Assessing Solvency by Brute Force *is* Computationally Tractable

(Applying High Performance Computing to Actuarial Calculations)

Mark Tucker and J. Mark Bull

M.Tucker@epcc.ed.ac.uk

Overview

- about us
- motivation
- optimisation
- progress
- where to next?

About Us

- who we are
  - Mark Tucker
    - 14 years at Aegon
    - currently writing software for real-time systems for military aircraft
  - Mark Bull
    - 15 years in EPCC
    - on OpenMP steering board
  - EPCC
    - 25 years within University of Edinburgh's School of Physics
    - UK's leading High Performance Computing centre
    - run ARCHER, UK's national academic supercomputer
    - services to businesses
      - ⊛ consultancy
      - ⊛ training
      - ⊛ by-the-hour hire of high performance machines
- what we are doing
  - applying HPC to profitability and reserving calculations
    - perform large volume of calculations in reasonable time scales

Motivation 1: Annuity Disinvestments

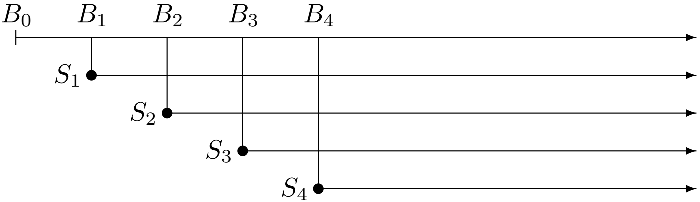
- estimate the amount of cash needed each month
  - payments to annuitants
  - investment expenses (as proportion of reserves held)
    $\Rightarrow$ need to know reserves at each step
- effectively, profitability with
  - one basis for calculating reserves
  - another basis for projecting
- industry-standard software on PCs
- separate data set for each cohort
- performance (for largest data set of each type)

Policy Type	Number of Policies	Run Time	Policies per Second
Immediate Annuities	$\approx$ 126,000	$\approx$ 35 hrs	1.0
Reversionary Annuities	$\approx$ 19,000	$\approx$ 13.5 hrs	0.4

$\circledast$ major bottleneck is calculating reserves

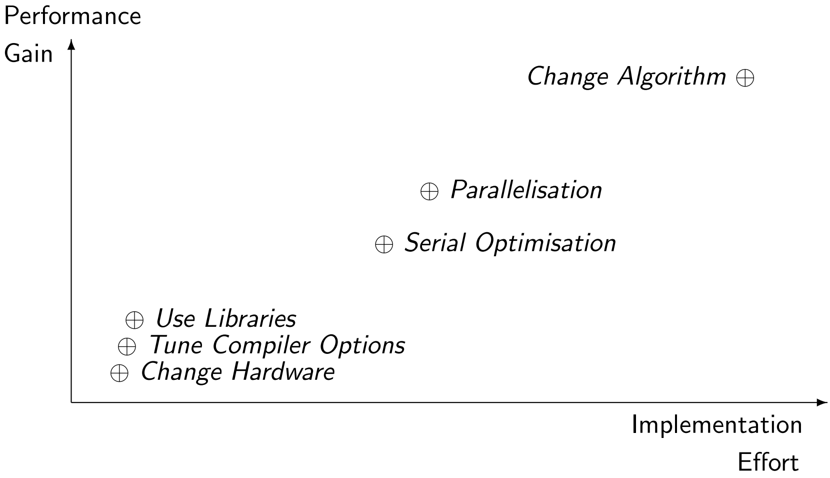
Motivation 2: Brute Force Annuity Reserves

- Solvency II $\Rightarrow$ thousands of scenarios
- based on (ex)Aegon's modelling actuary's interpretation



  - B_t is reserve on best-estimate basis at time t
  - 1000 scenarios at each future monthly time step
    S_t is 5th worst scenario at time t, i.e. "1-in-200 reserve"
  - require *additional capital* $= \sum_{t>0} v^t \cdot max(S_t - B_t, 0)$
- beyond contemplation ?
  - time for largest set of IA's is 12.6m core hours (≈ 1440 core years)
    - still more than 1 year when using 250 quad-core PCs
    - to obtain results in under a week, need more than 75000 CPU cores
      $\Rightarrow$ into the realms of "Top 500" supercomputer

Optimisation: Techniques



Optimisation: Implementation

- change hardware
  - until recently, standard practise in life offices
  - now have more cores per chip (cores no longer getting faster)
    $\Rightarrow$ need to embrace parallel processing
  - changing from *i5*'s to *Xeon*'s can still lead to small gain
  - ⊛ speedup of $1.8\times$ by moving from desktop PC to server
- change compiler switches
  - requires some knowledge of target hardware
  - use modern compiler $\rightarrow$ can benefit from modern hardware
  - need source code
    $\Rightarrow$ not always possible in packages which auto-generate code
  - ⊛ speedup of $3.8\times$ by selecting appropriate compiler options
- use libraries
  - none exist
- optimise serial code
  - replace calls to power function with repeated multiplication
  - simplify loop nests and other arithmetical steps
    $\Rightarrow$ not always possible in packages which auto-generate code
  - ⊛ speedup of $12.4\times$ from changes to serial code

Optimisation: Implementation 2

- parallelisation
  - use OpenMP
    - *de-facto* standard
    - shared memory / threaded API
    - standards exist for C, C++ and Fortran
      $\Rightarrow$ aimed at calculation-intensive codes
    - built into modern compilers
      $\Rightarrow$ portable
    - minimal changes to sequential code
    - helps if code being parallelised is well written
  - split loop over policies across multiple threads
    $\rightarrow$ each thread running on different core
  - some benefit in tuning the parallelisation parameters
  - speedup of $45.8\times$ using 48 cores (nearly 96% efficient)
- change the algorithm
  - "work smarter - not harder"

Recurrence Algorithm: Motivation

- level single life annuities
  - summation: $\ddot{a}_x = \sum_{t=0}^{\infty} v^t \; {}_tp_x$
  - recurrence: $\ddot{a}_x = 1 + v \, p_x \, \ddot{a}_{x+1}$
  - solution: assume $q_x = 1$ for $x > 120$ and work backwards
- level reversionary annuities
  - summation: $\ddot{a}_{x|y} = \sum_{t=0}^{\infty} v^t \; {}_tq_x \; {}_tp_y$
  - recurrence: $\ddot{a}_{x|y} = v \, p_x \, p_y \, \ddot{a}_{x+1|y+1} + v \, q_x \, p_y \, \ddot{a}_{y+1}$
  - combine with recurrence relation for single life to give the pair

$$
\begin{aligned}
\ddot{a}_{x|y} &= v \, p_x \, p_y \, \ddot{a}_{x+1|y+1} + v \, q_x \, p_y \, \ddot{a}_{y+1} \\
\ddot{a}_y &= 1 + v \, p_y \, \ddot{a}_{y+1}
\end{aligned}
$$

  - use matrix notation

$$
\begin{pmatrix} \ddot{a}_{x|y} \\ \ddot{a}_y \end{pmatrix} = v^0 \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} + v \begin{pmatrix} p_x \, p_y & q_x \, p_y \\ 0 & p_y \end{pmatrix} \begin{pmatrix} \ddot{a}_{x+1|y+1} \\ \ddot{a}_{y+1} \end{pmatrix}
$$

Recurrence Algorithm: Theory

- general case

$$\boxed{\mathbf{r}_{\mathbf{x},t} = v_t^f \, \mathbf{W}_{\mathbf{x},t,f} \, \mathbf{c}_{\mathbf{x},t} + v_t \, \mathbf{W}_{\mathbf{x},t,1} \, \mathbf{r}_{\mathbf{x+1},t}}$$

Tucker and Bull, *Algorithmic Finance* (2014), 3:3-4, 143-161.

  - based on time-inhomogeneous Markov chain
    - chain is formed by survival states of the lives involved
  - $\mathbf{x}$ = vector of ages of lives on which the policy depends
  - $\mathbf{r}_{\mathbf{x},t}$ = vector of reserves required depending on the state of the lives
  - $f \in [0, 1]$ = fraction through step where cashflows occur
  - v_t = time-varying interest rate (independent of the number of lives)
  - $\mathbf{W}_{\mathbf{x},t,g}$ = stochastic matrix of survivorship
  - $\mathbf{c}_{\mathbf{x},t}$ = vector of cash flows depending on the state of the lives

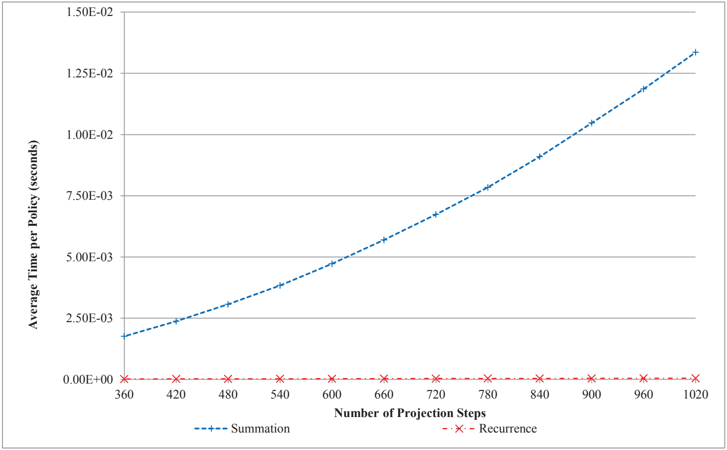
- variable interest and variable/improving mortality rate at each step
- computational complexity: for s outstanding time steps
  - summation: $O(s^2)$
  - recurrence: $O(s)$

Recurrence Algorithm: Implementation

- works for all (non-unit linked) policies with determinable cash flows
  - our use of annuities is purely because they provided our motivation
- stochastic matrix is straightforward to obtain
  - two states: annuities (level and increasing), endowments, ...
  - third state: assurances (term and whole life), ...
- extends to any number of lives using tensor products
- can ignore states which only ever lead to zero cash flows
- example: level, two life, last survivor annuity
  - $\mathbf{W}_{\mathbf{x},t,g} = \begin{pmatrix} {}_gp_x\ {}_gp_y & {}_gp_x\ {}_gq_y & {}_gq_x\ {}_gp_y & {}_gq_x\ {}_gq_y \\ 0 & {}_gp_x & 0 & {}_gq_x \\ 0 & 0 & {}_gp_y & {}_gq_y \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad g \in \{f, 1\}$
  - $\mathbf{c}_{\mathbf{x},t} = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}^T$
  - $\mathbf{r}_{\mathbf{x},t} \equiv \begin{pmatrix} a'^{(LS)}_{x,y} & a'_x & a'_y & 0 \end{pmatrix}^T$ where $'$ indicates generality of timing

Recurrence Algorithm: Processing Time

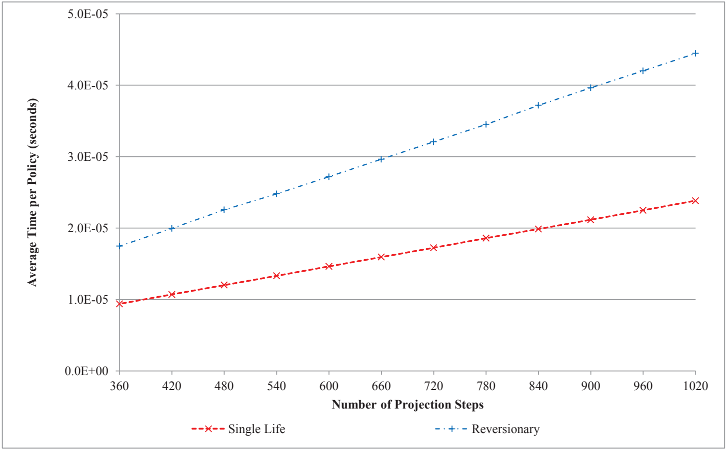
- processing times (per policy) for single life annuities



$\Rightarrow$ speedup of $100\times$ from use of recurrence

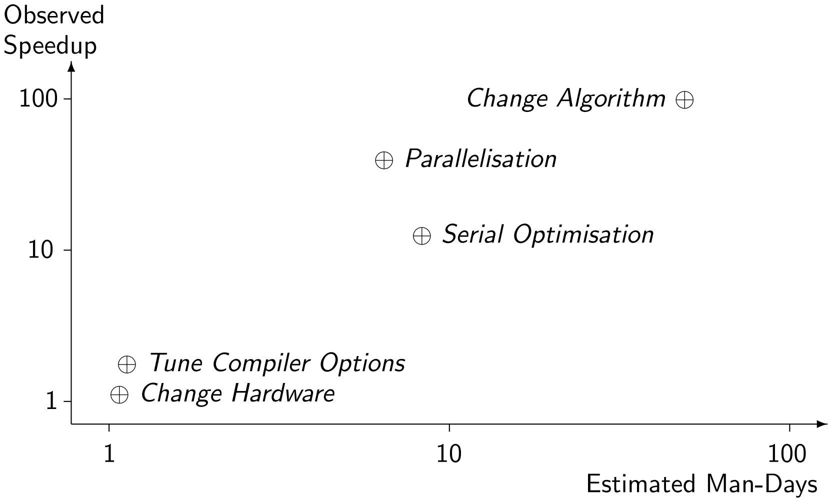
Recurrence Algorithm: Complexity

- processing times (per policy) for simple annuities



⇒ clear evidence of linearity over number of steps

Optimisation: Reward



Progress: Annuity Disinvestments

- speedup (Immediate Annuities)

Optimisation	Pols Per Sec
None	1.0
Increase Level of Compiler Optimisation	3.8
Manually Optimise Interpolation Routine	6.3
Manually Optimise Reserving Calculation	17
Remove Calls to Power Function	47
Implement Recurrence Algorithm	4,600
Change to Multi-Core Platform	8,600
OpenMP Parallelisation (48 threads on 48 cores)	390,000

- now limited by time taken to read data from disk
- conclusion: *write your own parallel code*

Progress: Brute Force Annuity Reserves

- have 1000 runs with one scenario at each future step
  - 1000 'tranches', each with 780 future steps
- 'representative' portfolio of 500,000 annuities

type	SL	RA	JL	LS
number of policies	300,000	100,000	50,000	50,000

- one machine with two 8-core chips

  $\Rightarrow$ acts as single 16-core shared memory machine
- timing (per tranche)
  - one scenario at each future monthly step for representative portfolio

type	SL	RA	JL	LS
overall wall-clock time (sec)	81.5	45.2	23.4	26.1

  $\Rightarrow$ actual wall-clock time ≈ 3 mins

  $\Rightarrow$ estimated wall-clock time for all 1000 scenarios ≈ 50 hours

  $\Rightarrow$ total CPU time ≈ 800 core hours
- conclusion: *write your own parallel code*

I/O Considerations

- performance of disinvestments is limited by time taken to read data
- policy data $\rightarrow$ 56.5MB for 500k annuity policies
- assumptions (per tranche)
  - 110MB for mortality tables
    - 2 sexes, 40 YoB's, 120 ages, 780 future time steps
  - 5.5MB for each of interest rate / inflation rate / investment exp pct
  - combine all assumptions for each tranche into one file
- overall input
  - one 56.5MB file to be read 1000 times
  - one thousand 126MB files, each to be read once

  $\Rightarrow$ likely to require to sustained input of 1GB per sec
- output reserves at each future step

  $\Rightarrow$ 1 file per tranche
  - one thousand 18KB files to be written once
  - post-processing step
    - read results and populate arrays
    - perform 780 sorts, each on 1000 elements
    - output 5^{th} largest at each step
  - time is insignificant

Future Work: Continuation

- other policy types
  - implement policy which requires 3 states ?
  - implement highly optimised code for common cases ?
  - implement the general case (cash flows defined by user) ?
- other CPU-based machines
  - same machine as used for disinvestments
    - four 12-core CPUs which can be used as single 48-core SMP
    - runtime should drop to around 17 hours (wall-clock)
      $\rightarrow$ estimator for runtime using hardware available within life offices
  - use one hundred of nodes of ARCHER
    - ARCHER can be rented by the hour
      $\Rightarrow$ see `www.epcc.ed.ac.uk`
      $\Rightarrow$ not beyond reach of commercial entities
    - each node has two 12-core CPUs
      $\Rightarrow$ runtime should drop to around 20 minutes (wall-clock)
      $\Rightarrow$ indicative cost at 10p per core hour $\approx$ £80
      $\circledast$ c.f. $\approx$ £1m using commercial software ...
      ... if you could run it on ARCHER

Future Work: Other Technologies

- Intel's *Xeon-Phi* chip
  - 60 cores, each with 4-way multi-threading
    $\Rightarrow$ effectively 240 cores on single chip
  - performance (with non-actuarial codes) is not spectacular
    $\Rightarrow$ don't expect to drop to 1 hour
    $\circledast$ expect this to be no worse than about 24 hours
- GPUs
  - a few thousand cores per chip, but each core is slower than CPU
  - researched in other scientific areas over past few years
    $\Rightarrow$ generally around 50 to 60 times faster than CPUs
    $\Rightarrow$ one GPU might be able to do all 1000 scenarios in one hour
  - small cluster of GPUs is relatively inexpensive option
    $\Rightarrow$ could do several brute force runs over lunch

Future Work: Other Tasks

- bases
  - dynamically generated (rather than read from file)

    either   can someone let us know how the bases are created?

    or   can someone give us real scenario info?
- other tasks
  - alternative interpretation (per conference, Royal Soc Edin, Apr2014)
    - 10^6 scenarios for first year
    - 10^3 nested scenarios to the end of the projection

    $\Rightarrow$ require clarification
    - do the bases within each nesting differ?
    - what is the interesting output from this setup?
  - assessing the accuracy of approximations
    - for a given set of bases, we "know" the correct answer
      $\Rightarrow$ can see how close we can get by sub-sampling
    - approach the "correct" answer by increasing the number of scenarios
      $\Rightarrow$ might be able to do $100\times$ the number of scenarios
      - each scenario uses $\frac{1}{100}\times$ the number of data points
    - might be able to guide the regulations

Future Work: Other Problems

- approximations
  - assume that all cash flows happen in advance
    - prudent
    - increases speed

  $\Rightarrow$ can assess which simplifications/approximations are worth making
- pricing
  - profitability of 1000 model points on each of 1000 bases

  $\Rightarrow$ a few seconds

  $\Rightarrow$ fully interactive
- sensitivity analysis
  - effect of changes to interest/mortality on reserve/profitability

  $\Rightarrow$ have small enough changes to perform numerical differentiation
- *your ideas*
  - what would *you* do with a program which runs this quickly?

  ⊛

  ⊛

  ⊛

  ⊛

Questions & Discussion

- thank you for listening

Prepared Answer: Upper Bound on Run Times

- for the 16-core SMP

type	SL	RA	JL	LS
number of policies	300,000	100,000	50,000	50,000
time excl i/o (sec)	78.57	42.09	20.15	22.9
time for 50000 pols	13.095	21.045	20.15	22.9
lives	1	2	2	2
non-zero entries in $\mathbf{W}_{\mathbf{x},t,g}$	1	3	1	5

- linear regression gives

$$runTime = 5.5125 + 6.895 \times lives + 0.6875 \times nonZeroes$$

$$R^2 = 0.9972$$

- fitted times:

type	SL	RA	JL	LS
time for 50000 pols	13.095	21.365	19.99	22.74

$\rightarrow$ not unreasonable

- can estimate the upper bound on run time for any policy type

  $\Rightarrow$ no real benefit in producing code for all types of policy

Prepared Answer: The Shape of Our Synthetic Data

- shape is that of cohort of recent retirees
- single life
  - all policies incepted in year preceding valuation date
  - age at inception $\sim U(57, 67)$
  - roughly 73% males
  - roughly 81% monthly payments, remainder annual
  - amount of each payment is s where $\ln s \sim N(5.0, 1.47^2)$
  - annual escalation rate is roughly

rate	0%	3%	4.25%	5%
proportion of pols	95%	3.5%	1%	0.5%

- reversionary annuity / joint life / last survivor
  - same major characteristics as single life
  - age difference $\sim U(-4, 4)$
    $\Rightarrow$ maximum difference is 4 years, with no regard to which is older
- effect of ages is to create 'long' outstanding terms
  $\Rightarrow$ run times are not unrepresentative