Automatic Parallelisation for Mercury

Paul Bone

pbone@csse.unimelb.edu.au
Department of Computer Science and Software Engineering
The University of Melbourne

December 6th, 2010

About my eye sight

I have roughly 1/8 of normal vision, so I probably won't see any raised hands.

If you have a question:

During the presentation Speak up / call out.

After the presentation Raise your hand, a facilitator will help me select questions.

The problem

Multicore systems are ubiquitous, but parallel programming is hard.

- Thread synchronisation is very hard to do correctly.
- Critical sections are not composable.
- Working out *how* to parallelise a program is usually difficult.
- If the program changes in the future, the programmer may have to re-parallelise it.

This makes parallel programming time consuming and expensive. Yet programmers *have* to use parallelism to achieve optimal performance on modern computer systems.

NICTA

THE UNIVERSITY OF MELBOURNE

Side effects

```
int main(int argc, char *argv[]) {
    printf("Hello ");
    printf("world!\n");
    return 0;
}
```

`printf` has the effect of writing to standard output. Because this effect is implicit (not reflected in the arguments), we call this a *side* effect.

When you are looking at unfamiliar code, it is often impossible to tell whether a call has a side effect without looking at its *entire call tree*.

Making all effects visible and therefore easier to understand would make both parallelization and debugging *much* easier.

NICTA

THE UNIVERSITY OF MELBOURNE

Mercury and Effects

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).
```

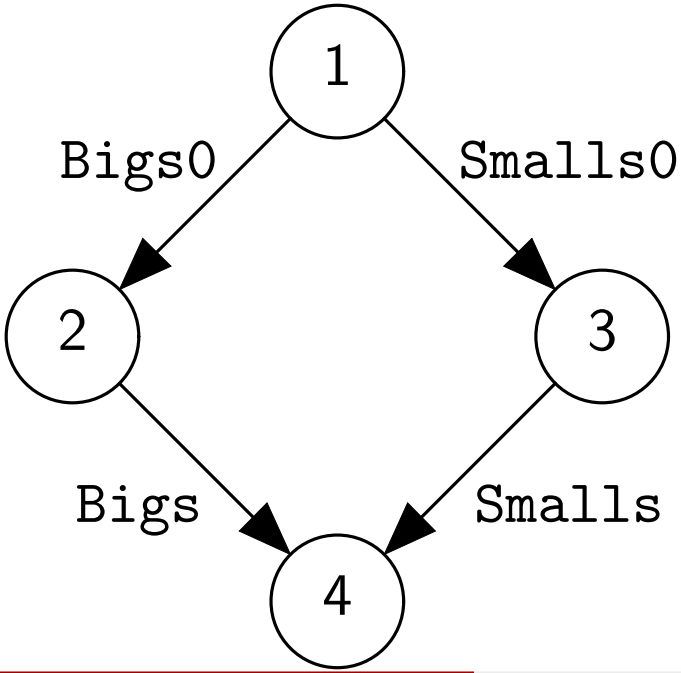
The I/O state represents the state of the world outside of this process. Mercury ensures that only one version is alive at any given time.

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`.

Effect Dependencies

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



- Steps 2 and 3 are independent.
- This is easy to prove because there are never any *side* effects.
- The compiler may execute them in parallel.

NICTA

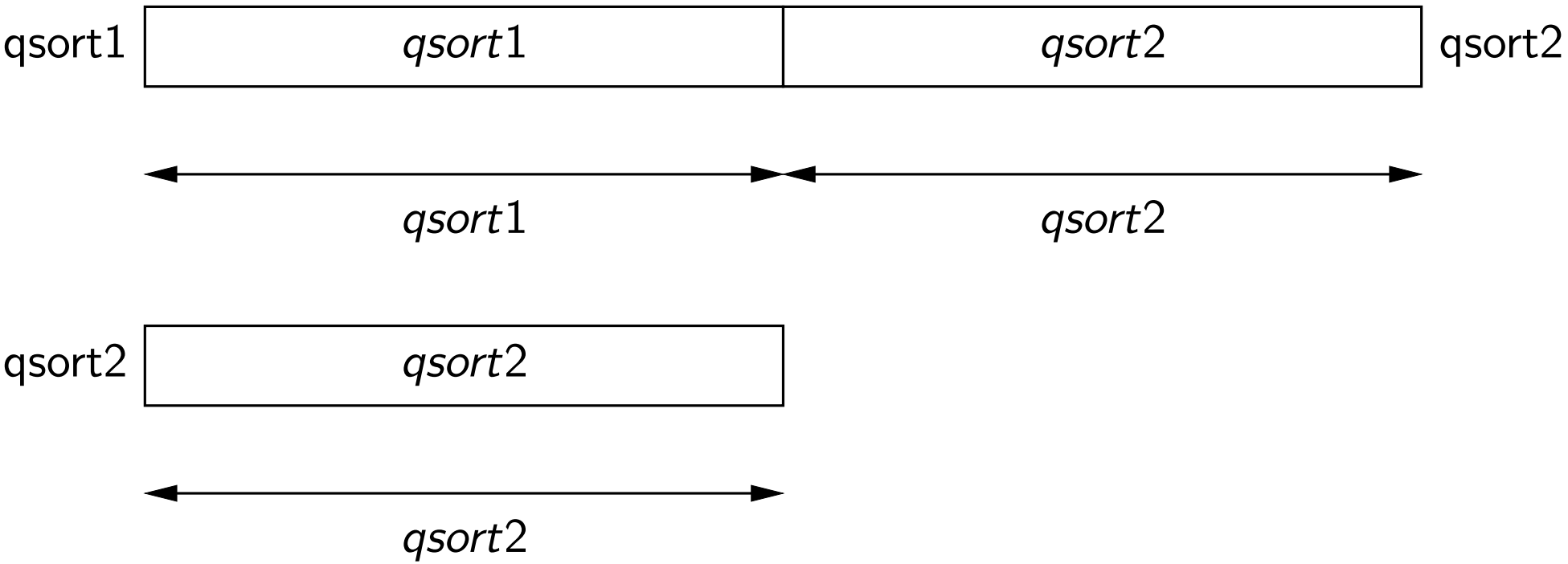
THE UNIVERSITY OF MELBOURNE

Explicit parallelism

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

The comma separates goals within a conjunction. The ampersand has the same semantics, except that the conjuncts are executed in parallel.

Parallelism overlap



Quicksort can be parallelised easily and reasonably effectively. However, most code is much harder to parallelise, due to dependencies.

NICTA

THE UNIVERSITY OF MELBOURNE

map_foldl

```
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 parallelised despite these dependencies, and if yes, how?

NICTA

THE UNIVERSITY OF MELBOURNE

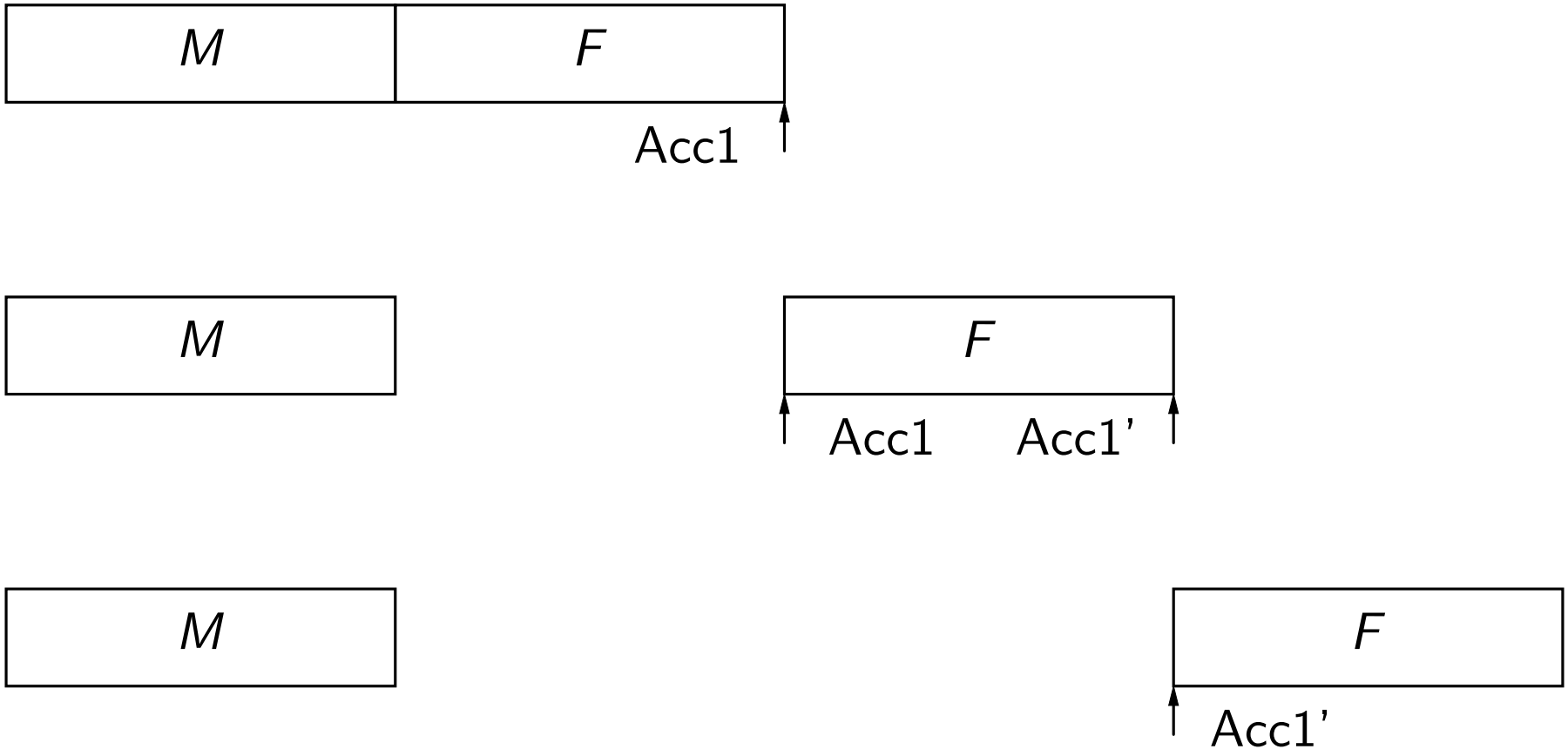
Parallelisation of `map_foldl`

`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.

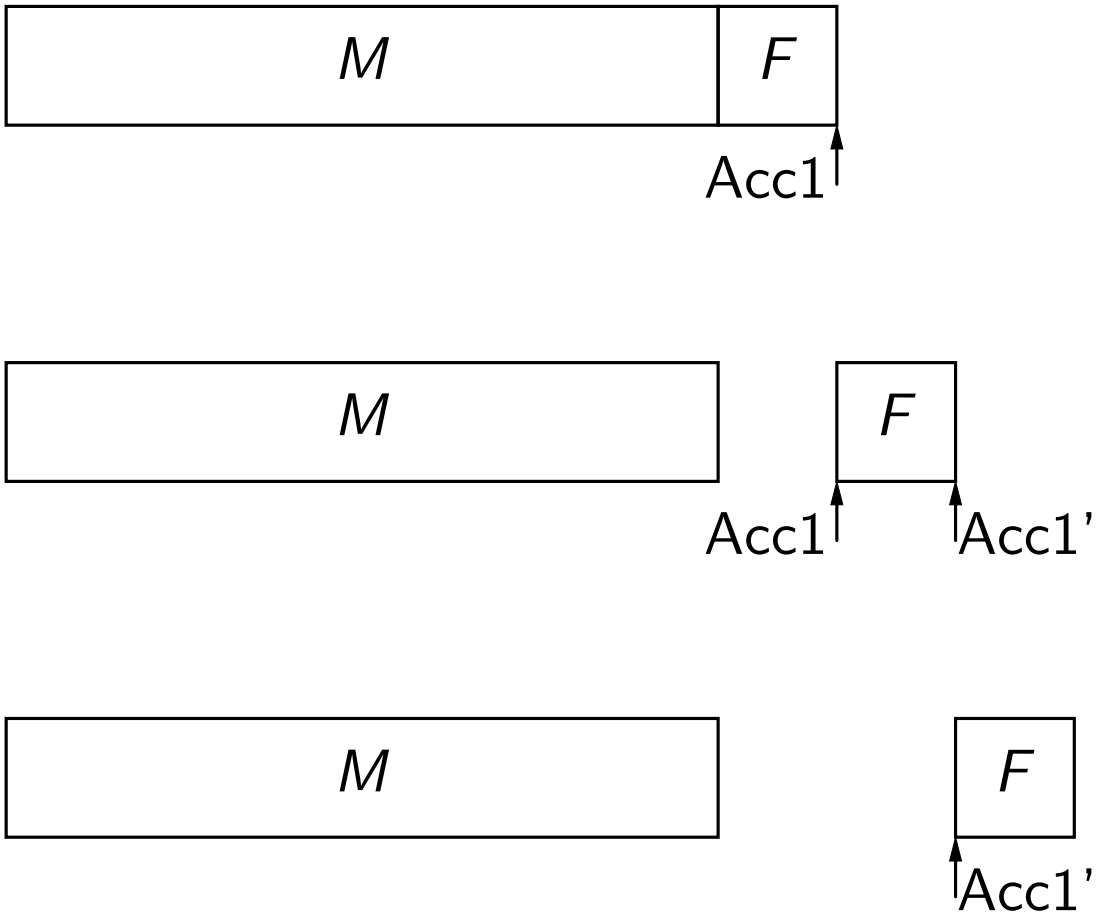
```
map_foldl(_, _, [], Acc, Acc).
map_foldl(M, F, [X | Xs], Acc0, Acc) :-
    (
        M(X, Y),
        F(Y, Acc0, Acc1)
    &
        map_foldl(M, F, Xs, Acc1, Acc)
    ).
```

map_foldl overlap



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

map_foldl overlap



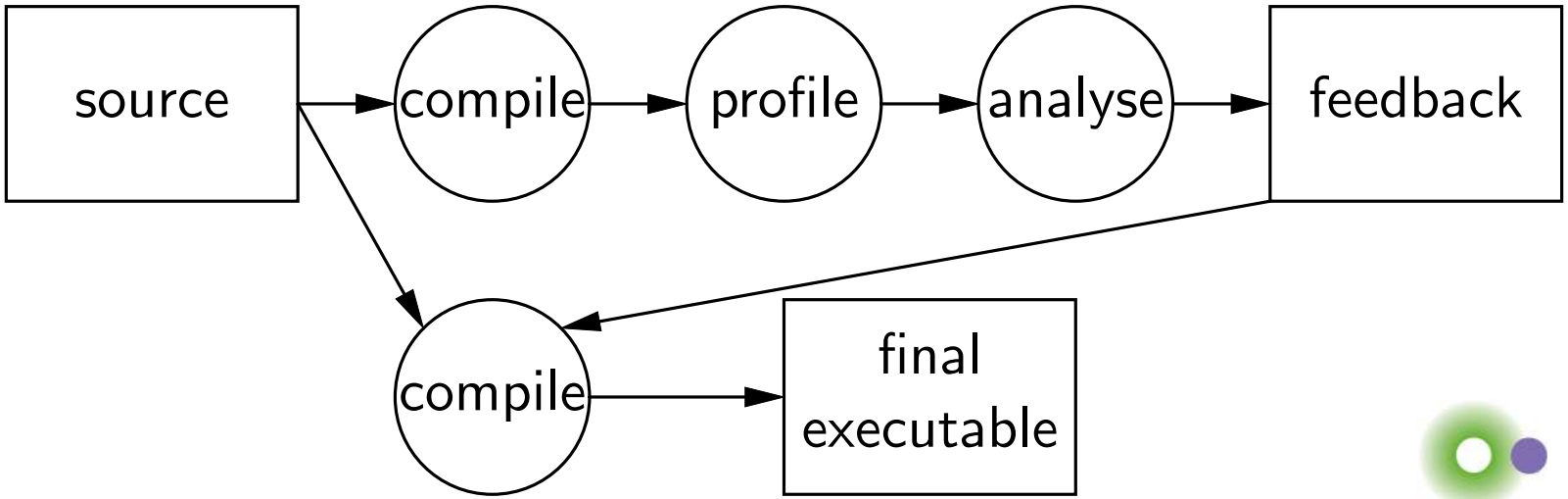
The more expensive `M` is relative to `F`, the bigger the speedup.

Profiler feedback

We need to know:

- the costs of calls through each call site, and
- the times at which variables are produced and consumed.

We extended the Mercury profiler to give us this information, to allow programs to be automatically parallelised like this:

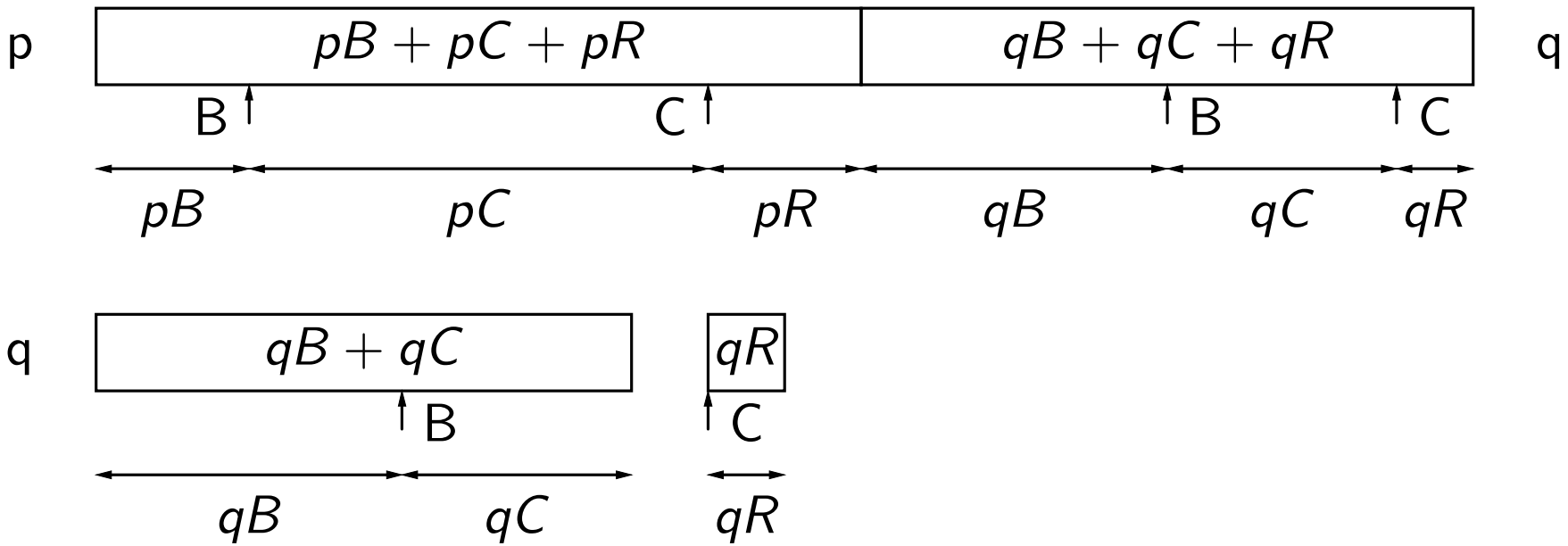


NICTA

THE UNIVERSITY OF MELBOURNE

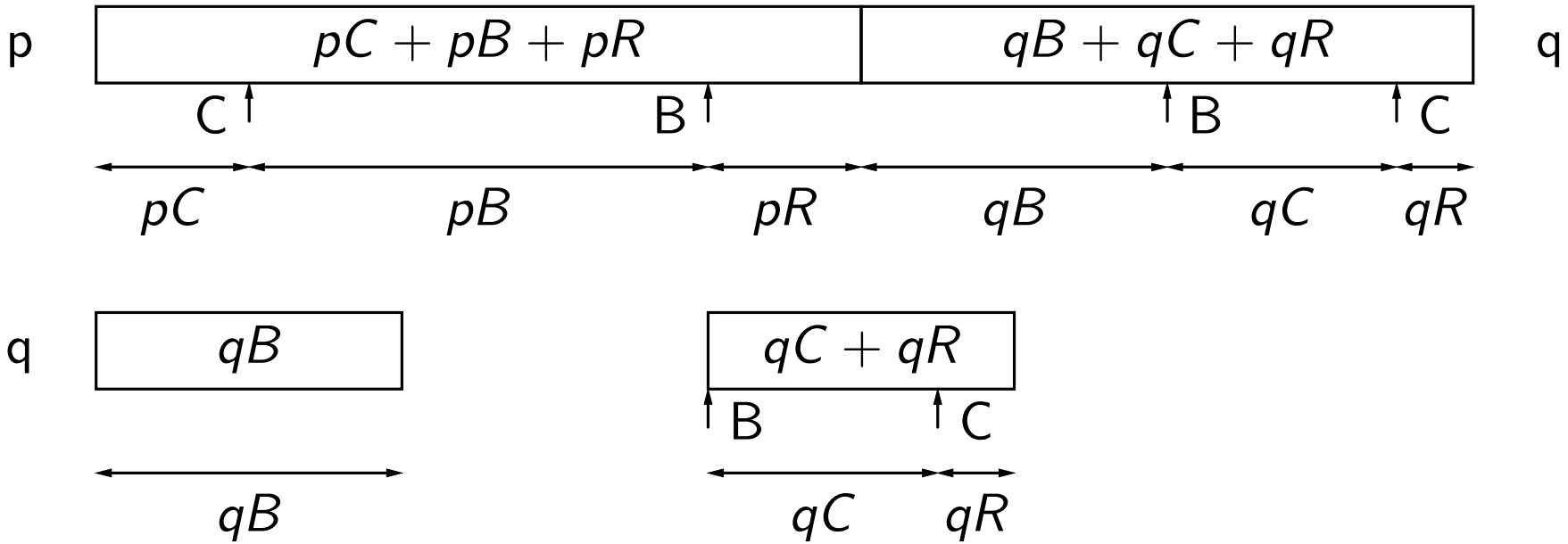
Overlap with more than one dependency

We calculate the execution time of q by iterating over the variables it consumes in **the order that it consumes them**.



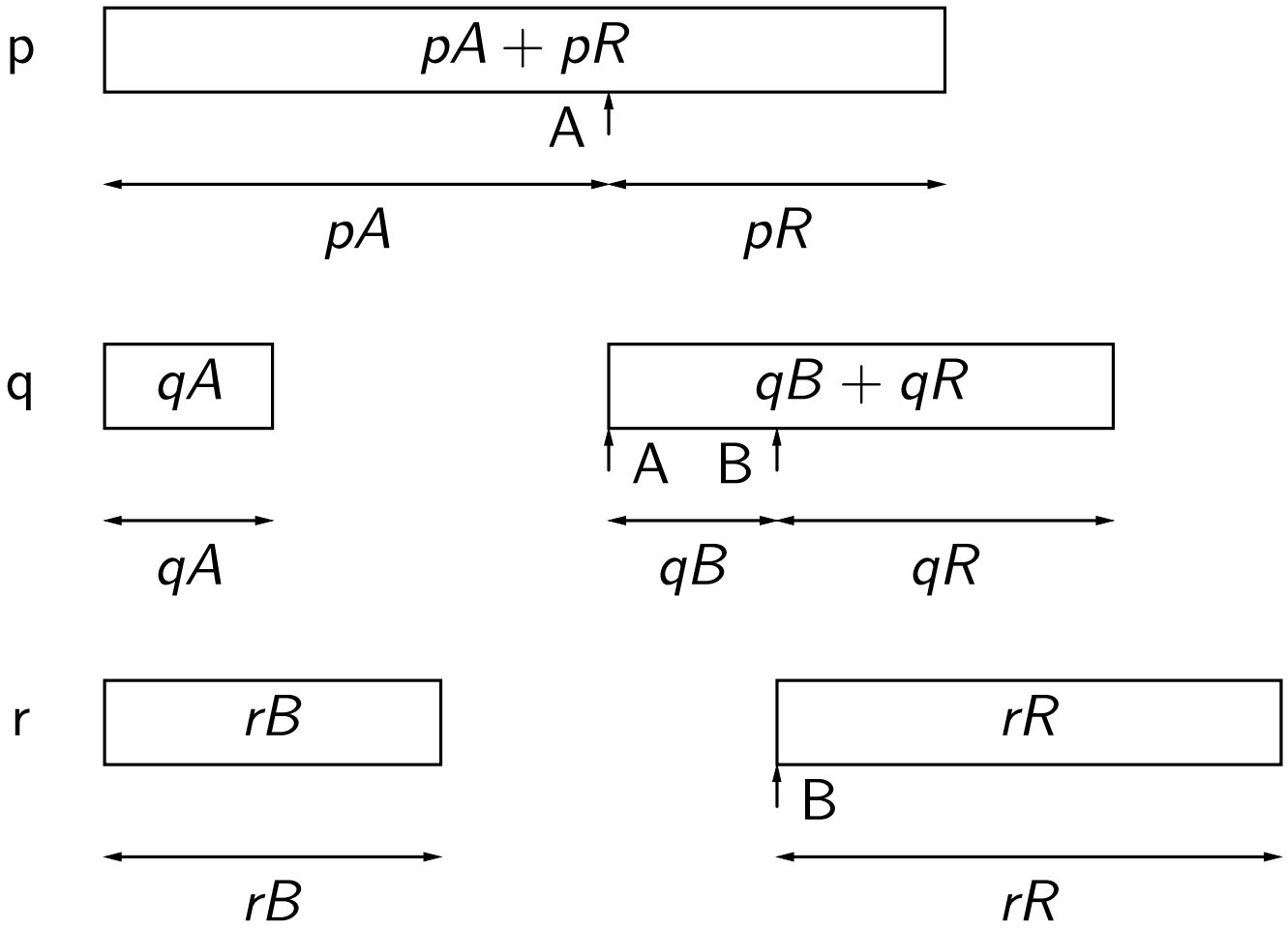
Overlap with more than one dependency

The order of consumption may differ from the order of production.



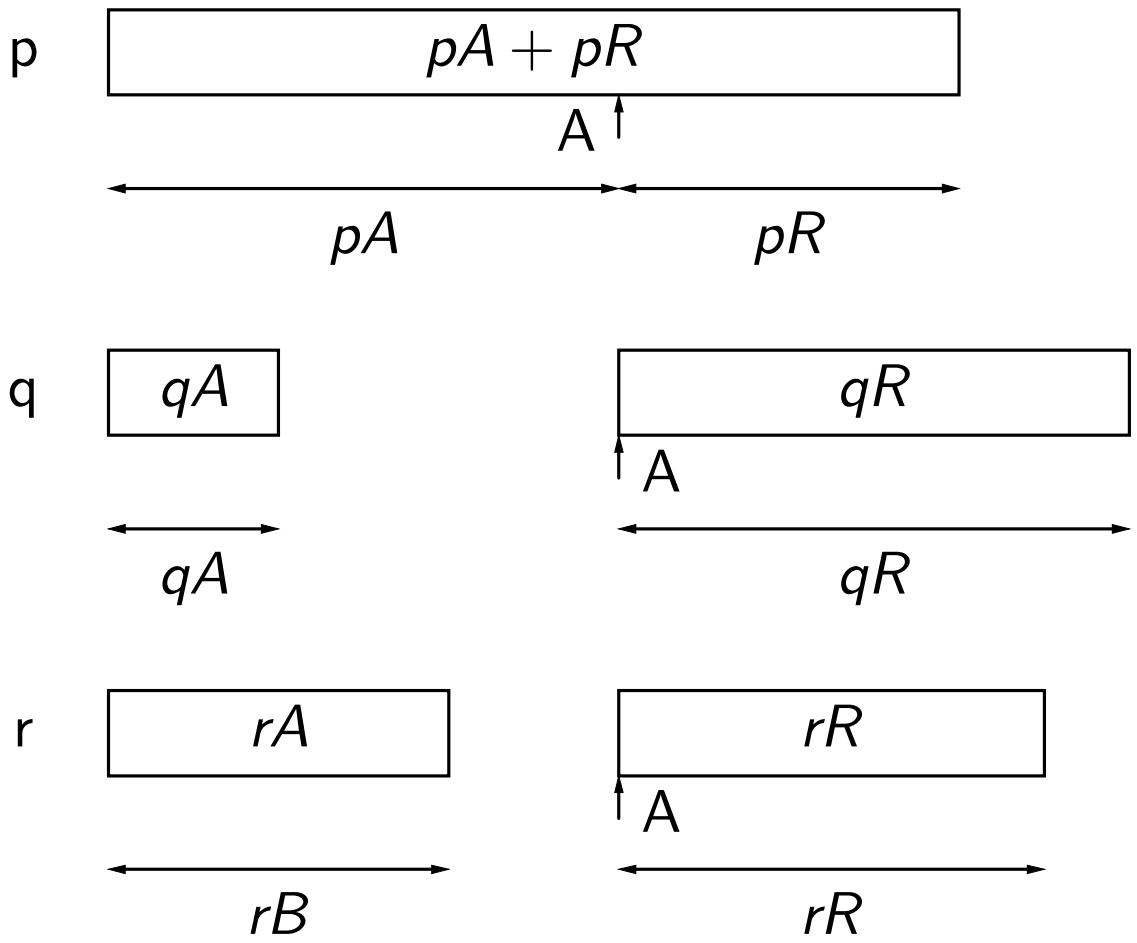
Overlap of more than two tasks

A task that consumes a variable must be *after* 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.



NICTA

THE UNIVERSITY OF MELBOURNE

How to parallelise

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

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

If there is a g4, you can (a) execute it after all the previous sequential conjuncts, or (b) put it as a new goal into the last parallel 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.

NICTA

THE UNIVERSITY OF MELBOURNE

How to parallelise

```
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.

How to parallelise

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")
- If the search is still taking to long, then switch to a greedy search that is approximately linear.

NICTA

THE UNIVERSITY OF MELBOURNE

Where to parallelise

We should only explore the parts of the program that might contain profitable parallelism.

We therefore start at the entry point of the program, and do a depth-first search of the call graph until either:

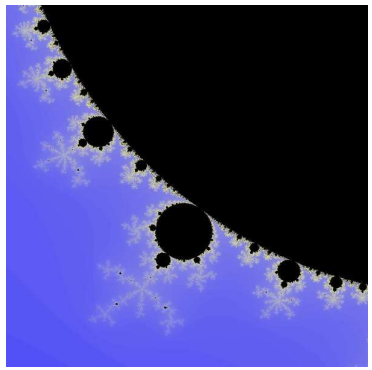
- the current node's execution time is too small to contain profitable parallelism, or
- we have already identified enough parallelism along this branch to keep all the CPUs busy.

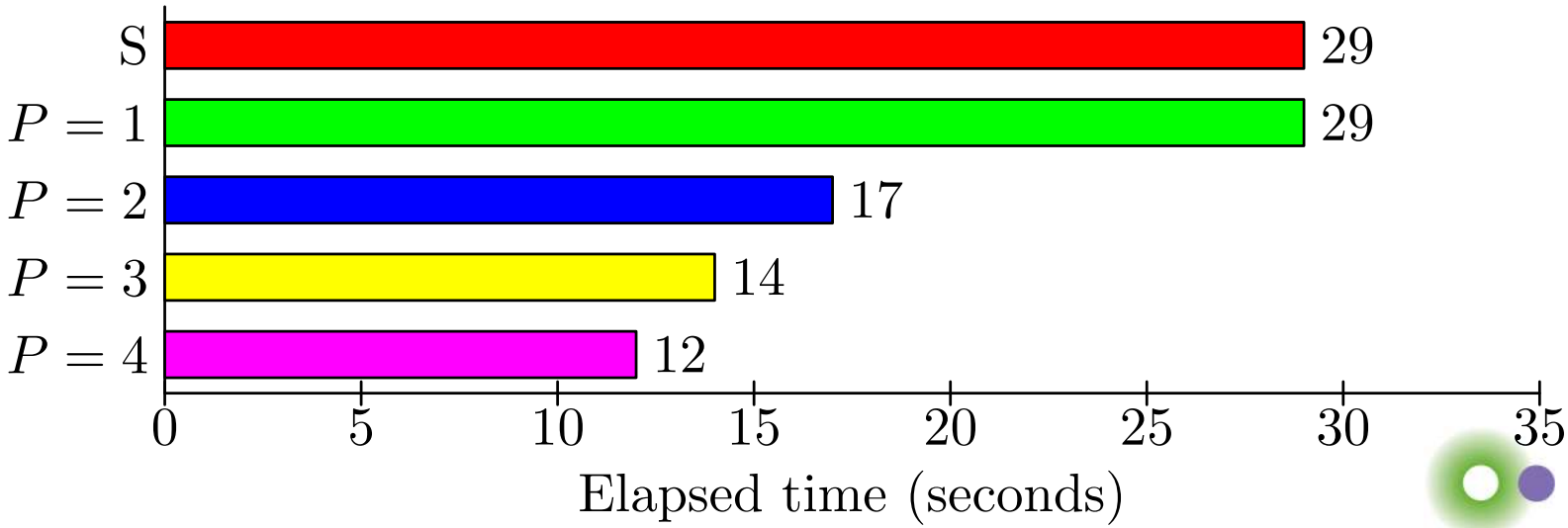
NICTA

THE UNIVERSITY OF MELBOURNE

Benchmarks — Mandelbrot image generator

- dependant parallelism using `map_foldl`.
- 280 LoC.
- Automatically parallelised.
- Light garbage collector usage.



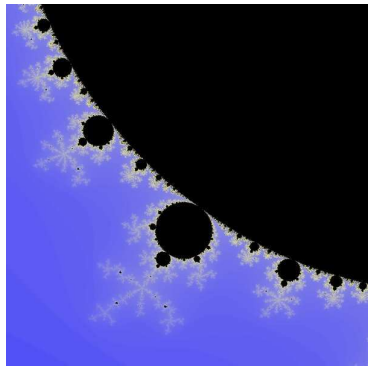


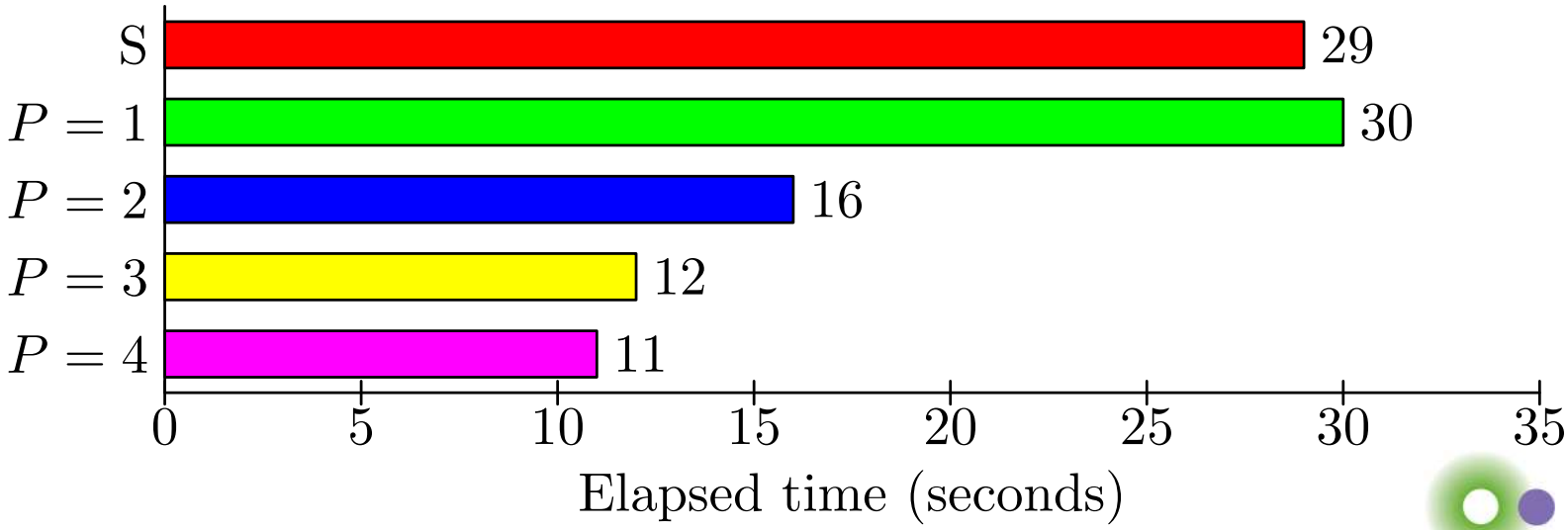
NICTA

THE UNIVERSITY OF MELBOURNE

Benchmarks — Mandelbrot image generator

- Modified so that independant parallelism is used.
- Automatically parallelised.



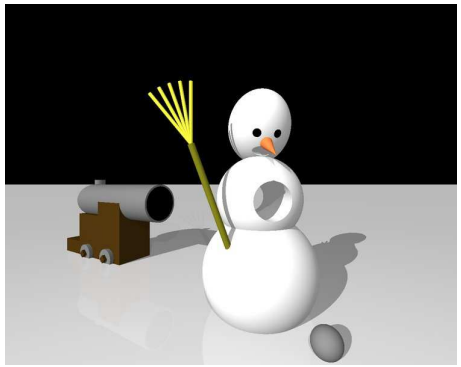


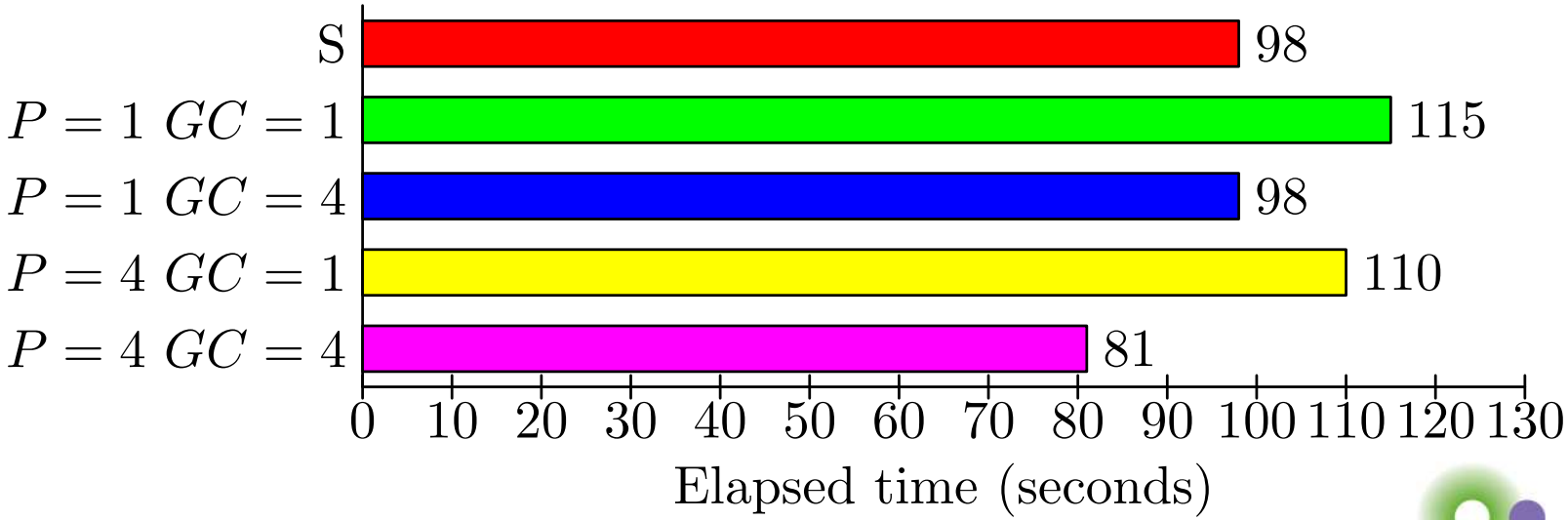
NICTA

THE UNIVERSITY OF MELBOURNE

Benchmarks — ICFP 2000 raytracer

- 6,200 LoC.
- Automatically parallelised.
- Heavy garbage collector usage.
- Code was altered to make it less stateful.



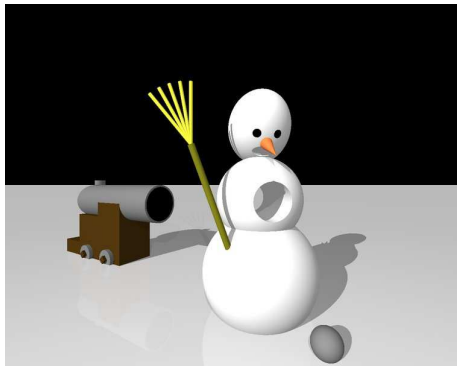


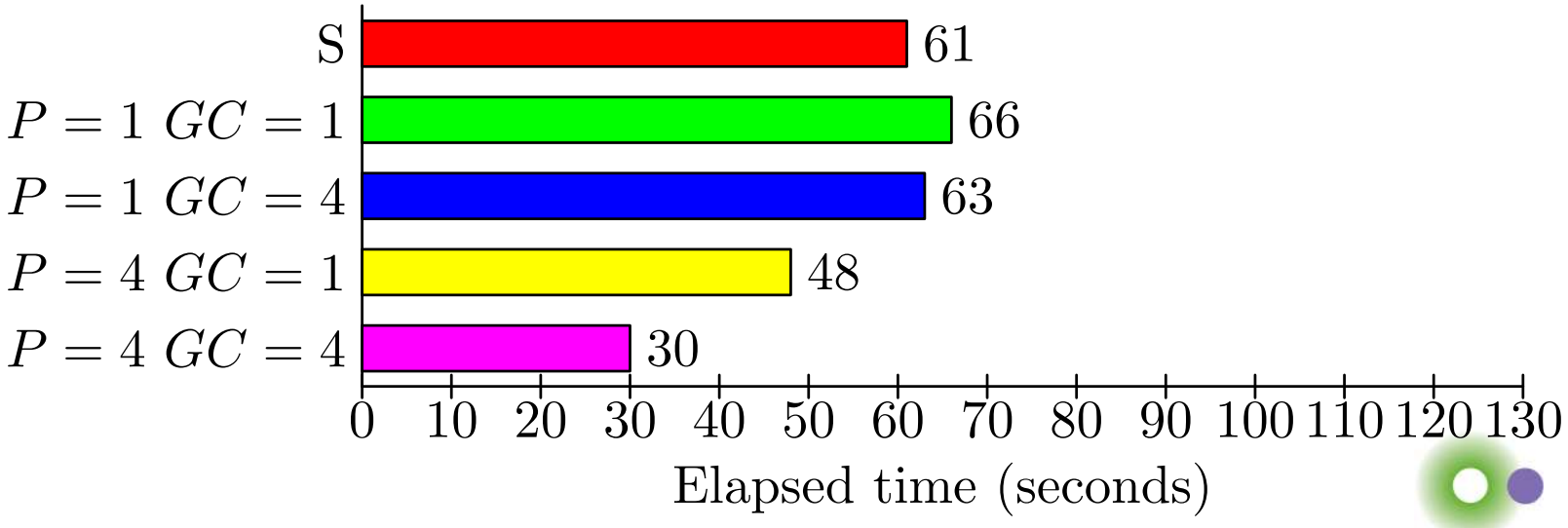
NICTA

THE UNIVERSITY OF MELBOURNE

Benchmarks — ICFP 2000 raytracer

- Increasing the initial heap size for the Boehm GC reduces the number of "stop the world" events.
- Increasing the size of the thead-local free lists reduces the contention on global locks.





NICTA

THE UNIVERSITY OF MELBOURNE

Conclusion

Progress to date:

- Can analyse program profiles, and find places where parallelism is probably profitable.
- Can explore a large search space of possible parallelisations efficiently.
- Auto-parallelisation already yields speedups for some small programs.

Future work:

- Build an *advice* system that informs programmers why something cannot be parallelised.
- Handle loops and divide-and-conquer code more intelligently.
- Test alternative ways of exploring the program's call graph.
- Account for barriers to effective parallelism, including garbage collection and memory bandwidth limits.

NICTA

THE UNIVERSITY OF MELBOURNE

Questions?

Mercury `http://www.mercury.csse.unimelb.edu.au`

State variable notation

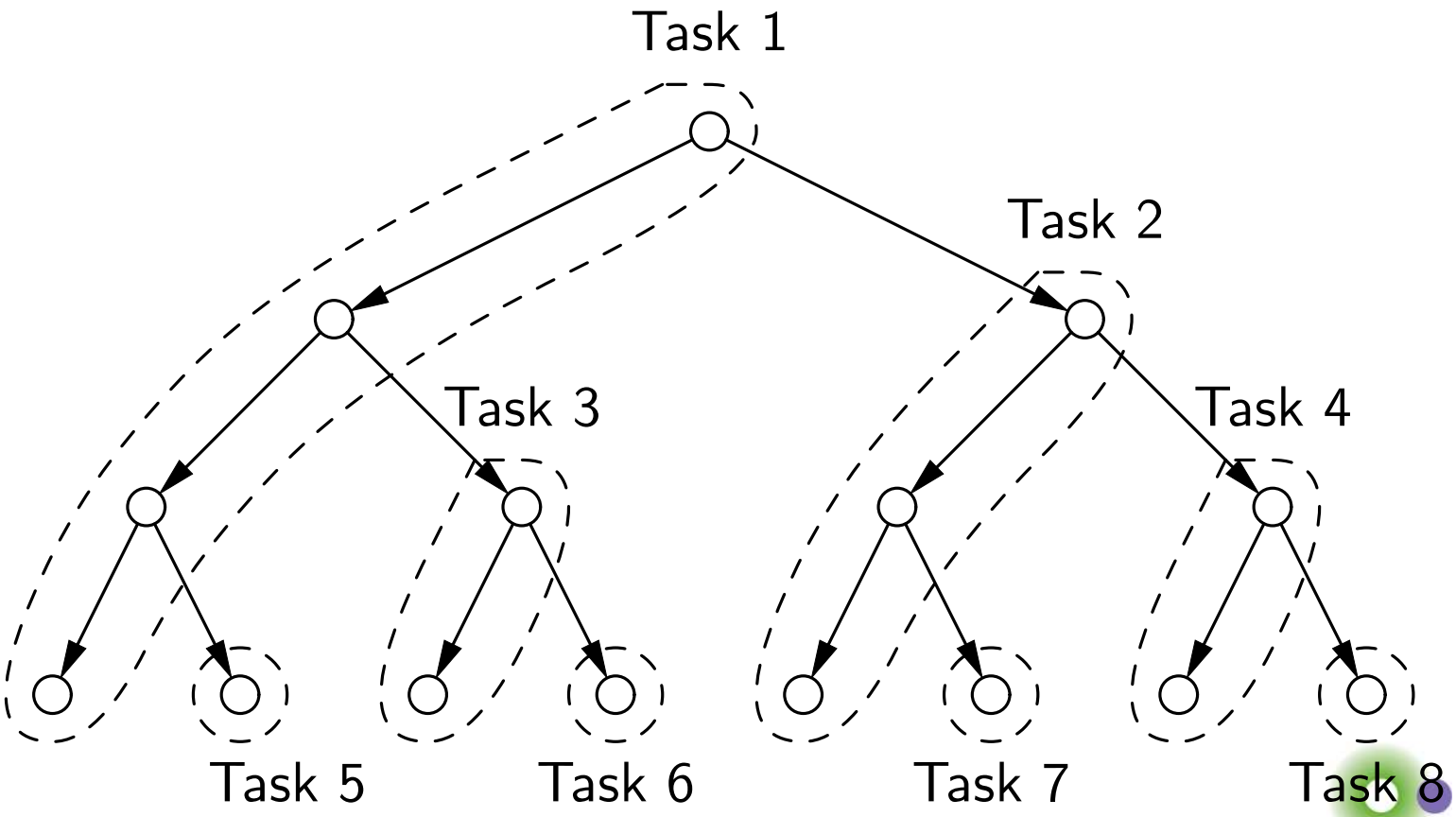
```
main(!IO) :-
    write_string("Hello ", !IO),
    write_string("world!\n", !IO).
```

`!VarName` is syntactic sugar for a pair of variables. The compiler will create as many variables as their are versions of the state they represent, and thread them through calls where `!VarName` appears.

This is not limited to the I/O state.

Divide and conquer

On average, this creates $O(N)$ small parallel tasks. This is far too many since most systems have far fewer than N cores.

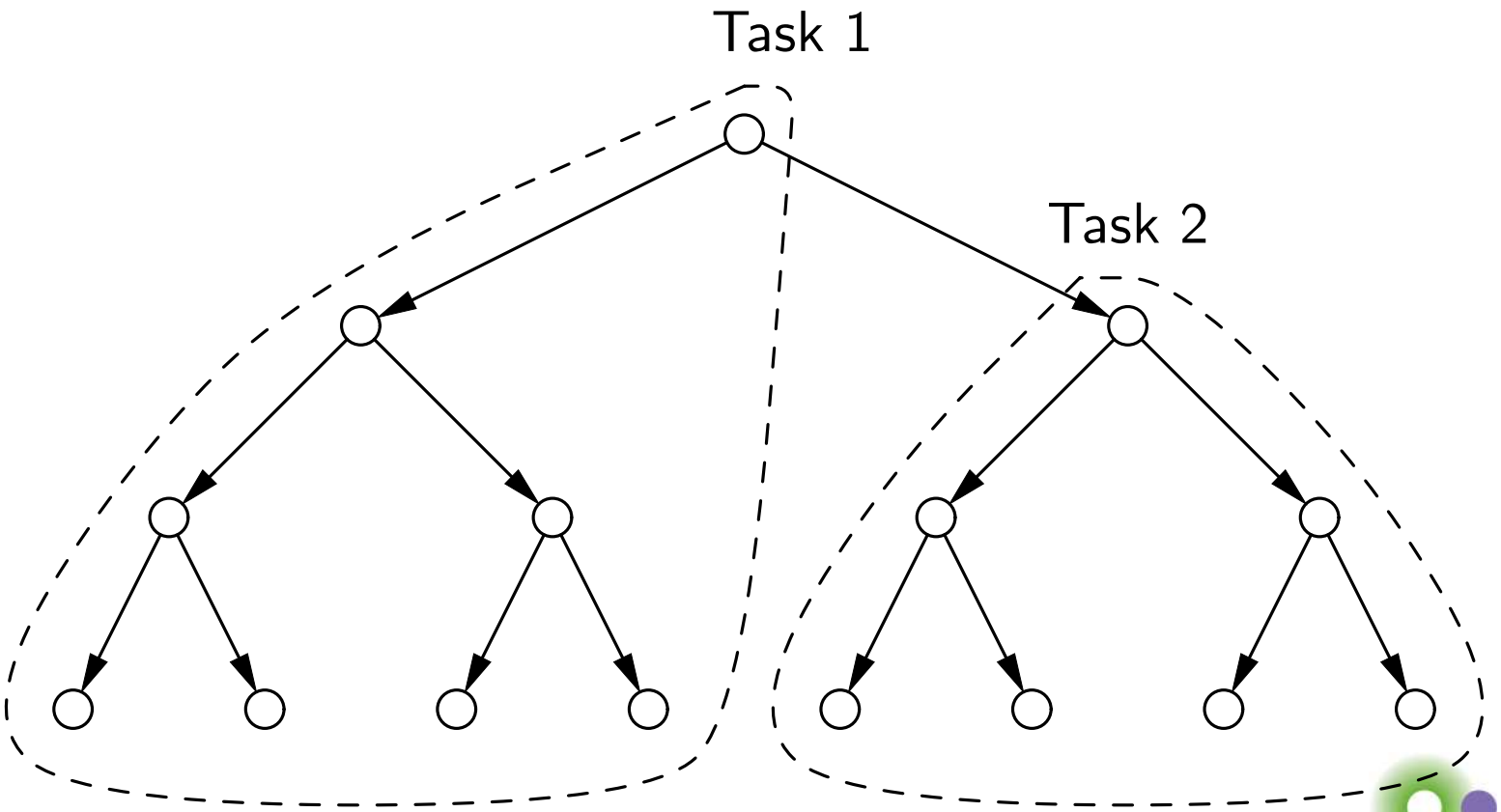


NICTA

THE UNIVERSITY OF MELBOURNE

Divide and conquer

It is much better to parallelise the first $O(log_2 P)$ levels of the tree.



NICTA

THE UNIVERSITY OF MELBOURNE