Automatic Parallelism for Mercury

Paul Bone

The University of Melbourne

National ICT Australia

Ph.D. Completion Seminar

May 2nd, 2012

Motivation — Multicore computing

Computing has traditionally seen a logarithmic increase in CPU clock speeds. However, due to physical limitations this trend no-longer continues.

Manufacturers now ship multicore processors to continue to deliver better-performing processors without increasing clock speeds.

Programmers who want to take advantage of the extra cores on these processors must write parallel programs.

Motivation — Threaded programming

Threads are the most common method of parallel programming. When using threads, programmers use critical sections to protect shared resources from concurrent access.

Critical sections are normally protected by locks, but it is easy to make errors when using locks.

- Forgetting to use locks can put the program into an inconsistent state, corrupt memory and crash the program.
- Using multiple locks in different orders in different places can lead to deadlocks.
- Critical sections are not composable, nesting critical sections may acquire locks in different orders in different places.
- Misplacing lock operations can lead to critical sections that are too wide (causing poor performance) or too narrow (causing data corruption and crashes).

Automatic parallelism

A good compiler performs many optimisations on behalf of the programmer. Programmers rarely think about:

- register allocation,
- inlining,
- simplification such as constant propagation & strength reduction.

We believe that parallelisation is just another optimisation, and it would be best if the compiler handled it for us; so that, like any other optimisation, we wouldn't need to think of it.

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).
```

Effects in Mercury

In Mercury, all effects are explicit, which helps programmers as well as the compiler.

```
main(IO0, IO) :-
    write_string("Hello ", IO0, IO1),
    write_string("world!\n", IO1, IO).
```

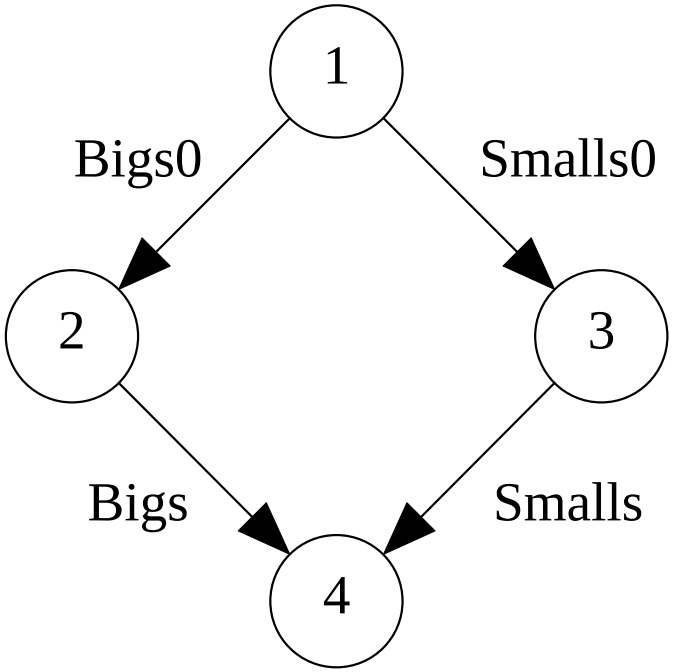
The I/O state represents the state of the world outside of this process. Mercury ensures that only one version is alive at any given time.

This program has three versions of that state:

- `IO0` represents the state before the program is run
- `IO1` represents the state after printing `Hello`
- `IO` represents the state after printing `world!\n`.

Data dependencies

```
qsort([], []).
qsort([Pivot | Tail], Sorted) :-
    partition(Pivot, Tail, Bigs0, Smalls0),      %1
    qsort(Bigs0, Bigs),                          %2
    qsort(Smalls0, Smalls),                      %3
    Sorted = Smalls ++ [Pivot | Bigs].           %4
```



- Steps 2 and 3 are independent.
- This is easy to prove because there are never any *side* effects.
- They may be executed in parallel.

Explicit Parallelism

Mercury allows explicit, deterministic parallelism via the parallel conjunction operator &.

```
qsort([], []).
qsort([Pivot | Tail], Sorted) :-
    partition(Pivot, Tail, Bigs0, Smalls0),
    (
        qsort(Bigs0, Bigs)
    &
        qsort(Smalls0, Smalls)
    ),
    Sorted = Smalls ++ [Pivot | Bigs].
```

Why make this automatic?

We might expect parallelism to yield a speedup in the quicksort example, but it does not.

The above parallelisation creates N parallel tasks for a list of length N. Most of these tasks are trivial and the overheads of managing them slow the program down.

Programmers rarely understand the performance of their programs, even when they think they do.

Runtime system changes

Before we can automatically parallelise programs effectively we need to be able to manually parallelise them effectively. This meant making several improvements to the runtime system.

The RTS has several objects used in parallel Mercury programs.

Engines represent abstract CPUs, the RTS will create as many engines as there are processors in the system, and control each one from a POSIX Thread.

Contexts represent a computation in progress, they contain the stacks for that computation, and a copy of the engine's registers when the context is suspended.

Sparks are a very small structure representing a computation that has not yet been started, and therefore has no allocated stack space.

Work stealing

Peter Wang introduced sparks and a partial work stealing implementation.

Work stealing reduces contention on a global queue of work by allowing each context to maintain its own work stack. Contexts can:

- Push a spark onto their own stack.
- Pop a spark off their own stack.
- Steal a spark from the cold end of another's stack.

All of these operations are lock free, the first two operations are wait free and do not use any atomic operations. The stealing operation uses an atomic compare-and-swap that may busy-wait.

Credit: 80% Peter Wang, 20% myself, excluding the queue data structure.

Dependent Parallelism

Mercury can handle dependencies between parallel conjuncts. *Shared variables* are produced in one conjunction and consumed in another.

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

`Acc1` will be replaced with a *future*, If the second conjunct attempts to read from the future before the first conjunct writes the future, its context will be blocked and resumed once the first conjunct has placed a value into the future.

Right-recursive parallel code

Mode correctness requires that all producers of variables occur before consumers in conjunctions.

Programmers are encouraged to make their code tail-recursive. This means that the recursive call is placed lasted in a conjunction so that it can become a tail call.

A parallel conjunction G_1 & G_2 & ... & G_N will be executed by spawning off G_2 & ... & G_N, then executing G_1 immediately. In the common case that the forked-off task is not taken up by another engine then, a dependency between the tasks does not require a context switch.

However, if the forked-off task was taken by another engine, the original context must be suspended until that task completes. When the last conjunct is a tail call, it often takes far longer to execute than the other conjuncts. Causing the original context to be blocked for a long time.

Decomposing a parallel conjunction

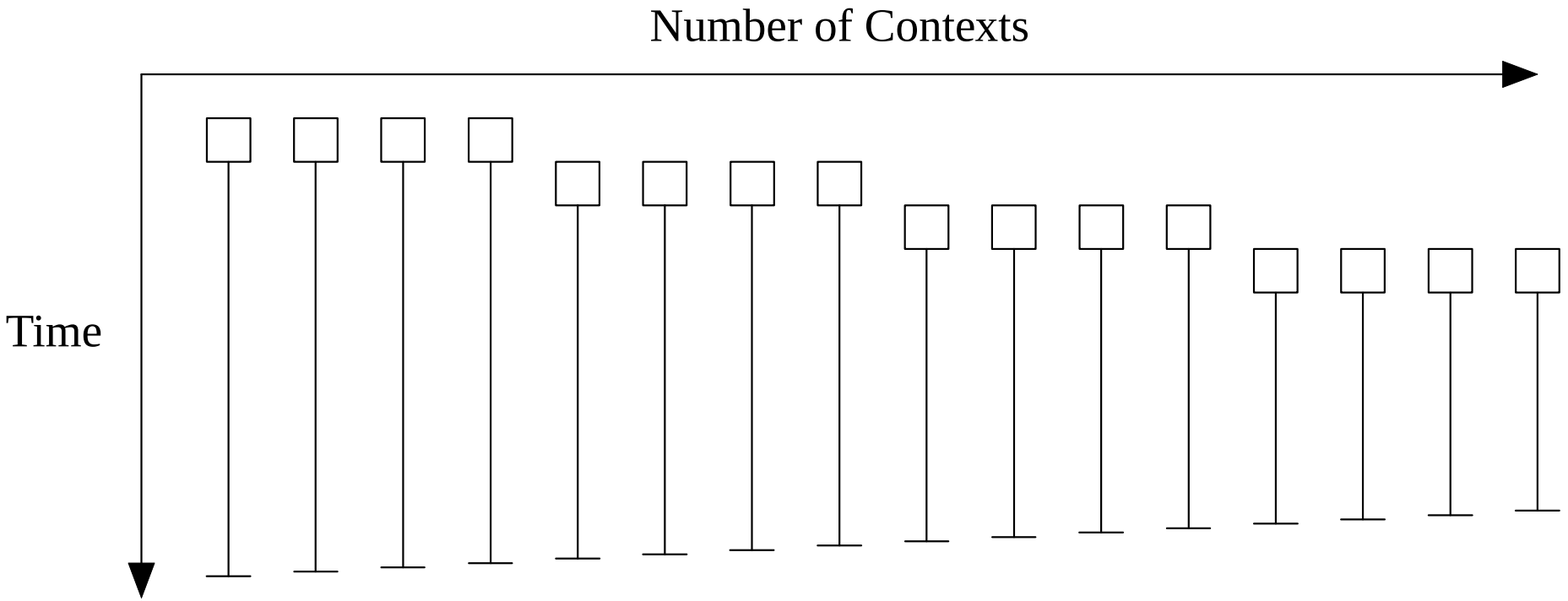
Pseudo compiler output:

```
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_continue(resume_label, &st);
resume_label:
    return;
```

Execution of right-recursive parallel code

Blocking the original context can create a pathological worst-case behaviour: the same behaviour will occur at each level of recursion.

This will cause it to use a number of contexts **linear** in the depth of the recursion.



If each context contains 4MB of stack space, a loop only of 256 iterations will consume 1GB!

Workaround — Reorder conjuncts

The compiler understands recursion including mutual recursion. Therefore, it can move conjuncts with recursive calls to the left of those without. This avoids the problem above.

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

However, where dependencies exist conjuncts cannot be moved without violating the program's mode-correctness. Therefore, this solution only works in the rare cases that parallel conjunctions are independent.

Workaround — The max-contexts limit

To reduce the impact of this, Peter added a maximum limit on the number of contexts that could be in memory at the same time.

When the limit is reached no context can be created to handle the spawned off computation.

This limit trades memory usage for sequential execution.

Management of work queues

Work queues where originally owned by contexts, if there where 100's of contexts active then there where 100's of work queues to steal work from, most of which are usually empty. This makes work stealing slower.

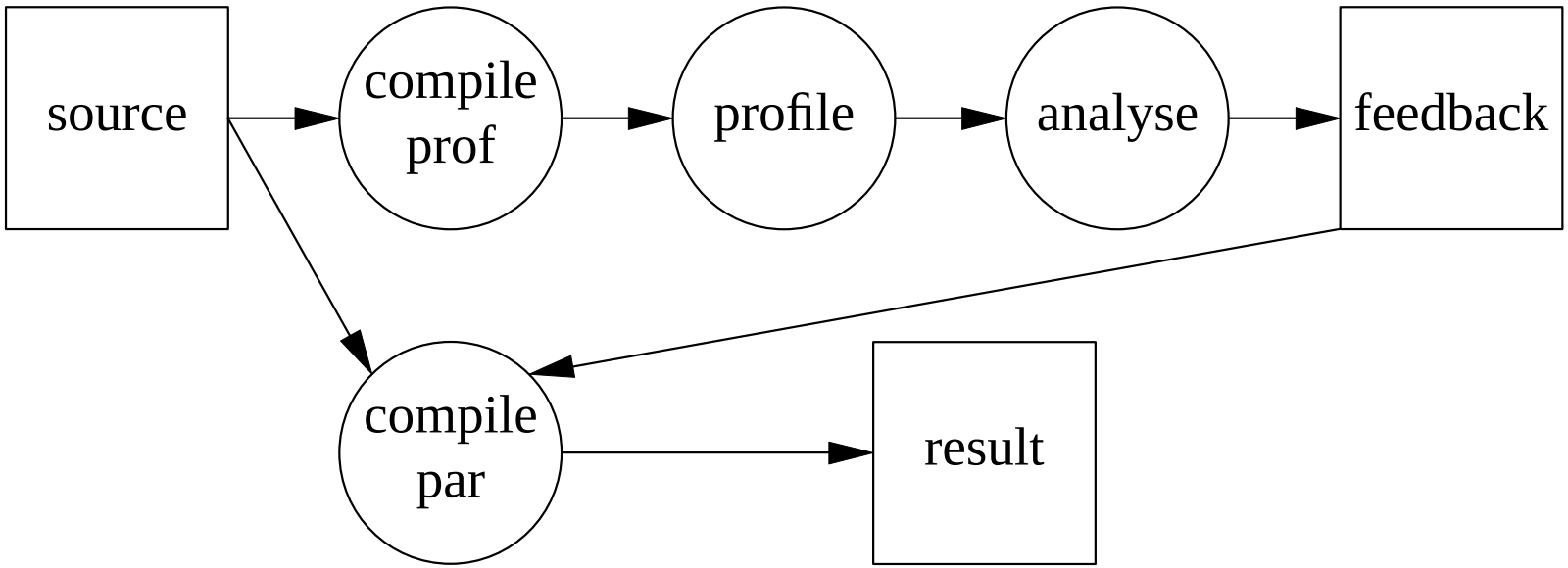
Furthermore, as contexts are created and destroyed the number of work queues changes. This required a global lock to manage the set of work queues,

Work queues are now owned by engines, which means that there are a small fixed number of queues. Greatly simplifying the work stealing code.

More importantly it is faster in the pathological right-recursive case.

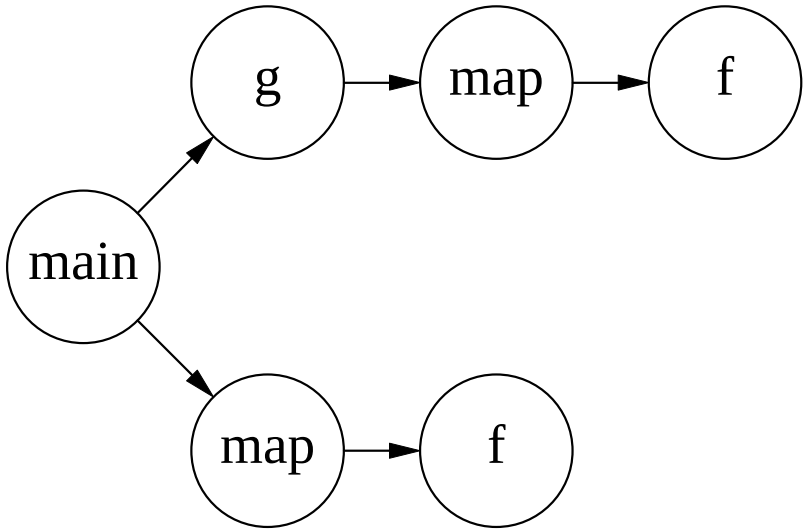
Our approach

- Profile the program to find the expensive parts.
- Analyse the program to determine what parts *can* be run in parallel.
- Select only the parts that can be parallelised *profitably*. This may involve trial and error when done by hand.
- Continue introducing parallel evaluation until the all processors are fully utilised or there is no profitable parallelism left.



Deep profiler call graphs

The deep profiler records profiling data not just for a call, but for a call-chain. The chain is an alternating list of call and call-site objects.

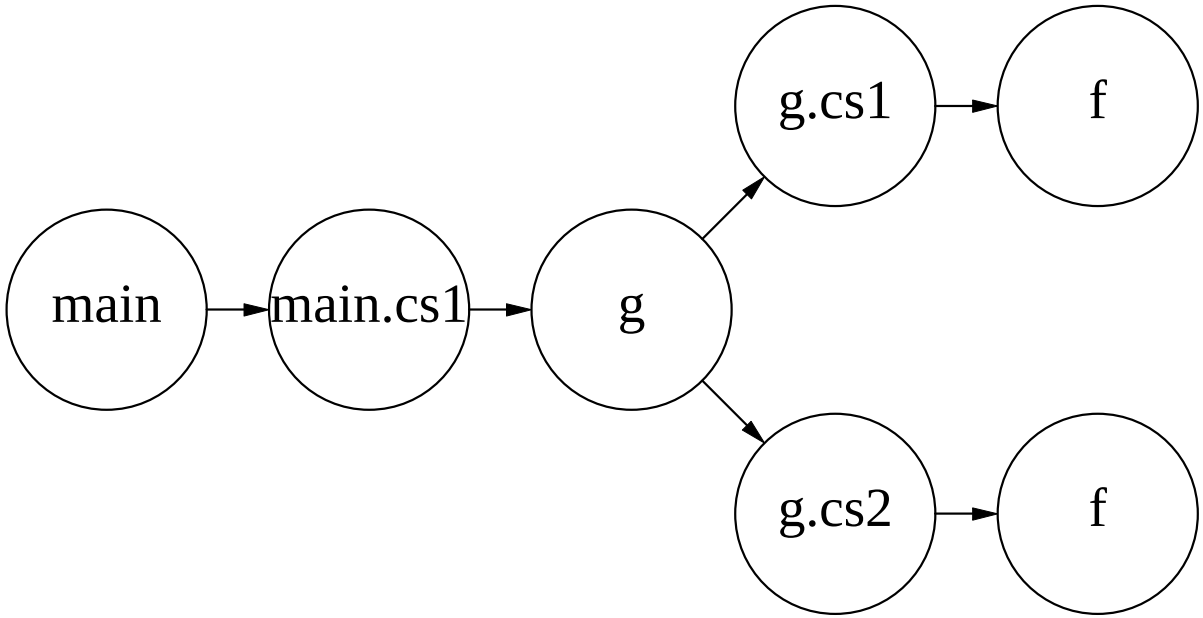


The two calls to *f* are recorded separately as the first one goes via *g* and the second does not.

This even works with higher order calls such as the one in map.

Deep profiler call graphs

Because recursive calls are included in this chain, multiple calls to the same procedure from a single procedure will have their profiling data recorded separately.

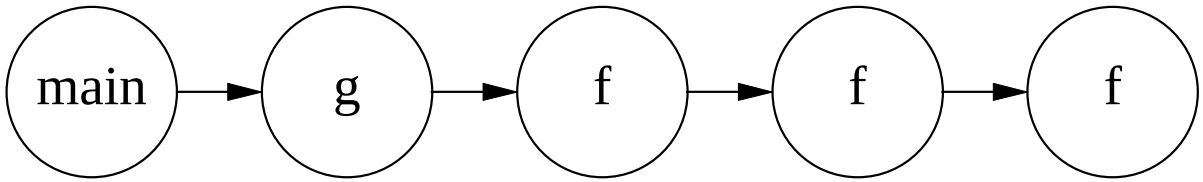


If g contains two call sites that both call f, then these also record separate profiling data for f and its descendants.

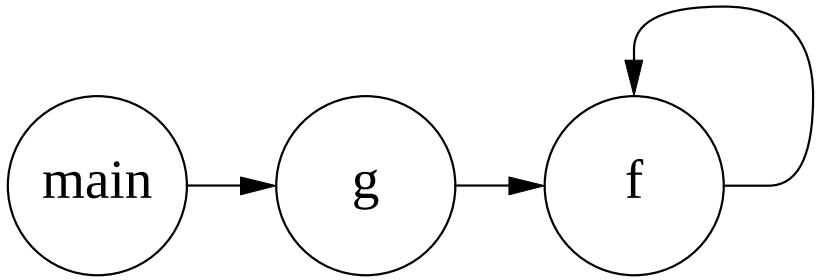
Deep profiler call graphs

This is very powerful and helps to collect a lot of very useful data about a programs performance. However, it does not separate recursions or mutual recursions.

The call tree:



Is actually recorded as:



This creates a tree of strongly connected components (SCCs).

Recursion types

Recursion type	% of procedures	% of runtime
Not recursive	78.04%	57.53%
Single recursion	12.91%	28.96%
Mutual recursion (3 procs)	2.52%	4.78%
Mutual recursion (2 procs)	1.09%	2.20%
Divide & Conquer	0.51%	3.27%
Mutual recursion (4 procs)	0.29%	1.39%
Mutual recursion (5 procs)	0.28%	1.10%
Recursion with levels 0, 2, 3, 4	0.15%	1.04%
Unclassified/unknown	3.09%	0.26%

Taken from an analysis on the mercury compiler.

Unclassified/unknown recursions include builtin code and code with determinisms the analysis cannot handle.

Most recursions with multiple levels such as 0, 2, 3, 4 are in the `tree234` module of the standard library.

Cost of recursive calls

For a singlely-recursive procedure we know:

no. of outside calls	no. of inside (recursive) calls
cost of base case	cost of recursive case (minus recursive call)
cost of shared code	

We can calculate:

$$
\begin{aligned}
MaxDepth &= \frac{NumInsideCalls}{NumOutsideCalls} \\
AvgDepth &= \frac{MaxDepth}{2} \\
RecCost(Depth) &= Shared + Base + Depth(Shared + RecBranch)
\end{aligned}
$$

Depending on the situation, we can calculate the cost of a recursive call at any depth.

Different equations can be derived for each recursion type.

Finding parallelisation candidates

The deep profiler's call graph is a tree of strongly connected components (SCCs). Each SCC is a group of mutually recursive calls. The automatic parallelism analysis follows the following algorithm:

- Recurse depth-first down the call graph from `main/2`.
- Analyse each procedure of each SCC, identify conjunctions that have two or more goals whose cost is greater than a configurable threshold.
- Stop recursing into children if either:
  - the child's cost is below another configurable threshold; or
  - there is no free processor to exploit any parallelism the child may have.

Overlap

Dependencies between variables can greatly effect the amount of parallelism that can be gained.

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

`F` needs `Y` from `M`, and the recursive call needs `Acc1` from `F`.

There are 3 ways to parallelise the conjunction in `map_foldl`.

Parallelising `map_foldl`

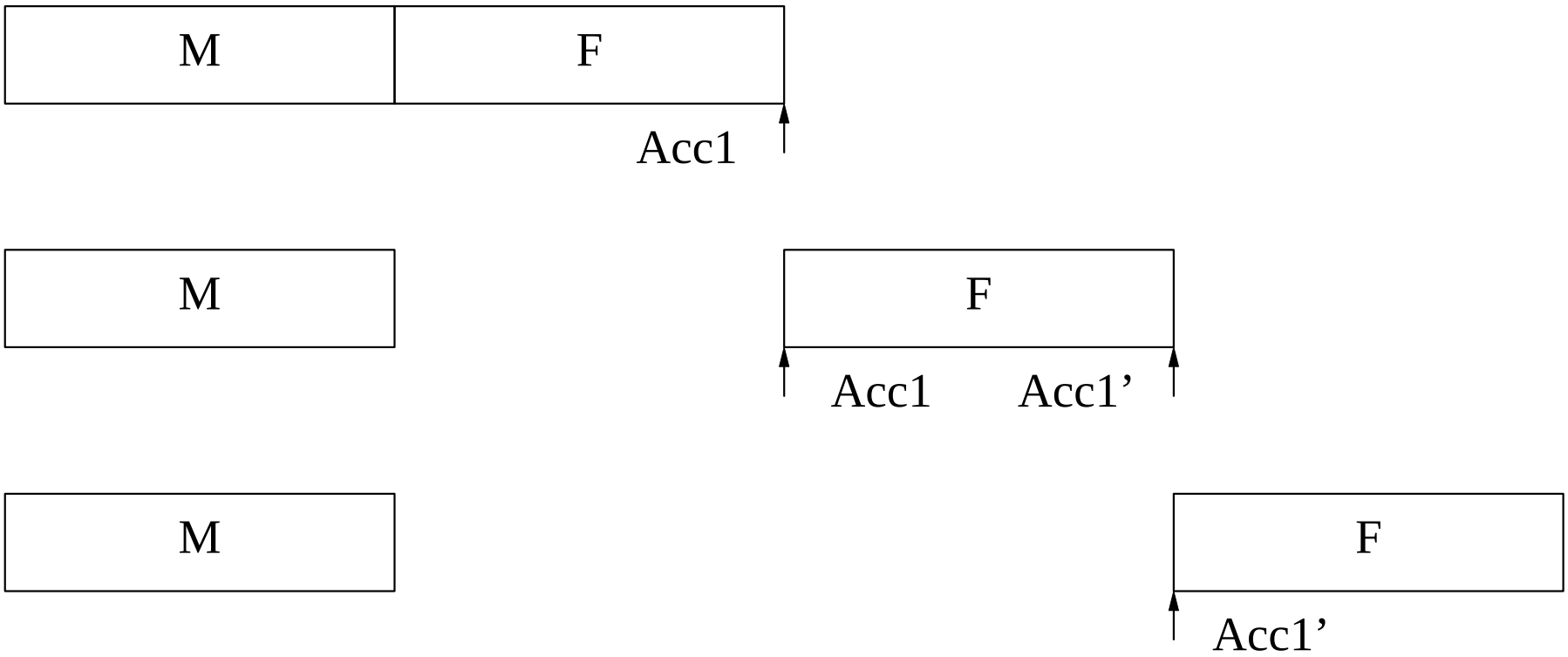
`Y` is produced at the very end of `M` and consumed at the very start of `F`, so the execution of these two calls cannot overlap.

`Acc1` is produced at the end of `F`, but it is *not* consumed at the start of the recursive call, so some overlap *is* possible.

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

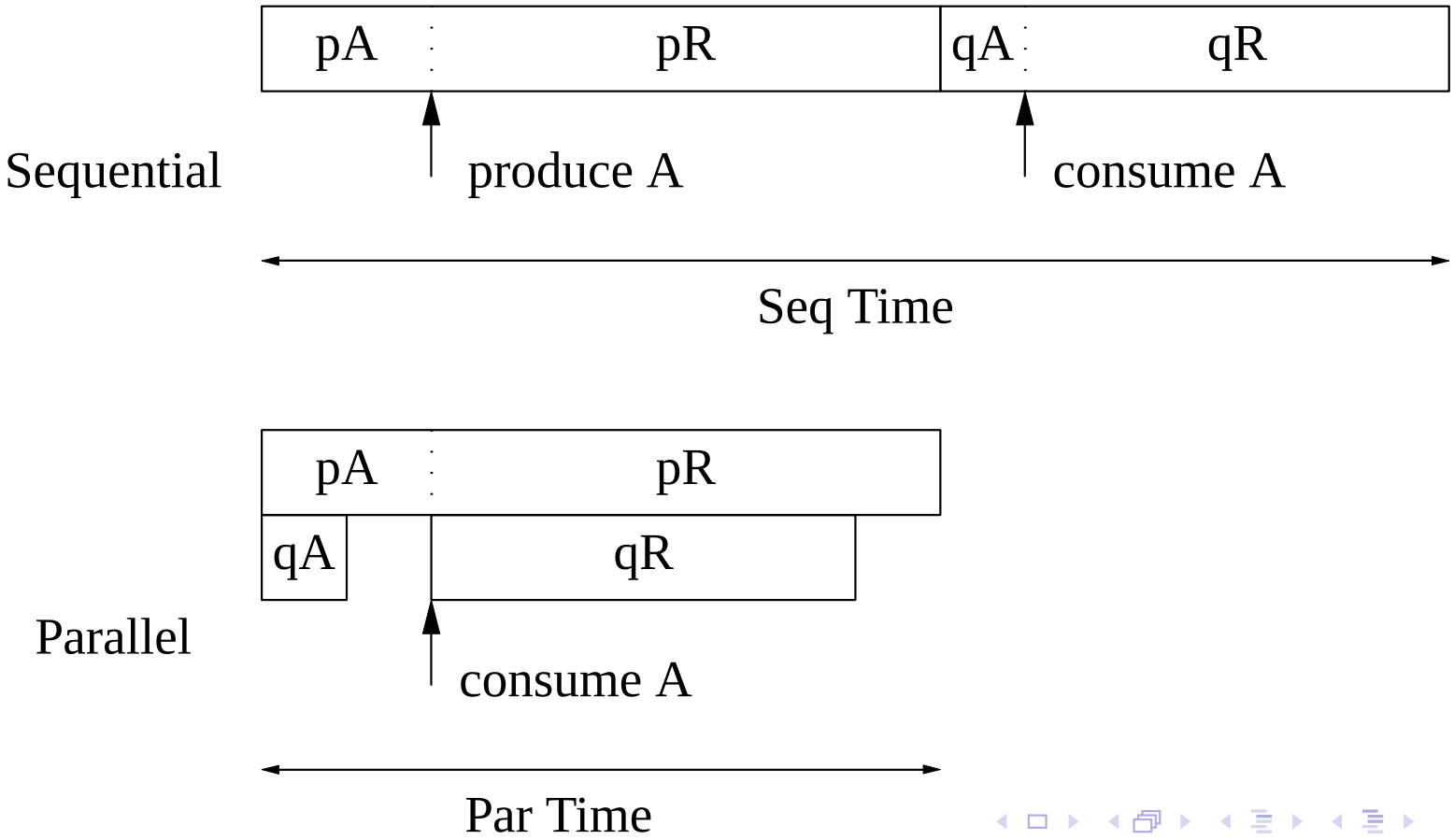
`map_foldl` overlap

The recursive call needs `Acc1` only when it calls `F`. The calls to `M` can be executed in parallel.



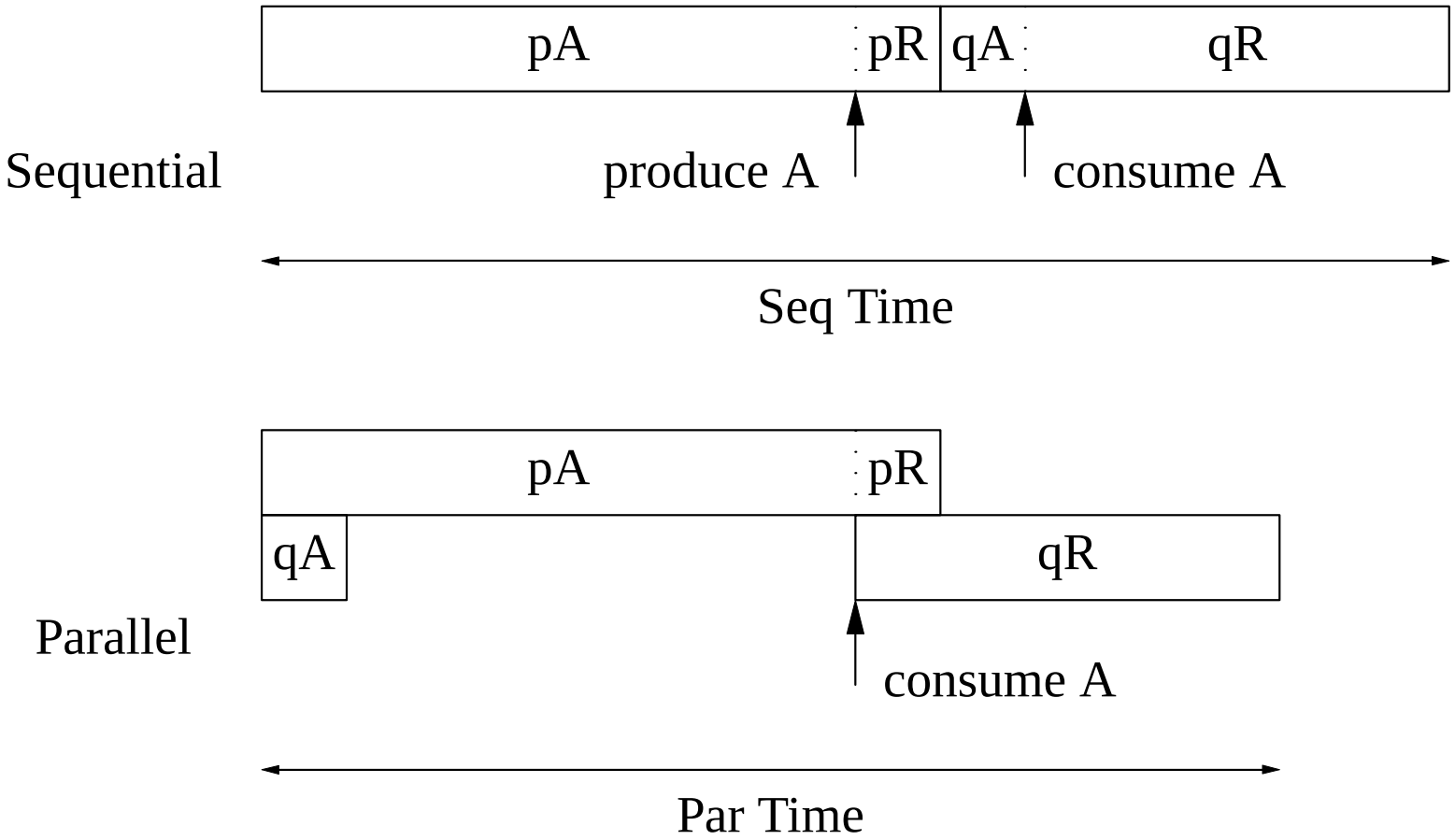
Simple overlap example

We conceptually split each task split into sections, each section ended by the production or consumption of a shared variable. *pR* and *qR* denote the time that the tasks take to compute any non-shared variables needed after the conjunction.



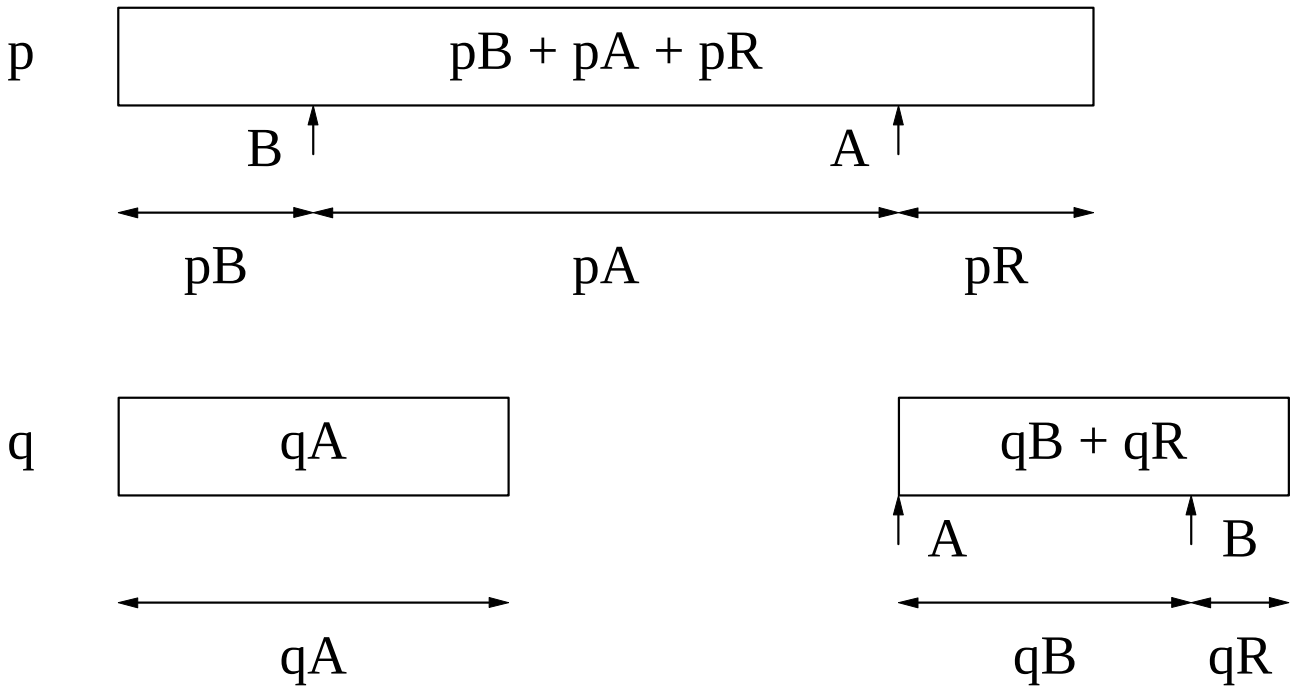
Pessimal overlap example

If variables are produced or consumed at different times within goals, then the overlap can vary greatly.



Overlap with more than one dependency

We calculate the execution time of q by iterating over its sections. In this case, that means iterating over the variables it consumes in the order that it consumes them.



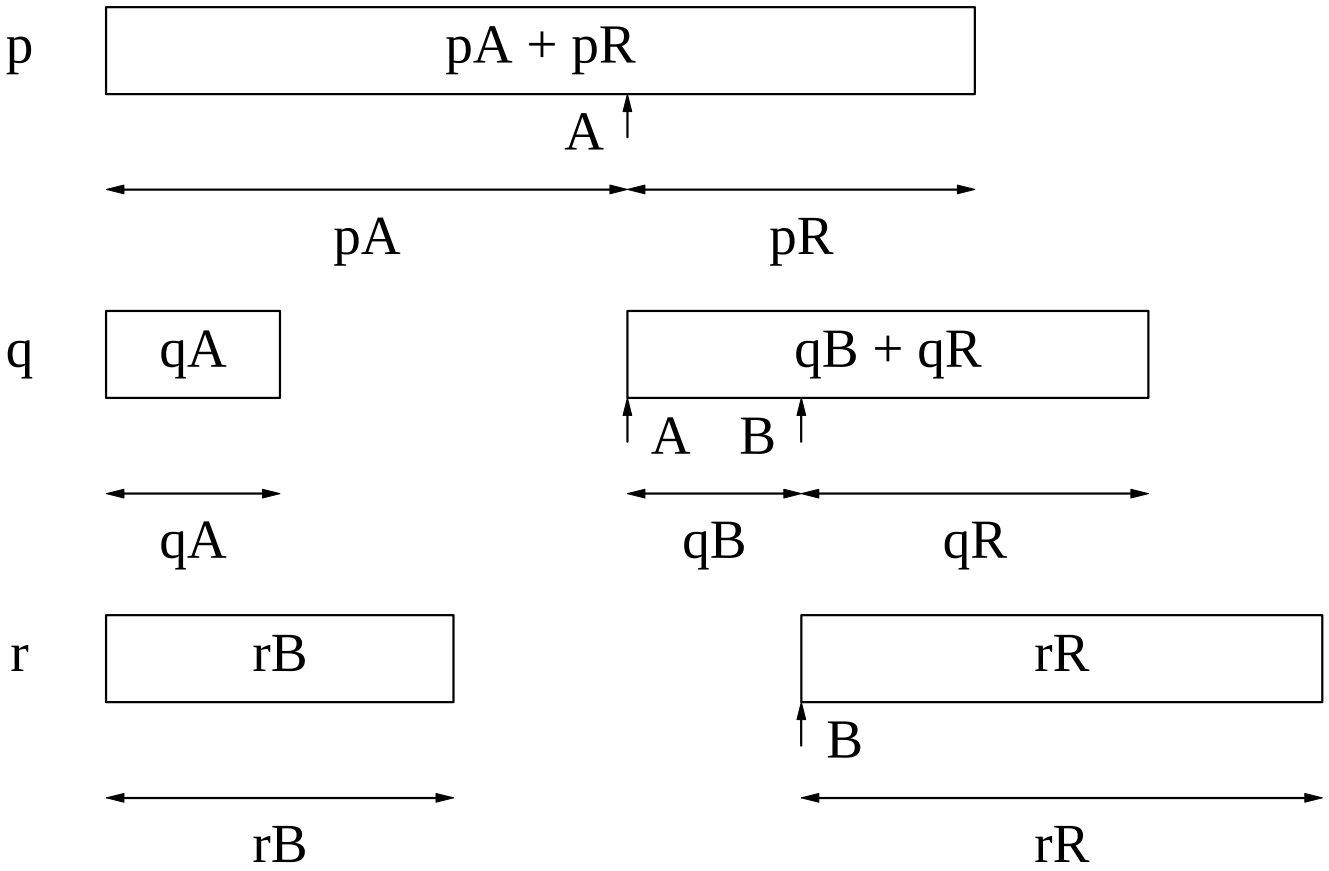
Overlap algorithm — Loop over variables

```
find_conjunct_par_time(Goal, SeqTime,
    inout CurParTime, inout ProdTimeMap):
  ProdConsList := get_sorted_var_uses(Goal)
  CurSeqTime := 0
  forall (Var_j, Time_j) in ProdConsList:
    Duration_j := Time_j - CurSeqTime
    CurSeqTime := CurSeqTime + Duration_j
    if Goal produces Var_j:
      CurParTime := CurParTime + Duration_j + ...
      ProdTimeMap[Var_j] := CurParTime
    else Goal must consume Var_j:
      ParWantTime := CurParTime + Duration_j + ...
      CurParTime := max(ParWantTime, ProdTimeMap[Var_j]) + ...
  DurationRest := SeqTime - CurSeqTime
  CurParTime := CurParTime + DurationRest
```

The ...s represent estimates of overheads.

Overlap of more than two tasks

A task that consumes a variable can occur only on the *right* of the task that generates its value. Therefore, we process conjuncts from left to right.



Overlap algorithm — Loop over conjuncts

```
find_par_time(Conjs, SeqTimes) returns TotalParTime:
  N := length(Conjs)
  ProdTimeMap := empty
  TotalParTime := 0
  for i in 1 to N:
    CurParTime := 0 + ...
    find_conjunct_par_time(Conjs[i], SeqTimes[i],
      CurParTime, ProdTimeMap)
    TotalParTime := max(TotalParTime, CurParTime)
  TotalParTime := TotalParTime + ...
```

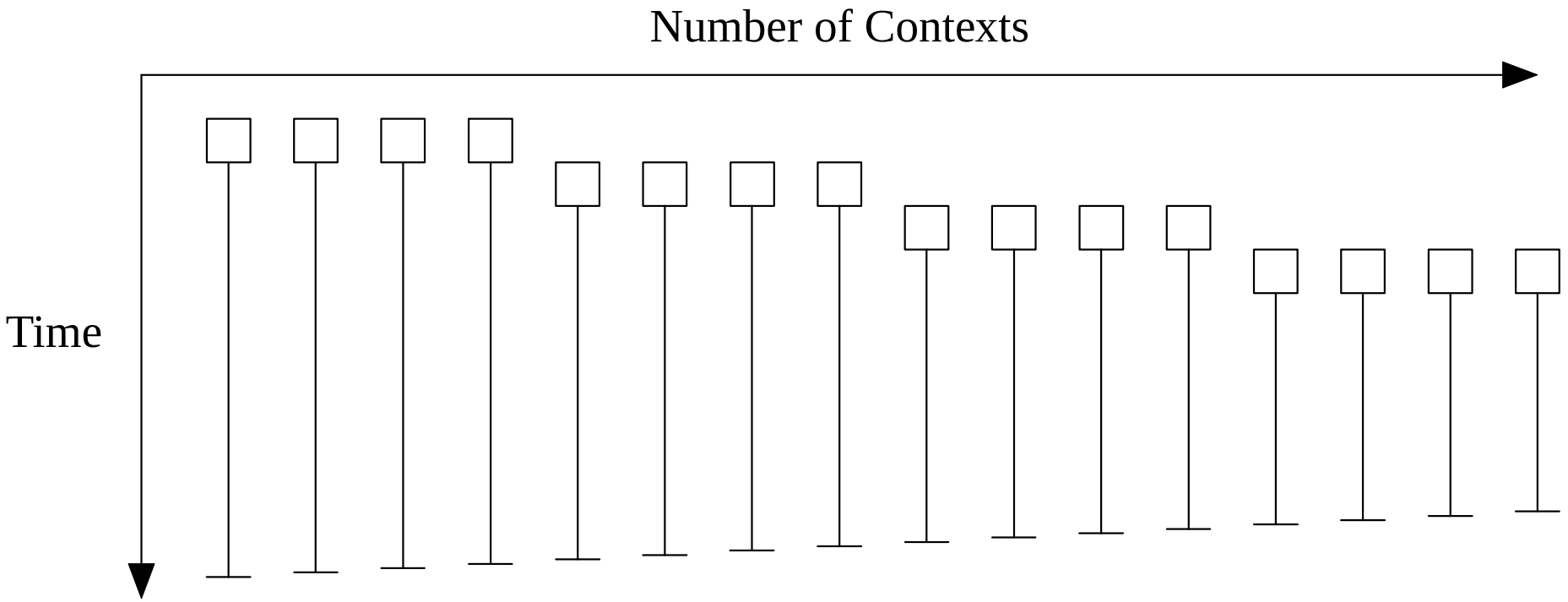
The ...s represent estimates of overheads.

Results

	mandelbrot		raytracer		matrixmult	
seq	19.6	(0.95)	17.8	(1.26)	7.7	(1.43)
thread-safe seq	18.6	(1.00)	22.4	(1.00)	11.0	(1.00)
thread-safe p1						
Indep only	18.6	(1.00)	22.3	(1.00)	11.0	(1.00)
Naive	18.6	(1.00)	22.4	(1.00)	11.0	(1.00)
Overlap	18.5	(1.01)	22.4	(1.00)	11.0	(1.00)
thread-safe p2						
Indep only	18.6	(1.00)	22.4	(1.00)	5.5	(1.99)
Naive	9.4	(1.98)	13.0	(1.72)	11.0	(1.00)
Overlap	9.5	(1.96)	12.8	(1.75)	5.5	(1.99)
thread-safe p4						
Indep only	18.7	(0.99)	22.6	(0.99)	2.9	(3.85)
Naive	4.8	(3.88)	7.9	(2.84)	11.0	(1.00)
Overlap	4.8	(3.88)	7.8	(2.87)	2.9	(3.85)

Execution of right-recursive parallel code

Recall the problem with parallel right recursion.



If each context contains 4MB of stack space, a loop only of 256 iterations will consume 1GB!

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

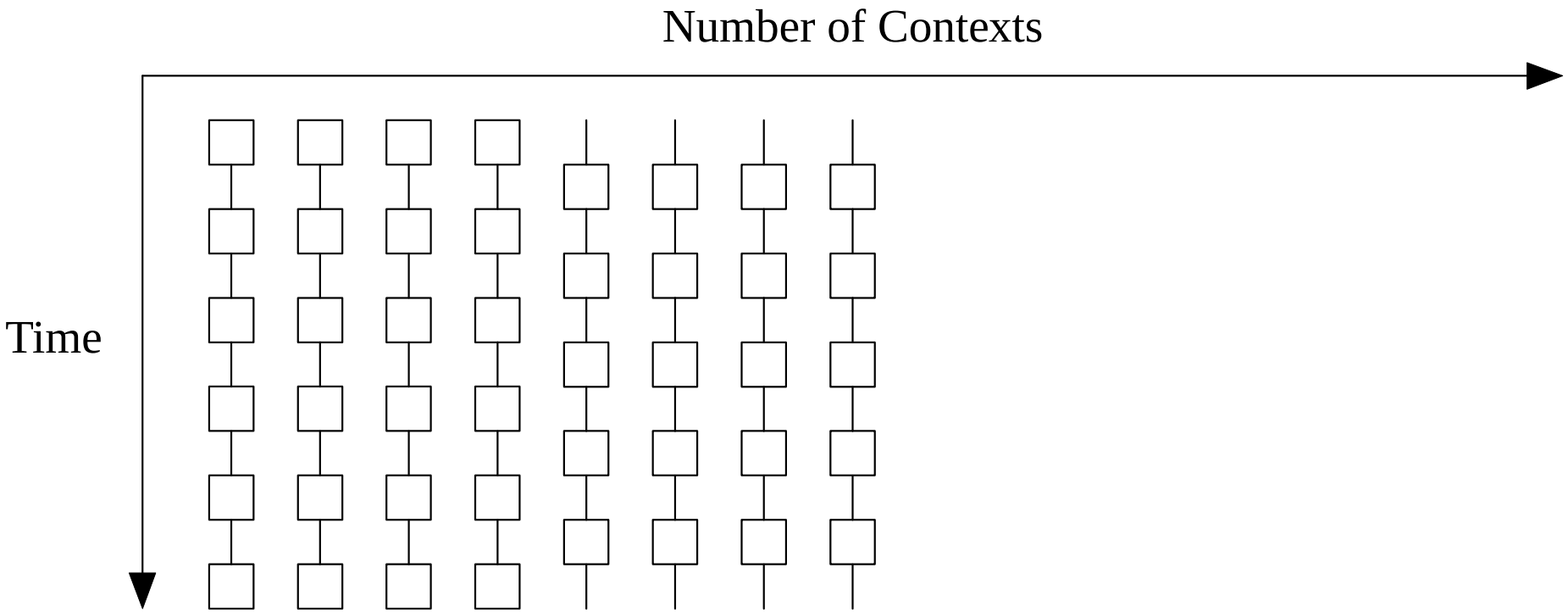
```
map_foldl_lc(LC, M, F, [X | Xs], Acc0, Acc) :-
    LCS = lc_wait_for_free_slot(LC),
    lc_spawn_off(LC, LCS, spawn_off_label),
    map_foldl_lc(LC, M, F, Xs, Acc1, Acc).  % Tail call
spawn_off_label:
    M(X, Y);
    F(Y, Acc0, Acc1);
    lc_free_slot(LC, LCS);

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 (1.00, 0.97)	19.50 (1.00, 1.21)	16.07 (1.00, 1.19)
par, no &	18.75 (1.03, 1.00)	23.55 (0.83, 1.00)	19.07 (0.84, 1.00)
1c, nolc, c128	18.74 (1.03, 1.00)	23.46 (0.83, 1.00)	19.30 (0.83, 0.99)
1c, nolc, c512	18.74 (1.03, 1.00)	23.43 (0.83, 1.00)	19.30 (0.83, 0.99)
1c, lc2	18.74 (1.03, 1.00)	23.54 (0.83, 1.00)	19.30 (0.83, 0.99)
1c, lc2, tr	18.74 (1.03, 1.00)	23.79 (0.82, 0.99)	n/a
2c, nolc, c128	17.82 (1.09, 1.05)	25.68 (0.76, 0.92)	19.25 (0.83, 0.99)
2c, nolc, c512	9.60 (2.02, 1.95)	20.34 (0.96, 1.16)	18.54 (0.87, 1.03)
2c, lc2	9.69 (2.00, 1.94)	14.14 (1.38, 1.67)	9.96 (1.61, 1.91)
2c, lc2, tr	9.78 (1.98, 1.92)	14.04 (1.39, 1.68)	n/a
4c, nolc, c128	8.35 (2.32, 2.25)	26.93 (0.72, 0.87)	18.91 (0.85, 1.01)
4c, nolc, c512	4.84 (4.01, 3.88)	14.12 (1.38, 1.67)	16.83 (0.95, 1.13)
4c, lc2	4.74 (4.09, 3.96)	9.35 (2.09, 2.52)	4.98 (3.23, 3.83)
4c, lc2, tr	4.76 (4.07, 3.94)	9.41 (2.07, 2.50)	n/a

Conclusion

- We're able to find and exploit profitable parallelism in small programs.
- The analysis explores only the parts of the call graph that might be profitably parallelised.
- Our novel overlap analysis allows us to estimate how dependencies affect parallel execution.
- Our loop control transformation eliminates excessive memory usage and maintain tail recursion
- Several modifications to the runtime system have improved efficiently.
- We have adapted the ThreadScope parallel profile visualisation system for use with Mercury (not shown).

Thank you

Old engine wakeup code

When a new context is created or an existing one becomes runnable and an engine is sleeping, the engine is woken up.

If the context must be executed on a particular engine (because foreign code needs to use the C-stack owned by that engine) then all the engines are woken up so that the correct one can execute the context.

Engines sleep using a POSIX condition variable associated with the runqueue lock. Sleeping engines wakeup periodically to attempt to steal sparks.

New engine wakeup code

In the new system each engine has a semaphore, Engines wait on the semaphore when they are idle.

When a context becomes runnable and an engine is sleeping the runtime system wakes the engine by posting to the semaphore. This prevents races that could occur in the previous system.

Individual engines can be targeted specifically, so if a context has only one valid engine, then only that engine will be woken.

We can also pass contexts directly to engines, avoiding the runqueue and synchronisation in many cases.

When a spark is created a sleeping engine is woken and told which spark queue contains a spark it can execute.

The hew code is much more responsive.

Choosing how to parallelise

```
g1, g2, g3
g1 & (g2, g3)
(g1, g2) & g3
g1 & g2 & g3
```

Each of these is a parallel conjunction of sequential conjunctions, with some of the conjunctions having only one conjunct.

If there is a g4, you can (a) execute it in parallel with all the other parallel conjuncts, or (b) execute it in sequence with the goals in the last sequential conjunction.

There are thus 2^{N-1} ways to parallelise a conjunction of N goals.

If you allow goals to be reordered, the search space would become larger still.

Even simple code can have many conjuncts.

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

Flattening the above expression gives 12 small goals, each executing one primitive operation:

```
V1 = 0              V5 = 4              V9 = sqrt(V8)
V2 = V1 - B         V6 = V5 * A         V10 = V2 + V9
V3 = 2              V7 = V6 * C         V11 = V3 * A
V4 = pow(B, V3)     V8 = V4 - V7        X = V9 / V11
```

Primitive goals are not worth spawning off. Nonetheless, they can appear between goals that should be parallelised against one another, greatly increasing the value of N.

Reducing the search space.

Currently we do two things to reduce the size of the search space from 2^{N-1}:

- Remove whole subtrees of the search tree that are worse than the current best solution (a variant of "branch and bound").
- During search we always follow the most promising-looking branch before backtracking to the alternative branch.
- If the search is still taking to long, then switch to a greedy search that is approximately linear.

This allows us to fully explore the search space when it is small, while saving time by exploring only part of the search space when it is large.

Expensive goals in different conjunctions

The call to typecheck and the call to typecheck_preds are expensive enough to be worth parallelising. But the if-then-else that contains the call to typecheck has a typical cost 1/10th of the cost of typecheck. It is not worth parallelising the if-then-else against typecheck_preds.

```
   typecheck_preds([], [], ...).
   typecheck_preds([Pred0 | Preds0], [Pred | Preds], ...) :-
       ( if should_typecheck(Pred0) then
 10        typecheck(Pred0, Pred, ...)
       else
 90        Pred = Pred0
       ),
100    typecheck_preds(Preds0, Preds, ...).
```

Push later goals into earlier compound goals

We can push the call to typecheck_preds into the if-then-else and parallelise only the then-part:

```
typecheck_preds([], [], ...).
typecheck_preds([Pred0 | Preds0], [Pred | Preds], ...) :-
    ( if should_typecheck(Pred0) then
        typecheck(Pred0, Pred, ...) &
        typecheck_preds(Preds0, Preds, ...)
    else
        Pred = Pred0,
        typecheck_preds(Preds0, Preds, ...)
    ).
```

Our analysis can perform this transformation as part of deciding whether this parallelisation is worthwhile.