Effective parallelization of Monte Carlo calculations for the solution of certain types of problems on multi-core systems using Erlang and other functional programming languages

Brett Cameron

March 2016

Abstract

Modern Monte Carlo methods were devised in the 1940s as a means of solving certain types of mathematical problems that could not be readily evaluated using conventional numerical methods. Since this time these methods have evolved into a useful and powerful tool for the solution of a wide variety of problems spanning many disciplines, including the physical sciences, computer science, engineering, and finance/business, and gaming. In more recent times Monte Carlo methods have found applicability to certain classes of Big Data problems where processing can be efficiently distributed and parallelised across multiple processors or multiple cores. In this talk the speaker will discuss some of these problems and will describe how functional programming languages such as Erlang are well-suited to the solution of these problems and to the efficient exploitation of multiple cores.

About me

Brett Cameron works as a senior software engineer at VMS Software (http://www.vmssoftware.com), helping to define and implement the company's Open Source strategy for the OpenVMS operating system. Before joining VSI Brett worked as a senior software architect with HP's Cloud and Enterprise Services groups. Brett lives in Christchurch, New Zealand and has worked in the software industry since 1992, and in that time he has gained experience in a wide range of technologies, many of which have long since been retired to the software scrapheap of dubious ideas. Over the past decade Brett has spent considerable time travelling the world helping organisations to modernize their legacy application environments and to better leverage Open Source technologies. In more recent times, his involvement with various Open Source projects and his work in the cloud computing space has caused him to develop a liking for functional programming languages, and Erlang in particular, which he has ported to several exotic operating systems such as OpenVMS. Brett holds a doctorate in chemical physics from the University of Canterbury, and maintains close links with the University, delivering guest lectures and acting as an advisor to the Computer Science and Electronic and Computer Engineering departments on course structure and content. In his spare time Brett enjoys listening to music, playing the guitar, and drinking beer.

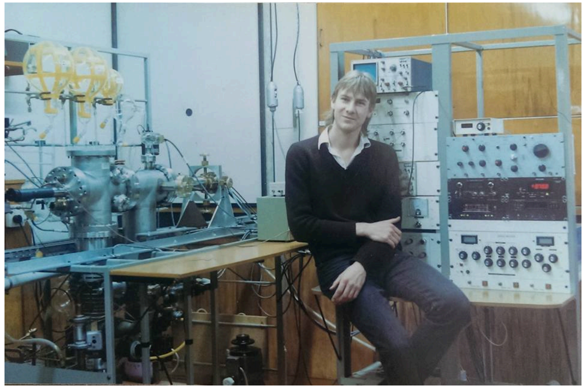
AGENDA

- **Introduction and background**
- Introduction to Monte Carlo methods
- Some terminology
- A simple experiment
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- Questions

Introduction and background

A long time ago at a university far, far away…

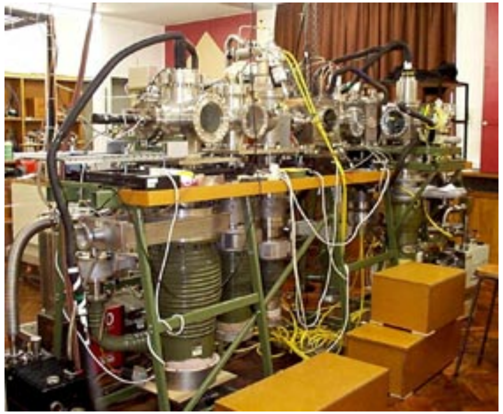
- A much younger me was embarking on his PhD in chemical physics (1988)
  - "*Investigations of the Flow Dynamics of Supersonic Molecular Beams and the Ionization of Molecular Clusters by Electron Impact*"
  - Involved the construction of a supersonic molecular beam machine for the investigation of various chemical reaction processes
    - Primarily ion-molecule/atom and electron-molecule/atom reactions



Introduction and background

- The molecular beam machine was a rather complex and temperamental apparatus
  - A lot of downtime waiting for components to arrive or to be built (or fixed)
  - When it was working it could take several days to establish a good enough vacuum (better than 10,000,000th of an atmosphere) for experiments to proceed

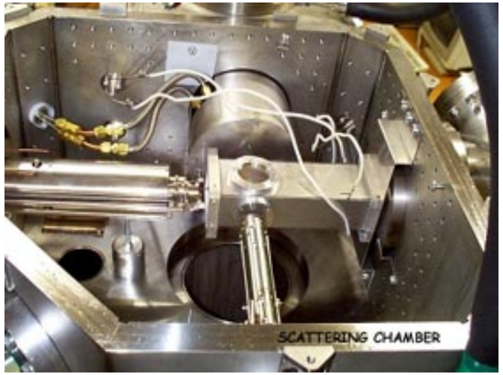
- So with all this downtime time I figured that I'd try to write some programs simulate the experiment
  - Molecular dynamics simulations
  - Fortran 77 running on VAX/VMS, single CPU, very small RAM (26MB in the 8600 class systems), batch processing…
  - Calculations not particularly amenable to standard numerical methods
    - Many variables
    - Multi-dimensional integrals, evil-looking differential equations, …
  - Could use arrays to hold discrete sets of values for each variable and loop over all permutations
    - Limited memory restricted maximum array sizes and therefore accuracy
    - ...
  - What to do?
    - Off to the library (a thing with books in it)...
    - Stumbled across a couple of books on Monte Carlo methods
    - Problem solved
    - PhD nailed… eventually

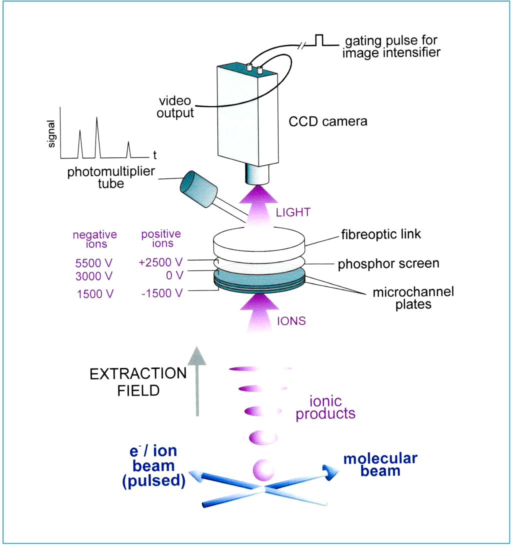


Introduction and background

The molecular beam machine revisited…

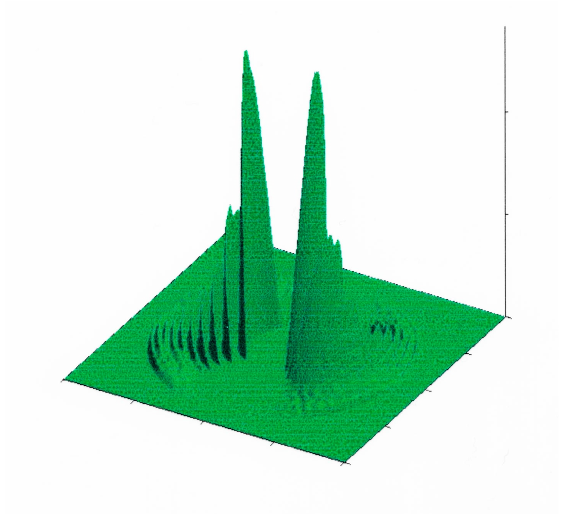
- Postdoctoral research, 2000-2001
- Ion-imaging
  - A cunning technique that allows the spatial distribution of a reaction product to be directly visualized
  - With a bit of clever mathematics and some image processing it is possible to recover the entire 3-D angular and velocity distributions of the product in a single measurement

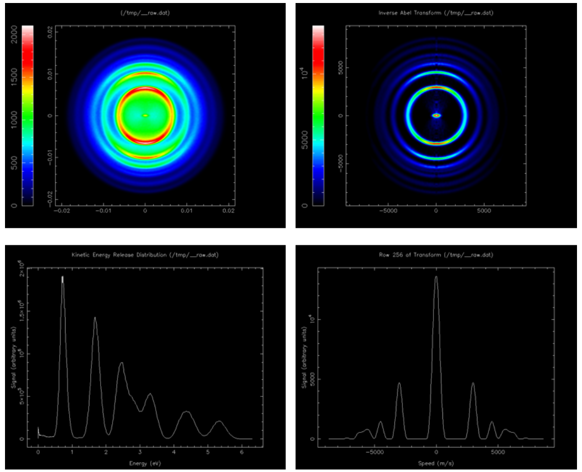




Introduction and background

- Used Monte Carlo simulations to validate experimental results and help with refinement of the experiment
  - C/C++ on Linux, single CPU, 128MB RAM
  - Still took a long time





AGENDA

- Introduction and background
- **Introduction to Monte Carlo methods**
- Some terminology
- A simple experiment
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- Questions

Monte Carlo methods

Putting it simply, Monte Carlo algorithms are randomized algorithms… computational experiments. They can be used to yield approximations where analytical computation is not tractable.

- Branch of experimental mathematics concerned with the approximate solution of mathematical and physical problems by the simulation of random quantities
  - The solution being some parameter taken from a hypothetical population constructed using a random sequence of numbers

- In essence…
  - Assume a known world
  - Assume something about the statistical distribution of certain events within that world
  - Numerically draw many realisations from this distribution (or distributions)
  - Calculate the outcome for each realisation/trial
  - Analyse the results

- A universal technique for the solution of mathematical and physical problems
  - Simulation of natural phenomena
  - Simulation of experimental apparatus
  - Numerical analysis
  - ...

<ins>Trivial example</ins>: What is the probability that 10 dice throws add up exactly to 32?

<ins>Exact solution</ins>:

- Calculate exactly by counting all possible ways of making 32 from 10 dice

<ins>Approximate (Monte Carlo) way</ins>:

- Simulate throwing the 10 dice *N* times (*N* being suitably large), count the number of times the results add up to 32, and divide this by *N*
- Can get quite close to the correct answer quite quickly; universally applicable approach

Monte Carlo methods

- Long been recognized as an extremely powerful technique for the solution of many mathematical problems
  - Evaluation of integrals of arbitrary functions of any number of dimensions
  - Predicting future values of stocks, simulating financial risk, …
  - Solving partial differential equations
  - Image enhancement
  - Molecular dynamics simulations
  - Bayesian analyses of large data sets (certain types of Big Data problems)
  - …
- Facilitate analysis of a range of problems that are not directly tractable to an analytical solution
  - Problems too complicated to be dealt with by a somewhat more classical approach

- Makes possible
  - The simulation of any process that is influenced by random factors
  - The solution of any mathematical problem involving absolutely no chance whatsoever for which it is possible to artificially devise a probabilistic description of the problem

- Applicable to both probabilistic and deterministic problems

History

- The name and the systematic development of Monte Carlo methods dates from around 1944, although the theoretical basis of the method has long been known

- One of the earliest real Monte Carlo calculations was that performed in 1777 by French mathematician Georges-Louis Leclerc, Comte de Buffon
  - The value of π was inferred from the number of times a needle of length *l* thrown in a random fashion onto a board ruled with parallel straight lines separated by a distance *l* hits or misses a line
  - After a large number of trials, π is estimated by twice the total number of trials divided by the number of hits

The formulation of Buffon's needle problem is based on the simple fact that the probability of a hit is exactly $2/\pi$, which can be explained as follows:

- If the angle between the needle and the perpendicular to the parallel lines is equal to ϑ, then the projection of the needle onto this perpendicular has a length of $l|\cos\vartheta|$, and for a given angle ϑ, the probability of a hit is clearly the ratio of this length to the length *l* separating the lines – that is $|\cos\vartheta|$. Since all angles are equally likely, the average value of $|\cos\vartheta|$ can be calculated by integrating it over its range and then dividing by the range. By symmetry it is sufficient to integrate over the one quadrant from 0 to $\pi/2$, where the integral is unity, and the probability is therefore $2/\pi$ as required.

History

- Derives its name from the city of Monte Carlo in the principality of Monaco, famous for its casino

- The name was first used by American mathematician John von Neumann during the Second World War as the code name for his secret work on the atomic bomb at Los Alamos Scientific Laboratory
  - This work involved the simulation of various problems concerned with random neutron diffusion in fissile material
  - Represented the first real use of Monte Carlo methods as a research tool

- These early simulations utilized many refinements of the method that are still in use today
  - Computer algorithms
  - Methods of transforming non-random problems into random forms that would facilitate their solution via statistical sampling
  - ...

- Physicists Maxwell and Boltzmann probably also deserve some credit
  - However they did not formalise the technique
  - Were more interested in the results of their investigations than the rather clever methods they used to obtain them

Since the use of computers became widespread, a great deal of theoretical investigation has been undertaken and practical experience has been gained with Monte Carlo methods. The only feasible limit to Monte Carlo is one of computing power, and the advent of more and faster computers has had a significant impact on adoption and use of Monte Carlo methods and the types and scale of the problems to which they are being applied.

Types of Monte Carlo

- Probabilistic
  - Observe random values chosen in such a way that they directly simulate the physical process of the original problem
    - Values must be chosen with appropriate probability
  - Infer the desired solution from the behaviour of these random values
  - Equates to direct simulation of the physical problem

- Deterministic
  - Replace theory by experiment whenever the former falters
  - Problems that can be formulated in theoretical terms but cannot readily be solved by deterministic means can often be described in terms of some apparently unrelated random process
  - Whereupon it is possible to solve the original deterministic problem by a Monte Carlo simulation of some concomitant probabilistic problem

Random numbers

- The essential feature common to all Monte Carlo computations is that at some point it is necessary to substitute for a random variable a corresponding set of actual values that have the statistical properties of the random variable
  - Random numbers
    - A set of numbers that have no correlation with one another (no defined sequence)
    - No such thing as a single random number

- Computer-generated random numbers come from a fixed deterministic sequence that is highly non-linear so that the numbers appear random
  - Pseudo-random numbers
  - Periodicity is essentially determined by the largest integer that can be represented on a given machine
  - Algorithms generally involve the use of large prime numbers and modulo arithmetic
  - Generally uniformly distributed

- Inherently random processes such as radioactive decay, thermal noise, or cosmic ray arrival times are not a particularly practical source of random numbers for computer-based simulations!

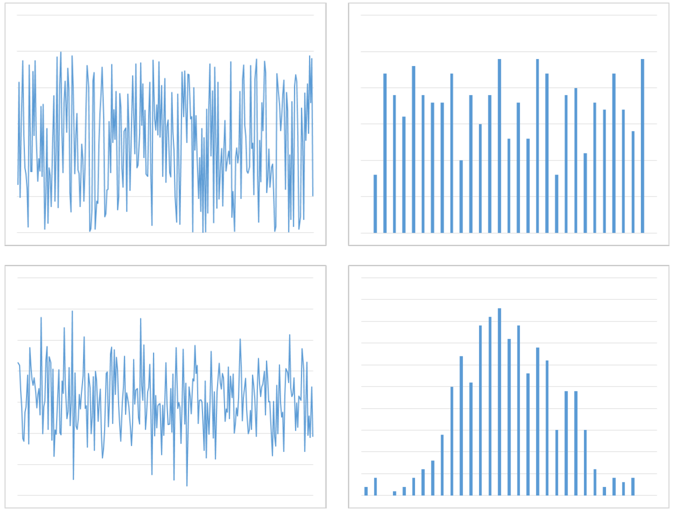
Most pseudo random number algorithms have some state that can be initialized using a seed value. The default state will always generate the same sequence of numbers; specifying different seed values provides a means of performing experiments (simulations) using different sequences (different samples). A common method of seeding is to use the lower-order bits of the system clock.

Generation of uniform random numbers

- Generation of sequences of random numbers generally breaks down into the following two problems:
  - How to generate sequences of random numbers uniformly distributed on some interval
  - How to transform uniformly distributed random numbers into random variables conforming to other probability distributions

- Uniform deviates are simply random numbers that lie within a specified range, with any one number in the range being equally as probable as any other
  - What most people consider random numbers to be

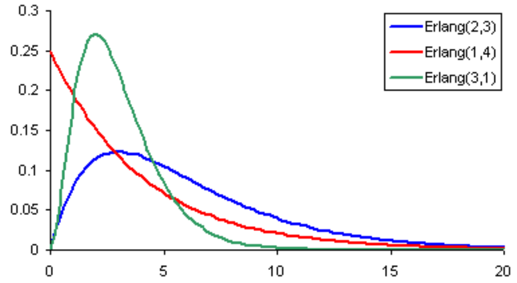
- It is important to distinguish uniform deviates from other sorts of random numbers such as those drawn from a Gaussian distribution
  - These other sorts of deviates are invariably generated by performing appropriate operations on one or more uniform deviates
  - A reliable source of uniform deviates is therefore an essential aspect of any Monte Carlo work


Reconstruction of uniform and Gaussian distributions from respective noise spectra.

Non-uniform random numbers

- The generation of non-uniform distributions is important in many applications where the physical phenomena being investigated are known to follow other distributions
  - Gaussian, exponential, Poisson, binomial, Erlang (!)
  - And many others

- Generating random values for these non-uniform distributions is generally achieved by performing appropriate transformations on uniformly-distributed values
  - Direct functional transformation
  - The principle of compound probabilities (complicated)
  - The rejection method
    - Reject some of the sampled values according to an appropriate test or rule
    - Powerful and quite general
    - Applicable to essentially any distribution



The Erlang distribution was developed by A. K. Erlang to examine the number of telephone calls that might be made at the same time to the operators of the switching stations. This work on telephone traffic engineering has been expanded to consider waiting times in queueing systems in general. The distribution is now used in the fields of stochastic processes and of biomathematics.

Image adapted from http://www.epixanalytics.com/modelassist/AtRisk/Model_Assist.htm#Distributions/Continuous_distributions/Erlang.htm.

Improving accuracy and efficiency

- A key concern in all Monte Carlo work is to obtain a respectably small standard error in the final result with a minimum amount of effort

- Uncertainty can always be reduced by performing more trials
  - There is generally a square law relationship between the error in the final answer and the requisite number of observations/ trails
    - To reduce the error associated with the final answer by a factor of *N* it is necessary to increase the sample size by N^2
    - Slow convergence
  - Often inefficient
    - Variance (noise) decreases slowly
    - Using more samples only removes a small amount of noise

Improving accuracy and efficiency

- In many instances it is possible to change the original problem in such a way that the level of uncertainty in the solution is reduced
  - So-called variance-reducing techniques
  - Reduce the variance without introducing any bias into the estimation; results are more precise without sacrificing any reliability in the results
  - Difficult to generalize
    - Best approach is often very problem-specific
  - Some commonly employed techniques to reduce variance include (there are books on this stuff):
    - Stratified sampling
    - Importance sampling (more samples in important/significant regions of the function)
    - Control variates
    - Antithetic variates
    - ...

Improving accuracy and efficiency

- Parallelism
  - Monte Carlo calculations are often very amenable to parallelism
    - Multiple cores
    - Distributed calculations

  - Find an estimate ~*N* times faster, where *N* is the number of cores/processors
  - Or reduce error by a factor of $N^{1/2}$

  - … hold this thought!

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- **Some terminology**
- A simple experiment
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- Questions

Some definitions

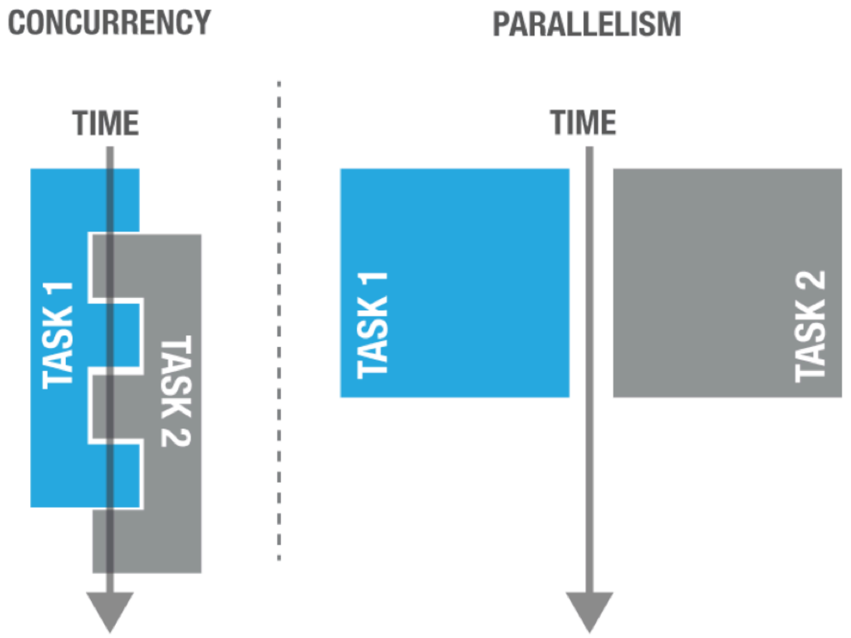
- Concurrency
  - A property of systems in which several computations are executing simultaneously, and potentially interacting with each other

- Parallel processing
  - The simultaneous use of more than one CPU or processor core to execute a program or multiple computational threads
  - Ideally makes programs run faster because there are more engines (CPUs or cores) running it
  - In practice it is often difficult to divide a program in such a way that separate CPUs or cores can execute different portions of the code without interfering with each other

- Bottlenecks
  - A phenomenon where the performance or capacity of an entire system is limited by a single or small number of components or resources

Uses material adapted from http://www.slideshare.net/PeterMilne1/principles-of-high-load-vilnius-january-2015.

Concurrency versus parallelism (sort of)



- In either scenario locks of one form or another are often a necessary evil
  - Increase latency
  - Decrease concurrency
  - Can introduce bottlenecks

- Keeping locks short-lived improves concurrency and reduces latency

Uses material adapted from http://www.slideshare.net/PeterMilne1/principles-of-high-load-vilnius-january-2015.

From an Erlang perspective…

- Erlang's concurrency is based on message passing and the actor model

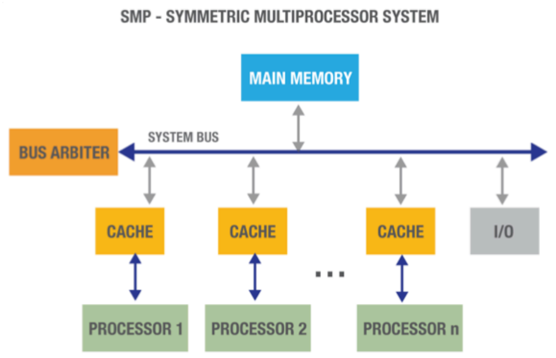
> *"The Actor model adopts the philosophy that everything is an actor. This is similar to the everything is an object philosophy used by some object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent."* (https://en.wikipedia.org/wiki/Actor_model)

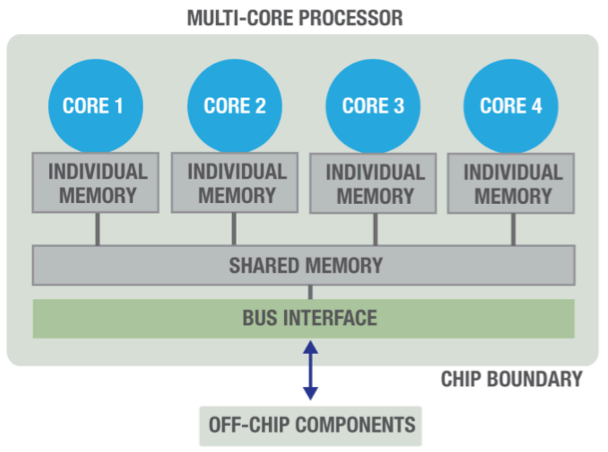
- As Erlang essentially defines them:
  - Concurrency refers to having many actors running independently but not necessarily at the same time
  - Parallelism refers to many actors running at exactly the same time

Multi-processor and multi-core

- Multi-processor
  - More than one processor sharing bus and memory

- Multi-core
  - More than one processor in a chip
  - Each with local Memory
  - Access to shared memory





Uses material adapted from http://www.slideshare.net/PeterMilne1/principles-of-high-load-vilnius-january-2015.

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- Some terminology
- **A simple experiment**
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- Questions

A simple experiment

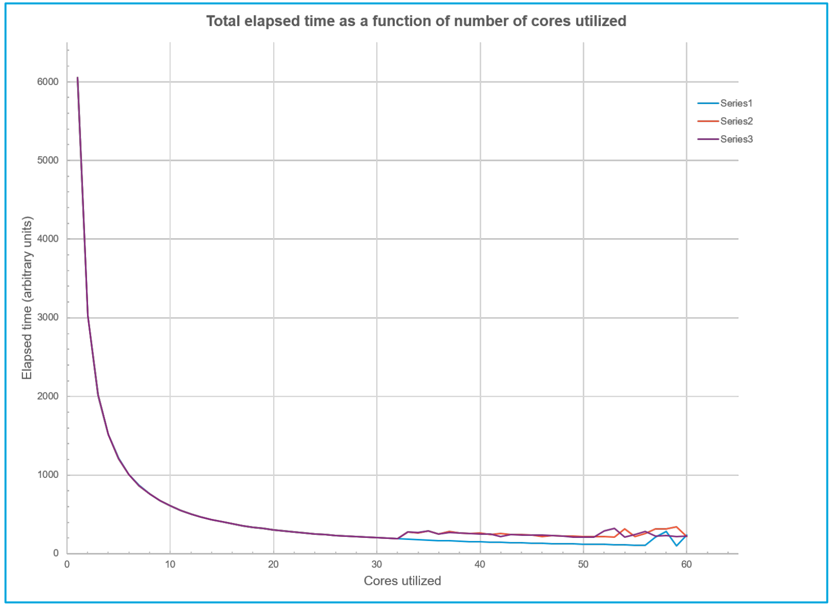
- A trivial example that uses Buffon's Needle Problem to estimate π
  - Calculation can be readily distributed across multiple cores
  - Embarrassingly parallel

- http://joef.co.uk/blog/2009/07/estimating-pi-with-monte-carlo-methods/ provides a nice Erlang implementation of the algorithm
  - Simply need to specify the number of Erlang processes *P* and total number of trials *N*
  - Each process will use a separate core
    - Assuming sufficient schedulers and cores
  - *N/P* trials will be done per core
  - Results from all processes are collated and the estimate is derived

- Tested on a 64-core server with Centos and Erlang/OTP R17
  - Tests done with 1-60 processes
    - Tests also performed using 1-60 processes with only 32 cores enabled and 32 cores plus hyper-threading
  - 100,000,000 total trials for each test
  - Schedulers not bound to cores (more on this topic later)

A simple experiment

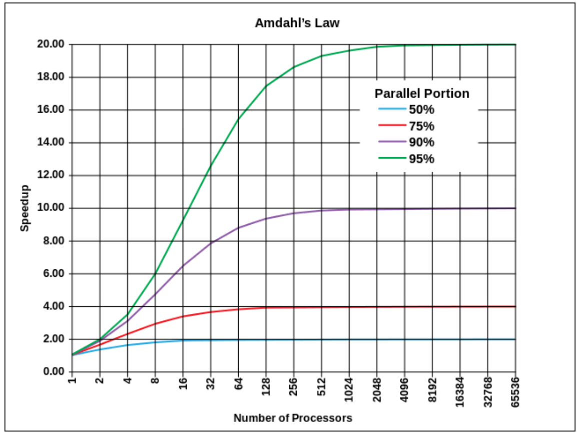
- Each Erlang process can consume 100% of a core ☺
- Blue line shows results for 1-60 processes with 64 cores available
  - As more processes are added, total time reduces largely as expected
    - Doubling the number of processes roughly halves the time, ...
  - Small glitch around 57
    - Probably some synchronization issue
    - Need to look into it
- Red and purple lines show results for 1-60 processes with only 32 cores enabled and 32 cores plus hyper-threading
  - Degradation visible for > 32 processes (as expected)
  - But not bad
    - May not be so fortunate with other programs
- Not exactly a great way to estimate π I suppose, but it does actually work!



```
$ erl +S 32:32
Eshell V6.3  (abort with ^G)
1> timer:tc(pi, start, [32,100000000]).
Started all processes.
Pi: 3.14127752
{5756467,ok}
2>
```

Comments about scalability through parallelism

- Some types of problems scale better than others when distributed across multiple cores or processors
  - Such problems are often referred to as being "embarrassingly parallel"
  - Generally involve number crunching
    - Numerical algorithms, which Erlang often doesn't handle so well
- Some problems are too sequential in nature to benefit from any form of concurrency or parallelism
- Absolute linear scalability is a bit of a pipe dream, but sometimes we can come close
- Amdahl's law comes into play…



The speedup of a program achievable by adding more processors is limited by the serial part of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- Some terminology
- A simple experiment
- **More complicated problems**
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- Questions

Monte Carlo methods and Big Data

- Not really too many good examples of Monte Carlo simulations using Erlang
  - Mostly small test and example programs
  - However Erlang can be a good option
    - Particularly from the perspective of leveraging multi-core and distributed systems

- One potential area of application is Big Data
  - Already something of a precedent for using Erlang in this domain (Riak, CouchDB, BugSense, …)

- Many Big Data problems are too large and complex to be mathematically analysed via conventional means
  - This has led to the use of large-scale statistical Monte Carlo simulations for the solution of Big Data problems
    - Finance industry
      - Risk/reward simulations for thousands of investment portfolios
      - Retirement plan simulations
      - …
    - Analytics on medical imaging data
    - Genome research
    - Predictive maintenance in the manufacturing industry
    - Marketing research

Monte Carlo methods and Big Data

- Interesting research paper by Google: "*Bayes and Big Data: The Consensus Monte Carlo Algorithm*" (2013)
  - See http://research.google.com/pubs/pub41849.html
  - Investigates the applicability of Bayesian Monte Carlo methods (BMC) to certain types of Big Data problems
    - BMC facilitates incorporation of prior knowledge into calculation
    - Generally more efficient that traditional Monte Carlo
    - Not always applicable
      - May not have relevant prior information
    - Basically a form of sharding

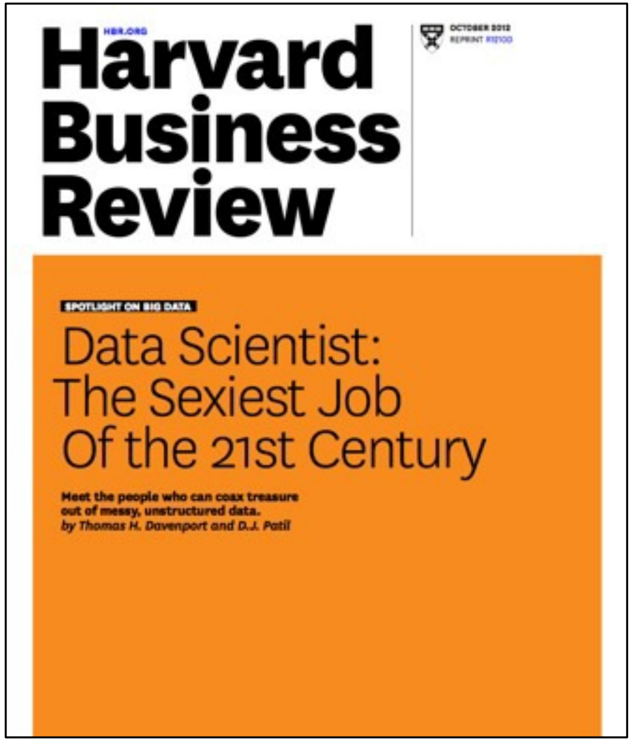
- Google paper talks more about distributed than multi-core, but this is of little consequence from an Erlang perspective

- Currently investigating using Erlang to implement these types of calculations

Job prospects…

"By 2018, the United States alone could face a shortage of 140,000 to 190,000 people with deep analytical skills as well as 1.5 million managers and analysts with the know-how to use the analysis of big data to make effective decisions."

Seems a bit melodramatic! This stuff takes me back to my university days… scientific programming, we used to call it... number crunching using various statistical and numerical algorithms… used to be in Fortran; now we've got all sorts of other fancy options and more data to play with, but the underlying mathematical principles have really not changed that much (IMHO).



HBR.ORG

Harvard Business Review

OCTOBER 2012
REPRINT R1210D

SPOTLIGHT ON BIG DATA

Data Scientist: The Sexiest Job Of the 21st Century

Meet the people who can coax treasure out of messy, unstructured data.
by Thomas H. Davenport and D.J. Patil

Erlang's approach to SMP & the role of Erlang

- Generally speaking, SMP/multi-core/parallel programming is non-trivial
  - Impacts the entire software stack and SDLC
    - Languages, runtime libraries, operating system, …
    - Testing, debugging, …

- Programming with threads, shared memory, and locks in C/C++ is too low-level
  - Requires considerable skill
  - Can take a lot of time to get things right
  - Code is often difficult to maintain and support
  - …

http://www.reddit.com/r/aww/comments/2oagj8/multithreaded_programming_theory_and_practice/

- Even with Java it's not easy, although to be fair…
  - Scala deals with some of the tricky stuff and provides some nice features
  - The Akka toolkit is also quite a nice piece of work
    - Does a pretty good job simplifying the construction of concurrent and distributed JVM-based applications
    - Derives much of its inspiration from Erlang

Erlang’s approach to SMP & the role of Erlang

- Several good API-style solutions for C/C++ have been around for some time and remain popular…
  - PVM (Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/)
  - MPI (Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi/)
  - …

- But these solutions are not transparent to developers and require specific knowledge of the API(s) and toolkits in question
  - … but they have been shown to perform very well

- In contrast, Erlang programs do not have to include any special code in order to take advantage of SMP
  - SMP is (generally) transparent to the Erlang programmer in much the same way as distribution is
  - Erlang inherently provides all of the necessary/desirable higher-level abstractions
    - Light-weight processes without shared state
    - Asynchronous message passing
    - Distribution transparency
    - Fault-tolerance
    - Shared-nothing, no mutexes, …

Newer languages such as Rust and Go include similar capabilities to Erlang, and this is arguably a major contributing factor to the increasing popularity and adoption of these languages.

Most developers consider thread-based applications particularly painful to write. Deadlocks, starvation, and race conditions are concepts that are all too familiar to the majority of developers doing concurrency. Erlang in particular takes away a lot of that pain.

Erlang's approach to SMP & the role of Erlang

- It was commented that Erlang is not such a great language for compute-intensive numerical calculations
  - That's not what Erlang was designed for!
  - HiPE can sometimes help… LLVM

- Calculations not conducive to Erlang can often be off-loaded to a driver/port implemented in a more appropriate language
  - Usually C/C++
  - Bitcask and LeveDB backends in Riak for example

- Must be careful to ensure that drivers operate in an asynchronous manner to avoid blocking scheduler threads
  - The `driver_async` callback can be used to off-load computation to another thread from a pool different from the scheduling threads
    - See http://erlang.org/doc/apps/erts/driver.html#id82769 for a simple example of an asynchronous driver
    - See Riak sources for some more sophisticated code

I'm reminded on Garrett Smith's 2014 talk to the Chicago Erlang User Group in which he considers the benefits of using Erlang in non-Erlang environments and presents a simple set of example code that illustrates how an Erlang/OTP application can be used to start, supervise, and stop a set of related applications (see https://github.com/gar1t/port_server).

Comparison with Scala

- Scala provides some cute features for parallelizing code
  - But it is not necessarily as transparent or "natural" as with Erlang

- Consider the following sequential Scala code fragment where doIt() is some tail-recursive method

```
def main(args: Array[String]) = {
    val trials = 1000000000
    val result = doIt(trials, 0.0)
    println(result / trials)
}
```

- The simplest way to introduce parallelism into Scala code is to parallelize a map over a collection:

```
def main(args: Array[String]) = {
    val N = 16                // N separate computations
    val trials = 1000000000
    val count = trials / N
    val sums = (1 to N).toList map {x => doIt(count, 0.0)}
    val result = sums.reduce(_+_)
    println(result / trials)
}
```

Material adapted from https://darrenjw.wordpress.com/2014/02/23/parallel-monte-carlo-using-scala/.

Comparison with Scala

- And the code in the previous example can then be made parallel to take advantage of multiple cores very simply as follows:

```
def main(args: Array[String]) = {
    val N = 16                // N seperate computations in parallel
    val trials = 1000000000
    val count = trials / N
    val sums = (1 to N).toList.par map {x => doIt(count, 0.0)}
    val result = sums.reduce(_+_)
    println(result / trials)
}
```

- The `par` method converts the collection to a parallel collection, whereupon subsequent maps and other operations can be computed in parallel

- Note that this is applicable to multi-core; not to distributed
  - Probably something like Akka would need to be introduced into the mix if we wanted to go distributed

Material adapted from https://darrenjw.wordpress.com/2014/02/23/parallel-monte-carlo-using-scala/.

Some other comments

- There is a great deal built into the Erlang/OTP that sets it apart from alternatives

- But I do also quite like Scala, Go, and Rust, and even R
  - Case of being aware of respective strengths and weaknesses
  - Use the right tool for the job
  - Not entirely sure what's happening with Scala…
  - Rust runtime is possibly still a little in flux
    - First stable release May 2015
    - Currently few examples of Monte Carlo simulations in Rust

- Functional programming offers many advantages over other programming styles for the implementation of highly reliable and scalable applications
  - Functional languages emphasize immutable state and referentially transparent (side-effect-free) functions
  - These characteristics are Ideal for concurrency and parallelism
  - Leads to the creation of naturally parallel code

The recent resurgence of functional programming languages is partly driven by the realization that functional programming provides a natural way to develop algorithms that can exploit multi-core parallel and distributed architectures, and can efficiently and reliably (and predictably) scale, making them ideal for many Monte Carlo simulations, which are often inherently embarrassingly (and naturally) parallel.

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- Some terminology
- A simple experiment
- More complicated problems
- **Comments on tuning the Erlang VM for multi-core**
- Summary/conclusions
- Questions

General remarks

- The SMP version of the Erlang VM is started automagically if the operating system reports more than one core
  - Default behaviour can be overridden with the `-smp [enable|disable|auto]` flag

- The SMP VM starts one thread per core (but by default does not bind threads to a core)
  - These threads act as schedulers
  - Each scheduler has its own run queue
    - List of Erlang processes on which the thread is to spend a slice of time
    - When a scheduler has too many tasks (large run queue) relative to other schedulers, tasks will be migrated to balance the workload

- If SMP is enabled, the `+S Schedulers:SchedulersOnline` option can be used to set the number of scheduler threads

- Generally nothing to gain from using more schedulers than there are CPU's/cores

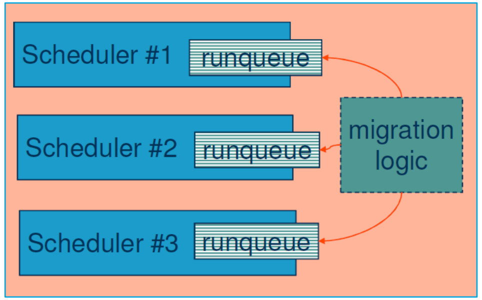


Image adapted from http://www.erlang-factory.com/upload/presentations/105/KennethLundin-ErlangFactory2009London-AboutErlangOTPandMulti-coreperformanceinparticular.pdf

Number of schedulers

- `+S Schedulers:SchedulersOnline`…
  - Unless instructed otherwise (via the relevant flags) the Erlang VM will attempt to automatically determine the number of configured and available logical processors (often these are the same value)
  - If the VM can determine these values:
    - `Schedulers` will default to the total number of processors configured
    - `SchedulersOnline` will default to the number of processors that are available
  - If the Erlang VM cannot make any such determination, both values will default to 1
    - … which is usually not such a good thing
  - Several other related command line options, some of which are experimental..

Scheduler wakeup interval

- An optional facility whereby schedulers are periodically scanned to determine whether they have fallen asleep (have an empty run queue)
  - Scheduler threads will be put to sleep by the operating system if there are no available jobs
  - A system call is needed to wake them up again
    - This takes time
  - Numbers of processes can be important here
    - For good SMP scalability it is generally desirable to always have enough runnable processes to keep all schedulers busy

- The interval at which schedulers are scanned can be set using the `+sfwi` flag
- Disabled (`+sfwi 0`) by default

- Be aware that some older Erlang distributions (pre-R16) can have a tendency to put schedulers to sleep too often

The `+sfwi` flag was introduced as a temporary workaround for long-executing native code. When these bugs have been fixed the `+sfwi` flag will/may be removed. For good scalability it is desirable to always have enough runnable processes to keep all schedulers busy!

Scheduler compaction and balancing

- Two methods are available for distributing workload across schedulers
  - Compaction of load
    - The Erlang VM will attempt to fully load as many scheduler threads as possible (attempts to ensure that scheduler threads do not run out of work)
    - The VM takes into account the frequency with which schedulers run out of work when making decisions about which schedulers should be assigned work
    - The default
  - Utilization balancing of load
    - The Erlang VM attempts to balance scheduler utilization as equally as possible, without taking into account the frequency at which schedulers run out of work

  - Cannot use both algorithms simultaneously
    - Choice of algorithm determined by the `+scl true | false` flag

General SMP tuning recommendations

- Can often do better than simply taking the default SMP beam settings

- Scheduler tuning can be a bit of a dark art
  - Lots of knobs to play with
  - RTFM
  - Experimentation is often required in order to find an optimal configuration for a particular program

- But in general…
  - Bind your schedulers (`+sbt`)
    - Scheduler context switching has undesirable performance ramifications
    - Binding schedulers to cores can often give a big performance boost
    - Unfortunately this will not work on some virtual (and other) configurations
  - Turn off load compacting (`+scl`)
    - Don't let schedulers sleep
    - Occasional severe performance drops have been observed by Basho and others that seem to be associated with load compacting

CPU topology examples

- On my laptop

```
$ erl
Eshell V6.2  (abort with ^G)
1> erlang:system_info(cpu_topology).
[{processor,[{core,[{thread,{logical,0}},
                    {thread,{logical,1}}]},
             {core,[{thread,{logical,2}},{thread,{logical,3}}]}]}]
2>
```

Single dual-core processor with hyper-threading.

CPU topology examples

- A dual 8-core processor system with hyper-threading

```
$ erl
Eshell V6.3  (abort with ^G)
1> erlang:system_info(cpu_topology).
[{node,[{processor,[{core,[{thread,{logical,0}},
                           {thread,{logical,16}}]},
                    {core,[{thread,{logical,1}},{thread,{logical,17}}]},
                    {core,[{thread,{logical,2}},{thread,{logical,18}}]},
                    {core,[{thread,{logical,3}},{thread,{logical,19}}]},
                    {core,[{thread,{logical,4}},{thread,{logical,20}}]},
                    {core,[{thread,{logical,5}},{thread,{logical,21}}]},
                    {core,[{thread,{logical,6}},{thread,{logical,22}}]},
                    {core,[{thread,{logical,7}},{thread,{logical,23}}]}]}]},
 {node,[{processor,[{core,[{thread,{logical,8}},
                           {thread,{logical,24}}]},
                    {core,[{thread,{logical,9}},{thread,{logical,25}}]},
                    {core,[{thread,{logical,10}},{thread,{logical,26}}]},
                    {core,[{thread,{logical,11}},{thread,{logical,27}}]},
                    {core,[{thread,{logical,12}},{thread,{logical,28}}]},
                    {core,[{thread,{logical,13}},{thread,{logical,29}}]},
                    {core,[{thread,{logical,14}},{thread,{logical,30}}]},
                    {core,[{thread,{logical,15}},{thread,{logical,31}}]}]}]}]
2> erlang:system_info(schedulers).
32
3> erlang:system_info(schedulers_online).
32
4>
```

Tweaking scheduler settings…

- Default behaviour

```
$ erl
Eshell V6.3  (abort with ^G)
1> erlang:system_info(scheduler_bindings).
{unbound,unbound,unbound,unbound,unbound,unbound,unbound,
         unbound,unbound,unbound,unbound,unbound,unbound,unbound,
         unbound,unbound,unbound,unbound,unbound,unbound,unbound,
         unbound,unbound,unbound,unbound,unbound,unbound,unbound,
         unbound,...}
2> erlang:system_info(schedulers).
32
3> erlang:system_info(schedulers_online).
32
4>
```

- Bind schedulers to cores using default bind type

```
$ erl +sbt db
Eshell V6.3  (abort with ^G)
1> erlang:system_info(scheduler_bindings).
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
 23,24,25,26,27,28,...}
2>
```

See http://erlang.org/doc/man/erl.html for an overview of the `+sbt` and related options (there are a few). Also be aware that as of R15B the Erlang VM does not by default try to bind schedulers to logical processors.

- Limit ourselves to 4 schedulers online

```
$ erl +S 4:4
Eshell V6.3  (abort with ^G)
1> erlang:system_info(schedulers).
4
2>
```

A comment about hyper-threading

- Intel's proprietary simultaneous multithreading (SMT) implementation used to improve parallelization of computations on x86 and ia64 (Itanium) processors
- For each processor core that is physically present, the operating system addresses two logical cores and shares the workload between them when possible

- Mileage may vary, depending on your operating system
  - Can be properly utilized only with an operating system specifically optimized for it

- Have observed mixed results

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- Some terminology
- A simple experiment
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- **Summary/conclusions**
- Questions

Summary

- Erlang has many traits that make it ideal for multi-core, for certain types of problems
  - Problems that conform to naturally-concurrent patterns
  - Includes many problems that can be modeled by Monte Carlo simulations of physical processes

- Most Monte Carlo calculations generally involve a lot of computationally intense number-crunching
  - Erlang is not very good at this sort of thing
  - But Erlang is very good at distributing and coordinating work across multiple cores and across multiple servers
  - Computationally intense aspects of an application can be written in a language better suited to the job
    - Can be invoked from Erlang

- Erlang's process model is inherently distributed and its concurrency model makes writing highly concurrent applications simpler than with many other languages
  - Utilization of multiple cores is simple
  - Facilitates more successful utilization of parallel hardware

Heterogeneous applications are a fact of life. Despite its lower raw performance speed, Erlang is able to find a home in high performance multi-core computing as a communications and coordination layer wrapped around compiled code that performs the heavy computation. It is the central nervous system that binds the application together, in an efficient, reliable, and elegant manner.

Summary

To quote Joe Armstrong (from “*Programming Erlang”*)…

“The world is parallel.

If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

Use a language that was designed for writing concurrent applications, and development becomes a lot easier.

Erlang programs model how we think and interact.”

AGENDA

- Introduction and background
- Introduction to Monte Carlo methods
- Some terminology
- A simple experiment
- More complicated problems
- Comments on tuning the Erlang VM for multi-core
- Summary/conclusions
- **Questions**

Questions