`bencherl`: A scalability benchmark suite for Erlang/OTP

Stavros Aronis[1] Nikolaos Papaspyrou[2] Katerina Roukounaki[2]
Konstantinos Sagonas[1,2] Yiannis Tsiouris[2] Ioannis Venetis[2]

[1]Department of Information Technology, Uppsala University, Sweden

[2]School of Electrical and Computer Engineering, National Technical University of Athens, Greece

Erlang Workshop 2012, Copenhagen

Motivation

Frustrated Erlang programmer

I thought my Erlang program was **100% parallelizable**, but when I made it parallel and ran it on a machine with **N CPU cores**, I got a **speedup** that was **much lower than N**. Why?

`bencherl`

- Serves both as a tool to run and analyze benchmarks and as an enhanceable benchmark repository
- Focuses on scalability, rather than on throughput or latency
- Examines how the following factors influence the scalability of Erlang applications
  - Number of Erlang nodes
  - Number of CPU cores
  - Number of schedulers
  - Erlang/OTP releases and flavors
  - Command-line arguments to `erl`
- Can be used to study the performance of any Erlang application, as well as the Erlang/OTP itself

Definitions

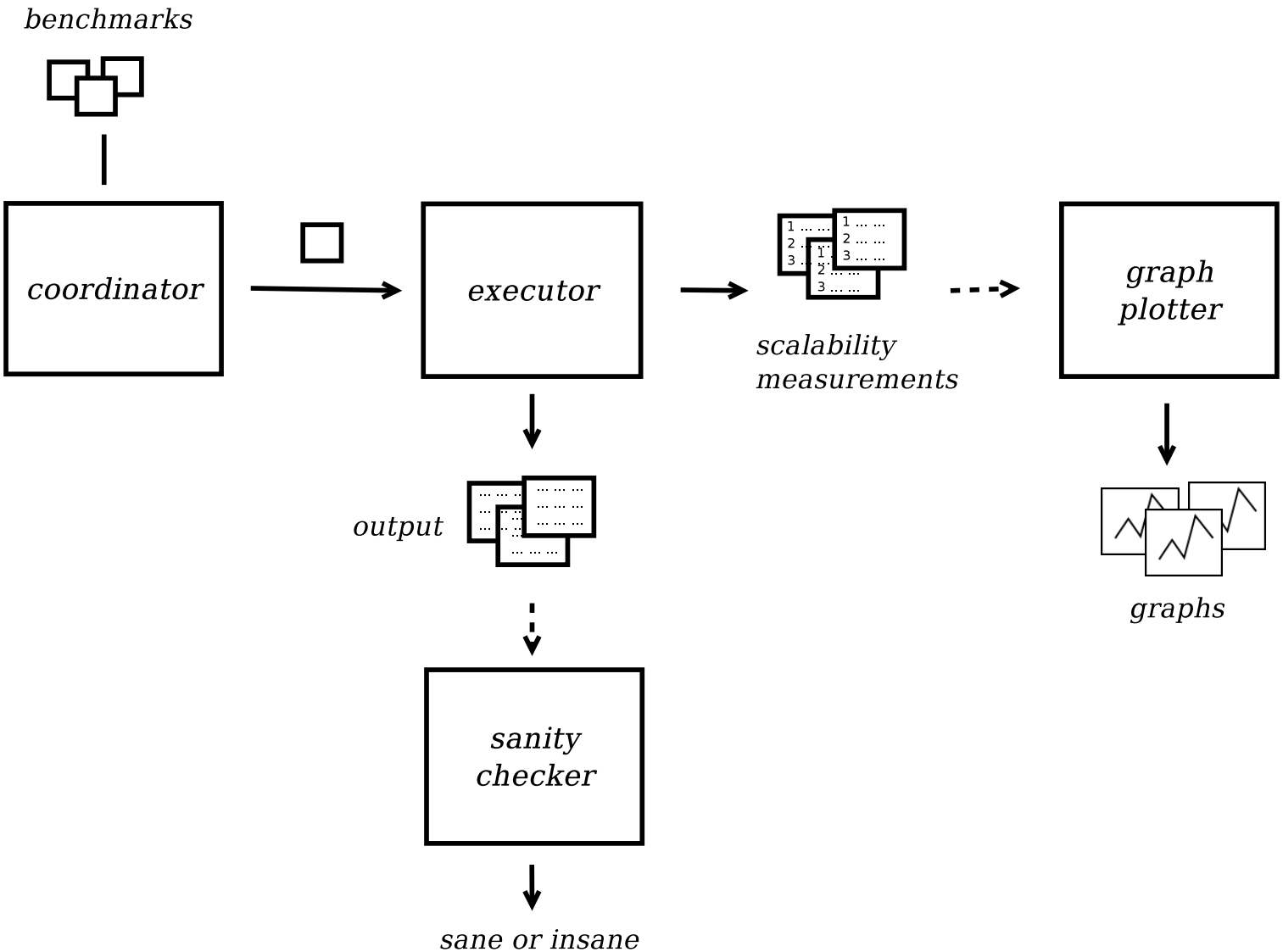
Application: The piece of software whose execution behaviour we intend to measure and analyze.

Benchmark: A specific use case of the application that includes setting up the environment, calling specific functions and using specific data.

Runtime environment: A specific combination of values for the scalability factors. E.g.

- 8 Erlang nodes
- each node runs on a machine with 8 CPU cores
- each node uses 8 schedulers
- each node runs the R15B02 release of Erlang/OTP
- each node passes "+sbt db" as command-line arguments to `erl`

Architecture



Coordinator

The module that coordinates everything during a `bencherl` run.

- Determines the benchmarks that should be executed
- Determines the runtime environments, where each benchmark should be executed
- Sets up each runtime environment before a benchmark is executed
- Prepares instruction files for the executor
- Performs any benchmark-specific pre- and post-execution actions

Executor

The module that executes a particular benchmark in a particular runtime environment.

- Receives detailed instructions from the executor about what to do
- Starts any necessary Erlang slave nodes
- Executes the benchmark in a new process
- Stops the Erlang slave nodes it started
- Makes sure that the output produced by the benchmark during its execution is written in an output file
- Makes sure that the measurements collected during the execution of the benchmark are written in a measurement file
  - Uses `erlang:now/0` and `timer:diff/2`

Sanity checker

The module that checks whether all executions of a particular benchmark produced the same output.

- Runs after a benchmark has executed in all desired runtime environments
- Examines the output produced by the benchmark in all runtime environments
- Decides whether the benchmark was successfully executed in all runtime environments
- Is based on the assumption that if a benchmark produces any output during its execution, then this output should be the same across all runtime environments, where the benchmark was executed
  - Uses `diff`

Graph plotter

The module that plots scalability graphs based on the collected measurements.

- Runs after a benchmark has executed in all desired runtime environments
- Processes the measurements that were collected during the execution of the benchmark
- Plots a set of scalability graphs
  - Uses `gnuplot`

Scalability graphs

- Both time and speedup graphs
- Graphs that show how benchmarks scale when executed with a specific version of Erlang/OTP and command-line arguments and with a different number of schedulers (nodes)
- Graphs that show how benchmarks scale when executed with a specific version of Erlang/OTP and with different number of schedulers (nodes) and runtime options
- Graphs that show how benchmarks scale when executed with a specific runtime options and with different number of schedulers (nodes) and versions of Erlang/OTP

Benchmarks

`bencherl` comes with an initial collection of benchmarks.

synthetic		real-world
`bang`	`orbit_int`	`dialyzer_bench`
`big`	`parallel`	`scalaris_bench`
`ehb`	`pcmark`	
`ets_test`	`ran`	
`genstress`	`serialmsg`	
`mbrot`	`timer_wheel`	

This collection can be extended in two simple steps.

Step 1: Add in `bencherl` everything that the benchmark needs for its execution.

- The sources of the Erlang application that it benchmarks
  - E.g. `dialyzer`
- Any scripts to run before or after its execution
  - E.g. a script that starts `scalaris`
- Any data that it needs for its execution
  - E.g. for `dialyzer_bench` the BEAM files
- Any specific configuration settings that it requires
  - E.g. a specific cookie that nodes should share

Step 2: Write the handler for the benchmark.

A benchmark handler is a standard Erlang module exporting two functions.

- bench_args: a function that returns the different argument sets that should be used for running a specific version of the benchmark

```
bench_args(Vrsn, Conf) -> Args
  when
    Vrsn :: 'short' | 'intermediate' | 'long',
    Conf :: [{Key :: atom(), Val :: term()}, ...],
    Args :: [[term()]].
```

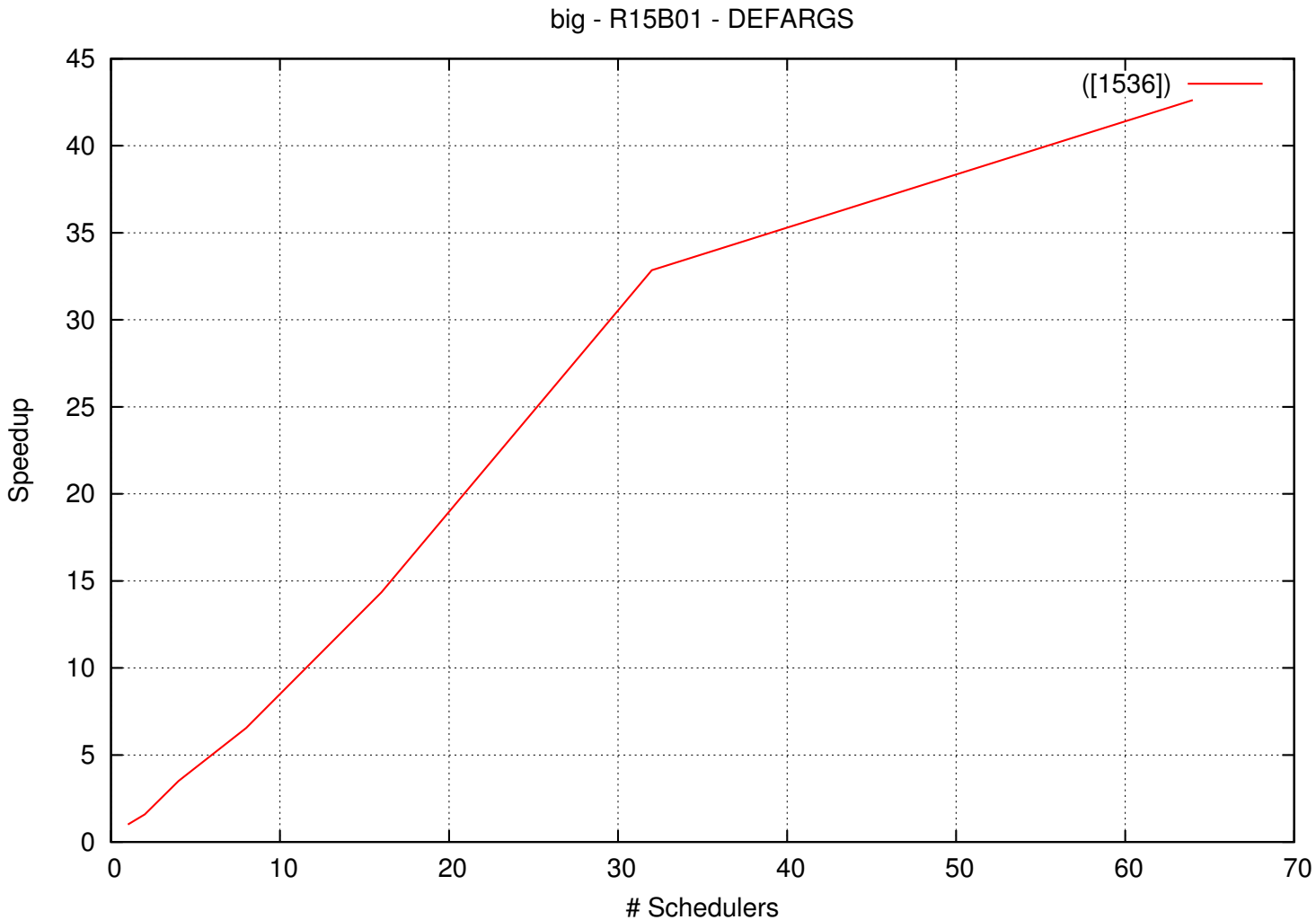
- run: a function that runs the benchmark on specific Erlang nodes, with specific arguments and configuration settings

```
run(Args, Slaves, Conf) -> 'ok' | {'error', Reason}
  when
    Args   :: [term()],
    Slaves :: [node()],
    Conf   :: [{Key :: atom(), Val :: term()}, ...],
    Reason :: term().
```

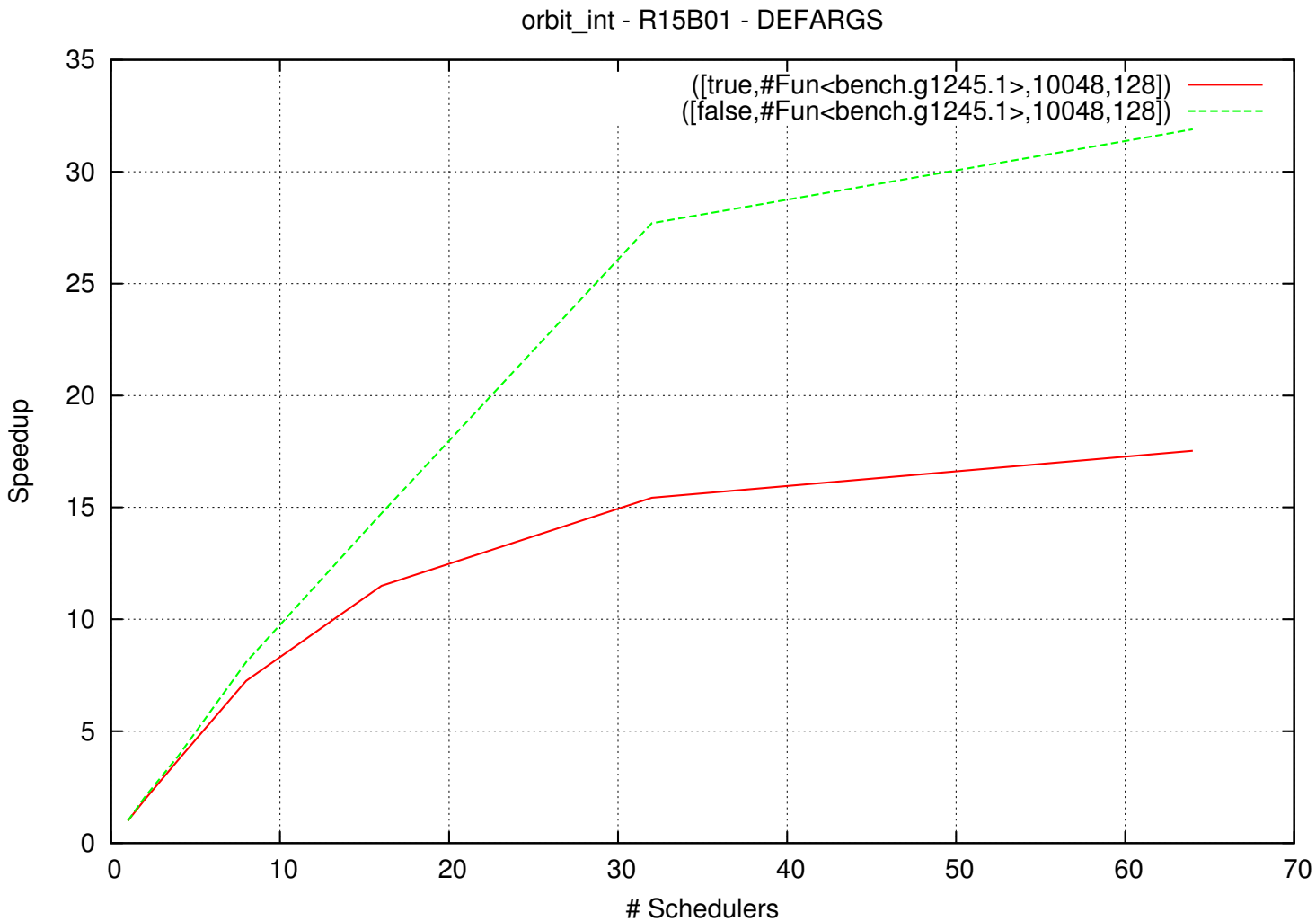
A benchmark handler example

```
-module(scalaris_bench).

-include_lib("kernel/include/inet.hrl").

-export([bench_args/2, run/3]).

bench_args(Version, Conf) ->
  {_, Cores} = lists:keyfind(number_of_cores, 1, Conf),
  [F1, F2, F3] = case Version of
                   short -> [1, 1, 0.5];
                   intermediate -> [1, 8, 0.5];
                   long -> [1, 16, 0.5]
                 end,
  [[T,I,V] || T <- [F1 * Cores], I <- [F2 * Cores], V <- [trunc(F3 * Cores)]].

run([T,I,V|_], _, _) ->
  {ok, N} = inet:gethostname(),
  {ok, #hostent{h_name=H}} = inet:gethostbyname(N),
  Node = list_to_atom("firstnode@" ++ H),
  rpc:block_call(Node, api_vm, add_nodes, [V]),
  io:format("~p~n", [rpc:block_call(Node, bench, quorum_read, [T,I])]),
  ok.
```

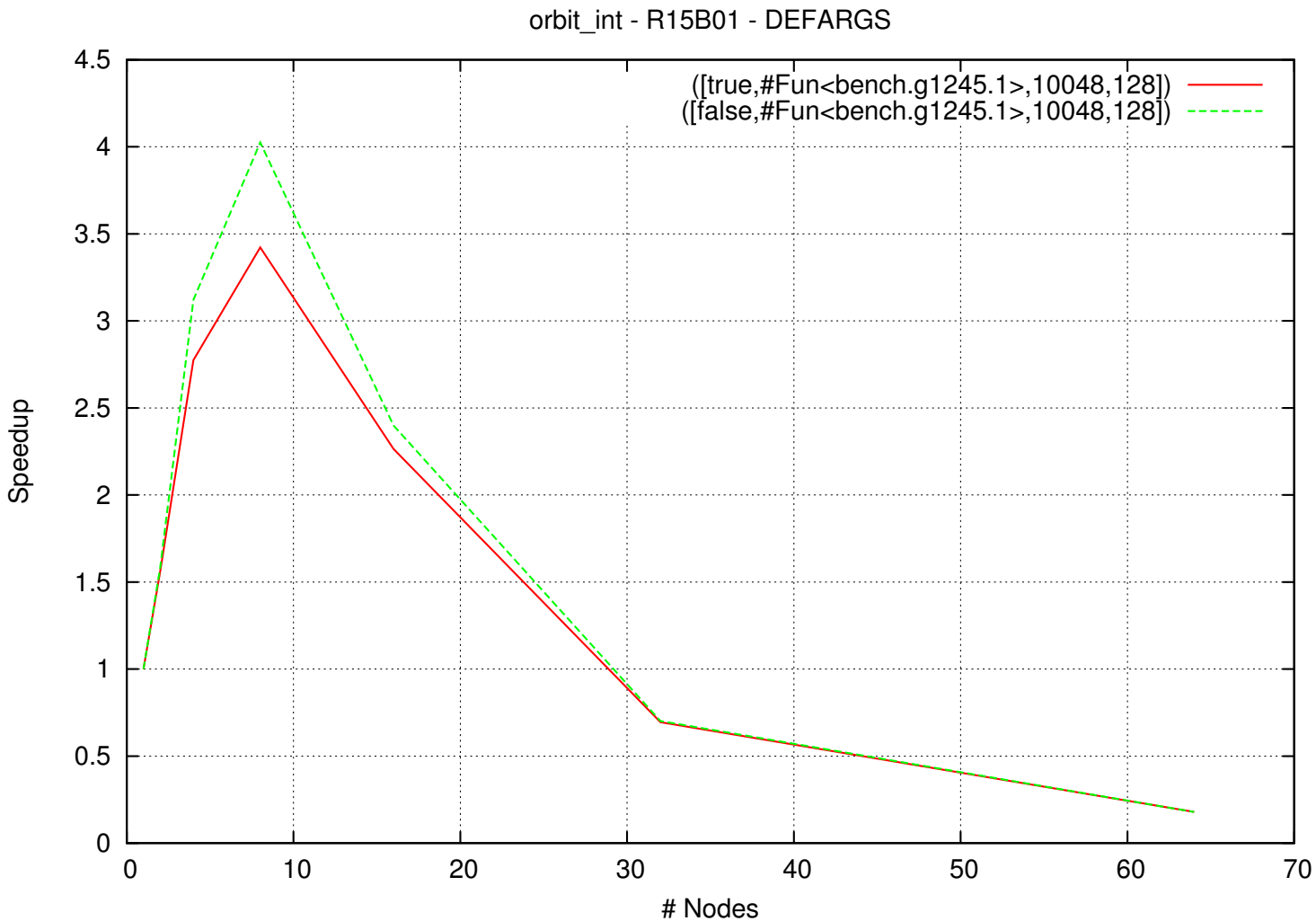
Experience #1: Some benchmarks scale well.



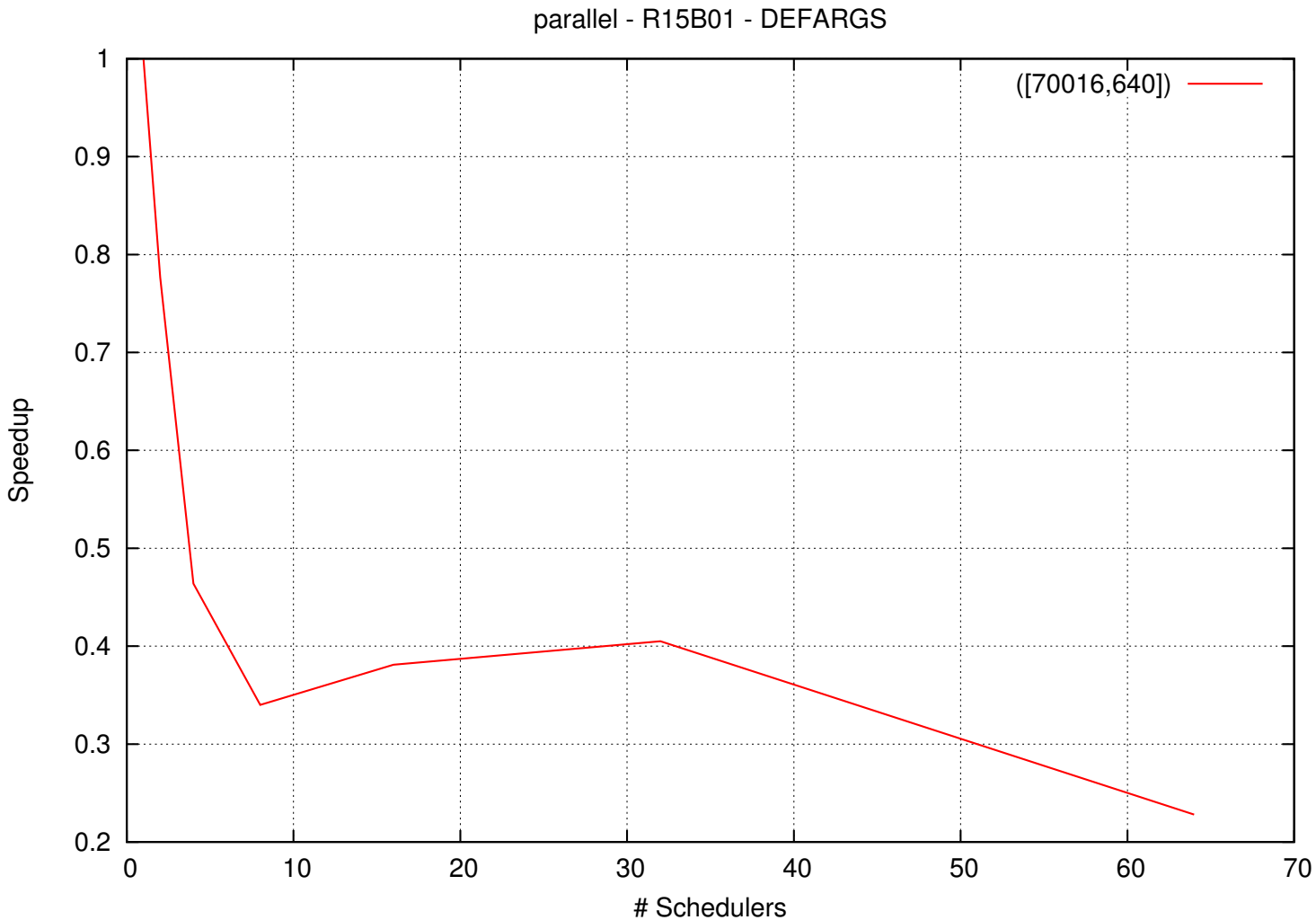
Experience #2: Some benchmarks do not scale well on more than one node.



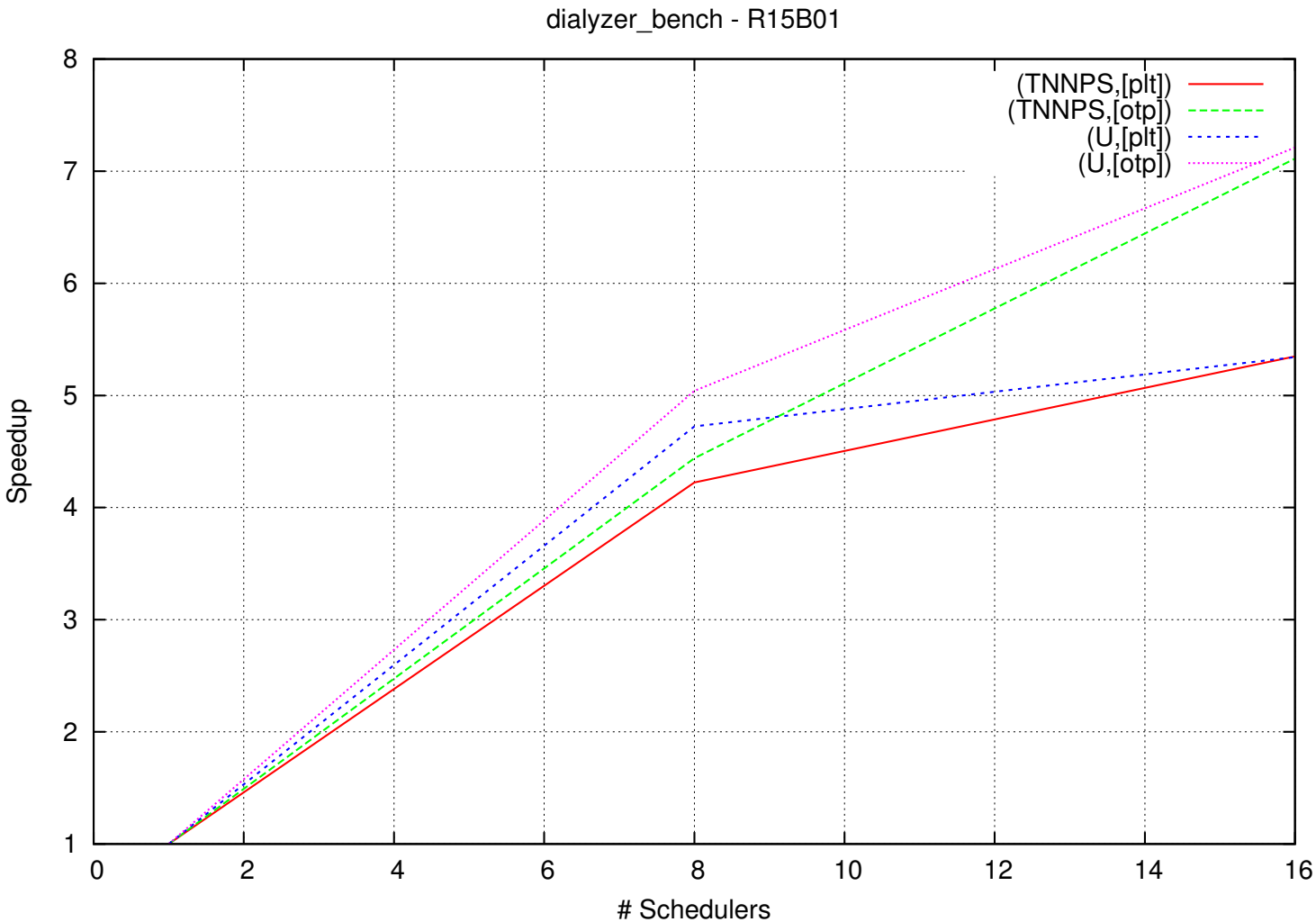
Experience #2: Some benchmarks do not scale well on more than one node.



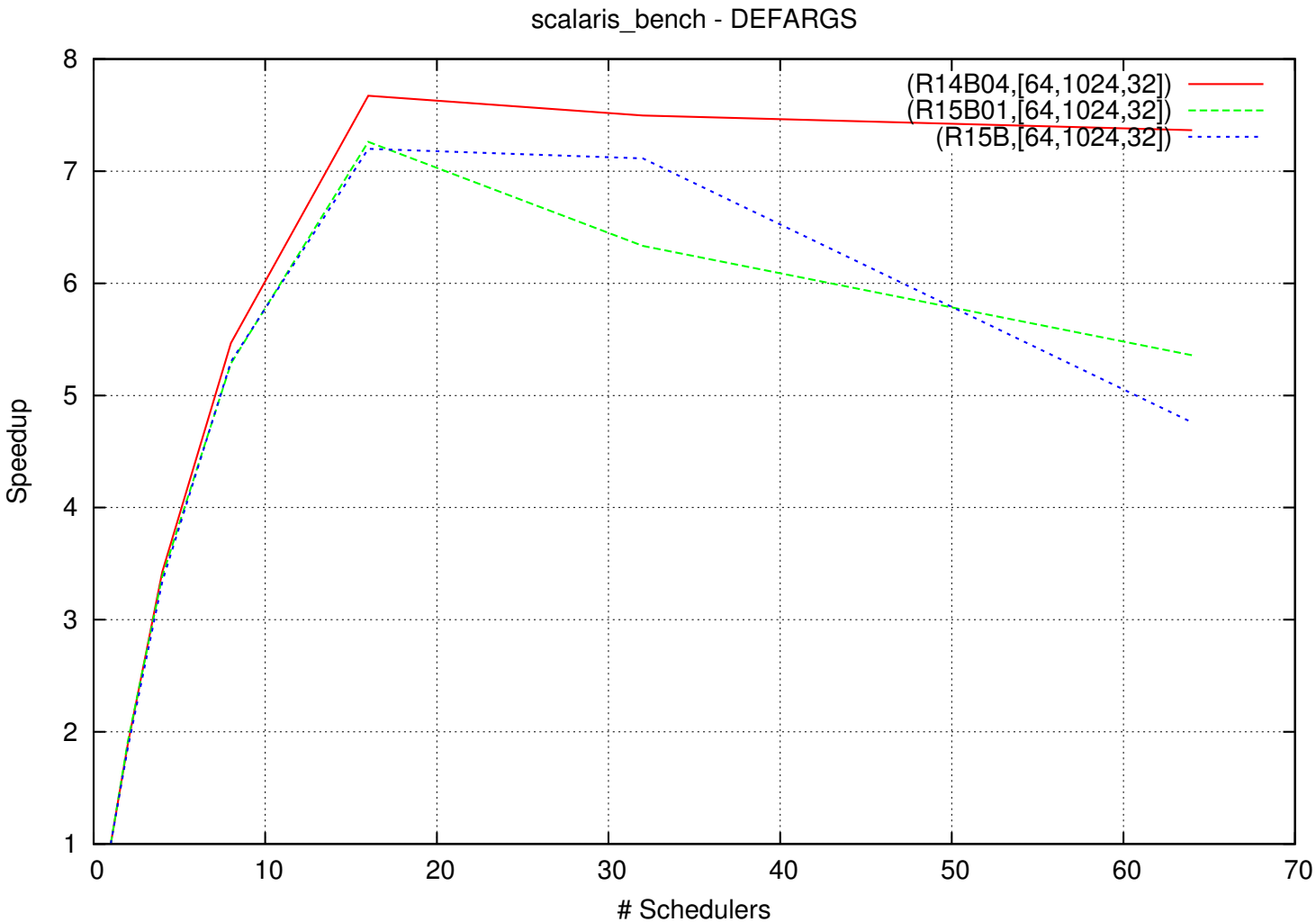
Experience #3: Some benchmarks do not scale.



Experience #4: Some benchmarks scale better with specific runtime options.



Experience #5: Some benchmarks scale better with specific Erlang/OTP releases.



Conclusions

- `bencherl` is a publicly available scalability benchmark suite for Erlang/OTP

  $\Rightarrow$ `http://release.softlab.ntua.gr/bencherl`

- Examines how nodes, cores, schedulers, Erlang/OTP versions and `erl` command-line options affect the scalability of Erlang applications
- Collects scalability measurements
- Plots scalability graphs
- Serves as a benchmark repository, where people can add their own benchmarks, so that they can be accessed and used by other people

Future work

- `bencherl` currently collects only execution times
  - $\Rightarrow$ Collect more information during the execution of a benchmark (e.g. heap size)
- `bencherl` currently can only answer questions like "Does this application scale well for this scenario?"
  - $\Rightarrow$ Try to answer questions like "Why doesn't this application scale well for this scenario?"
- `bencherl` could use `DTrace`

Thank you!