# CVA in the Cloud

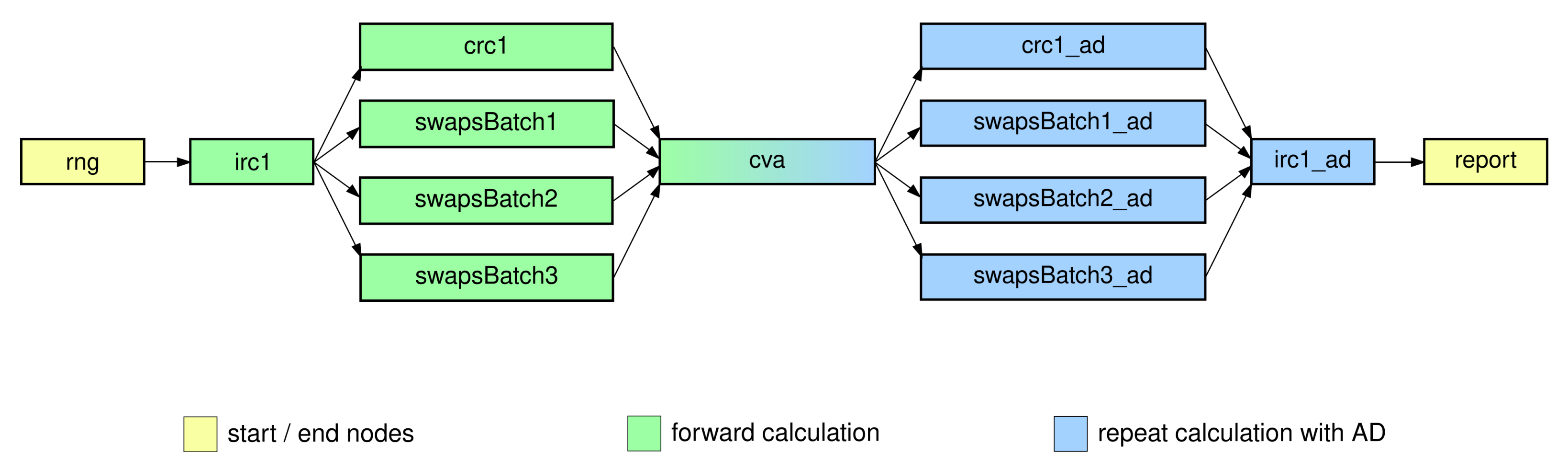
*Nick Dingle   Jacques du Toit   Ian Hotchkiss   Viktor Mosenkis   Justin Ware*

## Introduction

NAG has developed in collaboration with Xi-FINTIQ a CVA demonstration code to show how the NAG Library and the Algorithmic Differentiation (AD) tool dco/c++ can be combined with Origami to solve large scale CVA computations. Here we report our experiences running this demonstrator on a commercial cloud.

## CVA Demonstrator

Origami is a lightweight task execution framework. Users combine tasks into a task graph that Origami can execute on an ad-hoc cluster of workstations, on a dedicated in-house grid, on production cloud, or on a hybrid of all these. Origami handles all data transfers.



In our CVA demonstrator the trades in netting sets are valued in batches. CVA is calculated per netting set by running the code forward as normal. The graph is then reversed and the `dco/c++` adjoint version of each task is run to calculate sensitivities with respect to market instruments. The resulting graph has a large number of tasks with non-trivial dependencies which Origami automatically processes and executes.

## Optimizations

Before running in the cloud we profiled our demonstrator and identified two main opportunities for performance improvement:

- Reducing the amount of I/O, as this may perform poorly;
- Optimizing the adjoint calculations, which consumed a significant proportion of the execution time.

## Reducing I/O

Earlier versions of our CVA demonstrator relied on writing intermediate results (e.g. the outputs from the intermediate tasks in the graph) out to disk. Because this can introduce a large performance penalty, we modified the demonstrator to maintain these results in memory instead.

## Writing a Symbolic Adjoint

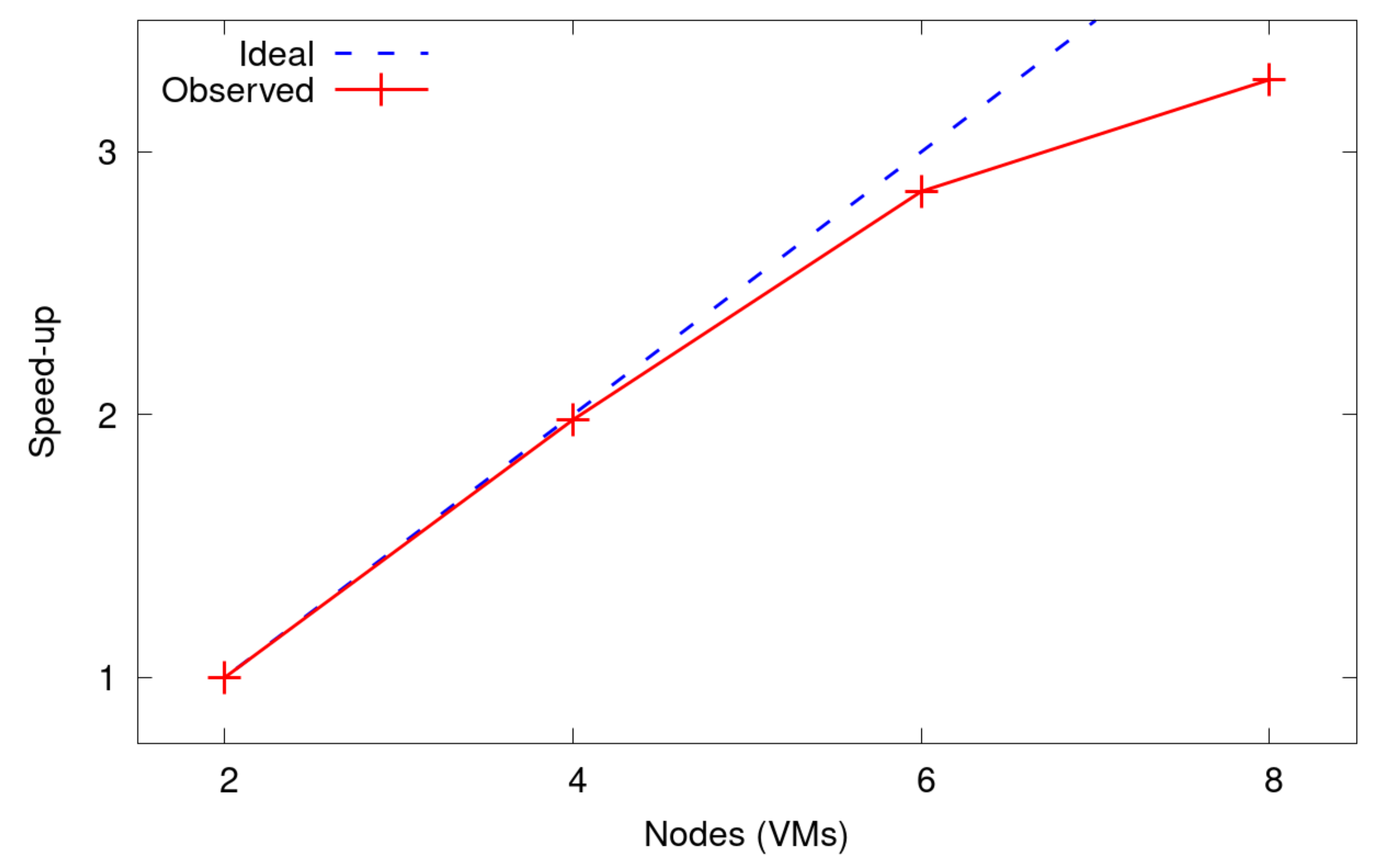
The original version of the CVA demonstrator spent approximately 25% of its runtime calculating the adjoint of the NAG Library linear regression routine `g02daf`. We therefore wrote a symbolic adjoint to improve the performance of the AD tasks:

```
! Assumes we are solving A^T*Ax = A^Tb (C:= A^T*A, d:=A^Tb)
! Compute adjoints of Cx = d and propagate adjoints of C & d to A & b
Subroutine g02da_sym (m, n, A, Aa, lda, b, ba, x, xa, Q, tau, &
     work, lwork, ifail)
  use nag_library, only: nag_wp
  implicit none
  Integer, Intent(in) :: m, n, lda, lwork
  Real(Kind=nag_wp), Intent(In) :: A(lda,n), b(m), x(n), Q(lda,n), tau(n)
  Real(Kind=nag_wp), Intent(Inout):: Aa(lda,n), ba(m), xa(n), work(lwork)
  Integer, Intent(Inout) :: ifail
  Real(Kind=nag_wp), Parameter :: one = 1.0_nag_wp

  Call dtrtrs('Upper','Transpose','Non-Unit',n,1,Q,lda,xa,n,ifail)
  Call dtrtrs('Upper','No␣Transpose','Non-Unit',n,1,Q,lda,xa,n,ifail)
  Call dgemv('No␣Transpose', m, n, one, A, lda, xa, 1, one, ba, 1)
  Call dger(m, n, one, b, 1, xa, 1, Aa, lda)
  work = 0.0_nag_wp
  Call dsyr2('Upper', n, -one, x, 1, xa, 1, work, n)
  Call dsymm('Right','Upper',m, n, one, work, n, A, lda, one, Aa, lda)
End Subroutine g02da_sym
```

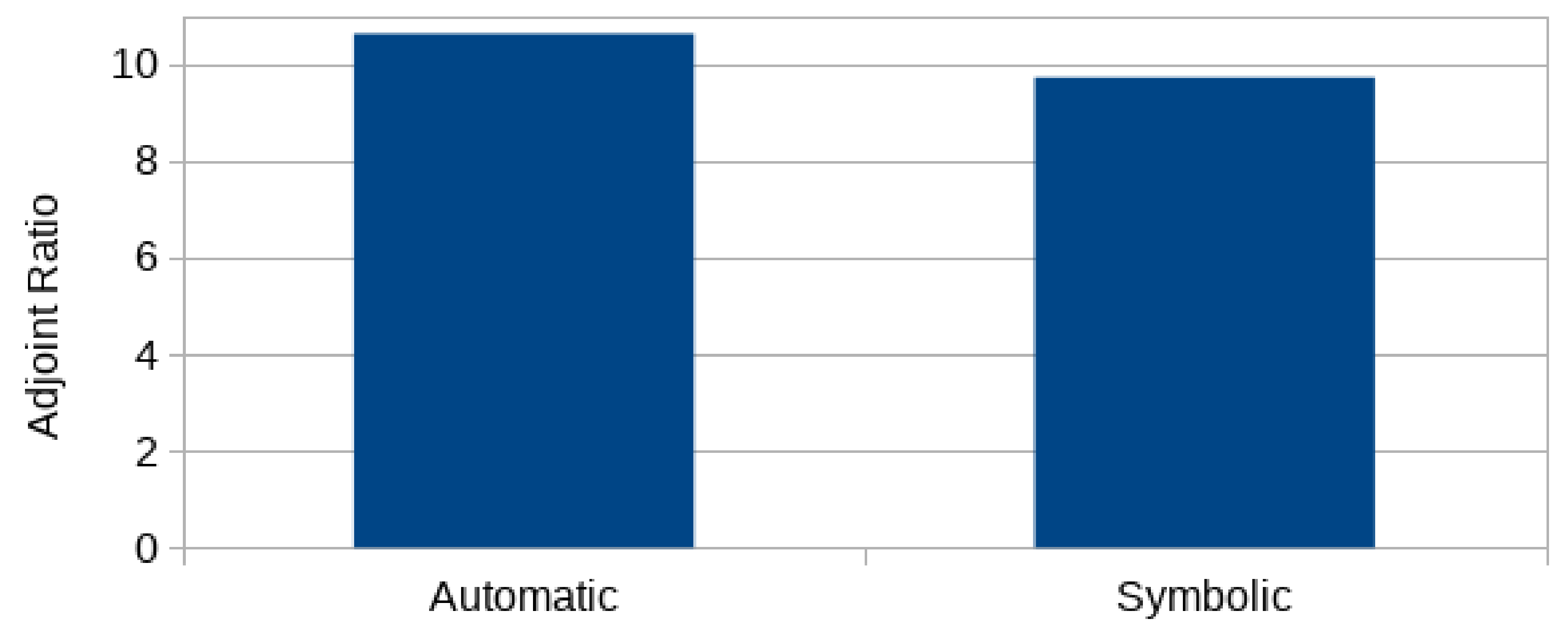
## Scaling

We ran the code on Microsoft's Azure cloud computing service. We used `D4s_v3` virtual machine (VM) images with 4 virtual CPUs and 16GB RAM running Ubuntu Server 18.04 LTS. The input data set contained 8 netting sets comprising a total of 28 843 swaps and 23 875 Bermudan swaptions. The Monte Carlo simulation used 2 000 paths. The code scales well as the number of VMs is increased.



## Adjoint Efficiency

We measure the efficiency of our AD scheme by the *adjoint ratio*: the runtime of the AD computation divided by the runtime of the forward computation (lower is therefore better). We observe that the symbolic adjoint reduces the adjoint ratio:



## Performance

Computing CVA and sensitivities using Origami and AD offers significant performance benefits compared with using finite differences and legacy grid execution software which might take many hours.

| | Elapsed Time |
|---|---|
| Origami (4 VMs x 4 cores) and AD | 49m 08s |
| Origami and AD, symbolic adjoint | 36m 23s |

## Possible Future Work

- Investigate different checkpointing strategies with the aim of reducing the code's memory consumption;
- Port the code to other operating architectures, e.g. GPUs.

## Availability

To find out more about this work and related NAG products please contact **support@nag.co.uk**.

## Acknowledgements

We gratefully acknowledge the support of the POP CoE which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 676553 and 824080, and has partially contributed to this work.

We would like to thank Microsoft for providing the Azure Sponsorship used to produce the results presented here.