Schedplot:
Design and Implementation

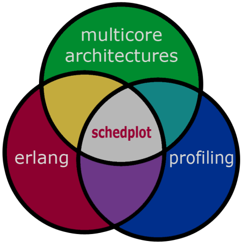
Athanasios Tintinidis
thanosqr@gmail.com

Software Engineering Laboratory
Division of Computer Science
Department of Electrical and Computer Engineering
National Technical University of Athens

September 18, 2012

What is schedplot?

An Erlang profiling tool aimed at visualizing the workload of schedulers (and garbage collection) in multi-core architectures.



Eventually, Schedplot aspires to answer a simple question: why a program doesn't run twice as fast if the number of available processors is doubled?

Motivation

- Shift to multi-core architectures
- Need for parallelization
- Lots of potential problems
- Tools required for debugging & optimization

Erlang is not a silver bullet

Even if by using Erlang many issues related to parallel programing (e.g. race conditions, deadlocks) are most of the times avoided thanks to Erlang's concurrency model, message passing and functional foundations, problems, such as bottlenecks and not ideal speedup, persist.

What is speedup?

Speedup expresses how faster a parallel algorithm runs when more processors are added; it is defined by the following formula:

$$S_p = T_p / T_1$$

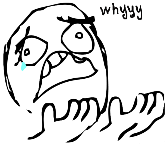
p	number of processors
T_1	execution time of the sequential algorithm
T_p	execution time of the parallel algorithm (p processors)

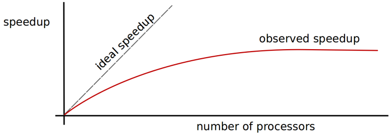
The values of the speedup usually range from 0 to p:

$S < 1$	the overhead outweighs the performance gain
$S = p$	ideal/linear speedup
$S > p$	superlinear speedup

Why speedup isn't always linear?

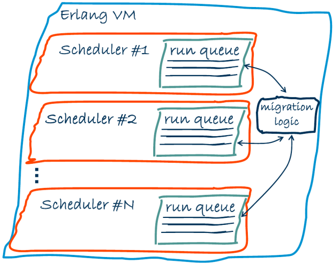
1. overhead caused by the parallelization and the addition of safety mechanisms such as locks
2. overhead caused by the need for communication between processes
3. Amdahl's law: the speedup of a parallel program is limited by its sequential part.
4. the size of the program's input (related to Gustavson's Law)





A few words about Erlang's schedulers

Support for symmetrical multi-processor (SMP) in Erlang is implemented by using schedulers (typically one for each processor) which pick jobs (i.e. Erlang processes) from the run-queue (currently, one per each scheduler) and execute them, similar to an OS scheduler.



Improved Erlang VM with SMP support

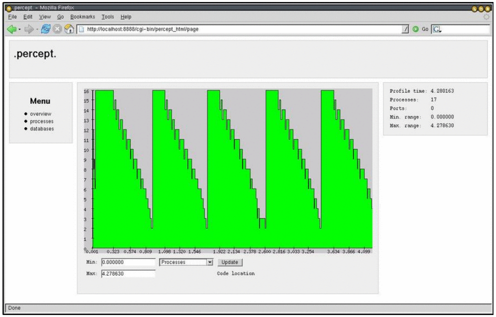
Caveats

Scheduler $\neq$ processor (but pretty close):

- each scheduler runs on an OS-thread: whether or not each scheduler runs in a different core is up to the operating system
- the number of schedulers may not correspond to the number of physical cores available to the Erlang VM
- the OS may move the OS-threads to different cores

Percept

Percept is a tool that utilizes trace information and profiler events to form a picture of the processes' and ports' runnability, showing the potential speedup of a program.



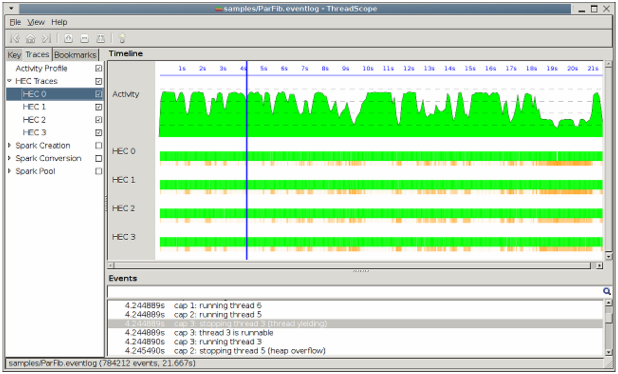
Profile time: 4.280163
Processes: 17
Ports: 0
Min. range: 0.000000
Max. range: 4.278630

Min: 0.000000
Max: 4.278630
Processes

percept execution overview

Similar tools in other languages

- Threadscope (Haskell)
- Threads View (C++AMP)
- Eden Trace Viewer (Eden)
- Thread Scheduling Visualizer (Java)



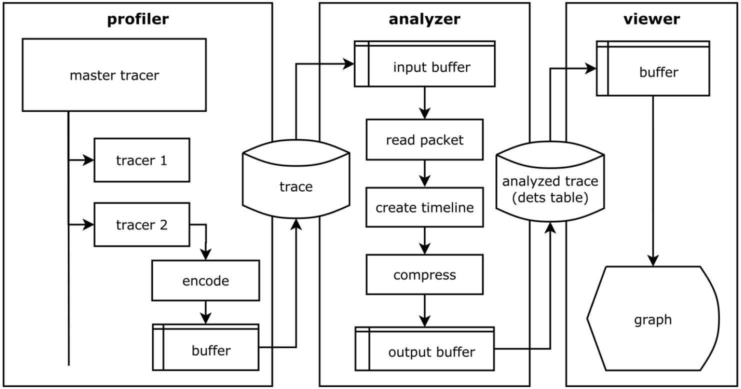
```
4.244889s   cap 1: running thread 6
4.244889s   cap 2: running thread 5
4.244889s   cap 3: stopping thread 3 (thread yielding)
4.244889s   cap 3: thread 3 is runnable
4.244890s   cap 3: running thread 3
4.245490s   cap 2: stopping thread 5 (heap overflow)
```

samples/ParFib.eventlog (784212 events, 21.667s)

Threadscope display

Demonstration

Implementation Overview

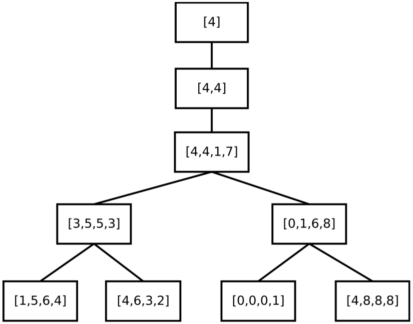


erlang:trace/3 flags

For extracting the required information from the Erlang VM, `erlang:trace/3` was used with the following flags:

1. running: send message when a process enters/exits a scheduler
2. scheduler_id: include the ID of the scheduler
3. timestamp: include a timestamp
4. set_on_spawn: trace the children of the process
5. trace, PID: send the trace messages to master tracer
6. gc (optionally): trace garbage collection

Analyzing trace files



- It is require to re-encode the traces in a format more suitable for real-time interaction
- 0 when the scheduler is inactive, 1 when it is active
- examining the execution at 1us resolution is rarely required so a initial zoom-out is performed by grouping the values
- finally, a variation of run-length encoding is done

Encoding Scheme

- If we encounter the same value two or more times we encode it as `V* N` where `V* = <<1:1, V:7>>` where V is the value encountered and N is the number of repetitions (this dictates that the value range is 0-127 so it can be stored in 7 bits resulting in 1 byte per value; this range is actually sufficient).
- Otherwise we simply keep the values as it is: `<<0:1,V:7>>`)

N is kept in 1 byte and hence varies from 2 to 255; if there is a larger sequence of same numbers it is encoded in multiple packets. This results in significant compression, mainly due to large sections of total activity (127) or inactivity (0).

Zooming-Out

The way data are zoomed-out in Schedplot differs from a lot of similar projects. Assume that the trace is:

[00, 00, 11, 11, 00, 00, 00, 11, 00, 11, 00, 11]

A simple approach to zoom-out would be combining the data in sets, calculate the average and round up. The result(for combining in sets of 2) would be:

1:2 [00, 11, 00, 11, 11, 11]

1:4 [11, 00, 11]

1:8 [11]

In the end, the scheduler appears 100% active while in reality it's just 58%!

Schedplot approach

There are three different values that we should make sure that are displayed at any scale :

1. the activity is maximum
2. the activity is 0
3. the activity is somewhere in-between.

Therefore, the minimum height for the display of one scheduler is 2 pixels: 11 for the first case, 00 for the second and 01 for the third. So, assuming a display with 2 pixels length we would have:

1:2	[00, 11, 00, 01, 10, 10]
1:4	[11, 01, 10]
1:8	[01]

Causes of overhead

1. the tracing itself (erlang:trace/3)
2. forwarding messages from the master tracer to tracers
3. encoding of messages
4. writing the messages to the disk

Worst case scenario

The worst case scenario, regarding overhead, is a program that achieves ideal speedup keeping all the schedulers busy and thus bombarding the tracers with messages which have to be encoded and stored. To study this case we used the following program:

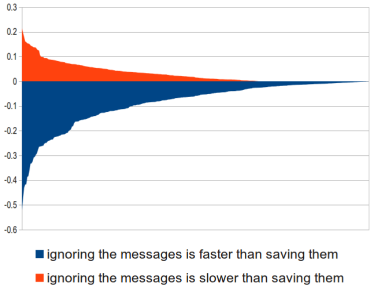
```
seq(N, M) ->
MM = M*1000000,
Self = self(),
L = lists:seq(1,N),
_ = [spawn(fun() -> seq_(Self, MM) end || _ <- L],
_ = [receive ok->ok end || _ <- L],
ok.

seq_(PID, 0) -> PID ! ok;
seq_(PID, M) -> seq_(PID, M-1).
```

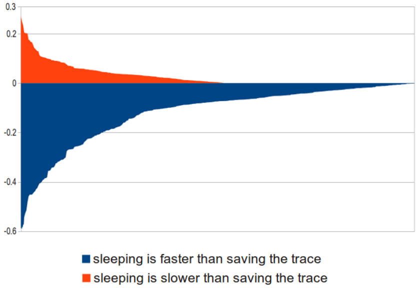
Modified Tracers

To calculate the overhead caused by the tracing we used the following modified tracers:

1. a tracer that ignores all received messages
2. a tracer that sleeps (the messages stay in the inbox)

Difference between modified and actual tracers

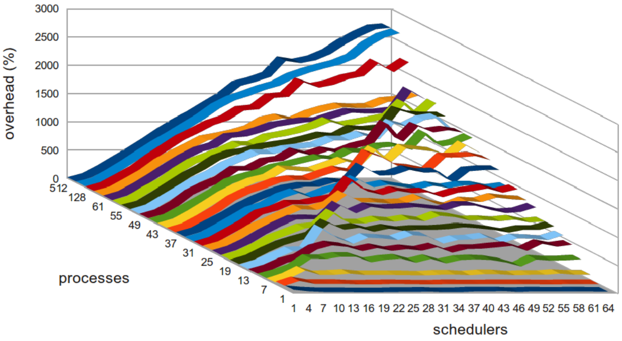




The difference of overheads is less that 0.4%; moreover, in some cases schedplot is faster.

Therefore, the overhead is introduced by the use of erlang:trace/3 and not from the way the traces are encoded or stored.

Total Overhead



Overhead (%) of profiling (3D)

Concluding Remarks

Pros:

+ Scallable for the average case
+ Highly customizable profiling
+ Fast trace analyzing & small output files
+ Real-time intuitive viewer
+ Minimal overhead caused by encoding & storing traces

Cons:

- Considerable overhead due to underlining meachanisms
- Noticeable bottleneck in profiling for the worst cases
- Profiling customization could be more user-friendly
- More information available in the viewer (e.g. code weaving) would be nice

Any questions?

Any questions?

Thanks!