Towards automatic parallelization of Mercury programs

Zoltan Somogyi
joint work with Tom Conway, Peter Wang and Paul Bone

Department of Computer Science and Software Engineering
The University of Melbourne

23 November, 2009

Mercury

Mercury is a purely declarative logic programming language. Syntactically, it is based on Prolog, but semantically, it is very different.

- Mercury has a strong Hindley-Milner type system with algebraic data types, parametric polymorphism, higher order types and typeclasses, very similar to Haskell.
- Mercury has a strong mode system. Each predicate has one or more modes, each of which says which arguments are input and which are output. Each mode is compiled into separate code (its own *procedure*), and the compiler reorders the goals in the body of each procedure to ensure that for each variable, its consumers are executed *after* its producer.
- Mercury has a strong determinism system that can say that a procedure has at least one solution, at most one solution, or both.

An example

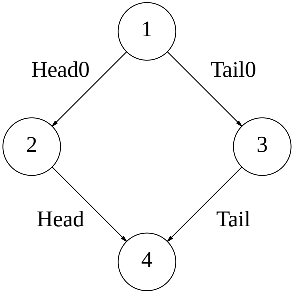
```
:- pred map_list(pred(X, Y)::in(pred(in, out) is det),
    list(X)::in, list(Y)::out) is det.

map_list(_, [], []).
map_list(P, [Head0 | Tail0], [Head | Tail]) :-
    P(Head0, Head),
    map_list(P, Tail0, Tail).
```

This predicate is the standard map function over lists.

The P(Head0, Head) represents a higher order call. The declaration of the first argument gives the type, the mode and the determinism of P.

Reordering

```
map_list(P, In, Out) :-
    (
        In = [],
        Out = []
    ;
        In = [Head0 | Tail0],      % 1
        P(Head0, Head),            % 2
        map_list(P, Tail0, Tail), % 3
        Out = [Head | Tail]        % 4
    ).
```



The data dependencies show that in the recursive case, (1) has to be done first and (4) has to be done last, but (2) and (3) can be done in either order, or in parallel.

OR-parallelism versus AND-parallelism

Logic programs can exhibit two kinds of parallelism: *OR-parallelism* and *AND-parallelism*.

With OR-parallelism, the parallel goals are computing alternate solutions (or alternate sets of solutions) of the same call to a predicate.

With AND-parallelism, the parallel goals are working on different parts of the same solution.

In typical Mercury programs, about 80-90% of procedures have determinism `det` (exactly one solution), about 10-20% are `semidet` (at most one solution), and only about 1-2% are `multi` (at least one solution) or `nondet` (any number of solutions). This makes OR-parallelism useful only in rare cases, so we implement only AND-parallelism.

We also require the conjuncts executed in parallel to be `det`.

AND-parallelism in Mercury

To execute two `det` conjuncts in parallel in Mercury, just replace the comma between them with an ampersand:

```
In = [Head0 | Tail0],
( P(Head0, Head) & map_list(P, Tail0, Tail) ),
Out = [Head | Tail]
```

The code generated for the parallel conjunction will

- create and initialize a barrier,
- spawn off all the conjuncts except the first for other threads running on other CPUs to pick up and execute,
- execute the first conjunct, and
- wait at the barrier for the spawned-off conjuncts to finish.

Dependent vs independent AND-parallelism

The parallelism opportunity in `map_list` is an example of *independent* AND-parallelism, because neither goal produces a variable that is consumed by the other.

Most programs have relatively few goals that are both (a) expensive enough to be worth executing in parallel and (b) independent. To find enough parallelism, we need to exploit *dependent* AND-parallelism as well.

In the 1980s, there were several logic language implementations based on dependent AND-parallelism. Most assumed that *all* conjuncts would be executed in parallel, and used completely different data representations and execution strategies from sequential systems.

These generated lots of overhead, mostly in the form of extra synchronization. These overheads swamped the available speedups.

Low-overhead dependent AND-parallelism

The Mercury compiler internally transforms code like this

```
p(A, B, C, D) &      % A => B, C, D
q(A, B, C, E)        % A, B, C => E
```

into code like this:

```
new_future(FutureB), new_future(FutureC),
(
    p(A, B, C, D),
    signal_future(FutureB, B), signal_future(FutureC, C)
&
    wait_future(FutureB, B2), wait_future(FutureC, C2),
    q(A, B2, C2, E)
)
```

This approach allows the system to pay the cost of synchronization only when synchronization is absolutely needed.

Pushing signals and waits

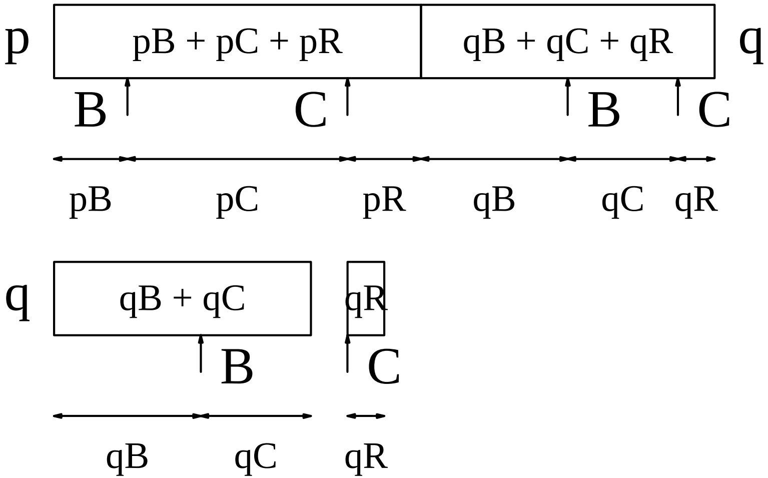
If the first conjunct signals the futures only at the very end, and the second waits for the futures at the very start, there will be no parallelism: instead of a speedup, you get a slowdown from the overhead.

We create specialized variants of p and q that also do the synchronization:

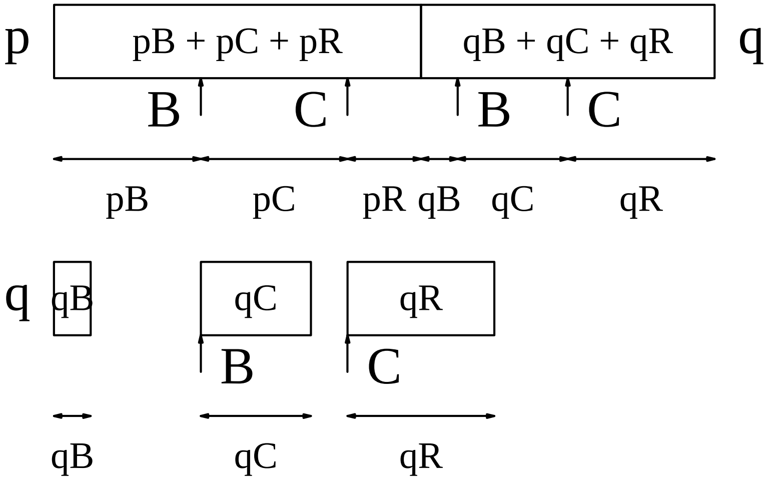
```
new_future(FutureB), new_future(FutureC),
(
    p_and_signal(A, FutureB, FutureC, D)
&
    q_and_wait(A, FutureB, FutureC, E)
)
```

In `p_and_signal`, we try to push the signal operations as early as possible, and in `q_and_wait`, we try to push the wait operations as late as possible. *How much* parallelism you get depends on how far you can push.

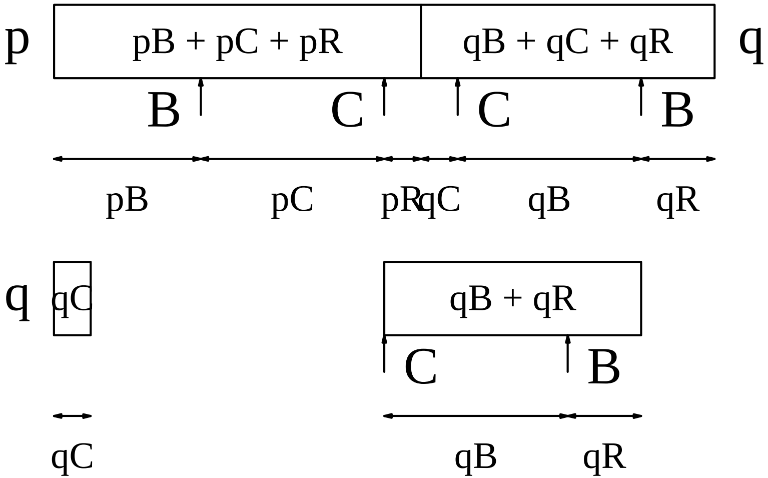
Good overlap



Not so good but sort-of OK overlap



Almost no overlap



Estimating times

```
p(A, FutureB, FutureC, D) :-
    p1(A, E),
    B = E * 2,
    signal_future(FutureB, B),
    p2(A, E, B, FutureC),
    p3(B, D).
```

To estimate the average time between the call of p and the signal on FutureB, we need to know the average time taken by the call to p1 *from this call site*. The Mercury deep profiler can give us this information.

Our estimate of the average time between the call of p and the signal on FutureC will be the time taken by the call to p1 plus the estimated average time between the call of p2 and the signal on FutureC.

This technique also works for estimating times until wait operations.

Coverage profiling

```
p(A, FutureB, FutureC, D) :-
    ( A = <value1>, ...
    ; A = <value2>, ...
    ; A = <value3>, ...
    ).
```

Different arms of the switch may signal e.g. `FutureB` at different times.

To compute the average time until any given future is signalled, we need to know how frequently each arm of the switch is executed. We recently extended the Mercury deep profiler to provide this information.

This adds very little overhead, since most switch arms have a call near the start whose call count we can reuse.

When is a parallel conjunction worthwhile?

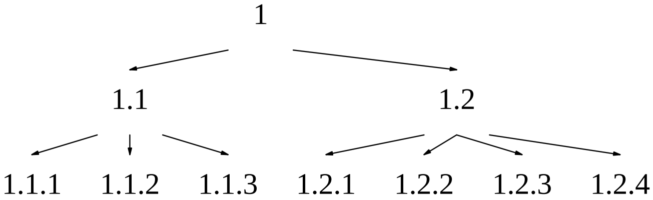
We want to execute two calls in parallel only if

- each call's time cost is above a threshold, and
- the speedup ratio of the conjunction (its sequential time divided by its parallel time) is above another threshold.

If the two calls are not adjacent, each piece of code between them has to be put into one parallel conjunct or the other. This will affect the speedup ratio.

With N pieces of code between the two calls, there are only N+1 sequence-preserving partitions, so it is a simple matter to try them all.

Using profiling feedback



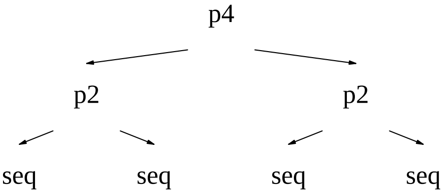
Our feedback tool walks the call tree from the top, with the parallelism level being 1 at the top node.

At each node, we look for two calls that satisfy both conditions.
If we find them, we update the current parallelism level within the two calls by multiplying it by their speedup ratio.

We keep going only until we reach the desired level of parallelism (which could be e.g. eight-fold parallelism for a four-core CPU). We also stop when the node's cost is itself below a third threshold.

Using profiling feedback on recursive code

If the two calls we want to execute in parallel are both recursive calls, we want to create several variants of the procedure:



If just one of the two calls is a recursive call, we want to modify the procedure body to test at runtime whether the runtime system has enough work packages already. If it does, we execute the calls in sequence; if it does not, we execute the calls in parallel.

Conclusion

We are just now working on getting the compiler to act on data fed back from the profiling tools.

When that is done, we will see how well the system works for real programs. Specifically, we want to see what the thresholds should be, whether we need to look for 3+ parallel conjuncts, and whether using averages loses too much precision.

Compiling a Mercury program with parallelism enabled but with no parallel conjunctions yields code that is only slightly slower than compiling that program for sequential execution. Such a program should be faster on one CPU than a parallel Haskell program fully using two or even three CPUs.

Mercury expresses I/O through dataflow. Our mechanism for synchronization based on dataflow therefore also ensures the preservation of the order of I/O actions.