Automatic Parallelisation

Paul Bone, Zoltan Somogyi and Peter Schachte

The University of Melbourne

National ICT Australia

Multicore World

March, 2012

Motivation

Multicore systems are ubiquitous, but parallel programming is hard.

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, currupt memory and crash the program.
- Using mulitple locks in different orders, including nested critical sections can lead to deadlocks.
- Misplacing locks can lead to critical sections that are two narrow, or two wide. Possibly causing poor performance or curruption crashes (as above).

There are analyses that can be used to safely parallelise programs. For example, SSA representations can be used to track changes to local variables and aliasing analysis can track the use of heap variables.

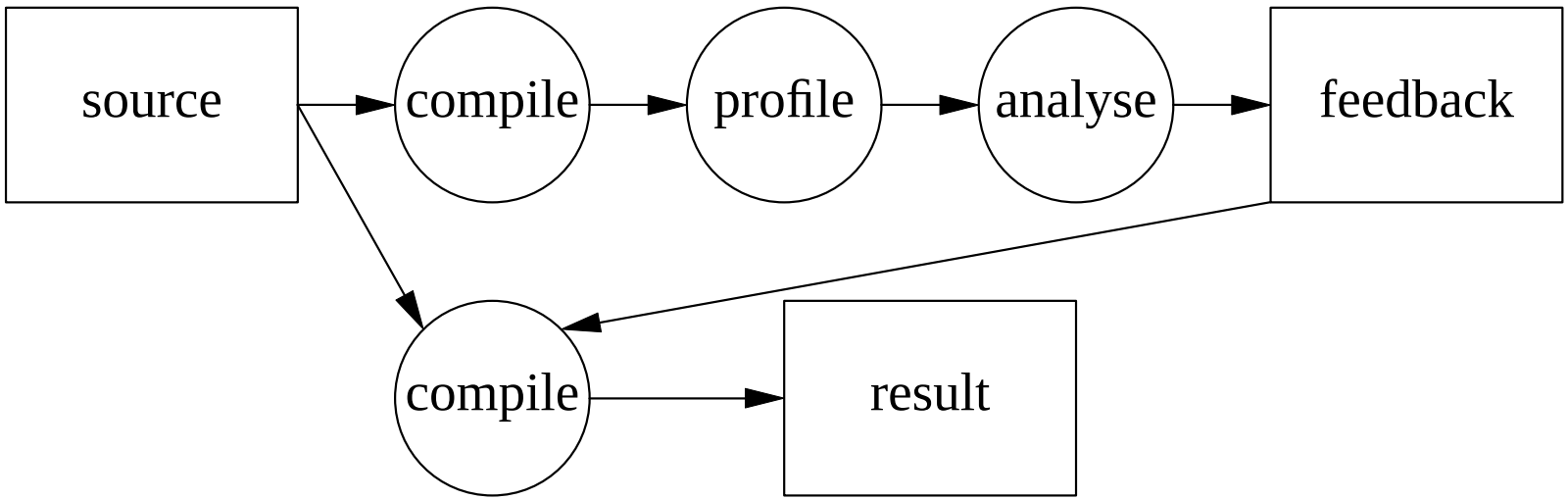
Optimal Parallelisation

Programmers introduce parallelism in their programs manually. Most other optimisations are performed automatically by the compiler, (eg: inlining and register allocation). What if a paralleliser could optimally parallelise a program the same way that compilers optimally allocate registers?

- A paralleliser can analyse and profile a program to accurately measure performance in different parts of the program.
- Programmers find it difficult to estimate the overheads of parallel execution, this can also be measured accurately by the automatic paralleliser.
- Parallelism in one part of the program can affect how many processors are available for another part of the program. This can be tracked easily by the automatic paralleliser.
- If the program changes in the future, the paralleliser can easily re-parallelise it, without modifying the source code.

How?

- 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 program'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`.
- Analyze each procedure of each SCC, identify sequential code with two or more statements or independent expressions whose cost is greater than a configurable threshold.
- Stop recursing into callees if either:
  - the callee's cost is below another configurable threshold; or
  - there is no free processor to exploit any parallelism that the callee may have.

Support for Dependent Parallelism

In our experience less than 2%[1] of potential parallelisation sites are completely independent. Therefore dependent parallelisation must be supported.

Variables that represent dependencies can be automatically transformed into *futures*[2] — special variables protected by mutual exclusion. Futures are passed into parallel tasks and used for communication.

```
fork LogA = task_a(...);
fork LogAB = task_b(..., LogA);   →
write_log_file(LogB);
```

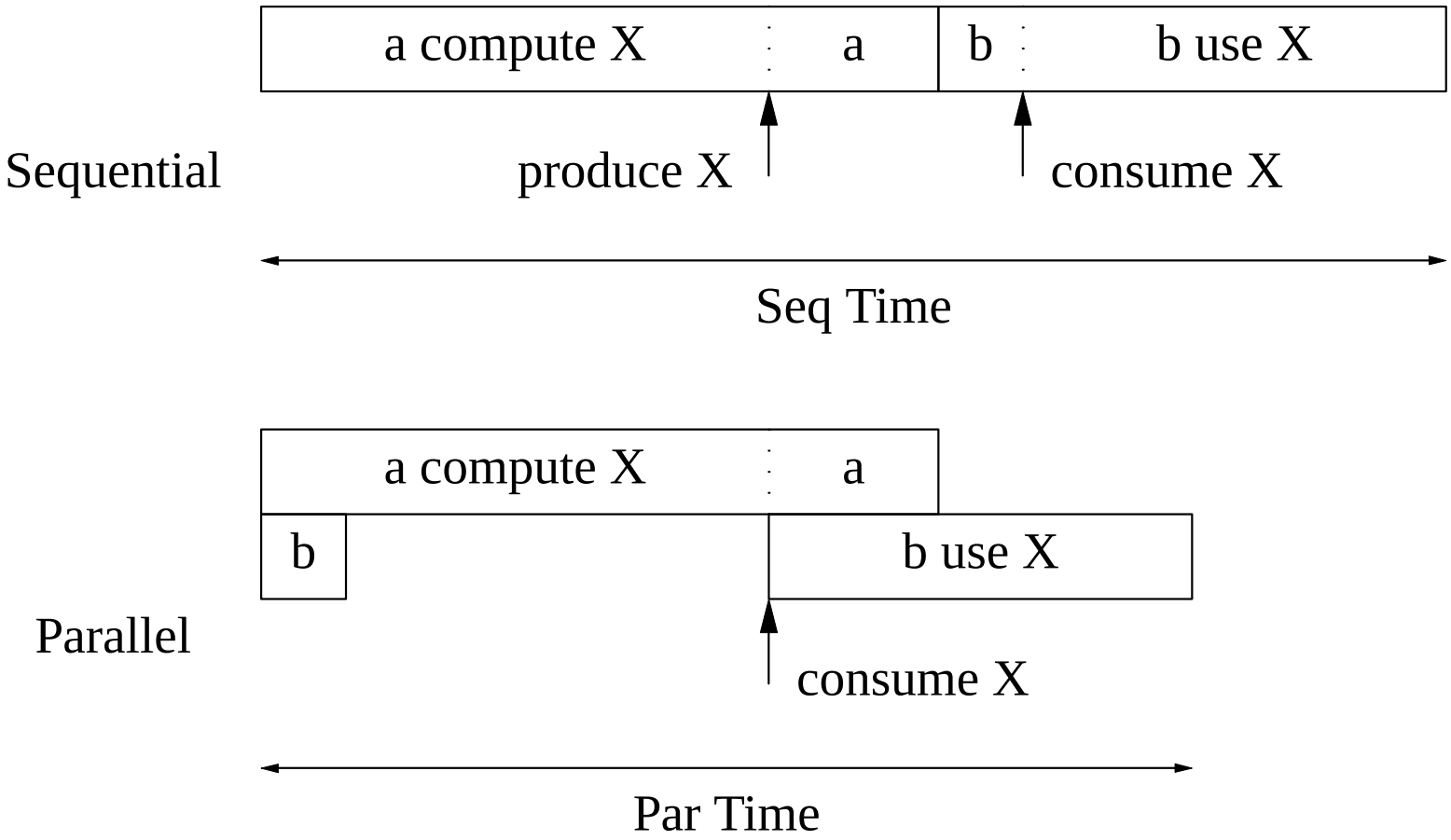
```
LogFutA = create_future();
fork task_a(..., LogFutA);
fork LogAB = task_b(..., LogFutA);
write_log_file(LogB);
```

[1] Data from analysis of the Mercury compiler

[2] Peter Wang and Zoltan Somogyi *Minimising the overheads of dependent AND-parallelism* ICLP 2010.

Overlap analysis

Tasks a and b have one shared variable X. We conceptually split each task split into sections, each section ended by the production or consumption of a shared variable.



Coverage Data

We need profiling data to determine the frequency that different branches are entered so that we can use a weighed average to determine when variables are produced and consumed in sub-computations.

Count	Goal	Call Cost
25	`void a(.., X) {`	
25	`  if (...)  {`	
5	`    X := ...;`	
	`  } else {`	
20	`    c(...);`	20
20	`    X := ...;`	
20	`    d(...);`	80
25	`  }`	
25	`}`	

Results

	mandelbrot		raytracer		spectral	
seq	19.37	(0.97)	19.50	(1.21)	16.07	(1.19)
t-safe seq	18.75	(1.00)	23.55	(1.00)	19.07	(1.00)
t-safe p1	18.74	(1.00)	23.54	(1.00)	19.30	(0.99)
t-safe p2	9.69	(1.94)	14.14	(1.67)	9.96	(1.91)
t-safe p3	6.29	(2.98)	10.72	(2.20)	6.62	(2.88)
t-safe p4	4.74	(3.96)	9.35	(2.52)	4.98	(3.83)

Paul Bone, Zoltan Somogyi and Peter Schachte *Controlling Loops in Parallel Mercury Code*, Declarative Aspects and Applications of Multicore Programming, Janurary 2012, Philadelphia PA, USA.

ThreadScope

ThreadScope[3] is a visual event-based profiler for parallel programs.

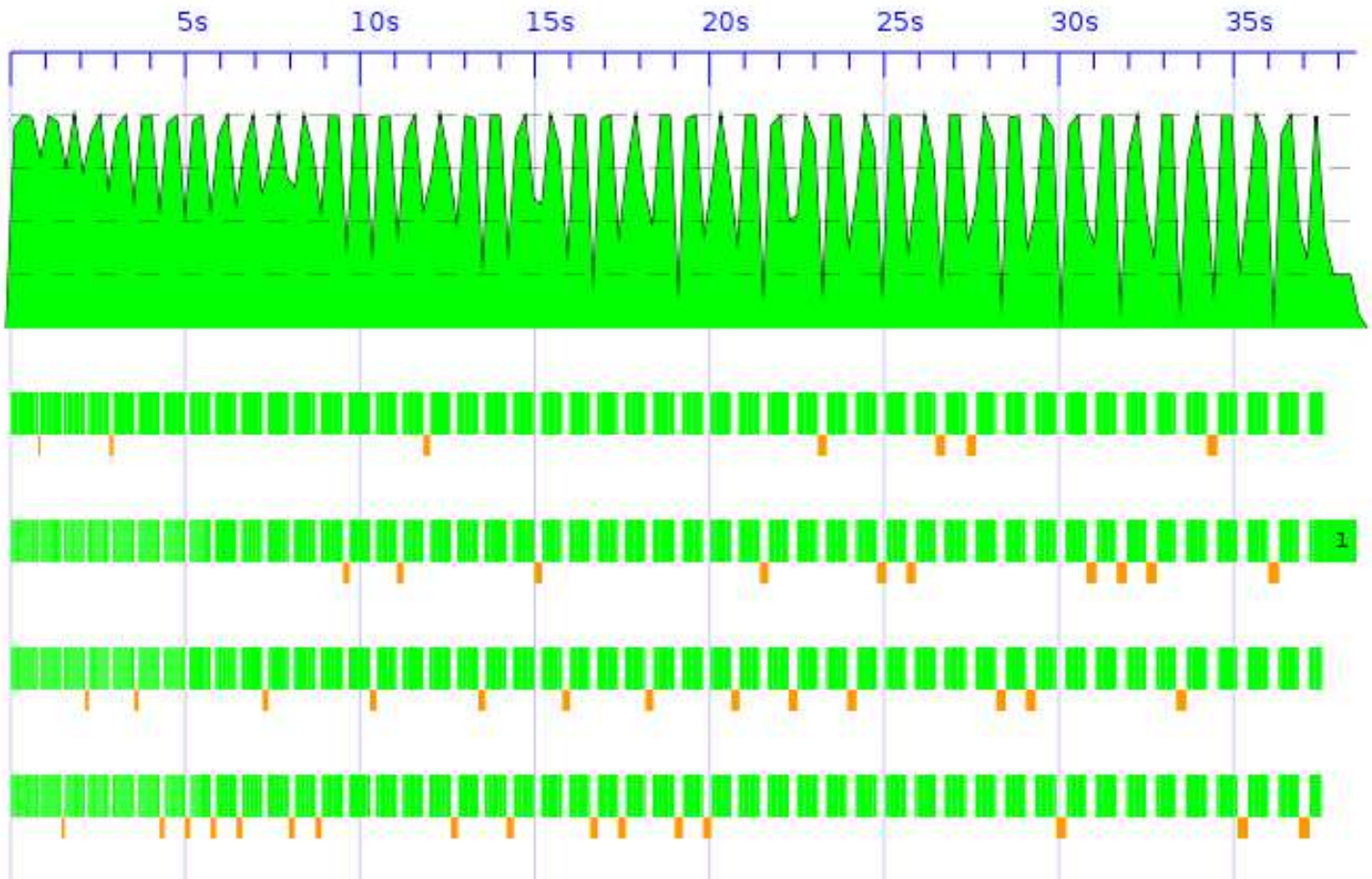
Some events include:

- Program starting/stopping, and the number of worker threads to use.
- Start / Stop garbage collection.
- Start / Stop thread.
- Thread has become blocked.
- Thread is yielding (to the garbage collector).
- Thread has become runnable.

[3] Don Jones Jr., Simon Marlow and Satnam Singh: *Parallel Performance Tuning for Haskell*, ACM SIGPLAN 2009 Haskell Symposium

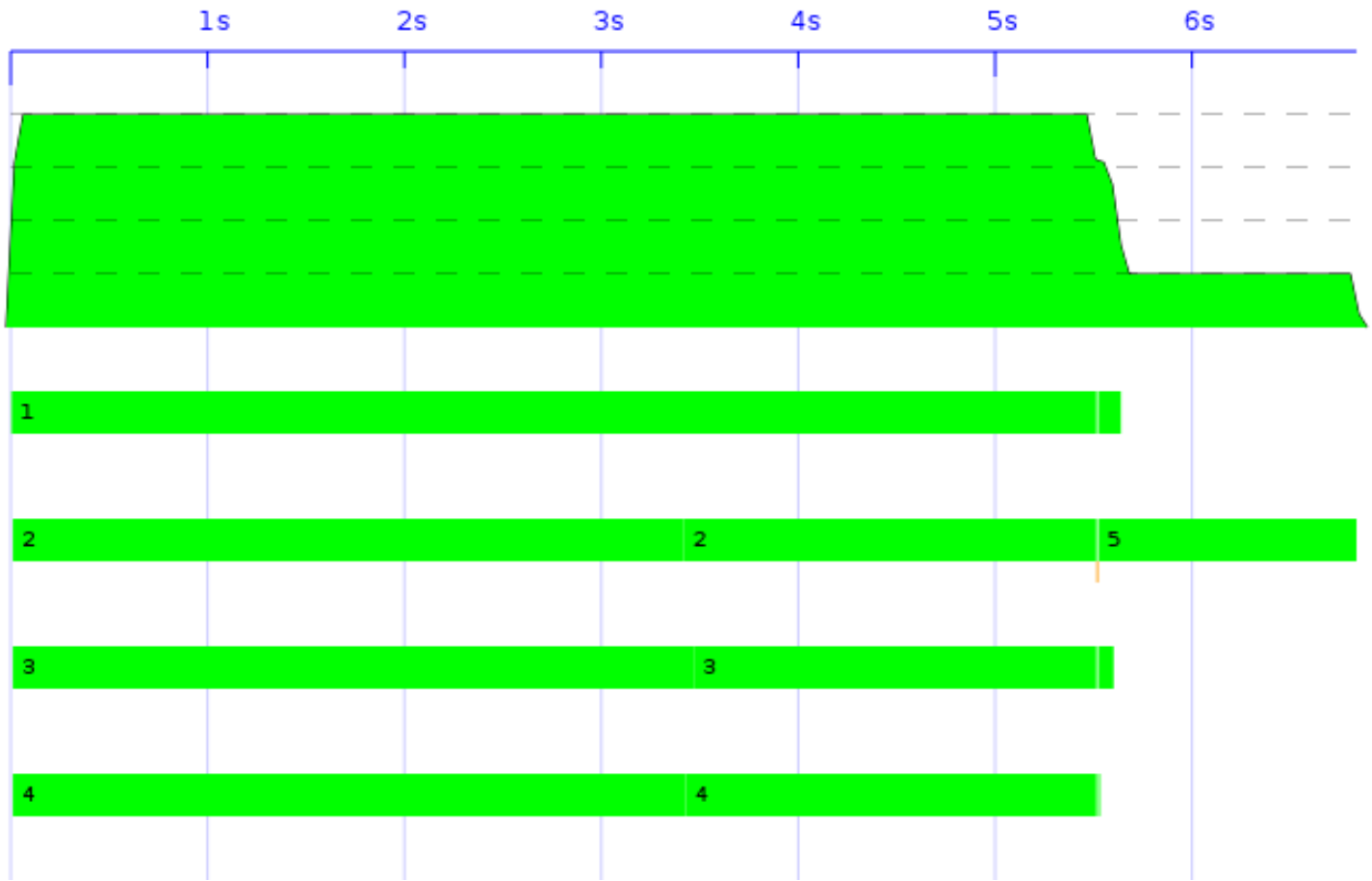
ThreadScope screenshot — Garbage collection

Profile of ICFP2000 raytracer benchmark.



ThreadScope screenshot — CPU utilization

Profile of naive parallel fibonacci calculation (synthetic).



ThreadScope gives uses a strong intuition into how their programs

CPU utilization

```
Basic Statistics
    Number of engines:          4
    Total elapsed time:         6.835s
    Runtime startup time:       10.468ms

CPU Utilization
    Average CPU Utilization:    3.44
    Time running 0 threads:     13.175ms
    Time running 1 threads:     1.204s
    Time running 2 threads:     37.639ms
    Time running 3 threads:     69.104ms
    Time running 4 threads:     5.512s
```

We intend to feed the data it captures back into our auto-parallelisation system.

Conclusions

- We have described the major parts of a working automatic parallelisation system.
- The profiler is the most important component, it provides the data needed by the analysis.
- An advice system for programmers can be built with only a profiler and an analysis tool.
- The algorithms are not specific to any programming language, although declarative langauges make some things easier.
- The best place to start such a project is with the profiler.

We have implemented our system for the open-source Mercury programming langauge.

`mercurylang.org`