Minimizing the overheads of dependent AND-parallelism

Zoltan Somogyi

The University of Melbourne

ICLP 2011

8 July 2011

Mercury: brief overview

Mercury has

- a strong type system;
- a strong mode system; and
- a strong determinism system.

Mercury supports AND-parallelism for conjunctions in which all the conjuncts are deterministic.

Before this work, Mercury required that all the conjuncts be independent. This work eliminates that restriction.

The problem with dependent AND-parallelism

The early systems that supported AND-parallelism, Concurrent Prolog, Parlog, and Guarded Horn Clauses, aimed to exploit all available parallelism.

Experience with these systems in the 1980s has taught us that levels of parallelism *beyond* what is needed to keep all available CPUs busy have only costs, not benefits.

In these dependent AND-parallel systems, and in many others, an access to a variable is more complicated than in the WAM, mainly because it may require synchronization.

The cost of synchronization is significantly greater than the cost of many primitive operations in the WAM.

Even a *test* to see whether synchronization is required would significantly increase overheads in the Mercury implementation, whose primitive operations are significantly simpler and faster than the WAM's.

Our approach

Even when a program runs on a 64-core processor, we expect the vast majority of conjunctions to be executed in sequence, not in parallel.

When deciding how to support dependent AND-parallelism, our main objective therefore was eliminating overheads *entirely* for sequential code.

The reason why we can (almost) achieve this is that the Mercury compiler has complete knowledge of

- exactly where the values of variables are produced and consumed, and
- what variables are shared between conjuncts in a parallel conjunction.

Our implementation consists of two components:

- a synchronization transformation, to ensure correctness
- a specialization transformation, to improve performance

Synchronization transformation example: before

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

% before transformation
p(A, B, C) :-
  (
    q(A, D),      % produces D
    r(D, E)       % produces E
  ) & (
    s(B, F),      % produces F
    t(D, F, G)    % produces G
  ),
  C = D + E + G. % produces C
```

Synchronization transformation example: after

```
% after transformation
p(A, B, C) :-
  new_future(FutureD),
  (
    q(A, D),     % produces D
    signal_future(FutureD, D),
    r(D, E)      % produces E
  ) & (
    s(B, F),     % produces F
    wait_future(FutureD, D'),
    t(D', F, G) % produces G
  ),
  C = D + E + G.  % produces C
```

The specialization transformation

A parallel conjunction like this will yield a slowdown, not a speedup, because there is no overlap between the executions of the two original conjuncts:

```
(
  p1(In1, S, Out1),
  signal_future(FutureS, S)
) & (
  wait_future(FutureS, S'),
  p2(In2, S', Out2)
)
```

We therefore have a specialization transformation that attempts to push

- each signal as early as possible in the calltree of the producer, and
- each wait as late as possible in the calltrees of the consumers.

Performance results

Our benchmark is a raytracer with two versions: one with independent AND-parallelism (i), and one with dependent AND-parallelism (d).

The different columns show speedups when the raytracer is told to divide the rows of the picture into 1, 8, 16 and 32 chunks respectively. Different rows within a chunk are processed in sequence, different chunks are processed in parallel.

	1i	8i	16i	32i	1d	8d	16d	32d
sequential	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99
parallel, 1 CPU	0.91	1.08	1.08	1.10	0.90	1.02	1.03	1.04
parallel, 2 CPUs	N/A	1.46	1.73	1.74	N/A	2.05	2.14	2.13
parallel, 3 CPUs	N/A	2.16	2.19	2.21	N/A	2.36	2.73	2.72
parallel, 4 CPUs	N/A	2.41	2.49	2.55	N/A	2.83	3.28	3.32

Conclusion

- Our system incurs synchronization overhead on accesses only to a very small proportion of variables; the fraction can be as low as tens out of billions.
- Unlike most other systems that allow multiple consumers, ours requires synchronization for consumers only up to the first one that waits for the variable on all paths.
- Since Mercury expresses I/O action sequencing as data dependencies, our system can also ensure that two I/O predicates called in a conjunction generate the exact same output when executed in parallel as when executed in sequence.

I/O as dataflow

In Mercury, input/output is done by predicates that

- take a unique reference to the initial state of the world as input, and
- return a unique reference to the updated state of the world as output.

The compiler ensures that there is exactly one "state of the world" variable live at any one time.

In this example, that variable is initially `S0`, then `S1`, then `S`.

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

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