Controlling Loops in Parallel Mercury Code

Paul Bone, Zoltan Somogyi and Peter Schachte
National ICT Australia
The University of Melbourne

Declarative Aspects of Multicore Programming

January 28, 2012

About Mercury

- Mercury is a **pure** logic/functional language designed to support the creation of large, reliable, efficient programs.
- It has a syntax similar to Prolog's, however the operational semantics are very different.
- It is strongly typed using a Hindley Milner type system.
- It also has mode and determinism systems.

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

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

Parallelism in Mercury

Introducing parallelism in Mercury can be done simply by replacing a comma with &, the parallel conjunction operator:

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

Parallel computations are handled by:

Engines Correspond to PThreads. One engine is created for each core on a multicore system. Each engine has a set of abstract machine registers.

Contexts Represent computations in progress. They are executed by engines. Although lighter than PThreads, contexts are still somewhat heavy: each one contains two stacks.

Dependent right-recursive parallel code

Programmers are encouraged to write tail recursive code. In Mercury, this means that the last call in a clause is often a recursive call.

Mercury allows dependent AND-parallelism. Variables such as `Acc1` are *shared* between the parallel conjuncts. Their synchronization is handled automatically.

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

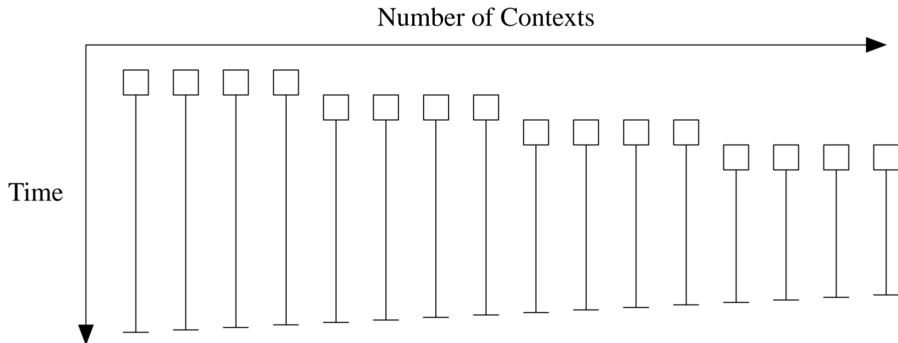
The general parallel conjunction transformation

A parallel conjunction G_1 & G_2 & G_3 is executed by spawning off G_2 & G_3 and then executing G_1 immediately in the current context. This mixed-level pseudo-code shows the operations that implement this.

```
case_label:
    SyncTerm st;
    init_sync_term(&st);
    spawn_off(spawn_off_label, &st);
    M(X, Y);
    F(Y, Acc0, Acc1);
    join_and_continue(resume_label, &st);
spawn_off_label:
    map_foldl(M, F, Xs, Acc1, Acc);
    join_and_terminate(&st);
resume_label:
    return;
```

Execution of dependent right-recursive parallel code

The original context has to stay around until the recursive call finishes, so it can resume. Parallelizing such a loop in this way will cause it to use a number of contexts **linear** in the depth of the recursion. If each context contains 4 megabytes of stack space, a loop only has to iterate 256 times to consume a gigabyte of memory!



Loop control structure

Our solution of this problem associates a *loop control structure* with each loop. This structure contains a fixed number of slots, each of which has a pointer to a single context.

Once a context is allocated to a slot, the context is not released until the loop has finished. Instead, it is reused for later iterations.

We replace the original looping procedure with code that creates the loop control structure, before calling a renamed and transformed version of its old self.

```
map_foldl(M, F, Xs, Acc0, Acc) :-
    create_loop_control(LC),
    map_foldl_lc(LC, M, F, Xs, Acc0, Acc).
```

Loop control transformation

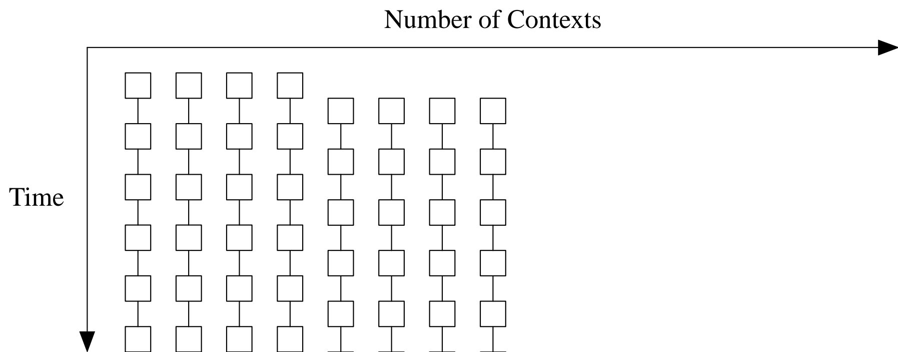
```
map_foldl_lc(LC, M, F, [X | Xs], Acc0, Acc) :-
    LCS = lc_wait_for_free_slot(LC),
    lc_spawn_off(LC, LCS, spawn_off_label),
    map_foldl_lc(LC, M, F, Xs, Acc1, Acc).  % Tail call
spawn_off_label:
    M(X, Y);
    F(Y, Acc0, Acc1);
    lc_free_slot(LC, LCS);

map_foldl_lc(LC, _, _, [], Acc, Acc).
    lc_finish(LC).
```

Only as many iterations of the loop can be active as there are slots in the loop control structure.

Execution of loop controlled code

The first time each slot is used, we create a context for that slot. After the initial rampup period, the loop always uses the configured number of contexts, never more. After the loop terminates, we free the contexts.



Memory usage results: contexts and megabytes

	mandelbrot		raytracer		spectral	
seq	1	0.62	1	0.62	1	0.62
par, no &	1	0.62	1	0.62	1	0.62
par, &, 1c, nolc, c128	1	0.62	1	0.62	1	1.12
par, &, 1c, nolc, c512	1	0.62	1	0.62	1	1.12
par, &, 1c, lc1	2	1.25	2	1.25	2	1.75
par, &, 1c, lc2	3	1.88	3	1.88	3	2.38
par, &, 1c, lc4	5	3.12	5	3.12	5	3.62
par, &, 2c, nolc, c128	257	160.62	257	160.62	257	161.12
par, &, 2c, nolc, c512	601	375.62	1025	640.62	1025	641.12
par, &, 2c, lc1	4	2.50	4	2.50	3	2.38
par, &, 2c, lc2	6	3.75	6	3.75	5	3.62
par, &, 2c, lc4	10	6.25	10	6.25	9	6.12
par, &, 4c, nolc, c128	513	320.62	513	320.62	513	321.12
par, &, 4c, nolc, c512	601	375.62	1201	750.62	2049	1281.12
par, &, 4c, lc1	6	3.75	6	3.75	5	3.62
par, &, 4c, lc2	10	6.25	10	6.25	9	6.12
par, &, 4c, lc4	18	11.25	18	11.25	17	11.12

Time results: seconds and speedups

	mandelbrot		raytracer		spectral	
seq	19.37	(0.97)	19.50	(1.21)	16.07	(1.19)
par, no &	18.75	(1.00)	23.55	(1.00)	19.07	(1.00)
1c, nolc, c128	18.74	(1.00)	23.46	(1.00)	19.30	(0.99)
1c, nolc, c512	18.74	(1.00)	23.43	(1.00)	19.30	(0.99)
1c, lc2	18.74	(1.00)	23.54	(1.00)	19.30	(0.99)
1c, lc2, tr	18.74	(1.00)	23.79	(0.99)		n/a
2c, nolc, c128	17.82	(1.05)	25.68	(0.92)	19.25	(0.99)
2c, nolc, c512	9.60	(1.95)	20.34	(1.16)	18.54	(1.03)
2c, lc2	9.69	(1.94)	14.14	(1.67)	9.96	(1.91)
2c, lc2, tr	9.78	(1.92)	14.04	(1.68)		n/a
4c, nolc, c128	8.35	(2.25)	26.93	(0.87)	18.91	(1.01)
4c, nolc, c512	4.84	(3.88)	14.12	(1.67)	16.83	(1.13)
4c, lc2	4.74	(3.96)	9.35	(2.52)	4.98	(3.83)
4c, lc2, tr	4.76	(3.94)	9.41	(2.50)		n/a

Conclusion

- We have prevented excessive memory usage.
- We can preserve tail recursion in parallel recursive code.
- We have also reduced the overheads of parallelism, resulting in greater parallel speedups.

Further work

- We plan to add support for profiling loop-controlled computations with ThreadScope.
- We also intend to add knowledge of the loop-control cost model to our automatic parallelization system.
- We would like to introduce new transformations that efficiently control parallelism for other common programming patterns such as divide and conquer.

Communication through stack frames

The variables used to communicate to and from spawned off computations, excluding shared variables, are stored on the parent's stack frame.

The code that is spawned off accesses these variables through an abstract machine register called the *parent stack pointer* rather than the normal stack pointer register. This mechanism existed before we introduced loop control.

However, this prevents tail recursion since a spawned off computation will need access to this stack frame even after the original context executed the recursive call.

Getting tail recursion back

In tail recursive code, we can create a stack frame on the child context's stack and copy over any variables it needs.

```
map_foldl_lc(LC, M, F, [X | Xs], Acc0, Acc) :-
    LCS = lc_wait_for_free_slot(LC),
    incr_child_stack_ptr(LC, LCS, NumSlots);
    child_stack_var(...)  = M;
    child_stack_var(...)  = F;
    ...
    lc_spawn_off(LC, LCS, spawn_off_label),
    map_foldl_lc(LC, M, F, Xs, Acc1, Acc).  % Tail call
```

In tail recursive code, we never need to manage communication from the spawned off code to the parent code. This is because there is no code after the recursive call, and therefore no variable can be consumed *after* the parallel conjunction.

Sparks

spawn_off actually creates a spark, which join_and_terminate may execute using the current engine if it has not been stolen by another engine.

```
case_label:
    SyncTerm st;
    create_sync_term(&st);
    spawn_off(spawn_off_label, &st);
    M(X, Y);
    F(Y, Acc0, Acc1);
    join_and_terminate(resume_label, &st);
spawn_off_label:
    map_foldl(M, F, Xs, Acc1, Acc);
    join_and_terminate(resume_label, &st);
resume_label:
    return;
```