A 'Hot Date' with *Julia*: Parallel Computations of Stochastic Valuations

By Charles Tsai

Meet "Julia," a free programming language licensed by MIT that may help you with parallel computing. It may be an alternative tool for those who are interested in nested stochastic processes for actuarial research (if not for regulatory compliance).

Nested stochastic processes may become more relevant and prevalent as stakeholders consider a broader spectrum of possible outcomes. Such "stochastic-in-stochastic" analyses often add color to actuaries' palette of tail risks and conditional tail dependencies (if any). However, they also introduce issues of runtime and memory allocation. The article "Nested Stochastic Pricing"[1] provides a comprehensive summary of nested stochastic applications in response to recent regulatory reforms. IFRS seems to require a comprehensive range of scenarios that reflects the full range of possible outcomes for calculating fulfillment cash flows. Economic capital calculations may likewise require stochastic-in-stochastic simulations. A practice that may have been previously deemed as a costly bonus may evolve into a minimum expectation for actuaries in the near future.

Nested stochastic processes may become more acceptable with parallel computations. One may boil down "parallel computing" to daily applications with an analogy. Imagine an investment banker who is planning a date with a lady. He barely has enough time to smoke, and he has completing the following four tasks in mind: 1) dress up, 2) buy flowers, 3) research a restaurant's menu, and 4) fold a thousand origami cranes. He has made these preparations in solo for all of his previous dates. Would it not be nice for him to have friends help him perform the latter three tasks *simultaneously*? Delegation may take some time, but it may be more efficient than performing all four tasks in sequence. Parallel computing is a form of dividing and conquering problems using multiple processes concurrently. It may help actuaries slam-dunk tasks like traversing a thousand scenarios, even if the tasks already take less time than folding a thousand origami cranes.

Julia allows users to distribute and execute processes (such as nested stochastic valuations) in parallel. In essence, a computer may have four Central Processing Units (CPUs) in resemblance to a soccer team with four members. Programmers can leverage Julia's multiprocessing environment to specify certain tasks to those CPUs on the bench. On the one hand, the art of scheduling may be a bulk process for infrequent and smaller tasks. On the other hand, the flexibility to pass messages to multiple processors may be one's niche in strategic scalability and performance. Actuaries may then manage disparate layers of stochastic simulations via a multiprocessing environment. Shorter runtimes may be a doomsday for a few students who use waiting time as an opportunity for studying. However, such efficiency opens doors to comprehensive iterations and widens windows of perspectives.

IS JULIA A DISRUPTIVE INNOVATION?

Julia has several features[2] that supplement its power in parallelism and distributed computation. Some features are for specialists like Sheldon Cooper (of *The Big Bang Theory*) while others may be easier for amateurs like me to appreciate.

- First, it is free and open sourced as licensed by MIT. Actuaries can share research results seamlessly at SOA/CAS events without worrying about whether the audiences have access to the same tools to review (and build upon) the findings.
- Second, users can define composite types that are equivalent to "objects" in other languages. These user-defined types can run "as fast and compact as built-ins".[3]
- Third, users can call C functions directly, and their programs' performances can approach those of languages like C. Such speed makes it a considerable alternative to proprietary computational software tools.[4]
- Fourth, one does not need to be a genius like Gaston Julia in order to learn the language. Justin Domke's blog post "Julia, Matlab, and C"[5] presents a crystal clear comparison of syntactic and runtime complexity tradeoffs. Learning Julia is like learning Matlab® and C++ for Towers Watson MoSes® simultaneously.

- Last but not least, Julia is a functional programming language like OCaml, which is adopted by niche firms like Jane Street. Functional programming frameworks can help actuaries adapt to and master recursions.

Julia also has several Achilles' heels that may significantly jeopardize its adoption among actuaries.

- One obstacle is communication. Due diligence may be lost in translation. A few know how to use and interpret proprietary actuarial software products due to limited availability. Fewer know how to read and review (or even find) its generated C++ codes. In a like manner, few have learned (or are willing to learn) the Julia language, and its graphical features are still under development. Some actuaries may still prefer parallel computations via multiple Microsoft Excel® sessions. Calibrations of Julia programs with validated Microsoft Excel® workbook models might just have exceeded paychecks.

- Another hindrance is the language's relative immaturity. Development commenced in 2009.[6] Its scale of recognition seems to be light years from the tipping point for a stabilized discussion ecosystem to exist. Online inquiries for relevant debugging notes make passing bills during gridlocks look easy. A tool may only be as valuable as its received appreciation.

- Lastly, the manipulation of processes in parallel computations requires an acute awareness of read-write conflicts. In light of the previous analogy, the banker may wish to match his suit with the flowers purchased, or the flowers purchased with the restaurant's cuisine. Tasks may not be completely independent from each other. Inexperienced users may inadvertently manipulate and designate processes in manners inconsistent with intentions.

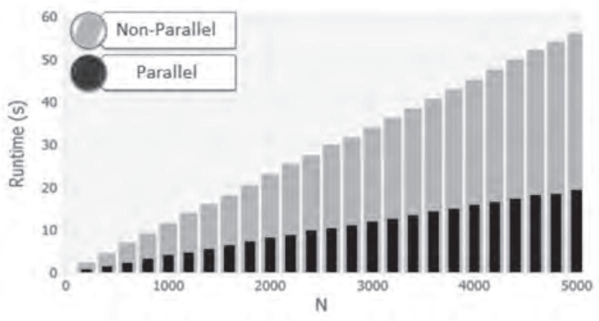
A SIMPLIFIED GMMB CASE STUDY

I have drafted an exemplary Julia application of an actuarial model. It is available at *https://github.com/Chuckles2013/GMMB_RSLN2_Julia*, and is an independent project for educational purposes only. All parameters and values have been arbitrarily chosen. The case study involves calculating the present values of liabilities for an extremely simplified Guaranteed Minimum Maturity Benefit (GMMB).

The scale of the project can be partitioned into two major layers. The first layer involves simulating parameters for N world scenarios. For simplicity, I have structured all key parameters to be the same across all N world scenarios. It is easy to see that one can simply modify the codes to utilize simulated parameter inputs for considering different world scenarios and economic environments. The second layer involves simulating fund returns for 1000 funds, from which one can derive a conditional tail expectation of liabilities. Both layers provide N figures of conditional tail expectations, from which one can extract a maximum level.

The superimposed bar graph below compares runtimes for non-parallel versus parallel computations under various numbers (*N*) of world scenarios. Four processors performed the parallel computations. The absolute values of the excess time elapsed are evident in the divergent gap.

Runtime Comparisons Across Different Numbers of Scenarios (When 4 CPUs Are Available)



CONTINUED ON PAGE 48

A '*HOT DATE*' WITH JULIA … | FROM PAGE 47

NEXT STEPS

One's vision for Julia in actuarial science can be the development of packages. A few companies were bold enough to have utilized R, and none has adopted (or even plan to leverage) Julia to my knowledge. Full adoption of Julia among actuaries within the next decade may be more of a fantasy than a reality, just as few actuaries have learned Python since its inception in 1991.[7] Nevertheless, open-source packages for broader usage are lower hanging fruit for intrigued actuaries to consider. To the best of my knowledge, there are no Julia packages similar to the lifecontingencies and actuar packages in R libraries. Templates of actuarial functions in Julia may capture more attention and appreciation for the beauty of parallel computations for nested stochastic valuations. ▼

Charles Tsai

Charles Tsai, ASA, is a Life Actuarial Analyst at AIG in Shanghai, China. He can be reached at *charles-cw.tsai@aig.com*.

ENDNOTES

1 "Nested Stochastic Pricing: The Time Has Come" by Milliman®'s Craig Reynolds and Sai Man is available at http://www.milliman.com/insight/insurance/pdfs/Nested-stochastic-pricing-The-time-has-come/

2 http://julialang.org/

3 http://nbviewer.ipython.org/github/bensadeghi/julia-datascience-talk/blob/master/datascience-talk.ipynb

4 Professor Fernández-Villaverde's "A Comparison of Programming Languages in Economics", which is available at www.econ.upenn.edu/~jesusfv/comparison_languages.pdf

5 http://justindomke.wordpress.com/2012/09/17/julia-matlab-and-c/

6 web.maths.unsw.edu.au/~mclean/talks/Julia_talk.pdf

5 This is a rather fun proof left for the reader. First, prove that each row of (I – S) sums to zero. What does this imply about the triangularized matrix?

7 http://svn.python.org/view/*checkout*/python/trunk/Misc/HISTORY