Profiling parallel Mercury programs with ThreadScope

Paul Bone and Zoltan Somogyi

The University of Melbourne

National ICT Australia

21st Workshop on Logic-based methods in Programming Environments

July 10th, 2011

Motivation

Parallel programs may suffer from

- any performance problems that may also occur in sequential programs, and
- some that are specific to parallel programs.

The second category of problems are hard to identify and diagnose without tools. The best tools offer visualizations of the program's behavior.

Mercury and ThreadScope

People want to understand the performance of their programs for these reasons:

- Users want to tune their explicitly parallel programs.
- Language implementors want to tune their run time systems.
- Researchers want to verify and improve their auto-parallelization systems.

Donnie Jones, Simon Marlow and Satnam Singh developed ThreadScope to help programmers tune parallel Haskell programs when using the Glasgow Haskell Compiler (GHC).

Mercury's RTS is similar to GHC's, making it easy for us to support ThreadScope simply by having Mercury write out compatible log files.

Basic events

Mercury implements many of the events that ThreadScope supports with GHC, as well as some events we added to ThreadScope just for Mercury.

Some of the basic events implemented by both systems include:

- Start / Stop the RTS.
- Start / Stop garbage collection.

How much time is spent in the garbage collector?

The ThreadScope viewer indicates a garbage collection with orange. All collections *stop the world* and the GC's own helper threads are used to perform marking in parallel.

Garbage collection metrics

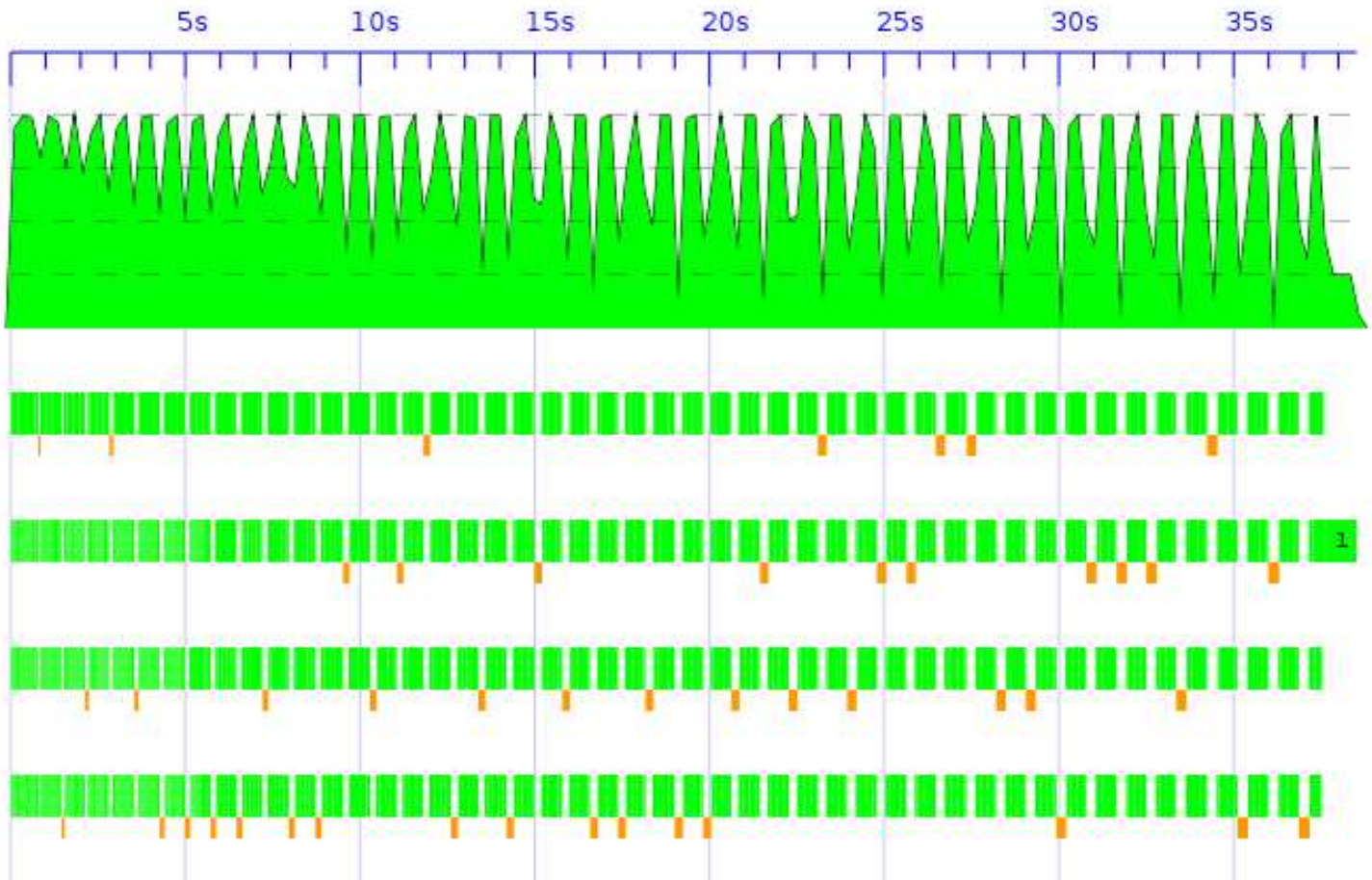
We can measure the time between Start GC and Stop GC events, and the time between the Startup and Shutdown events.

```
Basic Statistics
    Number of engines:        4
    Total elapsed time:    100.660s
    Runtime startup time: 203.698ms
    GC Stats Number:       420
             Total:         66.899s
             Average:      159.284ms
             Stddev:       175.342ms
```

The program spends roughly 66% of its elapsed time doing garbage collection!

Garbage collection metrics

The same program, but with a *much* larger initial heap size.



Garbage collection metrics

```
Basic Statistics
    Number of engines:        4
    Total elapsed time:     38.515s
    Runtime startup time:   10.357ms
    GC Stats Number:        46
             Total:         10.250s
             Average:      222.819ms
             Stddev:       229.569ms
```

The program now spends 85% less time in the garbage collector. Plus it is 2.6x faster! GC now accounts for 26% of the elapsed time.

Events for contexts

Mercury and GHC have a similar threading model.

In Mercury, *engines* correspond one-to-one with operating system threads. There are usually as many engines as there are processors.

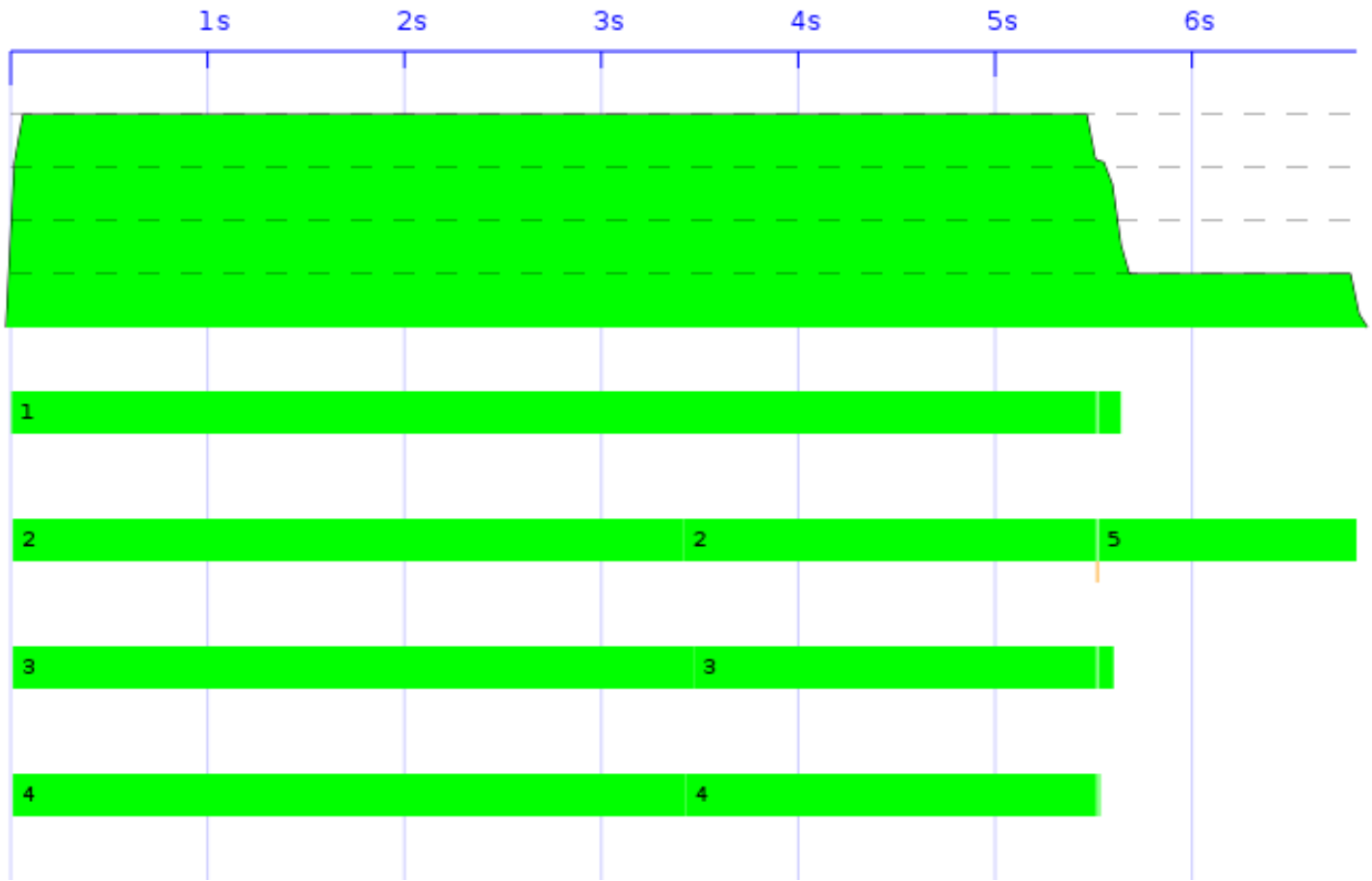
Contexts represent computations in progress. They include a stack and a saved copy of abstract machine registers for suspended computations.

We support the following original (Haskell) events for contexts:

- Start / Stop context.
- Context has become blocked.
- Context is yielding (to the garbage collector).
- Context has become runnable.

CPU Utilization

This program does not keep engines busy towards the end of its execution.



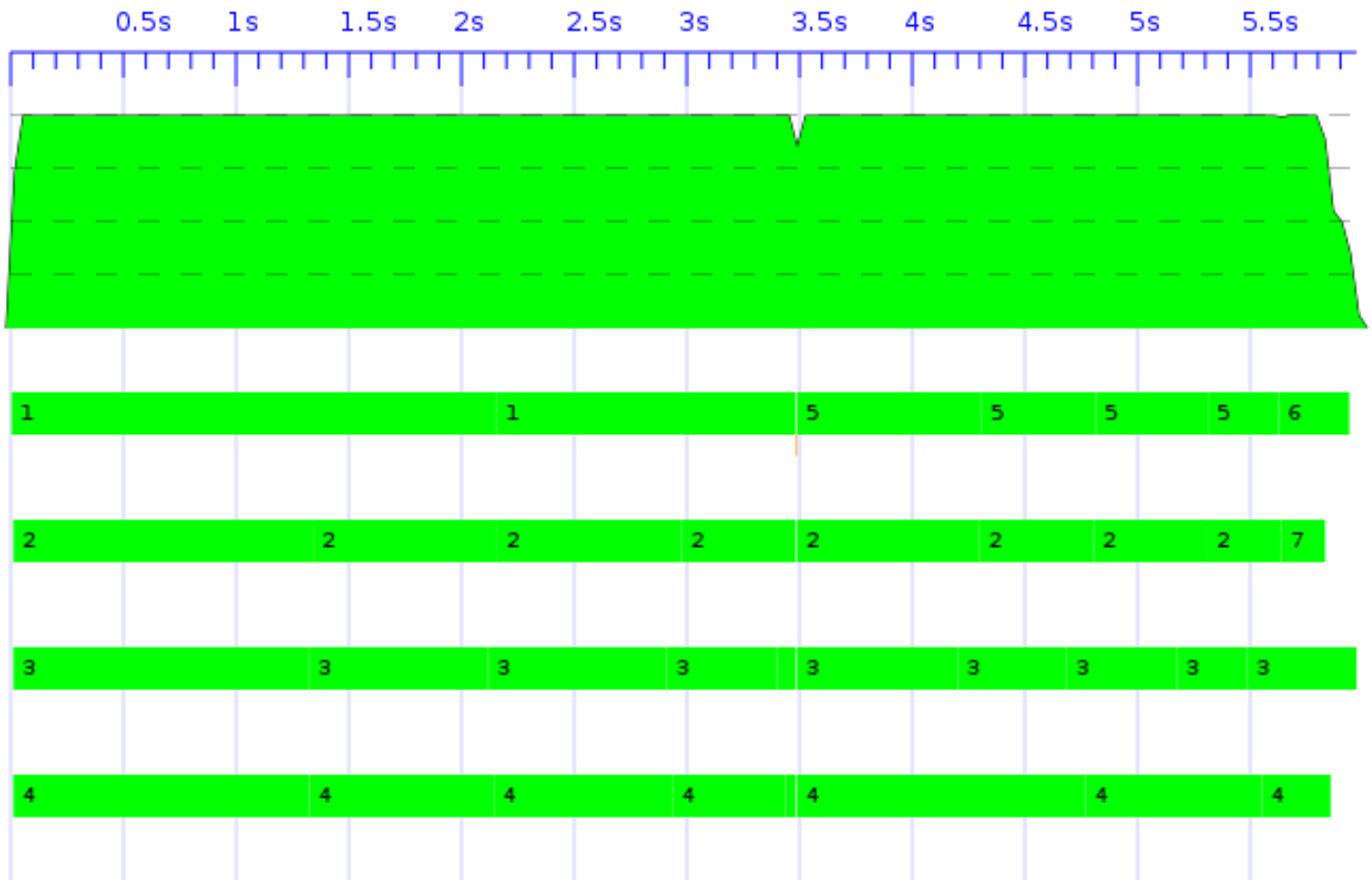
CPU Utilization

```
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
```

CPU Utilization

The same program has been modified to use finer-grained parallelism.



CPU Utilization

```
Basic Statistics
    Number of engines:          4
    Total elapsed time:         5.971s
    Runtime startup time:       10.854ms

CPU Utilization
    Average CPU Utilization:    3.94
    Time running 0 threads:     9.764ms
    Time running 1 threads:     39.369ms
    Time running 2 threads:     83.149ms
    Time running 3 threads:     25.946ms
    Time running 4 threads:     5.813s
```

Events for sparks

Both systems also support *sparks* — structures that represent parallel computations that have not yet started to execute, and therefore do not yet need a stack.

When a spark begins execution, we reuse an existing no-longer-needed context if we can, otherwise we allocate a new one.

ThreadScope defines two events for sparks:

- Run a spark taken from this engine's spark queue.
- Run a spark stolen from another engine's spark queue.

We have added an event to represent when a spark is created.

We have also added a spark id parameter to each of the pre-defined events. This will help us track individual sparks, and thus find out which parallel computations are related.

Metrics about contexts and sparks.

Earlier we measured the number of running contexts over time. We can also show the following:

- The number of *runnable* contexts plus sparks over time. This tells us if there is parallelism that could be exploited, and how much. (When it is too high, our computation may be too finely grained)
- The number of blocked contexts over time.
- When a context is blocked, how long it is blocked for.
- The number of live contexts over time — an indication of how much stack space is allocated.
- The rates that sparks are created and converted — another indication of granularity. These rates may be high even when the number of runnable contexts and sparks are low; this happens when work is created and consumed more synchronously.

New events for Mercury

Mercury supports the concept of parallel conjunctions and we want to measure their execution as well.

- Start / End parallel conjunction.

Using these events we can determine:

- The execution time of a particular instance of a parallel conjunction.
- The distribution of execution times for a parallel conjunction.

We have also added an event to mark the end of a parallel conjunct.

We did not need an event for the beginning of a parallel conjunct because it would be the same as a Run or Steal spark event with the correct spark id.

We can calculate the execution time and distribution of execution times for parallel conjuncts.

Metrics for Conjunctions

These events enable us to determine which parallel conjunct created which spark; and in turn, which context that spark uses. This allows us to attribute all the previous metrics to their parallel conjunct and conjunction.

Parallel conjunct $\rightarrow$ Spark $\rightarrow$ Context

Therefore a number of metrics can be derived based on sparks and contexts created for a specific conjunct and conjunction:

- Number of running contexts.
- Number of runnable contexts plus number of sparks.
- Number of blocked contexts.
- When a context is blocked, how long it is blocked for.

We can also use many of the metrics shown so far and restrict their input data to the events occuring only during a particular parallel conjunct or conjunction.

Conclusion

We have achieved the following so far:

- The Mercury RTS supports all the applicable basic ThreadScope events.
- All the Mercury-specific events in the paper have been implemented.
- Some analyses have been implemented.
- The complete and proposed work will make it easy for Mercury programmers, implementors and researchers to use parallel Mercury.

We will continue with the implementation of our proposal. In the future we could investigate the following:

- Allow the user to access the metrics while exploring the profile in the GUI.
- Gather extra data from the OS and CPUs such as page faults, cache-miss rates etc.

Events for futures

Mercury supports dependent parallel conjunctions by wrapping shared variables in *futures*.

- Create future.
- Wait (Suspend / No-suspend) on future.
- Signal future.

Create future events name the variable that is represented by the future. Create future events occur immediatly before their Start parallel conjunction event.

We can now measure the following:

- How often a particular future blocks a context.
- How long, on average, a future blocks a context.
- When a future does not block a context, by what margin is blocking avoided.

New events for Mercury's runtime system.

Implementing the following events allows us to profile Mercury's RTS, and therefore tune it.

- Looking for a global context to resume.
- Trying to run a local spark.
- Trying to steal a non-local spark.
- Going to sleep.

Each of these events can be paired with another to detect if the operation is successful and how long it took; except for going to sleep, which is always successful.