About Erlang/OTP and Multi-core performance in particular

Erlang Factory London June 26 2009

Kenneth Lundin
Manager of the Erlang/OTP team at Ericsson

ERICSSON
TAKING YOU FORWARD

Erlang/OTP R13B highlights

ERLANG

ERICSSON
TAKING YOU FORWARD

OTP release info R13B (April 22:nd)

Some Highlights in the Release

- **Multicore performance improvements**
  - (multiple run-queues)
  - Detecting CPU topology automatically at startup (Linux and Solaris only)
  - Possible to manually specify CPU topology to override info from the OS or OS:es where it is not possible to detect the topology correctly.
  - Possible to lock schedulers to logical CPUs (Linux and Solaris only)
  - Optimized message passing (reduced time for locking) (important when many senders to the same process)

- **Unicode support**
  - Impact on IO, new BIF's, support for UTF in bit-syntax,...)
- **New SSL implementation**
  - ready for limited use in products
- **WxErlang**
  - (beta version) (GUI library) planned to replace GS
- **RelTool**
  - a release management tool with graphical frontend, stepwise adding support for creation of standalone programs distributed as few files.
- **Dialyzer**
  - (support for opaque types)

OTP release info R13B01 (June 10:th)

Mainly some bug-fixes on top of R13B. Recommended to upgrade from R13B to this one.

Some Highlights in the Release

- **Multicore performance improvements:**
  - encoding and decoding messages over the Erlang distribution protocol is made more parallel
  - Improved SMP concurrency for ETS tables (more fine granular locking)
- **New functions in ETS** to transfer ownership of table.
- **New options added to `open_port`**
  - `spawn_executable` and `spawn_driver`
- **A brand new XML parser**
  - `xmerl_sax_parser` 3-4 times faster than the old one and not so memory hungry. Validation will be supported in R13B02.
- **Leex**
  - a lexical analyzer generator for Erlang, has been added as a complement to yecc in the Parsetools application.

Multicore and Erlang in more detail

Multi-core is a SW challenge

- **Impact on the entire software stack**
  - Tools, languages, libraries, runtimes, operating systems have support you in utilizing multi-core efficiently.
- **Shared memory threads and locks is too low level**
  - Programming with threads and locks in C/C++ and even Java requires great skills and it takes time to get it right.
- **Erlang already has higher level abstractions for this**
  - Very light-weight processes without shared state. Message passing.

The Erlang way

- **Continue program Erlang as before**
  - Many Erlang applications written long before the multi-core era will run and utilize the multiple cores without changes.
- **Use Erlang processes**
  - to represent parallel things in the application. E.g. calls, transactions, web-server requests, subscribers etc.
- **Make sure you use enough Erlang processes**
  - Requires some extra thinking to assure that there are enough processes ready to run and to avoid creating central processes which can become bottlenecks.
- **Possibly change some options to the Erlang startup**
  - Depending on your system and how many E-nodes and other processes which will run in your setup.
- **Possibly profile your application**
  - If the application does not scale as expected, profile with `percept` and other tools to find the application level bottlenecks.
- **That's it**
  - your program will run well on all kinds of systems from 1 core to more than 64 cores.

Some Processors that we have tested Erlang on

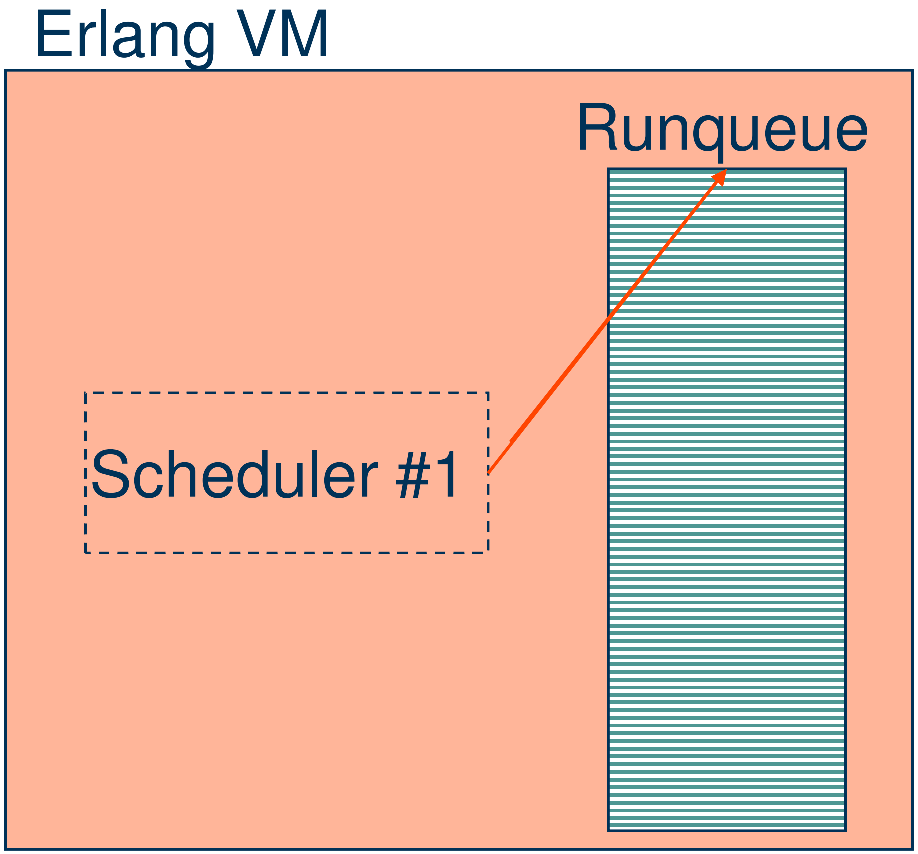
- AMD Opteron Dual and Quad core (NUMA) GEP
- Intel XEON Quad core
- Intel Nehalem Quad core with hyperthreads (NUMA)
- Tilera Pro 64 cores (NUMA)
- Sun Niagara T2000 , 8 cores with 4 threads each
- 4 x Intel Dunnington with 6 cores each
- Octeon II 16 or 32 cores
- Freescale PPC 2 or 4 cores
- Intel Core2 (in your laptops)
- Memory arch, cache-size are of great importance

History and evolution of SMP support in Erlang

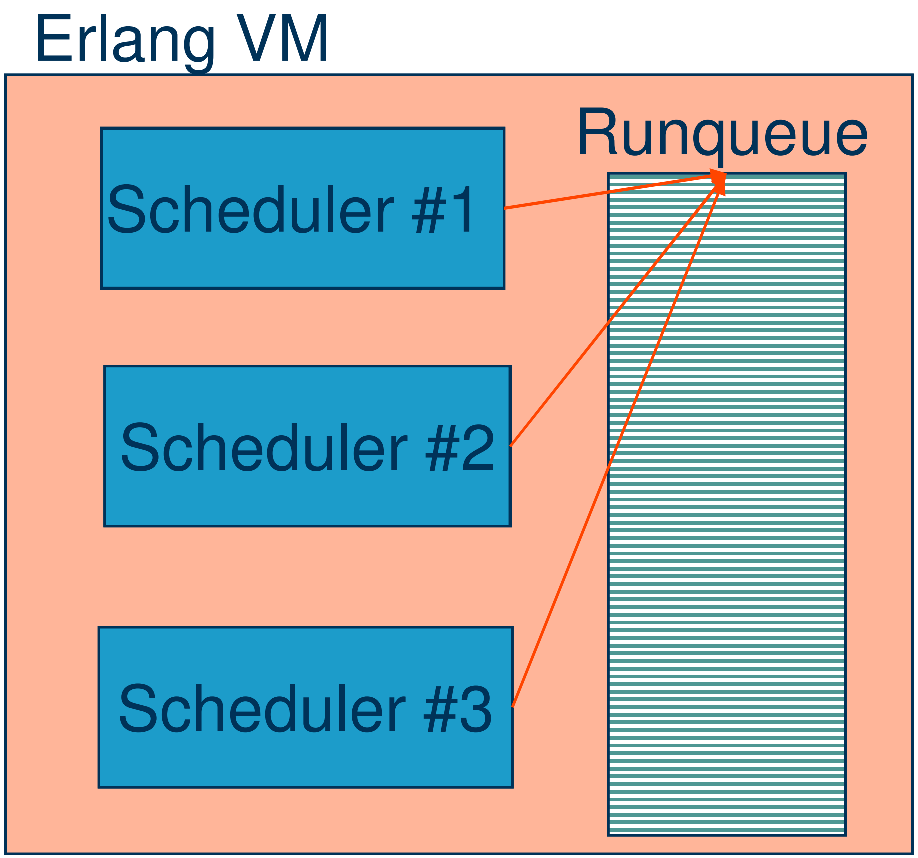
- Erlang SMP (Symmetrical Multi Processor) started 1997 with master thesis work by Pekka Hedqvist
  - Used a Compaq 4 x Pentium Pro 200 Mhz with Linux
- Erlang SMP work restarted 2005 as part of ordinary development, external coop with Uppsala University and Tony Rogvall (Synapse)
- First stable release in R11B May 2006
- Running in products March 2007, 1.7 scaling on dual core
- ...

How it used to work

Erlang (non SMP)



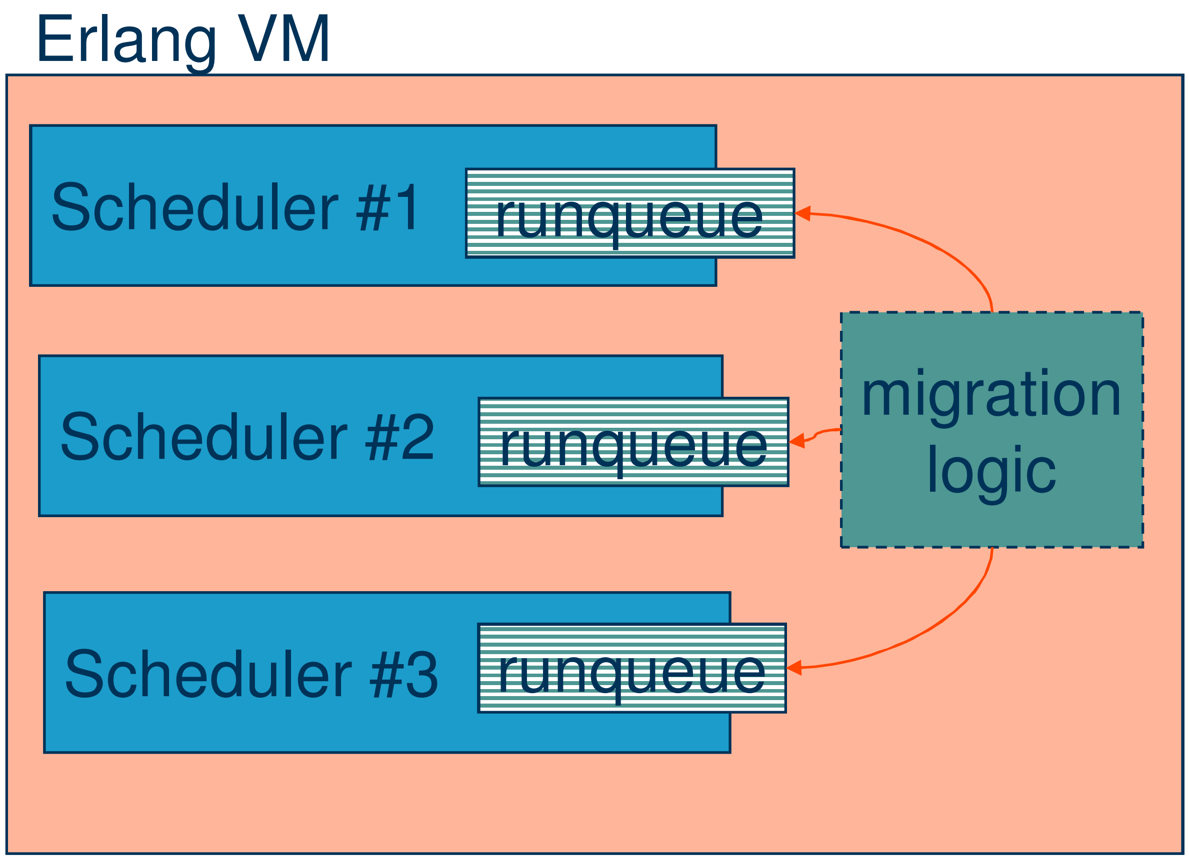
The first solution
Erlang SMP VM (before R13)



Our strategy with SMP

- Make it work -> measure -> optimize
- Hide the problems and awareness of SMP execution as much as possible for the programmer.
- Erlang should be programmed in the normal style using processes for parallelization and encapsulation
- An Erlang program should run perfectly well on any system no matter what number of cores or processors there are
- Fine grained parallelism as a later stage when running on really many cores >32?

Multiple runq-ueues
Erlang SMP VM (R13)

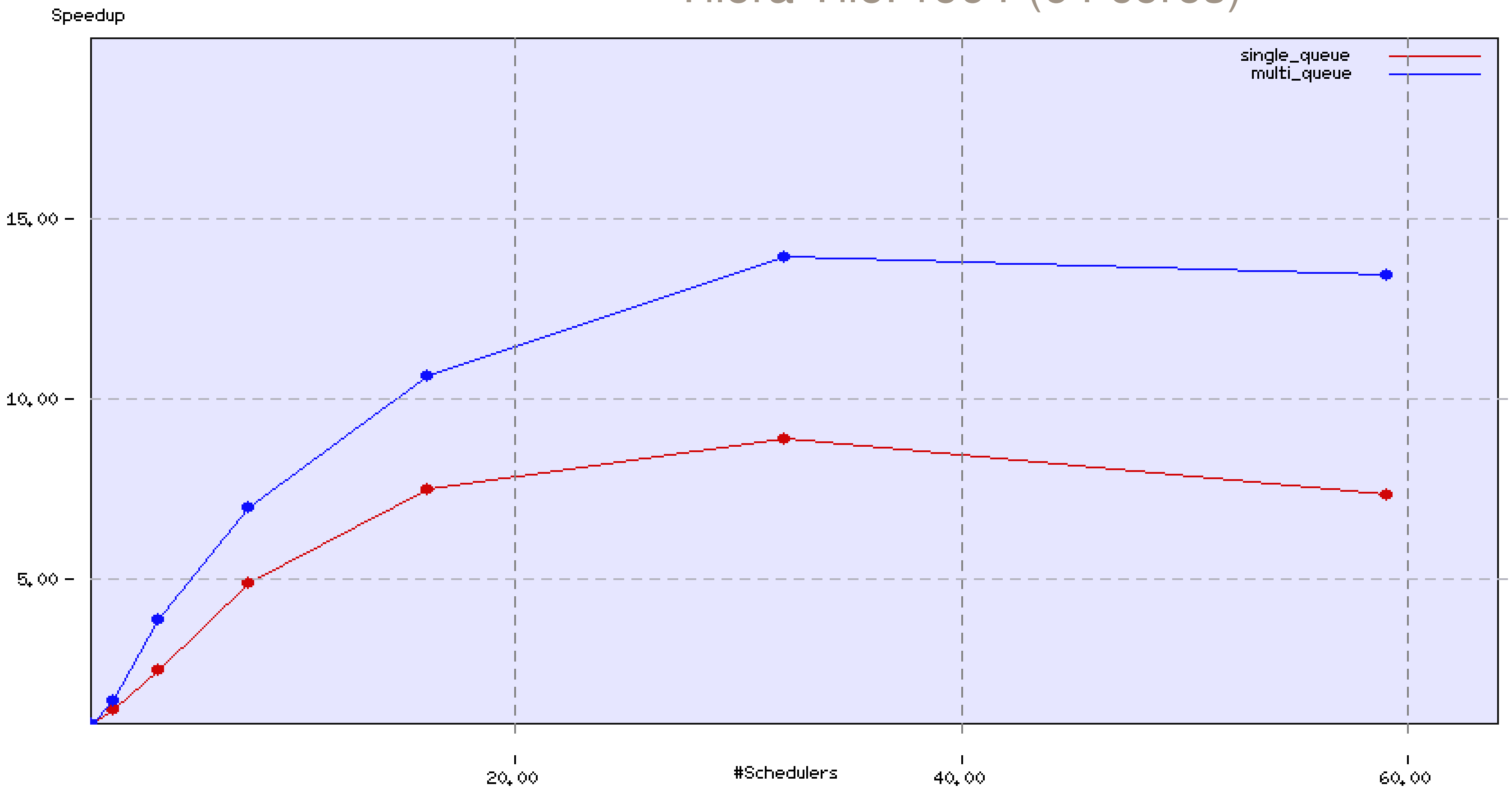


Example of improvement with multiple run-queues

big:bang

(multiple run-queue in blue)

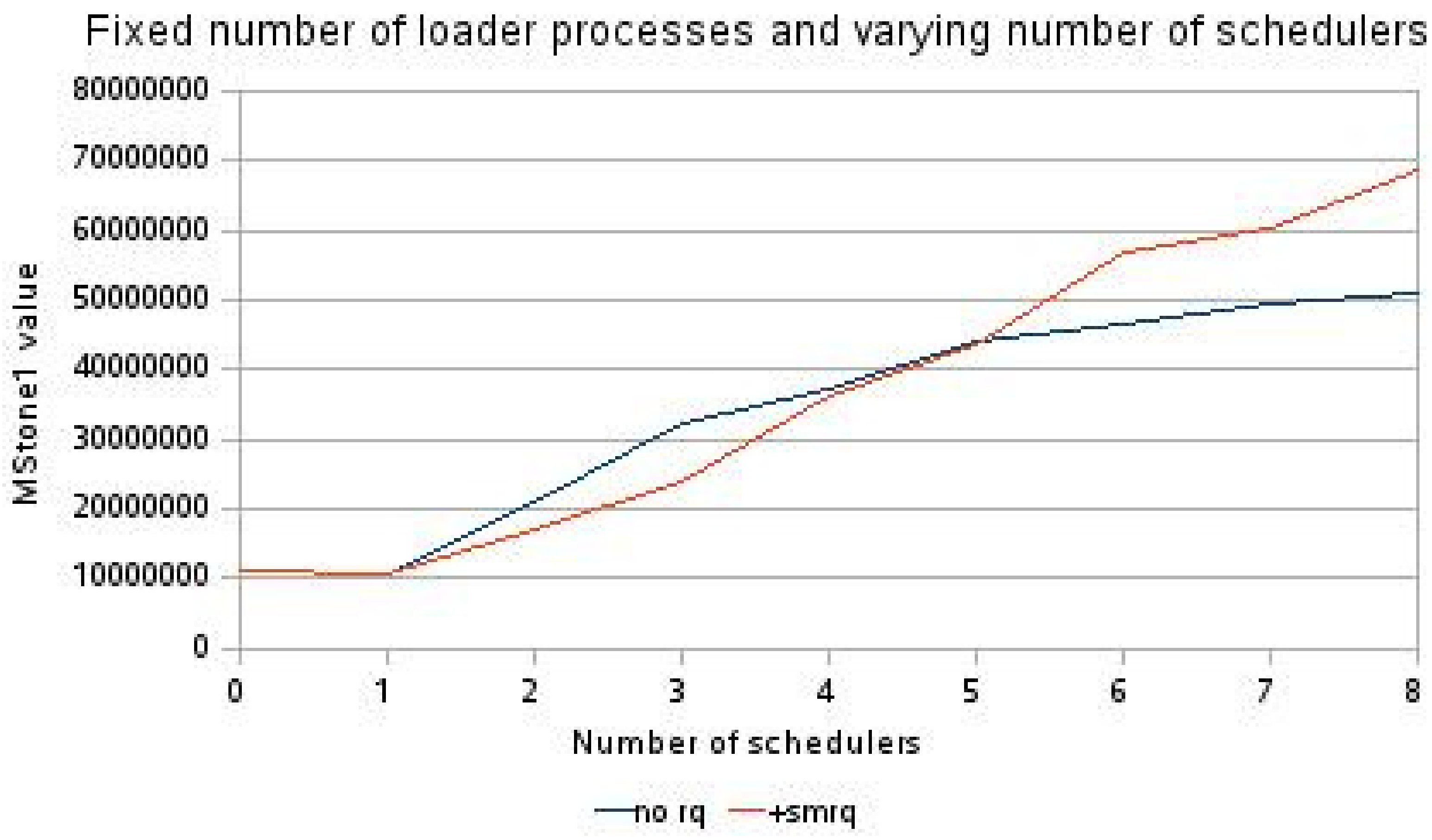
Tilera TilePro64 (64 cores)



Some Measurements

16 processes doing H.248 encode/decode in parallel (multiple run-queue in red)

2 x Intel Xeon E5310 Quad Core, SLES 10 x86_64



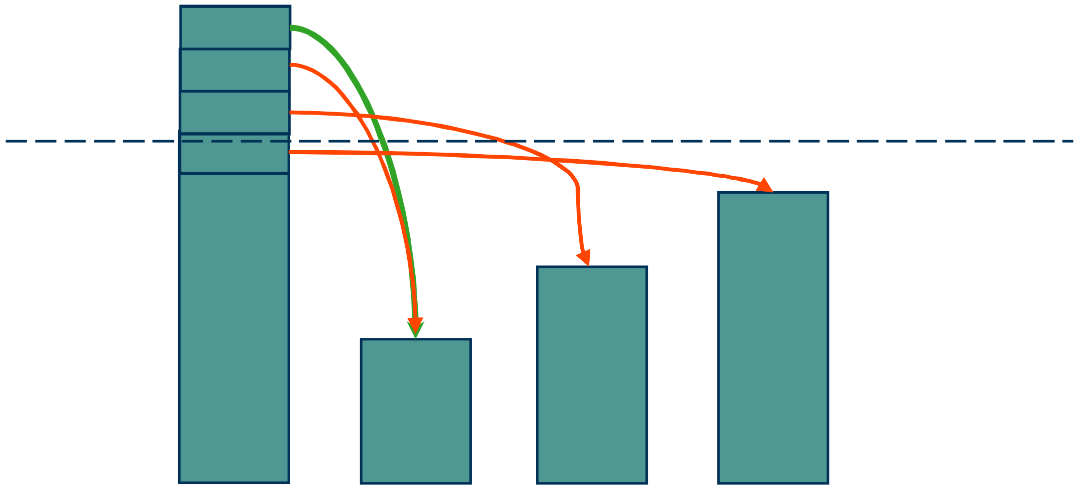
Migration logic

- Strive to keep the maximum number of run able processes equal on all schedulers
- Load balancing is performed by the scheduler that first reaches its max limit of reductions.

1. Collect statistics about the maxlength of all schedulers run-queues
2. Calculate the average limit per run-queue/prio and setup migration paths
3. **Give away jobs** from schedulers over the limit, **Take jobs** to schedulers under the limit

Migration logic (a sketch on how it works)

Full load

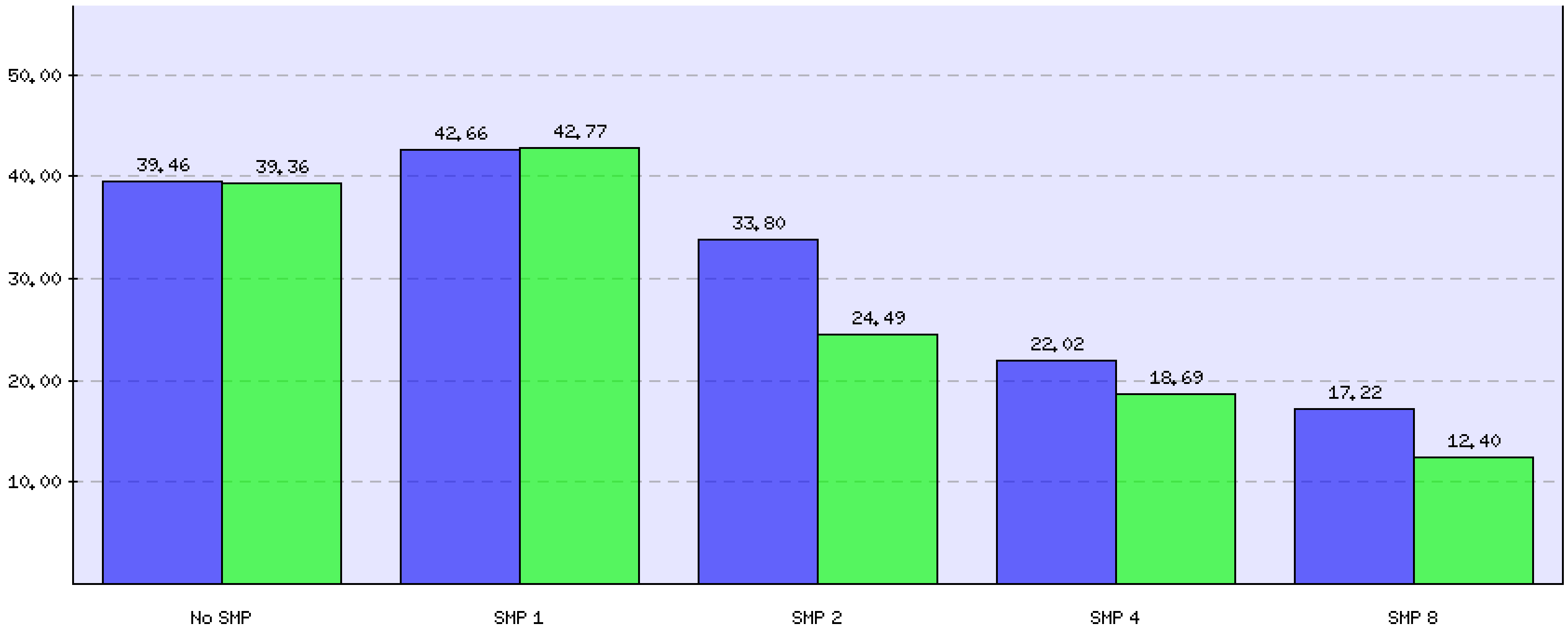


Migration logic continued

- Migrations occurs when the scheduler has finished a job and goes on until the limit is reached or a new loadbalancing takes place.
- There is also **work-stealing** , which occurs when a scheduler gets an emty run-queue
- **Running on full load or not!**
  - If all schedulers are not fully loaded, jobs will be migrated to schedulers with lower id's and thus making some schedulers inactive.

SMP support has its cost

- Non SMP is slightly faster than the SMP VM with 1 scheduler.
- Optimizing for many core systems will also slightly reduce performance on few core systems



The use of memory is very important

- **64 bit Erlang is slower than 32 bit Erlang**
  - This is because of almost twice as much memory used in the 64 bit version. And this is because most of the data contains pointers.

Tools for profiling

Erlang VM level

- Lock counter (special variant of VM)
- V-tune (Intel)
- Tools from Accumem
- Open Source tool that does the equiv of V-tune

Erlang application level

- Percept
- Lock counter (need to be documented and made official)

SMP in R12 and R13

- SMP version of VM is started automatically if the OS reports more than 1 cpu.
- Default can be overridden with the `-smp [enable|disable|auto]` flag. `-smp auto` is the default
- If `smp` is set to `enable` or `auto` use `+S Number` to set the number of schedulers (`+S 4` for 4 schedulers)
- **Normally nothing to gain from running with more schedulers than cpu's or cores.**
- Common mistake: The number of cores available might not be what you think. (might be limited with `taskset`)

Overview of SMP related options and functions

Options to ”`erl`” (the Erlang VM startup)

`erl -smp [enable|auto|disable]` default is `auto`

`erl +S Schedulers:SchedulerOnLine  (erl +S 4:4)`

Set Scheduler Bind Type

`erl +sbt db`

`erlang:system_flag(scheduler_bind_type,default_bind)`

Set CPU Topology

`erl +sct L0-3c0-3`

`erlang:system_flag(cpu_topology,CpuTopology).`

`erlang:system_info(cpu_topology).`

Get SMP properties in runtime with `erlang:system_info/1`

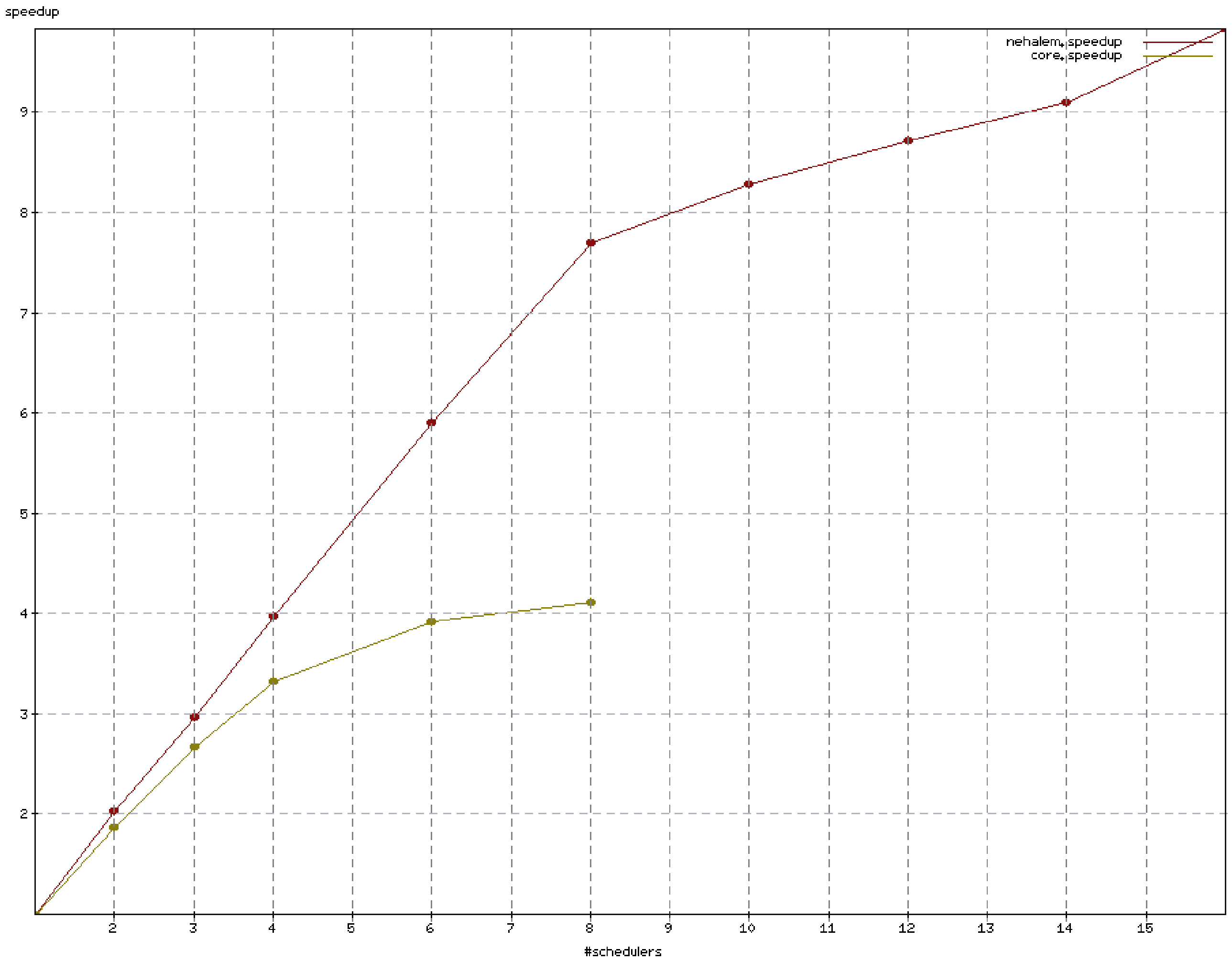
cpu_topology	Set with `system_flag/2`
multi_scheduling, block|unblock	Set with `system_flag/2`
scheduler_bind_type	Set with `system_flag/2`
scheduler_bindings	
logical_processors	
multi_scheduling_blockers	
scheduler_id	
schedulers	
schedulers_online	Set with `system_flag/2`
smp_support	

Overview of SMP related options and functions (examples)

```
erl +sbt db
or
erlang:system_flag(scheduler_bind_type,default_bind).
1> erlang:system_info(scheduler_bindings).
{0,3,1,2}
2>
>% erl
1> erlang:system_info(scheduler_bindings).
{unbound,unbound,unbound,unbound}
1> erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
             {core,{logical,3}},
             {core,{logical,1}},
             {core,{logical,2}}]}]
2>
```

This binds the schedulers to processor cores (normally in an optimal way)
Doing this will boost performance significantly and it is more important the more cores or cpus you have on the system.

Intel Nehalem 2 x 4 core with one hyperthread per core
Erlang and the Nehalem architecture goes very well together.



Multi-core tips and tricks (2)

Cpu topology presented for some real machines:

```
1> erlang:system_info(cpu_topology).
[{processor,{logical,0}},{processor,{logical,1}}]
```

A dual core processor

```
1> erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
             {core,{logical,1}},
             {core,{logical,2}},
             {core,{logical,3}}]}]
```

A quad core processor

Here is a dual processor quad-core:

```
 1>  erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
          {core,{logical,2}},
          {core,{logical,4}},
          {core,{logical,6}}]},
 {processor,[{core,{logical,1}},
          {core,{logical,3}},
          {core,{logical,5}},
          {core,{logical,7}}]}]
```

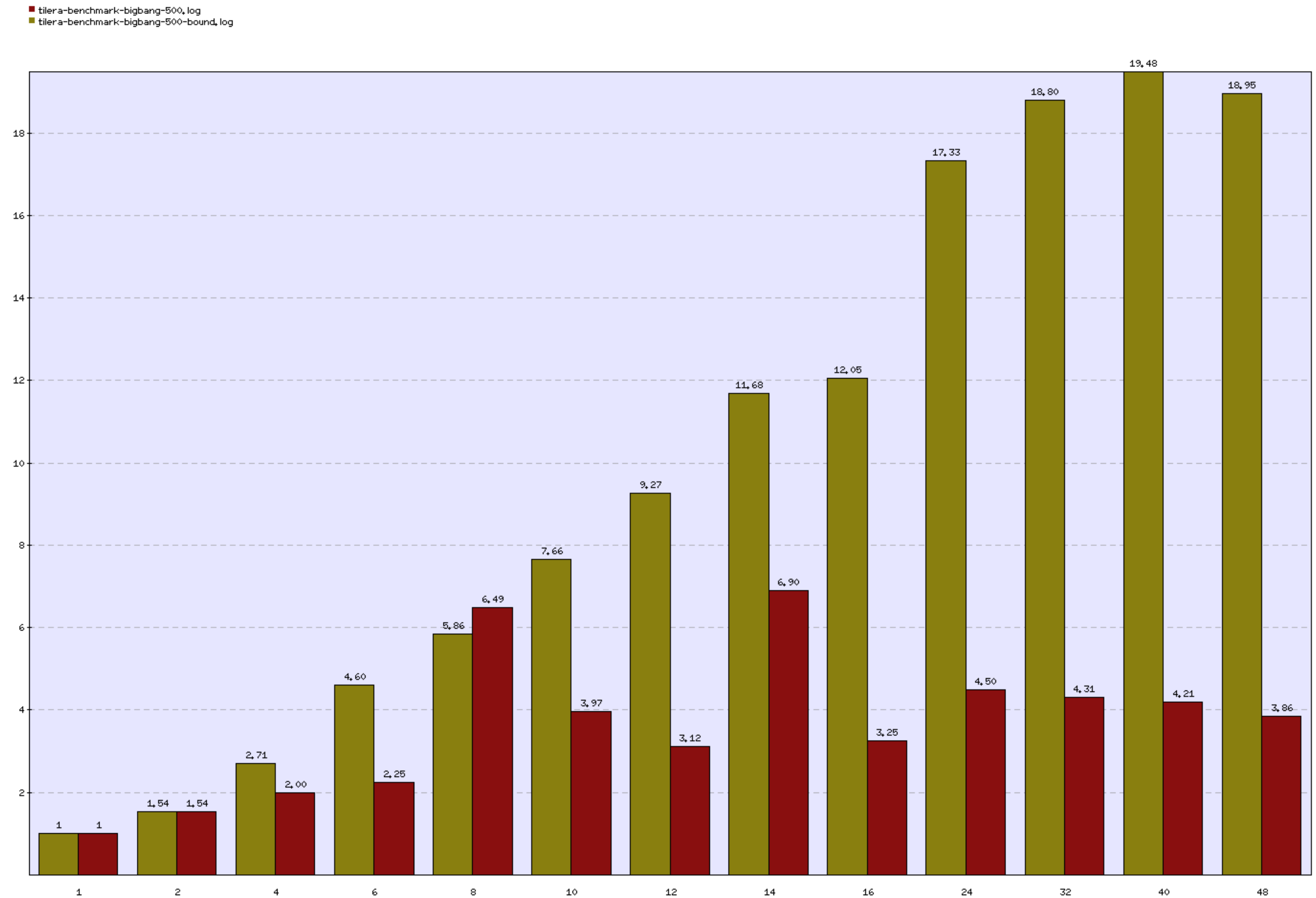
A dual quad core processor system

Multi-core tips and tricks (3)

Here is a dual processor (each processor in different numa nodes) quad-core with hyper-threads.

```
1> erlang:system_info(cpu_topology).
[{node,[{core,[{thread,{logical,0}},{thread,{logical,1}}]},
        {core,[{thread,{logical,2}},{thread,{logical,3}}]},
        {core,[{thread,{logical,4}},{thread,{logical,5}}]},
        {core,[{thread,{logical,6}},{thread,{logical,7}}]}]},
 {node,[{core,[{thread,{logical,8}},{thread,{logical,9}}]},
        {core,[{thread,{logical,10}},{thread,{logical,11}}]},
        {core,[{thread,{logical,12}},{thread,{logical,13}}]},
        {core,[{thread,{logical,14}},{thread,{logical,15}}]}]}]
```

Benchmark showing the positive effect if binding the schedulers (Tilera Pro 64 cores)



Next steps with SMP and Erlang

Some known bottlenecks to address

- Improved handling of process table
- Separate allocators per scheduler
- Delayed dealloc (let the right scheduler do it)
- Use NUMA info for grouping of schedulers
- Separate poll sets per scheduler (IO)
- Support Scheduler bindings, cpu_topology on Windows.
- Dynamically linked in BIF's (for C-code , easier to write and more efficient than drivers)
- Optimize Erlang applications in Erlang/OTP
- Fine grained parallelism, language and library functions.
- Better and more benchmarks

ERICSSON

TAKING YOU FORWARD