 100,000 random integers and sorts it using the naïve implementation of quicksort, i.e. using `append` to join the left and right sorted lists together. The time spent generating the random list is not counted.

fib and tak do almost no work apart from making procedure calls. They give us some insight into the overhead of parallel execution. The fib case marked "ngc" has no granularity control built in; it tries to perform parallel computations all the way down to `fib(2)`. As we can see, its performance is not good. To reduce the parallel overhead to reasonable limits, we must manually add in granularity control. Once we know that the amount of work underneath the current call is too small, we fall back to a sequential version of the procedure to avoid the parallel overheads. This does require a bit of guess work and is not entirely satisfactory. Research is being done to address this problem.

The rest of the benchmarks have granularity control built in when necessary. When we do that, we can get rather good speedups for fib on both machines. We get good relative speedup for tak as well, but on saturn tak takes a big performance penalty when parallelism support is enabled so the absolute speedup is much lower.

fib and tak were simple enough to be transliterated into equivalent sequential C programs, which were compiled with `gcc -O2`. We provide their run times as a reference point to gauge the speed of the machines, and also to show that sequential Mercury execution is very efficient already.

We get very satisfying speedups for mmatrix on mundara. We don't think that the overhead of supporting parallel execution is to blame for the more modest speedup on saturn, since the performance drop when parallelism support was enabled (i.e. from asm_fast.gc to asm_fast.par.gc) was not that great. Rather, it may be due to the more limited memory bandwidth of a PC.

The run times for hanoiapp are puzzling. It seems, however, to be due to the large amounts of memory allocated and the large amounts of garbage generated: the program builds up an exponentially large list of the moves required to solve the Towers of Hanoi puzzle, and uses `append` to concatenate intermediate move lists.

To check this, I modified the program to change the representation of the move list. Instead of working with lists of moves, hanoitree works with binary trees. Joining two subtrees of moves becomes a constant time operation instead of linear in the number of elements in the first list, and doesn't generate garbage. The move lists remain exponentially large, of course. As shown in the results, the speedups for hanoitree are far better.

qsortapp gets modest speedup up for a few processors, but doesn't improve thereafter. It could be due to problems mentioned before: memory bandwidth limitations, large amounts of memory being allocated, large amounts of garbage needing to be reclaimed. It could also be due to the nature of the implementation. The sorting predicate itself looks like this (omitting granularity control):

```
qsort([], []).
qsort([Pivot | Rest], Sorted) :-
    partition(Pivot, Rest, Less, More),
    ( qsort(Less, Left)
    & qsort(More, Right)
    ),
    append(Left, [Pivot | Right], Sorted).
```

When `partition` returns two unbalanced lists (one of `Less` or `More` being substantially longer than the other) then one of the recursive `qsort` calls will take longer to complete than the other. This could require the current context to suspend and the engine to switch to an unrelated task or sit idle.

In general, the benchmarks with regular, predictable loads and which didn't allocate huge amounts of memory were able to take advantage of the parallel hardware quite well.

6.2 Ray tracer

The ray tracer was the Mercury team's entry for the ICFP 2000 programming contest, in which it placed fourth. It renders scene descriptions, which are written in a simple functional language. Here we used the scene decription snowgoon.gml, which was used to judge contest entries.

The parallelisation of the rendering loop followed exactly the pattern of the `mapfold` predicate in section 3.3, so made use of dependent AND-parallelism. Every row index of the output image was mapped to a list of calculated pixel values for that row, so multiple rows could be rendered in parallel. The fold step then wrote each row of pixels out to a file. The "results" we accumulated as part of the fold step were actually successive I/O states as the rows were written out.

We achieve a decent speedup on saturn. The parallel version of the program takes a performance hit compared to the sequential version when using only one core, however we still get an absolute speedup of 1.50 when two cores are used.

On mundara, the results do not look as encouraging, until you consider that it only has a single floating-point unit. We believe that this is the main reason that the speedup does not scale as well as we would like on this machine.

The results in both cases show that if you had a multi-core machine available and you were going to render many scenes, enabling parallel rendering would definitely be worthwhile.

6.3 Parallelising the Mercury code generator

The most practical way to exploit parallelism in the Mercury compiler would be to compile separate modules in parallel. We would be assured of having enough work to make parallel execution worthwhile. However, our focus here is on trying to exploit parallelism to speed up the compilation of a *single* module. We picked out the code generator pass of the Mercury compiler for parallelisation as it takes a fairly significant amount of time.

The process of generating code for two procedures is almost independent of one another, with one exception: to collect and share static data which is common to two or more procedures, a single `global_data` structure is threaded through the code generation pass,

	saturn		mundara	
Grade (engines)	Time (s)	Speedup	Time (s)	Speedup
asm_fast.gc (1)	1.01	1.00	4.74	1.00
hlc.gc (1)	0.74	1.36	4.98	0.95
asm_fast.par.gc (1)	0.97	0.96	4.76	1.00
asm_fast.par.gc (2), ngc	15.18	0.067	79.56	0.06
asm_fast.par.gc (2)	0.52	1.94	2.51	1.88
asm_fast.par.gc (3)			1.77	2.67
asm_fast.par.gc (4)			1.40	3.38
asm_fast.par.gc (5)			1.20	3.96
asm_fast.par.gc (6)			1.05	4.50
asm_fast.par.gc (7)			0.96	4.96
asm_fast.par.gc (8)			0.89	5.35
C program (1)	0.78	1.30	4.86	0.98

Table 1: Benchmark results for fib(39). ngc = no granularity control.

so code generation of the next procedure could not begin until the previous procedure was completed.

To get around this, we introduced code to combine two `global_data` structures together, and code to go back and fix up the procedures which used the second of the `global_data`s to use the merged `global_data` instead. (Procedures using information from the first `global_data` wouldn't need to be fixed up due to the way `global_data`s are merged.) Fortunately, these extra steps take up very little time and, when compiling large modules, are compensated for by the speedup from parallelisation.

Table 8 contains the run times for compilation of three of the largest modules in the Mercury compiler. Results are given for a compiler built with no parallel support (asm_fast.gc), for a compiler with parallelism support built in but not enabled (asm_fast.par.gc), and for a compiler with parallelism support and the parallel code generation option enabled.

If we look just at the run times of the code generator pass, we see that we do get a decent speedup in the code generator pass by using two processors. Unfortunately, the overall runtimes for a parallelism-enabled compiler are worse than for a compiler with no parallelism support.

Why does this happen? It turns out that much of the performance loss in the parallelism-enabled compiler can be attributed to garbage collector library. A version of the asm_fast.gc compiler linked with a copy of Boehm GC with thread support enabled runs ∼23% slower on saturn, when compiling the six largest modules in the compiler. Hopefully future improvements to the garbage collector or a different garbage collection implementation will allow us to close the gap between the parallel and non-parallel grades, so that small scale parallelism, such as we attempted here, will become worthwhile.

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc (1)	3.03	1.00	23.15	1.00
hlc.gc (1)	3.98	0.76	21.70	1.07
asm_fast.par.gc (1)	4.93	0.62	23.16	1.00
asm_fast.par.gc (2)	2.78	1.09	12.11	1.91
asm_fast.par.gc (3)			8.18	2.83
asm_fast.par.gc (4)			6.22	3.72
asm_fast.par.gc (5)			5.05	4.58
asm_fast.par.gc (6)			4.27	5.42
asm_fast.par.gc (7)			3.73	6.21
asm_fast.par.gc (8)			3.33	6.95
C program (1)	3.91	0.78	18.39	1.26

Table 2: Benchmark results for tak(24, 20, 6).

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc	1.53	1.00	16.74	1.00
hlc.gc	2.04	0.75	65.80	0.25
asm_fast.par.gc (1)	1.61	0.95	18.37	0.91
asm_fast.par.gc (2)	1.10	1.39	9.44	1.77
asm_fast.par.gc (3)			6.42	2.61
asm_fast.par.gc (4)			4.93	3.40
asm_fast.par.gc (5)			4.03	4.15
asm_fast.par.gc (6)			3.48	4.80
asm_fast.par.gc (7)			3.08	5.44
asm_fast.par.gc (8)			2.74	6.11

Table 3: Benchmark results for mmatrix(500).

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc	3.41	1.00	18.73	1.00
asm_fast.par.gc (1)	4.09	0.84	28.84	0.65
asm_fast.par.gc (2)	2.30	1.48	13.07	1.43
asm_fast.par.gc (3)			11.91	1.57
asm_fast.par.gc (4)			9.26	2.02
asm_fast.par.gc (5)			9.95	1.88
asm_fast.par.gc (6)			11.26	1.66
asm_fast.par.gc (7)			10.54	1.78
asm_fast.par.gc (8)			9.68	1.94

Table 4: Benchmark results for hanoiapp(21).

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc	3.54	1.00	22.56	1.00
asm_fast.par.gc (1)	3.78	0.94	23.88	0.94
asm_fast.par.gc (2)	2.37	1.49	14.03	1.61
asm_fast.par.gc (3)			10.37	2.18
asm_fast.par.gc (4)			7.19	3.14
asm_fast.par.gc (5)			6.66	3.39
asm_fast.par.gc (6)			5.48	4.12
asm_fast.par.gc (7)			5.26	4.29
asm_fast.par.gc (8)			5.13	4.40

Table 5: Benchmark results for hanoitree(24).

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc (1)	0.169	1.00	1.11	1.00
asm_fast.par.gc (1)	0.210	0.80	1.19	0.93
asm_fast.par.gc (2)	0.141	1.20	0.75	1.48
asm_fast.par.gc (3)			0.58	1.92
asm_fast.par.gc (4)			0.51	2.18
asm_fast.par.gc (5)			0.46	2.42
asm_fast.par.gc (6)			0.43	2.61
asm_fast.par.gc (7)			0.41	2.69
asm_fast.par.gc (8)			0.40	2.77

Table 6: Benchmark results for qsortapp(100000).

Grade (engines)	saturn Time (s)	saturn Speedup	mundara Time (s)	mundara Speedup
asm_fast.gc	5.58	1.00	188.76	1.00
hlc.gc	7.25	0.77	187.86	1.01
asm_fast.par.gc (1)	6.24	0.89	224.84	0.83
asm_fast.par.gc (2)	3.72	1.50	126.91	1.48
asm_fast.par.gc (3)			92.56	2.03
asm_fast.par.gc (4)			76.26	2.47
asm_fast.par.gc (5)			66.28	2.84
asm_fast.par.gc (6)			59.73	3.15
asm_fast.par.gc (7)			56.12	3.36
asm_fast.par.gc (8)			53.62	3.51

Table 7: Benchmark results for the ray tracer.

File	Configuration	Code gen pass (s)	Speedup	Overall (s)	Speedup
1	asm_fast.gc (1)	0.415	1.00	4.65	1.00
	asm_fast.par.gc (1)	0.478	0.87	5.95	0.78
	w/parallel code gen (2)	0.311	1.33	5.33	0.87
2	asm_fast.gc (1)	0.889	1.00	8.71	1.00
	asm_fast.par.gc (1)	0.803	1.10	10.54	0.83
	w/parallel code gen (2)	0.567	1.57	9.17	0.95
3	asm_fast.gc (1)	0.394	1.00	3.76	1.00
	asm_fast.par.gc (1)	0.464	0.85	4.79	0.78
	w/parallel code gen (2)	0.209	1.89	4.32	0.87

Table 8: Compiling different modules with the parallel code generation option on saturn. The files were `llds_out.m`, `modules.m` and `mercury_compile.m`.

7 Conclusion and future work

In this thesis we have shown a simple but effective dependent parallel conjunction transformation that allows parallel execution of more Mercury programs than was previously possible. Our transformation has no negative impact on code which does not make use of parallelism, or even code that only uses independent AND-parallelism. We also described some changes in the way code is executed in parallel. Before our changes the Mercury implementation was really only suitable for running small benchmarks using coarse-grained parallelism. We showed that we are now able to get good speedup on both toy benchmarks and a real program, a ray tracer.

Still, there is plenty more work to be done on parallelism in Mercury. We have presented ideas for extending the dependent parallel conjunction transformation, for dynamic stack growth, using work stealing for scheduling and dealing with exceptions. As always, the efficiency of parallel execution could be improved further. In particular, memory management incurs quite a significant overhead even when parallelism is not used, but only enabled. This restricts our ability to incrementally exploit parallelism in otherwise sequential programs, as we showed with the Mercury compiler. Finally, granularity control is key to reducing parallel overheads. There is work being pursued now on using profiling information to help the programmer out in that regard.

8 Acknowledgements

Thanks to my supervisor, Zoltan Somogyi, and the guys in the Mercury office for their help and advice both this year and previously. In particular, thank you to Julien Fischer for putting up with my patches. And, of course, thank you to Thomas Conway for doing all the heavy lifting beforehand.

References

[1] Joe Armstrong, Bjarne Däcker, Robert Virding, and Mike Williams. Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services*, 1992.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, New York, NY, USA, 1998. ACM Press.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[5] Thorsten Brunklaus and Leif Kornstaedt. A virtual machine for multi-language execution. Submitted, November 2002.

[6] Francisco Bueno, María García de la Banda, and Manuel Hermenegildo. Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–239, 1999.

[7] Francisco Bueno, María J. García de la Banda, Manuel V. Hermenegildo, Kim Marriott, Germán Puebla, and Peter J. Stuckey. A model for inter-module analysis and optimizing compilation. In *LOPSTR '00: Selected Papers from the 10th International Workshop on Logic Based Program Synthesis and Transformation*, pages 86–102, London, UK, 2001. Springer-Verlag.

[8] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, New York, NY, USA, 2005. ACM Press.

[9] Keith Clark and Steve Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[10] Thomas Conway. *Towards Parallel Mercury*. PhD thesis, The University of Melbourne, July 2002.

[11] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[12] Tyson Dowd, Peter Schachte, Fergus Henderson, and Zoltan Somogyi. Using impurity to create declarative interfaces in Mercury. Technical report, The University of Melbourne, 2000.

[13] Seth Copen Goldstein, Klaus E. Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[14] Gopal Gupta, Enrico Pontelli, 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(4):472–602, 2001.

[15] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM Press.

[16] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual, 1996. A more recent version can be found at http://www.cs.mu.oz.au/research/mercury.

[17] Fergus Henderson and Zoltan Somogyi. Compiling Mercury to high-level C code. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 197–212, London, UK, 2002. Springer-Verlag.

[18] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006.

[19] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pages 280–289, New York, NY, USA, 2002. ACM Press.

[20] Manuel V. Hermenegildo and K. J. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. In *7th International Conference on Logic Programming*, pages 253–268, 1990.

[21] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *2000 Marktoberdorf Summer School*, 2000.

[22] Robert H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[23] David A. Kranz, Robert H. Halstead Jr., and Eric Mohr. Mul-T: a high-performance parallel Lisp. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 81–90, New York, NY, USA, 1989. ACM Press.

[24] Michael Mehl. *The Oz Virtual Machine — Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999.

[25] Mikael Pettersson, Konstantinos F. Sagonas, and Erik Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In *6th International Symposium on Functional and Logic Programming*, pages 228–244, 2002.

[26] K. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1998.

[27] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, 2006.

[28] Wolfgang Schreiner. Parallel Functional Programming — An Annotated Bibliography. Technical Report 93-24, Johannes Kepler University, Linz, Austria, 1993.

[29] Kish Shen. Exploiting dependent and-parallelism in Prolog: The dynamic dependent and-parallel scheme (DDAS). In *Joint International Conference and Symposium on Logic Programming*, pages 717–731, 1992.

[30] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[31] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.