Inside the Erlang VM

with focus on SMP

Prepared by Kenneth Lundin, Ericsson AB
Presentation held at
Erlang User Conference,
Stockholm, November 13, 2008

1 Introduction

The history of support for SMP (Symmetrical Multi Processor) in Erlang started around 1997-1998 as a master thesis work by Pekka Hedqvist with Tony Rogvall (Ericsson Computer Science Lab) as supervisor.

The implementation was run on a Compaq with 4 Pentium Pro 200 Mhz CPU's (an impressive machine in those days) and showed a great potential for scalability with additional processors but suffered from bad IO performance.

The work with SMP did not continue at that time since it was so easy to increase performance by just upgrading the HW to the newest processor. There simply was no business case for it at the time.

The SMP work was restarted at 2005 and now as part of the ordinary development. The work was driven by the Erlang development team at Ericsson with participation and contributions from Tony Rogvall (then at Synapse) and the HiPE group at Uppsala University.

The strategy was (and still is):

- First, ”make it work”
- Second, ”measure” and find bottlenecks
- Third, ”optimize” by removing bottlenecks

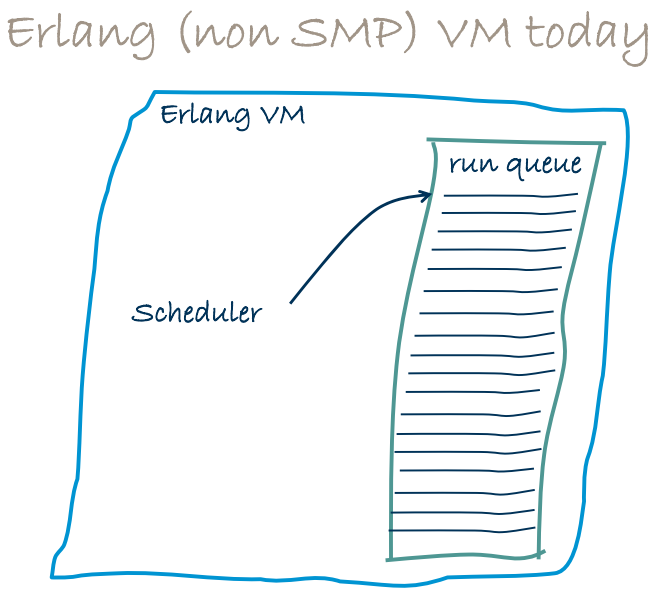
The first release of a stable runtime system with support for SMP came in OTP R11B in May 2006.

This ended the first cycle of the strategy and a new iteration with “measure”, “optimize” and “make it work” started. Read more about it in the next pages.

2 How it works

2.1 Erlang VM with no SMP support

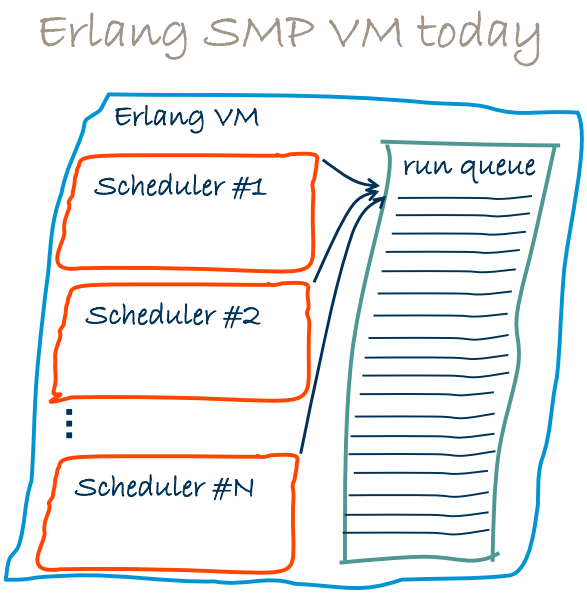
The Erlang VM without SMP support has 1 scheduler which runs in the main process thread. The scheduler picks run able Erlang processes and IO-jobs from the run-queue and there is no need to lock data structures since there is only one thread accessing them.



2.2 Erlang VM with SMP support (in R11B and R12B)

The Erlang VM with SMP support can have 1 to 1024 schedulers which are run in 1 thread each.

The schedulers pick run able Erlang processes and IO-jobs from one common run-queue. In the SMP VM all shared data structures are protected with locks, the run-queue is one example of a shared data structure protected with locks.



2.2.1 First release for use in Products, March 2007

Measurements from a real telecom product showed a 1.7 speed improvement between a single and a dual core system.

It should be noted that it took only about a week to port the telecom system to a new OTP release with SMP support, to a new Linux distribution and to a new incompatible CPU architecture, the Erlang code was not even recompiled.

It took a little longer to get the telecom system in product status, a few minor changes was needed in the Erlang code because Erlang processes now can run truly parallel which changes the timing and ordering of events which the old application code did not count for.

The performance improvements achieved on a dual core processor for a real telecom system where encouraging and after that several other telecom systems have also taken benefit from the SMP support in Erlang.

2.2.2 SMP in R12B

From OTP R12B the SMP version of the VM is automatically started as default if the OS reports more than 1 CPU (or Core) and with the same number of schedulers as CPU's or Cores.

You can see what was chosen at the first line of printout from the `erl` command. E.g.

```
Erlang (BEAM) emulator version 5.6.4 [source] [smp:4] .....
```

The `[smp:4]` above tells that the SMP VM is run and with 4 schedulers.

The default behaviour can be overridden with the "`-smp [enable|disable|auto]`" `auto` is default and to set the number of schedulers, if smp is set to `enable` or `auto` use "`+S Number`" where `Number` is the number of schedulers (1..1024)

Note ! It is normally nothing to gain from running with more schedulers than the number of CPU's or Cores.

Note2 ! On some operating systems the number of CPU's or Cores to be used by a process can be restricted with commands. For example on Linux the command "`taskset`" can be used for this. The Erlang VM will currently only detect number of available CPU's or Cores and will not take the mask set by "`taskset`" into account. Because of this it can happen and has happened that e.g. only 2 Cores are used even if the Erlang VM runs with 4 schedulers. It is the OS that limits this because it take the mask from "`taskset`" into account.

The schedulers in the Erlang VM are run on one OS-thread each and it is the OS that decides if the threads are executed on different Cores. Normally the OS will do this just fine and will also keep the thread on the same Core throughout the execution.

The Erlang processes will be run by different schedulers over time because they are picked from a common run-queue by the first scheduler that becomes available.

3 Performance and scalability

The SMP VM with only one scheduler is slightly slower (10%) than the non SMP VM. This is because the SMP VM need to use locks for all shared datastructures. But as long as there are no lock-conflicts the overhead caused by locking is not that high (it is the lock conflicts that takes time).

This explains why it in some cases can be more efficient to run several SMP VM's with one scheduler each instead on one SMP VM with several schedulers. Of course the running of several VM's require that the application can run in many parallel tasks which has no or very little communication with each other.

If a program scale well with the SMP VM over many cores depends very much on the characteristics of the program, some programs scale linearly up to 8 and even 16 cores while other programs barely scale at all even on 2 cores.

This might sound bad, but in practice many real programs scale well on the number of cores that are common on the market today, see below.

Real telecom products supporting a massive number if simultaneously ongoing "calls" represented as one or several Erlang processes per core have shown very good scalability on dual and quad core processors.

Note, that these products was written in the normal Erlang style long before the SMP VM and multi core processors where available and they could benefit from the Erlang SMP VM without changes and even without need to recompile the code.

4 Our strategy with SMP

Already from the beginning when we started implementation of the SMP VM we decided on the strategy:

"First make it work, then measure, then optimize".

We are still following this strategy consistently since the first stable working SMP VM that we released in May 2006 (R11B).

Another important part of the strategy is to hide the problems and awareness of SMP execution for the Erlang programmer. Erlang programs should be written as usual using processes for parallel tasks, the utilization of CPUs and cores should be handled by the Erlang VM. It must be easy and cost effective to utilize multicore and SMP HW with Erlang this is one of our absolute strengths compared to other programming languages.

There will be new BIF’s for SMP related stuff but we try to avoid that as much as possible. We think it is preferable to configure SMP related things such as number of cores to use, which cores to use on the OS level and as parameters to the Erlang VM at startup.

The principle is that an Erlang program should run perfectly well on any system no matter what number of cores or processors there are.

5 Next steps with SMP and Erlang

There are more known things to improve and we address them one by one taking the one we think gives most performance per implementation effort first and so on.

We are now putting most focus on getting consistent better scaling on many cores (more than 4).

The SMP implementation is continually improved in order to get better performance and scalability. In each service release R12B-1, 2, 3, 4, 5 , ..., R13B-0, 1, …, R14B etc. you will find new optimizations.

5.1 Some known bottlenecks

Below some of the most significant bottlenecks that we know of are described, there are for sure more bottlenecks than this and we intend to address them one after one. It is worth noting that after removal of one bottleneck there might be new ones coming up and the already known ones may have got changed importance.

5.1.1 The common run-queue

The single common run-queue will become a dominant bottleneck when the number of CPU's or Cores increase.

This will be visible from 4 cores and upwards, but 4 cores will probably still give ok performance for many applications.

We are working on a solution with one run-queue per scheduler as the most important improvement right now. Read more about this later in the document.

5.1.2 Ets tables

Ets tables involves locking. Before R12B-4 there was 2 locks involved in every access to an ets-table, but in R12B-4 the locking of the meta-table is optimized to reduce the conflicts significantly (as mentioned earlier it is the conflicts that are expensive).

If many Erlang processes access the same table there will be a lot of lock conflicts causing bad performance especially if these processes spend a majority of their work accessing ets-tables.

The locking is on table-level not on record level. An obvious solution is to introduce more fine granular locking.

Note! that this will have impact on Mnesia as well since Mnesia is a heavy user of ets-tables.

5.1.3 Message passing

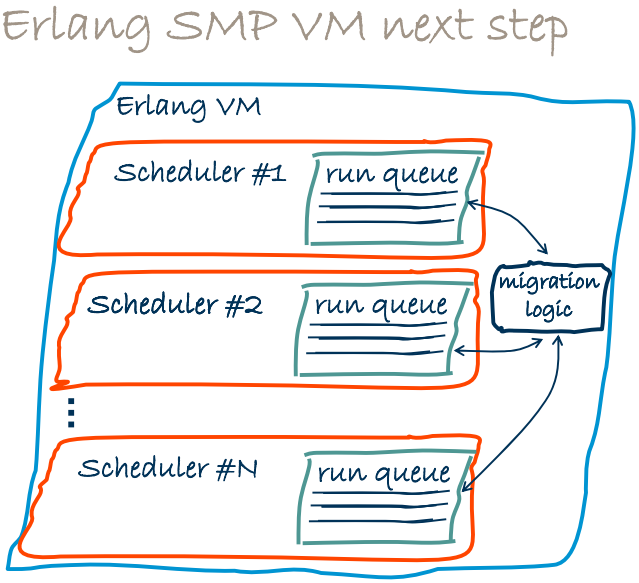
When many processes are sending messages to the same receiving process there will be a lot of lock conflicts. There are ways to optimize this by reducing the amount of work being done while having the lock.

5.1.4 A process can block the scheduler

If a process is blocked waiting to get a lock for example to access an ets-table the whole scheduler is blocked doing nothing until the lock is accuired and the process can continue it's execution. This can be improved by introducing what we call "*process level locking"* which means that if a process is blocked waiting to get a lock it will be scheduled out and the scheduler will schedule in the next process from the run-queue instead. We have already implemented and measured on this solution and concluded that it probably can be introduced when the separate run-queues are in place. We still need to verify that it does not degrade performance for certain special cases.

5.2 Separate run-queues per scheduler

The next big performance improvement regarding SMP support in the Erlang runtime system is the change from having one common run-queue to having a separate run-queue per scheduler. This change will decrease the number of lock conflicts dramatically for systems with many cores or processors. The improvement in performance will in many applications be significant already from 4 cores and will of course be even more noticeable in systems with 8, 16 or even more cores.



5.2.1 Migration logic

When there are separate run-queues per scheduler the problem is moved from the locking conflicts when accessing the run-queue to the migration logic which must be both efficient and reasonably fair.

The implementation we have so far will need a lot more benchmarking and fine tuning before it works optimally. It works roughly like this:

The maximum number of run able processes over all schedulers is measured approximately 4 times per second. This value divided by number of schedulers is then used to trigger migration of processes from one scheduler to another scheduler.

When a scheduler is about to schedule in a new process it will first check if its number of run able processes is above the max value described above and if it is it will migrate the process to another scheduler according to the migration path set up.

There are also 2 other occasions in addition to the “schedule in” of a new process when a process migration can occur:

1. If a scheduler gets out of jobs it will steal jobs from other schedulers.

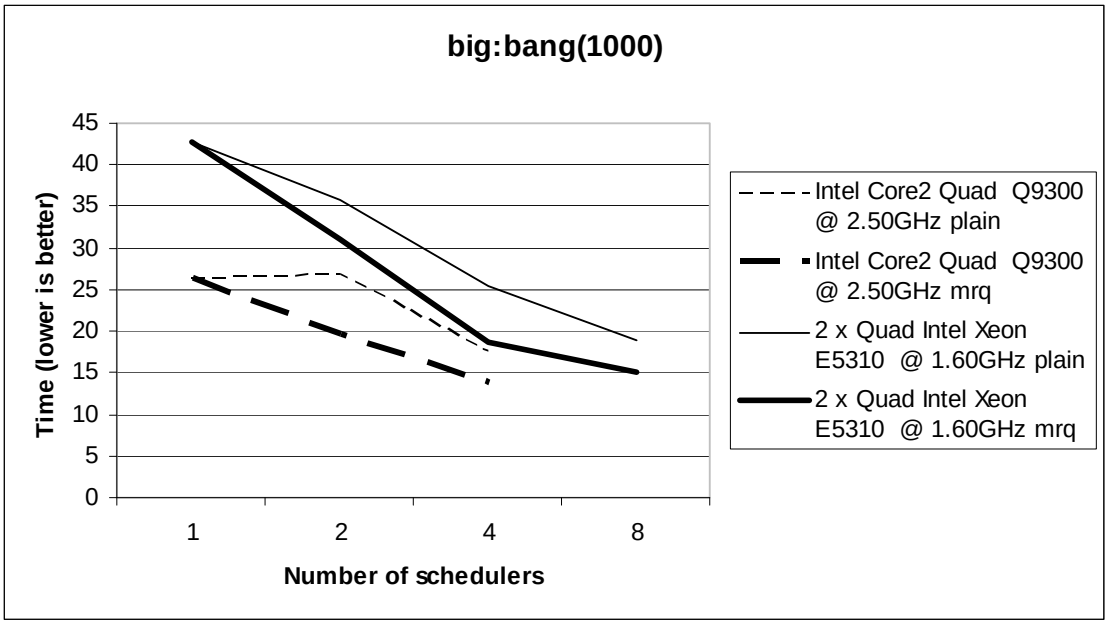
2. Underloaded schedulers will also steal jobs from heavily overloaded schedulers in their migration paths.

Below follows some measurements that show early indications of the improvements the system with separate run-queues per scheduler and the migration logic described above will give.

The graph below shows the results from running the benchmark “big bang” with 1, 2, 4, 8 schedulers on both the current system with one single run-queue and on the next to come system with multiple run-queues one per scheduler.

The benchmark spawns 1000 processes which all sends a ‘ping’ message to all other processes and answer with a ‘pong’ message for all ‘ping’ it receives.

The “fat” lines in the graph shows the multiple run-queue case and as can be seen the improvement is significant.



6 Frequently Asked Questions

6.1 Is there any difference in the .beam file depending on if it should run in a SMP or non SMP system?

As long as the module is not compiled with “native” option with a HiPE enabled system the .beam files are the same and can be run in both SMP and non SMP systems.

6.2 Can an Erlang process be locked to a specific processor core?

An Erlang process can not be locked to a specific processor by the programmer and this is intentional. In a future release it might be possible to lock a scheduler to a specific core.

6.3 What is the relation between asynch threads and SMP?

The asynch thread pool has nothing with SMP todo. The asynch threads are only used by the file driver and by user written drivers that specifically uses the thread pool. The file driver uses this to avoid locking of the whole Erlang VM for a longer time period in case of a lengthy file operation. The asynch threads was introduced long before the SMP support in the VM and works for the non SMP VM as well. In fact the asynch threads are even more important for a non SMP system because without it a lengthy file operation will block the whole VM.