

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

Suppose an Excel Solvency II Model Runtime of 1.52 Months Could Be Shortened to 8.58 Hours

By

William C. Scheel and Randy Smith

Introduction

This article is about Microsoft high performance computing (HPC) using Excel 2010. There are many Excel workbooks in the insurance world that have long runtimes. Some run on a desktop computer for hours—others run days, but this may be the first time you heard about one running 1.52 months. The chore involved a stochastic simulation of life and lapse contingencies to produce an aggregate loss distribution and identification of the 0.995 VaR. The probability distribution was derived from 100,000 evaluations of each of 50,000 equity-linked life insurance policies. The simulations involved generation of Vasicek mean-reverting interest rate paths and policy-specific cash flows using random mortality and lapsation. And, yes, it would have taken at least 1.5 months to solve the problem without grid computing. The HPC cluster running the job in 8.58 hours consisted of 224 cores. In terms of similar, server equipment this result is given an efficiency of 57% relative to the standalone estimate of 1.52 months running a single instance of Excel.

Many readers are quite familiar with the need to super-charge Excel using user-defined functions (UDFs) written in C/C++. This use of UDFs in cell functions enhances Excel performance by substituting VBA functions with quicker, compiled macros and the use of multiple cores on a single computer. But, the substitution of visual basic for applications (VBA) in an Excel workbook with C/C++ code often means a loss in the flexibility and simplicity VBA gives to actuarial programmers. But, there is no question that VBA code is slower, and usually significantly slower, than compiled code. So, if you wrote a VBA loop such as: For i=1 to 100000 ...do something ...Next i, you may have sat and waited a long time for Excel VBA code to “do something” repeating the computational loop 100,000 times. Microsoft has ushered a new Excel capability in its version 14 release. It enables actuaries to push do-something_i to one computer while at the same time the do-something_{i+1} is pushed to work on a different computer. Although do-something_{g100000} will execute on the Nth computer where N<100,000, there still could be many computers each with many cores that are operating in parallel over the course of 100,000 partitions of a problem that otherwise would be done serially on a single computer. When a loop contains independent iterations, it is embarrassingly parallel and is a candidate for parallelization wherein partitioned pieces of work are calculated in parallel on a grid rather than in the serial confines of a loop.

Just about all insurance processes, whether valuations or simulations, fall into this genre. Those of us who are not particularly facile with C/C++ and shy away from UDFs (and the authors include themselves

William C. Scheel, DFA Technologies, LLC and Randy Smith, Microsoft Corporation.

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

in that lot), will be delighted with Excel 2010 because the required asynchronous programming method is easy to learn and adopt in these repetitive types of problems. Asynchronous VBA means you throw away the for-loop, and think of it as a series of events.

This article describes asynchronous programming that now is required for HPC VBA Excel applications. We use the context of two Excel examples. The first one has been noted above; it is a delta-NAV calculation that might be done for Solvency II. The second one is a rate book pricing problem. Here there is examination of many combinations of parameters—each combination leads through simulation of censored frequency and severity to a chance-constrained premium calculation. The classification parameters are for frequency and severity probability distributions and policy censors. The pricing exercise is intended for policies covering multiple events. Each event is subject to a policyholder deductible. There is an occurrence maximum limit of liability as well as an aggregate limit of liability for all insured events. The frequency of events is modeled with a Poisson distribution and subject to a maximum number of insured events. Each event's severity is modeled using a truncated normal distribution.

The truncated normal has a lower and upper truncation point. This distribution is described by a simulation in which a draw from an underlying normal distribution is obtained. If it falls outside the truncation points, another draw is taken. In effect the probably area in the truncated tails is redistributed to the interior in proportion to interior, normal probabilities. This severity distribution was chosen more for its computational complexity than any particular actuarial application. The substance of the problem is largely unchanged regardless of what censored frequency and severity distributions are used.

Further, the severity distribution is subject to contagion. An increasing, additional loss amount is added to the liability for each additional insured event. The parallelization of this problem is the treatment of combinations of parameters in a sweep. The sweep of each parameter from a minimum to a maximum value over intervals is similar to pricing solutions for an entire rate book. Each combination is priced in parallel over the HPC cluster.

The authors are unaware of any closed solution to either problem. Each one must be solved using simulation. Both are embarrassingly parallel because one instantiation of investment returns can be evaluated for 50,000 policies on one computer while another instantiation can be evaluated for the same block of policies on a different computer. Similarly, various combinations of classification parameters in the second problem can be evaluated on different computers. Why use one computer when we can use N of them?

The second problem involved 153,747 parameter sweep combinations. This is analogous to rate book pricing calculations. It would have taken over 22 days on a very fast workstation computer. On the grid with about 225 cores, it took only 3.6 hours...that's why a cluster solution is needed.

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor’s permission.

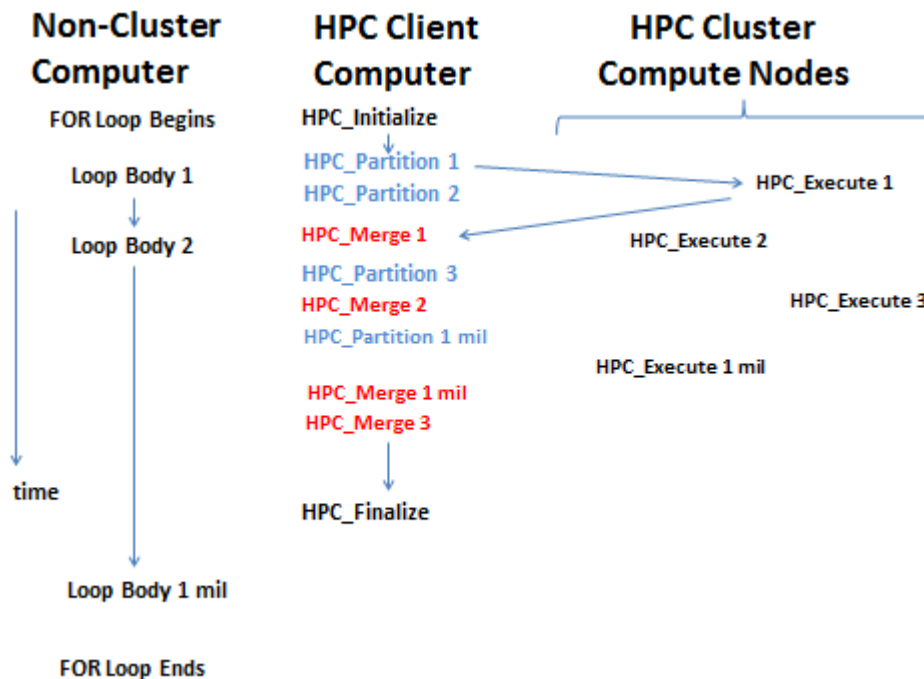
Excel VBA Asynchronous Programming

There are identifiable events within the for-loop. The body of the loop contains the “do-something,” code. Each traversal of the loop is an execution event that may be run on separate compute nodes within the computer cluster or grid. Other parts of the loop also are events. Actions precede entrance into the loop; these initialize the loop. The clockwork of the loop is deconstructed into partition events. Each partition or loop iteration is an event requiring information setup. We refer to those as partition events. Data with specificity to a subsequent execution is prepared in the partition, and each partition event results in data transport to a compute node.

The first operation is referred to as initialization—think of it as a job setup. The second, however, involves creation of a quantum of information needed by a compute node—there are many compute node operations and for each one there is a partition setup.

The asynchronous aspect of event processing is a callback by a mechanism within HPC Excel to various code blocks. Each callback is associated with an event. The idea can be seen in Figure 1 where events are identified with parts of the for-loop.

Figure 1 Deconstructed, Event-Driven For-Loop



The events each require a separate VBA function. There are four events: Initialize, Partition, Execute and Finalize. These can be seen in the lower portion of Figure 1. In the HPC background there is a process which brokers the job among computers. The figure illustrates a block of computers running on

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

a cluster, and it is the broker's job to shuttle information between these various nodes. This mechanism is addressed in Excel through a COM object; it is one added to HPC Excel 2010 through VBA references. The daemon determines when an event callback occurs. Neither the programmer nor user controls the exact timing of event processing. That is why HPC Excel 2010 is an asynchronous services oriented architecture (SOA).

HPC_Partition

Although the workbook setup may be done during the first callback to HPC_Initialize, the HPC_Partition callbacks are when the user will push a small bit of unique information to each compute node. This partition callback activity with partition creation will continue until the HPC_Partition function senses an end and notifies the HPC session broker. Each partition is used during an execution of a workbook that is on a file share accessible to all compute nodes. This execute workbook is opened by the HPC session manager prior to delivery of partition data. The client workbook is functionally idle between callbacks asking for partition data and HPC_Merge callbacks delivering results. The Excel GUI, however, is available during these idle periods. Some of these asynchronous, HPC events merit closer examination.

HPC_Execute

This is a VBA function. It will be called multiple times in every instance of Excel that is running and allocated to the HPC job. We will note in a moment that N instances of Excel running does not necessarily translate into N computers; the N instances likely are running on M computers where $M < N$, and maybe substantially so. We also know that the number of parallel simulations or valuations is much larger than N. Each of the N Excel instances will have its HPC_Execute code *executed multiple times*. The HPC broker is going to reuse instances of Excel; each time partition data is given to HPC_Execute during callbacks. The quantum of partition data plus whatever else is in the operational sphere of the Excel workbook is all that the execution code has to work with. This thought is captured in the pseudo code in Figure 2.

Figure 2 Excel HPC Execution Stream

```
Public Function HPC_Execute(data as Variant) as Variant
    'Take the Variant information and put it in cell range.
    'Do an Application.Calculate. Excel uses cellular
    logic, calls to macros or UDFs to calculate all
    dependency relationships.
    'Gather information to be passed back from the
    compute node to the HPC client.
    'Create and assign the HPC_Execute variant variable
    that is given to the HPC client during an HPC_Merge
    callback and after the end of this HPC_Execute
    subroutine.
End Sub
```

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

The operation of HPC_Execute procedures has an extremely important property. The **full** repertoire of an Excel workbook is available. Think of this step as pressing an F9 key and recalculation of the workbook. In Figure 2, the program function embodies a full workbook calculus. There is both write, calculate, and read.

The Execute on Each Compute Node Is a Robust One

Notice that a compute node can do whatever the user might ordinarily do within any VBA *subroutine*; but the procedure in Figure 2 is a function and returns a result. *The compute workbook can be calculated (equivalent to pressing F9) to produce this result, and that is why this truly is a full-functional Excel compute step.* In this case, the result passes from the compute node back through the broker(s) and head node to the client workbook during the HPC_Merge callback. This transfer from computational source to the original caller is done during asynchronous callbacks to the client workbook. The variant (a variable type in VBA programming that can be any datum, array or object) containing an HPC_Execute result is an argument in the HPC_Merge function call. A client workbook then can do anything it wants to do with the compute node(s) result(s) during this merge callback(s) operation. Note that the compute nodes have no memory of prior executions nor should they. The execute activity is a service—a partition of data is handled by this service and a result returned to the HPC_client.

HPC_Merge

After each execute there is an HPC_Merge callback; and the client workbook can do anything that may be required with the result including just remembering it. The remembrance could be manifest in transfer of the data variant to worksheet cells, writing them to a disconnected ODBC dataset or pushing them through an ODBC connection to an external database. Speedy handling, however, is important.

HPC_Finalize Callback

How does the HPC client know that processing is complete? The HPC_Merge data are received in an unknown order, so the client is given a final callback after the last HPC_Merge, and it tells the client that the HPC job is finished. The conclusion of a job could be a result of successful completion of the partition-execute-merge sequence. But, it also could be a result of abnormal termination because of error or job cancelation by other means. All of these circumstances will result in a callback to the HPC_Finalize function.

HPC Excel Components

The out-of-the-box HPC Excel runs on HPC-compliant operating systems. There are two of them:

1. Servers operating Excel 2010 using Windows HPC 2008 R2 operating system.
2. Workstations operating Excel 2010 using Windows 7 operating system.

Each piece of hardware requires a single copy of Excel 2010. But, HPC computing may have and likely will have multiple instances of Excel running simultaneously on the same computer and using multiple

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

cores within each computer. Fortunately, the number of copies of Excel you buy is not more than the number of computers. In fact, it is the same number. N computers; N copies of Excel. And, each of the Excel copies requires the purchase of an additional HPC Pack. The Pack operates both on HPC servers and workstations. There is an HPC Pack— N of them—that is required for each of the N computers in the grid. Cost neutrality is important, but difficult to judge. In the past, Microsoft never supported Excel on a headless server. HPC services now available for Excel is the first time server support in any capacity has been provided for it.

Operating systems such as XP or versions of Excel prior to 14 will not work in Microsoft HPC. However, computers running Windows 7 can be deployed as compute nodes. Conceptually, HPC is stealing compute cycles from these Win7 workstation nodes while they are otherwise being normally used. Off-hour usage of these Win7 machines when workers go home is an opportune time to turn on idle resources as additional HPC compute nodes on the cluster.

In summary, HPC Excel Version 14 requires many parts. Foremost is a collection of operating systems working on server computers (at least one and probably many), each with 4 or more cores. Possibly there are Win7 computers (each with multiple cores) working as compute nodes. All of this gear is working on the same network domain. Every operating system seat has a single Excel seat. Revamping Excel VBA code in a workbook to be asynchronous enables it to be opened in many instances of Excel on many computers and calculated on many more cores. Excel 14 has a new Microsoft object that is referenced in VBA. That object is used by the HPC client to open and run an HPC session on all this gear. Once it is running, there is a cascade of callbacks by the session manager to open a workbook on compute nodes and shuffle data back and forth to the HPC client instance of Excel. All-the-while `HPC_Execute` is the function operating in parallel in many instances--each using the full-kit, out-of-the-box Excel capabilities.

What Is in an HPC Excel Workbook?

The HPC client workbook can be run standalone. Containing the `HPC_Execute` code and supporting helper procedures plus all the other HPC procs (`Initialize`, `Partition`, `Merge` and `Finalize`), the HPC client workbook is written with asynchronous callback functions. To operate standalone, all of the HPC callback functions must be present in the client workbook. However, a compute node will only perform the `HPC_Execute` callback; none of the other Excel HPC functions are used in a compute node.

The reader may think of her workbook opening simultaneously on multiple instances of Excel, but she needs to consider the possibility that the HPC *compute* workbook may be different than the HPC *client* workbook. Caution is needed here. From a development and maintenance standpoint, one wants and must have a single workbook and not separate workbooks that easily could get out-of-sync.

The authors used an advanced programming technique to create a reduced size workbook on-the-fly from the larger, client workbook. There may be a motivation for running a smaller, quicker version of the HPC client workbook. The scope of an actuarial model and its accustomed workbook and be very large. The client workbook will fully contain the model. But, there certainly will be more in this client

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

workbook that actually is required on a compute node. That could be the motivation for two versions of the "workbook." One is a client workbook that runs on the computer launching the HPC job. The other may be the same workbook or a different, smaller workbook. The reason for a reduced-size, compute-node workbook is that it loads and executes faster. But, the authors do not want readers to conclude that this must be the case. A single workbook with all of the HPC callback functions should be the "master" workbook. If there is a compute workbook, it should be made on-the-fly prior to opening an HPC session, and it must be done with special coding techniques.

We will summarize what has been covered by making several observations designed to highlight this entire process.

1. The HPC client workbook may be similar to the original, non-HPC workbook; but that similarity fades because it is a parallelized version written with asynchronous VBA code. That means that the compute segment is stateless and that the sequence of merge operations with compute node results is *not* in the same sequential order as the partitions that lead to them. The future of actuarial VBA programming lies in the actuary asking himself whether and how a for-loop or do-loop should be replaced with a series of parallel events and pushed to a computer cluster.
2. An instance of Excel remains open on a compute node as long as it is needed. The compute node workbook also is opened once, but there are multiple calls to the HPC_Execute functions each with different partition data.
3. The compute node operations are designed to be independent and parallel. The workbook was parallelized so that each compute operation was independent of others. However, the process of formulating that parallel operation can sometimes be counter-intuitive, and this will shortly be touched upon.
4. Note that the compute workbook can contain methods and properties in the form of worksheets, VBA code procedures and the ability to launch SQL queries, read or write files and so forth. The HPC Excel SOA has a potentially wide-reaching handshake.
5. The client workbook receives an HPC_Merge callback for each HPC_Partition operation. Because these are asynchronous (but also because they are synchronous to the broker handling pass-offs to and from compute nodes), you do not want the HPC client workbook operations in the client workbook to take a lot of time either when it fashions partitions or merges computational results. And, extended tinkering in either HPC_Partition or HPC_Merge would tie the hands of the HPC broker. Every partition and merge is just a real-time slice lost to the compute nodes.
6. Careful thought must be given what is done during either HPC_Partition or HPC_Merge. Here is where the art of programming will play a dominant role. The authors have used programming tricks both in these functions and in the HPC_Execute function. The parallelization process is often dependent on careful analysis of the model. At times the best parallelization technique is not the obvious one.
7. Recall that all or a subset of the HPC client workbook is placed on a file share and used on compute nodes. Even if there is only one workbook used both on client and compute node computers, it may be useful to think of them as two different workbooks. The client workbook

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

always will do the setup (HPC_Initialize), create independent, parallel operations by formulating data blocks (HPC_Partition) and receive the results of computations (HPC_Merge). This client workbook is notified when all executions are complete and the last HPC_Merge has been received (HPC_Finalize callback).

8. During design and debugging, the client workbook also will have HPC_Execute and be run in a standalone mode.
9. The compute node workbook is a copy of the HPC client workbook or a different workbook. It will be opened on compute nodes and will require all of the resources that HPC_Execute would use on the client workstation. For example, there may be objects that are referenced in the HPC_Execute code; these objects must be installed and available on the compute nodes as well as the client computer. For example, MathWorks Matlab is widely used by actuaries in Excel applications. If it were used in HPC_Execute, it must be available to every compute node.

The Model in the Workbook

The model most likely is codified in HPC_Execute. However, there really is not, a *model* workbook. Consider the life insurance example. The modeling chore is market-consistent valuations of insurance liabilities. The cash flow that arises from a particular scenario realization is discounted in a manner prescribed by regulation. That discount is thought to produce a market consistent valuation. Another aspect of the model is determining cash flows under a wide variety of scenarios resulting in policy termination. The model comprises both financial and non-financial simulations, and it determines policy provisions reflective of embedded value to the enterprise of being at risk on a block of policies. Different scenarios, both financial determinants and physical ones, result in different realizations of cash flows. The “model,” therefore, encompasses considerable properties, methods, and system tools. The portion of this model that actually is run under HPC is almost certainly less than its totality. It is easy to characterize the model workbook as a parallelization wherein each probabilistic realization is what is separately valued during an HPC_Execute. But, sometimes there is a bit of legerdemain in deciding what should be the object of parallelization.

The Life insurance example is a good because the parallelization treatment could be done in at least two markedly different ways. The one taken by the authors was not obvious (at least to them), and their choice was greatly influenced by programming considerations. In this example, there are many policies each of which is dissected into various probabilistic, cash flow waves. Many cash flow vectors are possible for a single equity-linked life insurance policy. But, of course, there are many policies in the block each with potentially different underwriting characteristics. Is “parallel” looking at each policy separately? Or, is “parallel” looking at each scenario separately? The answer to this question materially affects how HPC_Execute is written.

A Policy Is Not Necessarily Parallelization Fodder for an HPC_Execute

A careful consideration of how a simulation of contingencies for an equity-linked life policy unfolds helps to unearth a parallelization approach. Over a future horizon looms an investment realization. A financial scenario may be, for example, an n-year simulation of a mean reverting process such as the

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

Vasicek model. But, each policy during an investment horizon has a cash flow realization, and it could take many forms depending on whether the policy terminates through death or lapse and at what point in time termination occurs. And, it is unlikely that a block of policies will influence a financial realization such as equity-linked investment returns, so investment and physical risk are deemed independent. The authors decided to choose a parallelization along investment simulations rather than along policy-specific, physical simulations. But, the solution could have been done either way.

A compute node could, for example, be given information about a single policy. Or, it could be given information about possible financial outcomes. That means that an HPC_Partition would send to the HPC broker a VBA variant containing either necessary information about a single policy or a single investment scenario. Consequently, the compute node workbook either needs to identify the investment scenarios given a policy, or it must know about the block of policies given a scenario. Parallelization could occur either way, and what pre-existing data are at hand in each compute node workbook will be affected by the parallelization choice. The parallelization approach chosen fed a scenario to a compute node rather than a policy. This solution required information about the entire block of policies being present in the compute node workbook.

The investment scenario used in the example is a quickly simulated Vasicek interest rate path. Each of these paths was created during an HPC_Partition step. The path was the unique data sent as the inbound variant argument for HPC_Execute. The compute node functions as a special service provider. It does the derivation of the market consistent valuation of policy cash flows for every policy given the investment scenario. The reason for this approach is because during the workbook open on a compute node, a "setup" may be done for the policies, and it can be reused during other HPC_Execute cycles on the same compute node. Because the block of policies is known during workbook open, the VBA code can create a collection, a VBA object that can hold variant information, be indexed and rapidly traversed. That provides a rapid lookup for what otherwise would be a repetitive, time-consuming operation—looking up attained age mortality probabilities, lapsation probabilities, surrender charge schedules and expense charges. For example, a policyholder's attained age provides a unique entry into a set of mortality or other tabular probabilities. This lookup is likely non-trivial in terms of compute cycles were it to be done over and over again for the same policy and block of policies. So, each compute node endures a one-time overhead during formation of a collection object containing indirect addresses to the appropriate tabular entries. The lookup operation, even for thousands of policies, is trivialized once this collection is prepared.

The same technique could have been used for a block of scenarios instead of for a block of policies. The difference lies in that there are many *policy* parameter locations (mortality, lapse, surrender charges, and so forth) whereas there is only one location for a given investment scenario. The benefit of indirect addressing with a collection of policy information dominates in an overall runtime reduction. And, the parallelization approach is defined.

Parallelization for the Rate Book Pricing Example

The non-life example involves pricing given a certain parameter profile. The parallelization is approached on a profile basis. The compute node chore is to do the necessary workup for a policy class

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

meeting certain conditions. This example involves a simulation for each parameter combination or profile; the simulation is uniquely defined by the policy parameters. The pricing is based on a quantile of the aggregate loss probability distribution, and that distribution is derived in a compute node by simulation of a frequency-severity convolution in which censoring of both frequency and severity is done during the simulation. In the example problem, one million simulations for each sweep or parameter combination were done. This aggregate loss derivation is rendered within the HPC_Execute procedure, and it is not parallelized. Rather the parallelization requires routing each of 153,747 combinations to a compute node where the simulation of the aggregate loss distribution occurs.

The information transmitted from the HPC client workbook during each partition is one of the profiles—a combination of parameter values. This includes information for the Poisson frequency and maximum frequency cutoff sensor. It also includes information about the severity truncation as well as all other sensors that figure in the determination of cash flows. This information is embedded in the HPC_Partition variant data that is passed to the compute node workbook procedure, the HPC_Execute function's argument. The pricing information (in the form of aggregate loss distribution statistics and quantiles) is passed to the client from the HPC_Execute function as the inbound argument for HPC_Merge.

Conclusion

Excel is an important tool in an actuary's toolkit. But, it often is too slow reaching a solution to complex problems involving large seriatim valuations or requiring high-volume simulations. And, actuaries historically resort to compiled code, approximations and other remedies do deal with this runtime problem. HPC Excel 2010 opens new opportunities for them to consider solutions previously rejected as too time consuming.

This article describes what the authors deem to be the most difficult obstacle for the actuarial programmer, asynchronous or event programming. This means that the serial processing using well-known for-loops is abandoned. The object of parallelization, the execution procedure, is run on a massively parallel basis in many instances of Excel as a SOA service. These executions were, perhaps, a *former* member of a loop. Each instance of Excel running on a cluster compute node uses the same workbook containing the same VBA code. There is no relationship between the outbound order of the partitions fashioned by the HPC client to the compute nodes and the inbound merging of compute node results.

The actuary must fashion the execute algorithm from the interior of a repetitive looping. This requires identifying exactly what will be done in parallel across the grid. There is no loss of Excel capabilities because techniques deployed on a standalone basis are available during the HPC_Execute. There often will be several ways to accomplish parallelization. Somewhat paradoxically, the separation of simulations may not always be the best parallelization; in the rate book example simulations were not parallelized. In this example, parallelization was based on policy classification attributes. Similarly, what seems like a policy-based parallelization was rejected for the life insurance example. Seriatim valuations are repetitive and often would be the focus of parallelization. However, this was not the case for the

This paper has been accepted for publication in *Contingencies*. It has been posted with the expressed permission of the Editor, and it may not be copied or distributed without the Editor's permission.

derivation of an aggregate distribution for a market-consistent valuation of a block of life insurance policies. Because the entire block was valued for each financial simulation, the partition was based on investment path simulation and not by policy.

Microsoft HPC Excel 2010 has dramatically changed the worthiness of this time-honored, widely used actuarial tool. VBA programming coupled with cellular logic offers exceptional flexibility, and this remains in HPC. There are few actuarial solutions that do not have their genesis in an Excel workbook. But, slow runtimes have led actuaries to ramp speed up using compiled add-Ins. These still can be used in parallel on many compute nodes. The concept of a powerful, but slow actuarial workbench has now been replaced by an actuarial studio cluster, one that allows the actuary to produce Excel solutions that can be designed, tested and placed in the production environment of a high performance computational computer cluster.