# Estimating the overlap between dependent computations

Paul Bone, Zoltan Somogyi and Peter Schachte

The University of Melbourne

National ICT Australia

# Motivation

Multicore systems are ubiquitous, but parallel programming is hard.

**Pure** declarative languages do not allow side-effects; all effects must be declared, including IO actions. Therefore, it is trivial to determine if parallel execution is safe.

However, working out how to optimally parallelize a program is much more difficult.

- Programmers rarely understand the performance of their programs.
- They find it difficult to account for the overheads of parallelization, such as the costs of spawning a task and sharing data.
- Parallelism in one part of the program can affect how many processors are available for another part of the program.
- If the program changes in the future, the programmer may have to re-parallelise it.

# About Mercury

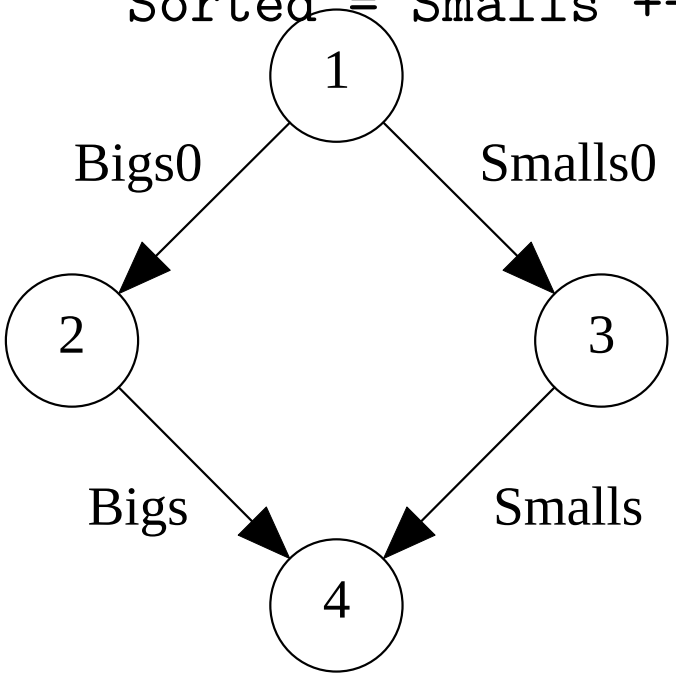
- Mercury is a **pure** logic/functional language designed to support the creation of large, reliable, efficient programs.
- It has a syntax similar to Prolog's, however the operational semantics are very different.
- It is strongly typed using a Hindley Milner type system.
- It also has mode and determinism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

# 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.

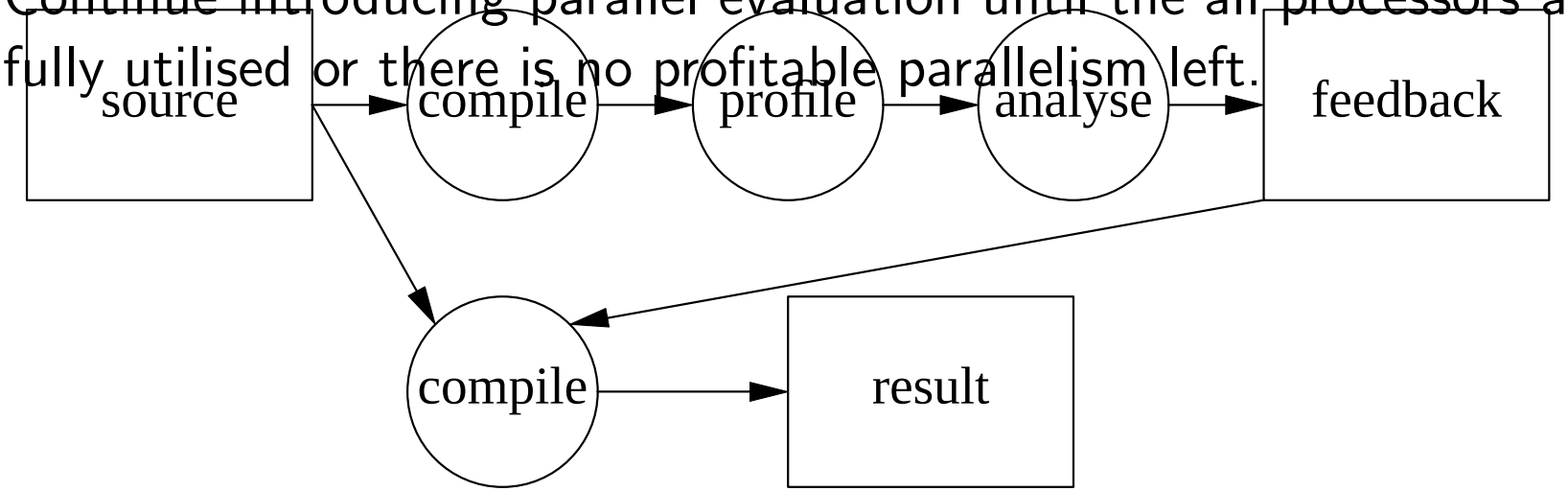
The above parallelization creates $N$ parallel tasks for a list of length $N$. Most of these tasks are trivial and therefore the overheads of managing them slow the program down.

Programmers rarely understand the performance of their programs, even when they think they do. It is best if the compiler optimizes the program on behalf of the programmer.

The same is true for parallelization — which is just another optimization.

# 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 parallelized *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.



# Finding parallelization 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`.
- Analyze 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.

# How would you parallelize this?

```
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).
```

During parallel execution, a task will block if a variable it needs is not available when it needs it.

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

Can `map_foldl` be profitably parallelized despite these dependencies, and if yes, how?

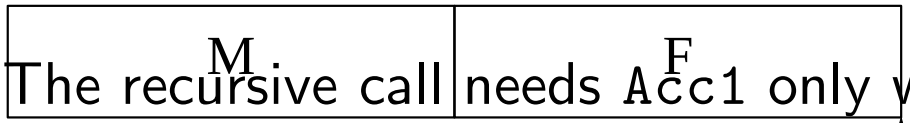
# Parallelizing `map_foldl`

`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.

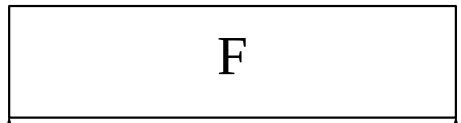
```
map_foldl(_, _, [], Acc, Acc).
map_foldl(M, F, [X | Xs], Acc0, Acc) :-
    (
        M(X, Y),
        F(Y, Acc0, Acc1)
    ) &
    map_foldl(M, F, Xs, Acc1, Acc).
```

# `map_foldl` overlap

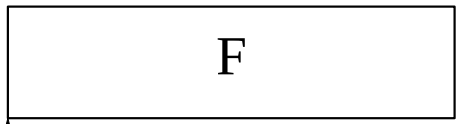



The recursive call needs `Acc1` only when it calls `F`. The calls to `M` can be executed in parallel.

Acc1


Acc1 Acc1'

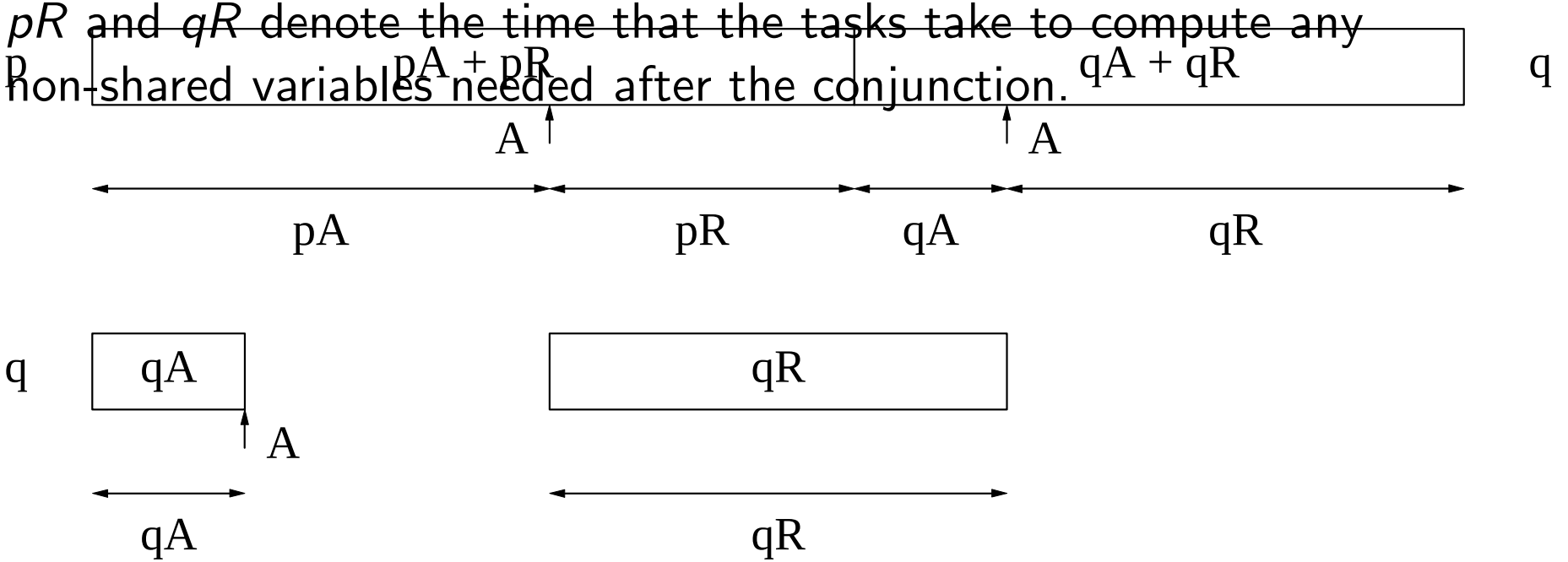

Acc1'

# Simple overlap example

Tasks p and q have one shared variable.

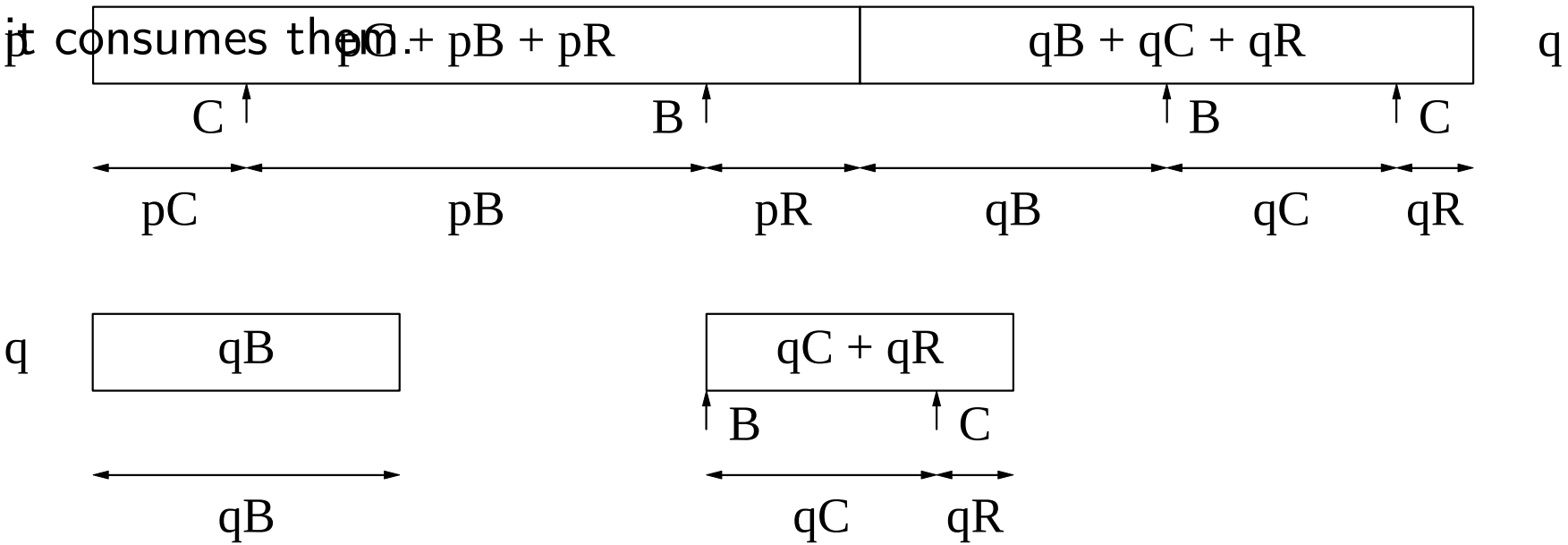
We conceptually split each task split into sections, each section ended by the production or consumption of a shared variable.

*pR* and *qR* denote the time that the tasks take to compute any non-shared variables needed after the conjunction.



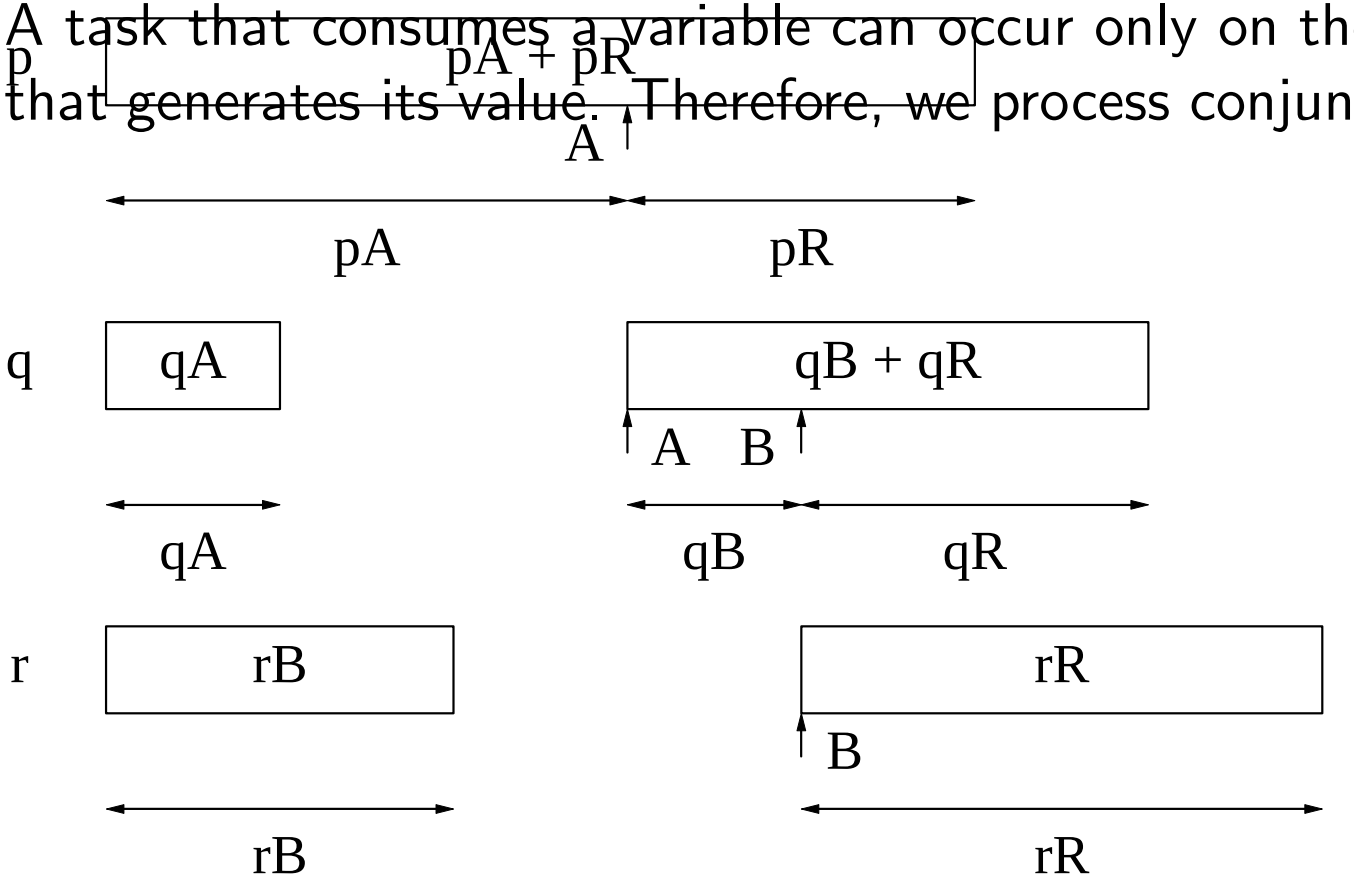
# Overlap with more than one dependency

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 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.

# Overlap algorithm — Loop over variables

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

# Results on some small programs

| Program | Seq | 1 CPU | 2 CPUs | 3 CPUs | 4 CPUs |
|---|---|---|---|---|---|
| matrixmult | 11.0 | 14.6 (0.75) | 7.5 (1.47) | 6.2 (1.83) | 5.2 (2.12) |
| raytracer | 22.7 | 25.1 (0.90) | 16.0 (1.42) | 11.2 (2.03) | 9.4 (2.42) |
| mandelbrot | 33.4 | 35.6 (0.94) | 17.9 (1.87) | 12.1 (2.76) | 9.1 (3.67) |

Parallel code needs to use a machine register to point to thread-specific data, so enabling parallel execution but not using it leads to slowdowns.

Matrixmult has one memory store for each FP multiply/add pair. Its speedup is limited by memory bus bandwidth, which it saturates relatively quickly.

Raytracer generates many intermediate data structures. The GC system consumes 40% of the execution time in stop-the-world collections when using 4 Mercury threads and 4 GC threads. When using 1 Mercury thread and 4 GC threads it uses only 5% of the program's runtime.

# Results on the Mercury compiler

- There are 53 conjunctions in the compiler with two or more expensive conjuncts.
- 52 of these are dependent conjunctions.
- 31 of these have a predicted speedup of greater than 1% (the default speedup threshold).
- Therefore, our analysis can prevent the parallelization of 22 conjunctions that are not profitable.

Unfortunately, many parts of the compiler must be executed sequentially. Due to Amdahl's law, this limits the overall speedup of the compiler.

# Progress to date

- Our analysis is able to find profitable parallelism in small programs and generate advice for the compiler.
- The analysis explores only the parts of the call graph that might be profitably parallelized.
- Our novel overlap analysis allows us to estimate how dependencies affect parallel execution.
- The compiler can act on this advice, and can profitably parallelize small programs.

Not shown in this presentation:

- We can rearrange some computations to make it easier to take advantage of parallelism.
- We can efficiently search a large space of possible parallelizations.

# Further work

- Account for barriers to effective parallelism, including garbage collection and memory bandwidth limits.
- Build an *advice* system that informs programmers why something they think can be profitably parallelized in fact cannot be.
- Support parallelization as a specialization.

Questions?

# Choosing how to parallelize

```
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 parallelize 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 parallelized 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 parallelizing. 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 parallelizing 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, ...).
```

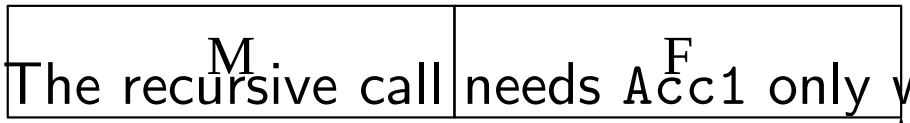
# Push later goals into earlier compound goals

We can push the call to typecheck_preds into the if-then-else and parallelize 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, ...)
    ).
```

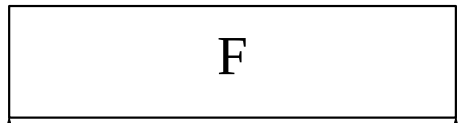
Our analysis can perform this transformation as part of deciding whether this parallelization is worthwhile.

# `map_foldl` overlap

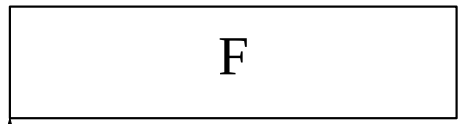



The recursive call needs `Acc1` only when it calls `F`. The calls to `M` can be executed in parallel.

Acc1

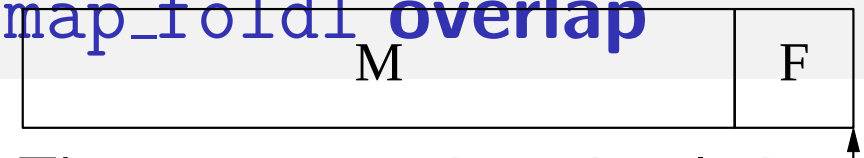




Acc1 Acc1'

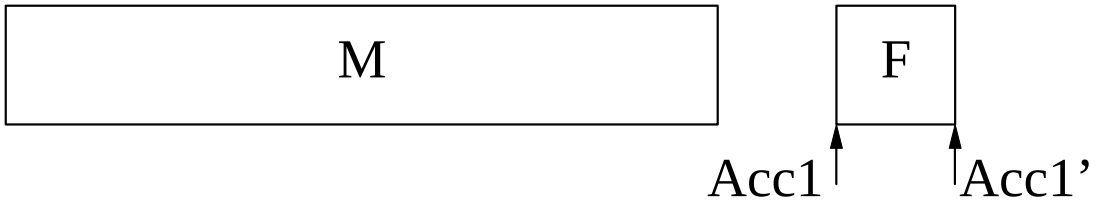



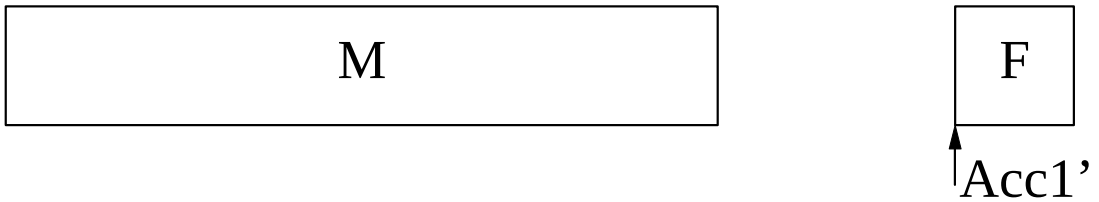

Acc1'

# `map_foldl` overlap



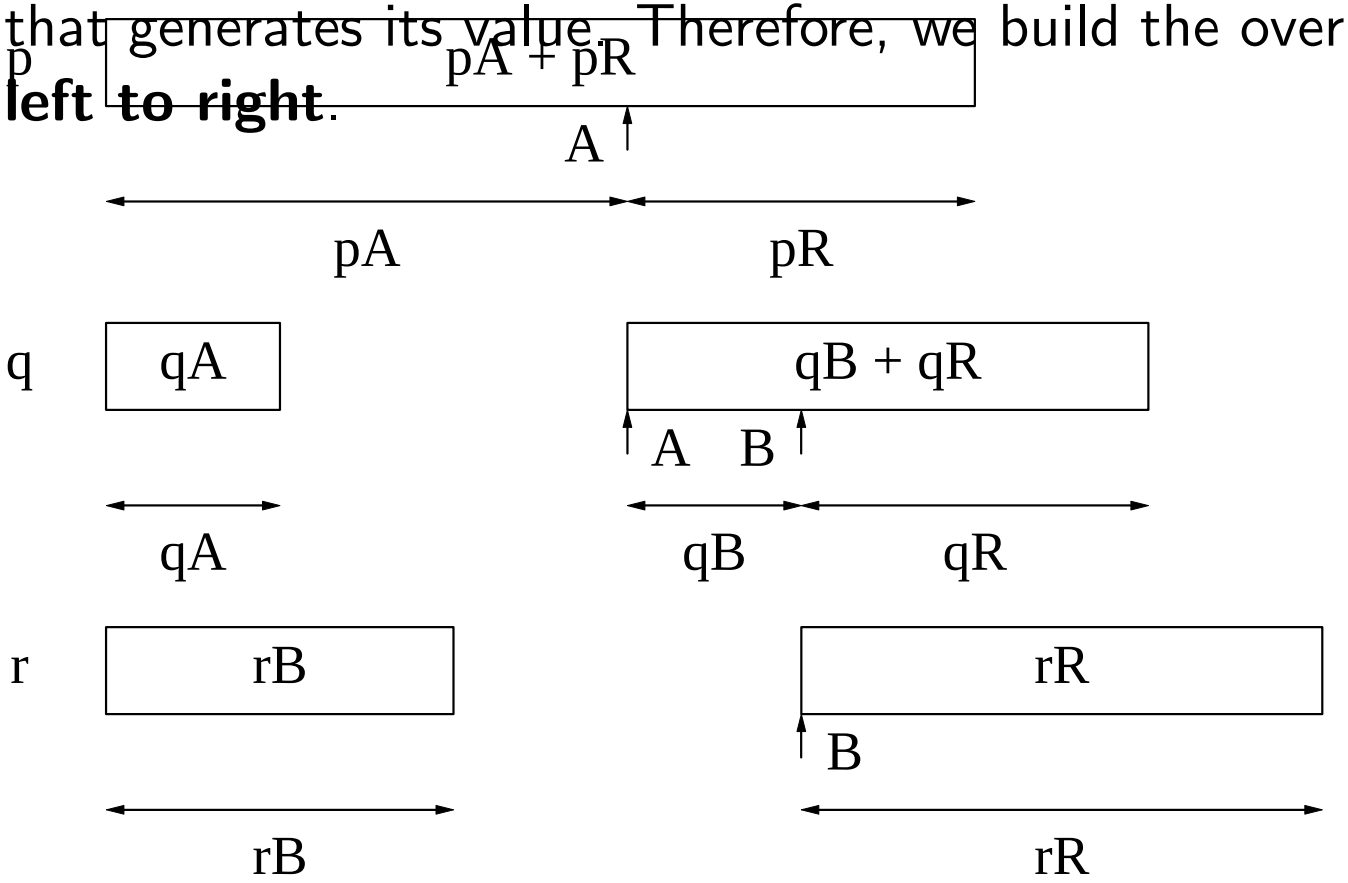

The more expensive `M` is relative to `F`, the bigger the overall speedup.



# 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 build the overlap information from **left ~~to right~~**.



# Overlap of more than two tasks

In this example, the rightmost task consumes a variable produced by the leftmost task.