

Erlang Multicore support

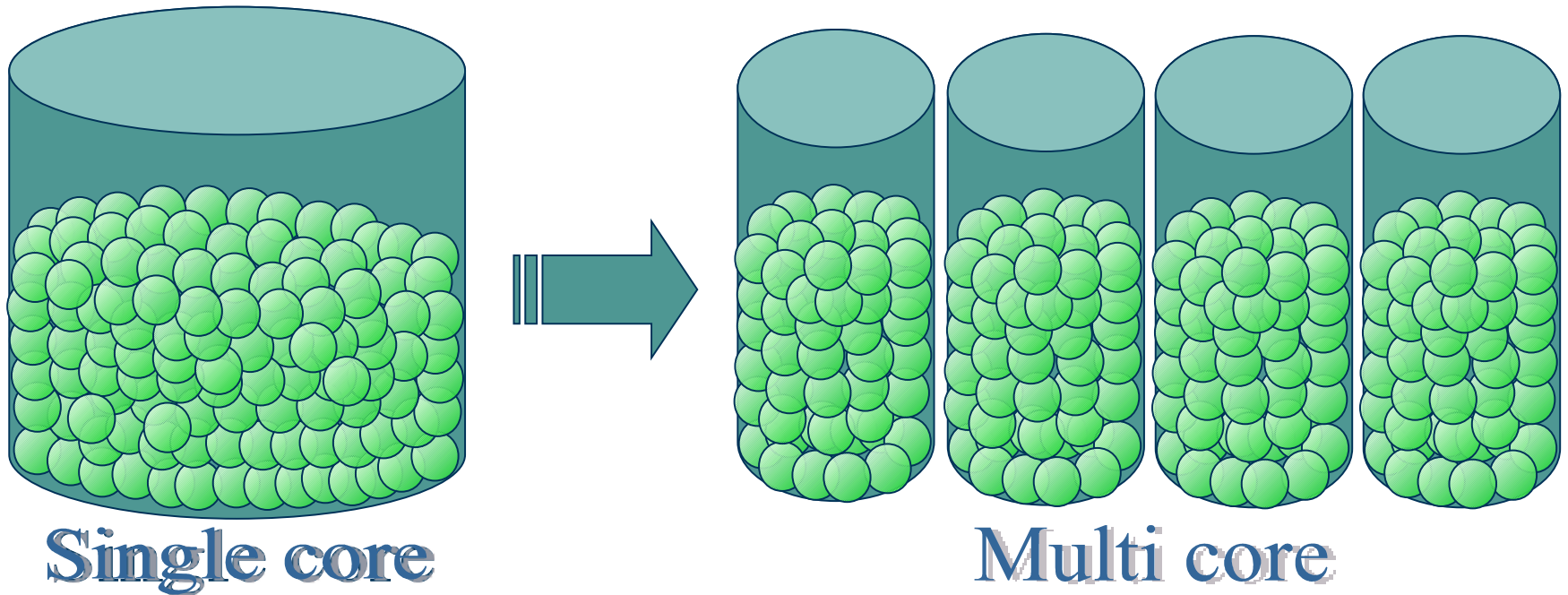
Behind the scenes

Erlang VM (BEAM) when we started

- Virtual register machine which scheduled light weight processes
 - One single process scheduler and one queue per priority level
 - Switching between I/O operations and process scheduling
- I/O drivers and “built in functions” (native functions) had exclusive access to the data structures
 - Network code
 - ETS tables
 - Process inspection etc

Perfect program for using multicore

- A lot of small units of execution
- The parallel mindset has created applications just waiting to be spread over several physical cores



Conversion steps

- Multiple schedulers
- Parallel I/O
- Parallel memory allocation
- Multiple run-queues and generally less global locking

Multiple schedulers

- Tools
 - Locking order and lock-checker
 - Ordinary test cases
 - Benchmarks (synthetic)
- Techniques
 - Own thread library (Uppsala University)
 - Lock tables and custom lock implementation for processes
 - Lots of conventional mutexes
- Result
 - One scheduler per logical core
- Insights
 - You will have to make memory/speed tradeoffs
 - Lock order enforcement is very helpful

Parallel I/O

- Tools
 - More simple benchmarks
 - Customer systems
 - Intuition (or – the problem was obvious...)
- Techniques
 - More fine granular locking
 - Locking on different levels depending on I/O driver implementation
 - Scheduling of I/O-operations
- Result
 - Real applications parallel...
- Insight
 - Doing things at the right time can vastly reduce complexity

Multiple allocators

- Tools
 - Even more benchmarks
 - vTune (Intel-specific)
 - Thread profiler (Intel-specific)
- Techniques
 - Each scheduler has it's own instance of memory allocators
 - The “malloc” implementation was already our own
 - Locks are still needed as one scheduler might free another schedulers memory
- Result
 - Greatly improved performance for CPU intense applications
- Insight
 - Not only execution has to be distributed over cores

Multiple run-queues and generally less global locking

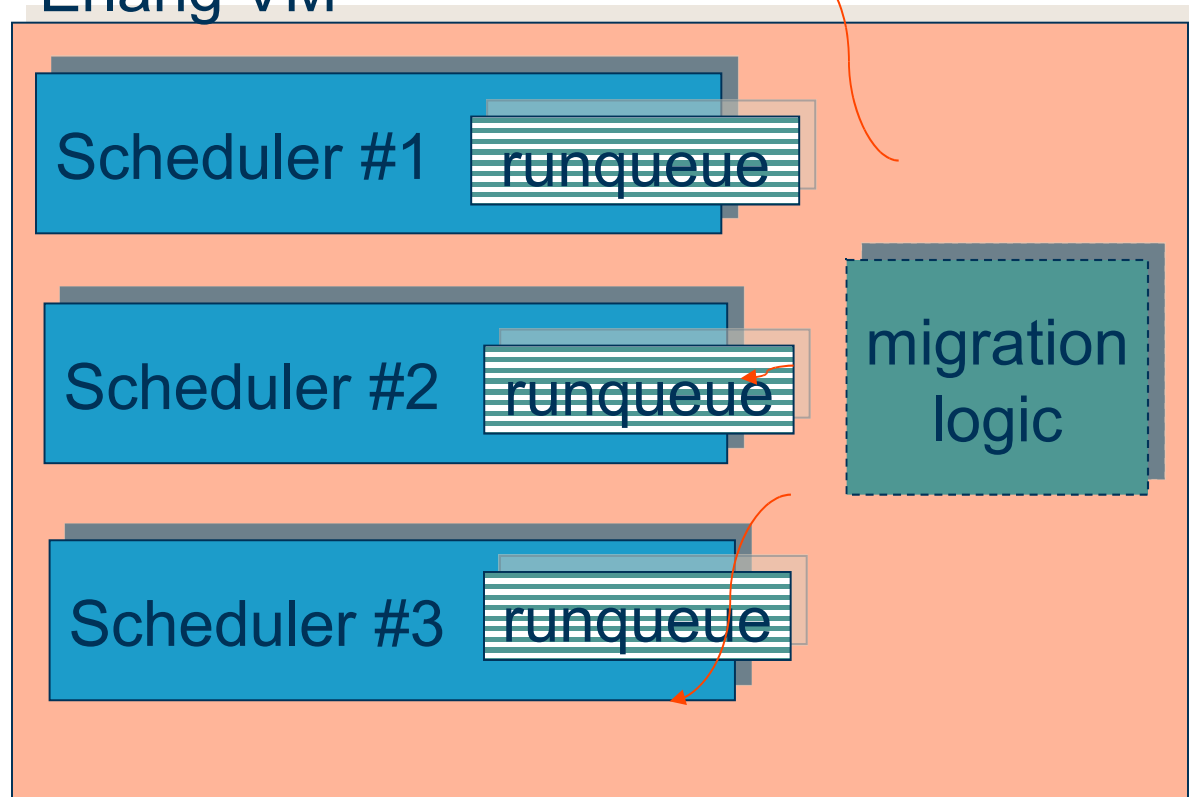
- Tools
 - Custom lock counting implemented (not cpu-specific)
 - More massive multicore CPU's to test on (Tilera, Nehalem)
 - More customer code from more projects
- Techniques
 - Distributing data over the schedulers
 - Load balancing at certain points
 - More fine granular locking (ETS Meta- and shared tables)
 - Reimplementation of distribution marshaling to remove need for sequential encode/decode
- Results
 - Far better performance on massive multicore systems
 - Nehalem performance great, but core2 still problematic
- Insight
 - No global lock will ever fail to create a bottleneck



Multiple runq-ueues

Erlang SMP VM (R13)

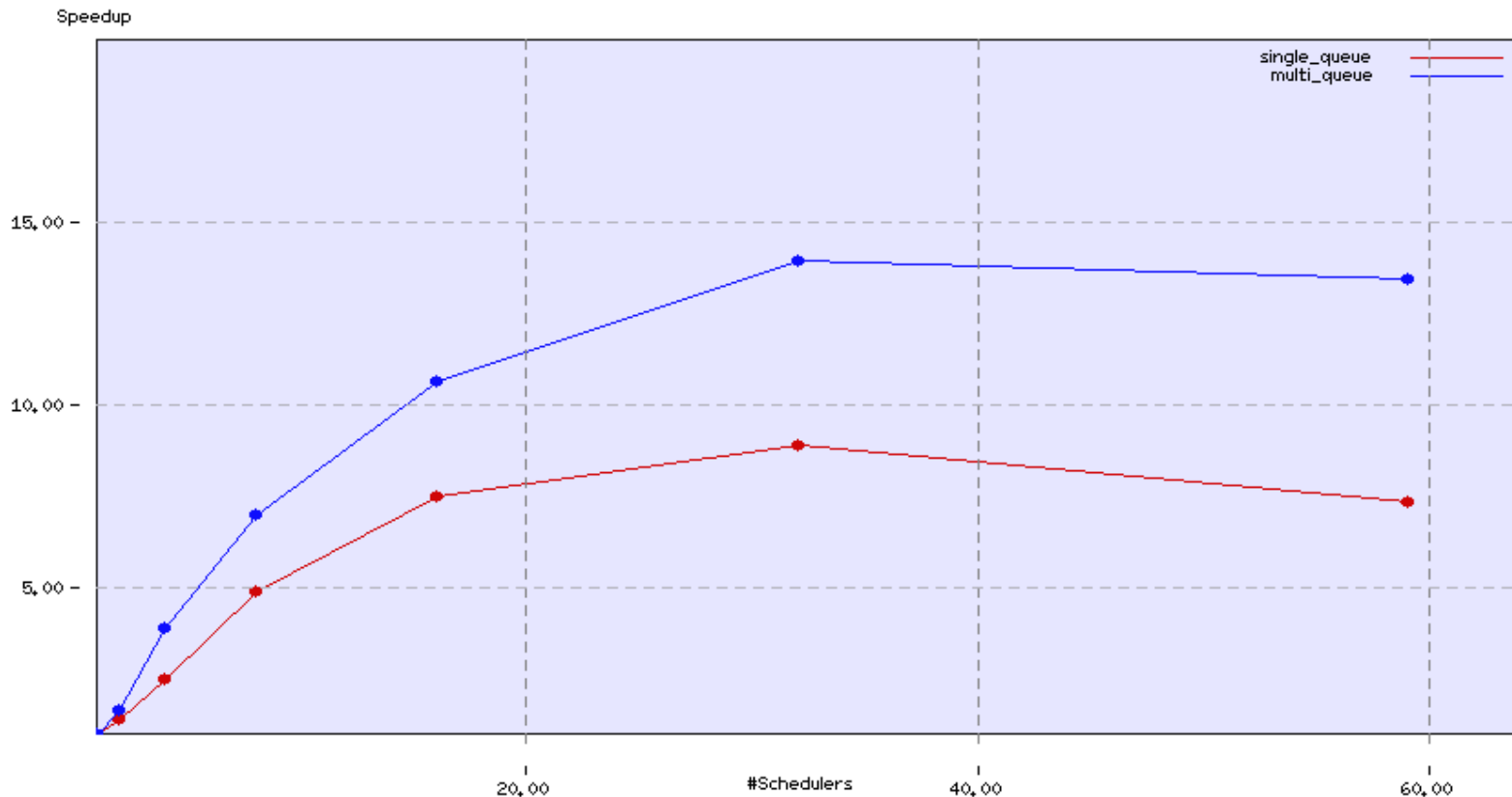
Erlang VM



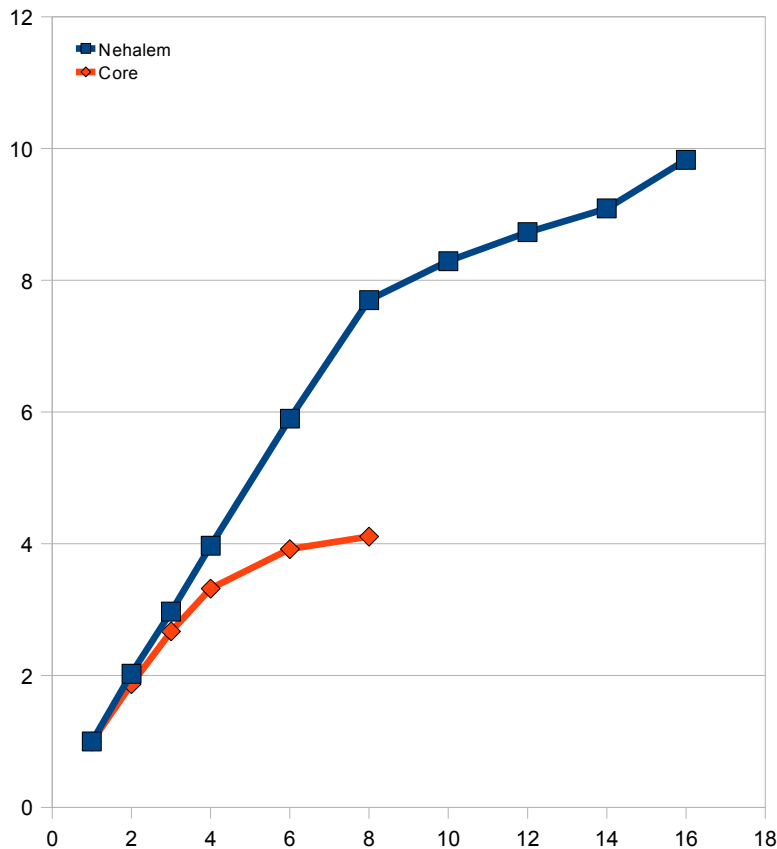
Migration logic

- Keep the schedulers equally loaded
- Load balancing after a specific amount of reductions
 - Calculate an average run-queue length
 - Setup migration paths
- Each scheduler migrates jobs until happy
- If a scheduler has suddenly no job, it starts stealing work
- If, at load-balancing, the logic detects too little work for the schedulers, some will be temporarily inactivated

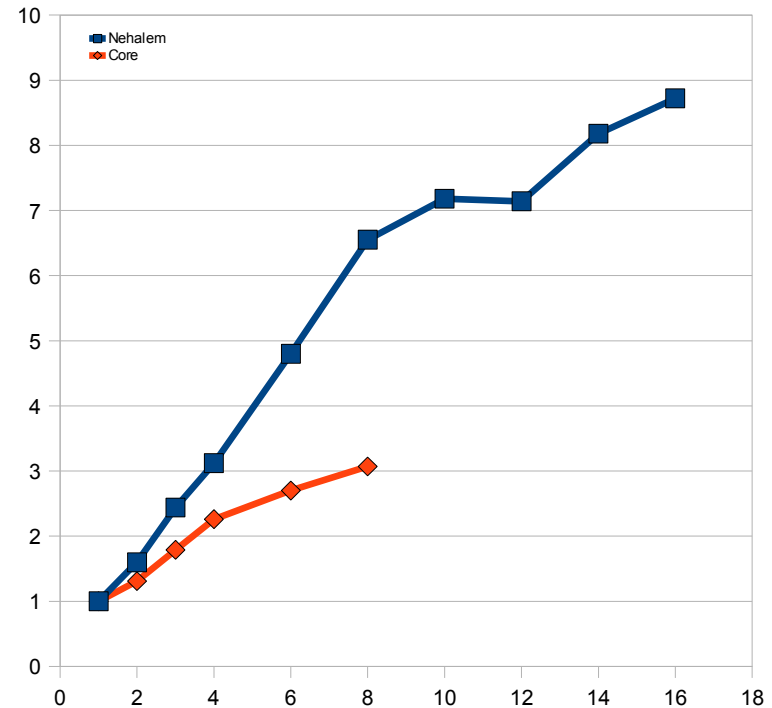
Example of performance gain w/ multiple run-queues in TilePro64



Comparing “Clovertown” Xeon E5310 to “Gainstown” Xeon X5570



Memory/CPU



Interprocess communication

Insights

- No global lock ever goes unpunished
- Data as well as execution has to be distributed over cores
 - Malloc and friends will be a bottleneck
- You will have to make memory/speed tradeoffs
- New architectures will give you both new challenges and performance boosts
 - Revise and rewrite as processors evolve
- Doing things (in the code) at the right time can reduce complexity as well as increase performance
- Take the time to use third party tools and to write your own.
- Work incrementally

Tools we've used

- Lock checker (implemented in VM) and strict locking order
- vTune and thread profiler
- oProfile
- Lock counter (implemented in VM)
- Acumem (www.acumem.com)
- Valgrind
- Benchmarks
 - Customers
 - Open Source
- Percept (Erlang application parallelism measurement tool)

What now?

- Non uniform memory access
 - Schedulers private memory near core
 - Distribute processes smarter, taking memory access into account
 - ...
- Delayed deallocation to avoid allocator lock conflicts
 - Especially important for Core systems
- Developing our libraries
- Using less memory?
- More measuring, benchmarking, customer tests...

ERICSSON



TAKING YOU FORWARD