Parallel and other simulations in R made easy: An end-to-end study

Marius Hofert

(joint work with Martin Mächler)

2015-05-30

Department of Statistics and Actuarial Science

UNIVERSITY OF WATERLOO

A motivating example from QRM

- Consider the value of a portfolio of d stocks $S_{t,1}, \dots, S_{t,d}$

$$V_t = \sum_{j=1}^{d} \beta_j S_{t,j}.$$

- The one period ahead loss can be expressed in terms of the risk-factor changes $X_{t+1,j} = \log S_{t+1,j} - \log S_{t,j} = \log(S_{t+1,j}/S_{t,j})$ as

$$L_{t+1} = -(V_{t+1} - V_t) = -\sum_{j=1}^{d} w_{t,j}(\exp(X_{t+1,j}) - 1), \quad w_{t,j} = \beta_j S_{t,j}.$$

Key tasks: 1) Find an adequate (copula) model for $\boldsymbol{X}_{t+1}$

2) Find the df $F_{L_{t+1}}$ of L_{t+1}

3) Compute risk measures such as $\mathrm{VaR}_\alpha(L_{t+1}) = F^-_{L_{t+1}}(\alpha)$

How to conduct simulation studies?

- Embarrasingly parallel problems (without machinery like Apache Hadoop)
- Package `simsalapar` (simulations simplified and launched parallel)
- For students (master/PhD), researchers (new models), practitioners (time constraints, validating internal models)

A simulation consists of the following parts:

1) Setup:
   - Scientific problem/question
   - Translating it to R (determining input variables and their "type")
   - Implementing the main, problem specific function (`doOne()`)
2) Conducting the simulation: sequentially? in parallel? nodes or cores?
3) Analyzing the results: Computing and presenting statistics (with tables or graphics)

Example: Computing VaR_α via Monte Carlo

Recall that

$$L_{t+1} = -(V_{t+1} - V_t) = -\sum_{j=1}^{d} w_{t,j}(\exp(X_{t+1,j}) - 1), \quad w_{t,j} = \beta_j S_{t,j}.$$

Scientific problem: Compute $\mathrm{VaR}_\alpha(L_{t+1})$ via Monte Carlo under various setups and investigate its behavior (w.l.o.g. $w_{t,j} = 1$)

In R:

Variable	expression	type	value
`n.sim`	N_{sim}	N	32
`n`	n	grid	64, 256
`d`	d	grid	5, 20, 100, 500
`varWgts`	$\mathbf{w}$	frozen	1, 1, 1, 1
`qF`	F^{-1}	frozen	qF
`family`	C	grid	Clayton, Gumbel
`tau`	τ	grid	0.25, 0.50
`alpha`	α	inner	0.950, 0.990, 0.999

N: The variable of type N gives the number of simulation replications.

frozen: These variables remain the same throughout the study.

grid: A unique combination of these variables builds the (physical) grid. The simulation will iterate N_{sim} times over all rows of this grid (virtual grid). The computations for one row in the virtual grid form a sub-job, possibly dealt with in parallel.

inner: Variables of this type are all dealt with within a sub-job.

```
require("simsalapar")
varList ←
    varlist( # constructor for an object of class 'varlist'
        n.sim = list(type="N", expr = quote(N[sim]), value = 32), # replications
        n = list(type="grid", value = c(64, 256)), # sample size
        d = list(type="grid", value = c(5, 20, 100, 500)), # dimension and weights
        varWgts = list(type="frozen", expr = quote(bold(w)),
                       value = list("5"=1, "20"=1, "100"=1, "500"=1)),
        qF = list(type="frozen", expr = quote(F^{-1}), value=list(qF=qnorm)), # margins
        family=list(type="grid", expr = quote(C), value = c("Clayton", "Gumbel")), # C
        tau = list(type="grid", value = c(0.25, 0.5)), # Kendall's tau
        alpha = list(type="inner", value = c(0.95, 0.99, 0.999))) # confidence levels
toLatex(varList) # method for constructing the above table
```

(Physical) grid

n	d	C	τ
64	5	Clayton	0.25
256	5	Clayton	0.25
64	20	Clayton	0.25
256	20	Clayton	0.25
64	100	Clayton	0.25
256	100	Clayton	0.25
⋮	⋮	⋮	⋮
64	500	Gumbel	0.50
256	500	Gumbel	0.50

sub-job

Virtual grid

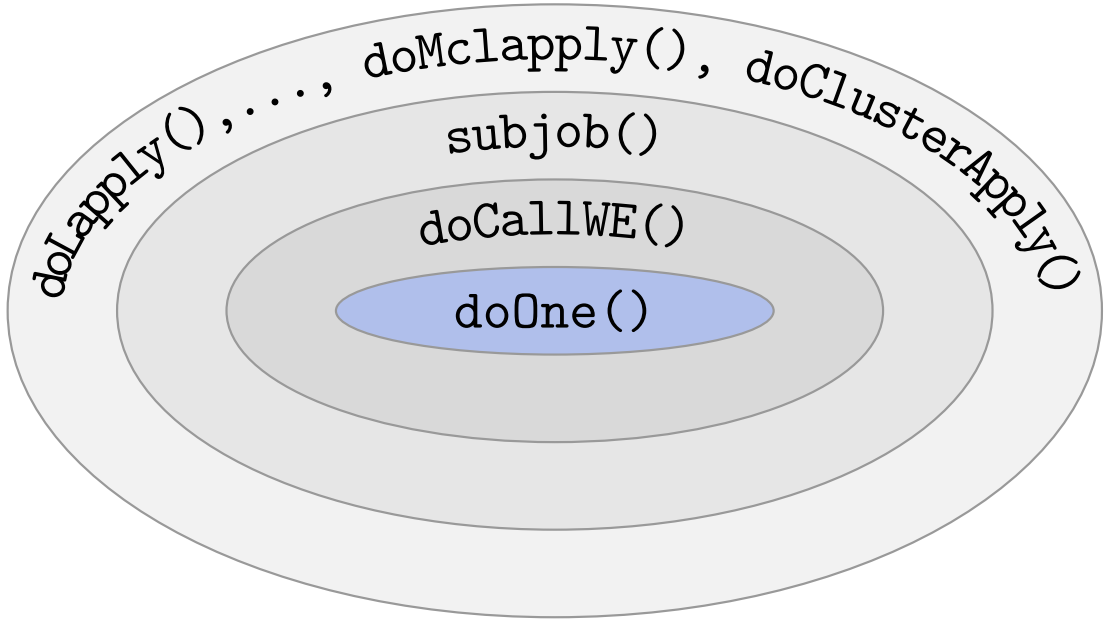
n	d	C	τ
(Physical) grid 1			
(Physical) grid 2			
⋮			
⋮			
(Physical) grid N_{sim}			

⇒ Use the same seed for different lines in the physical grid, but different seeds for different blocks of physical grids.

The workhorse doOne(): Computation for one set of variables

```
1  doOne ← function(n, d, qF, family, tau, alpha, varWgts, names=FALSE)
2  {
3      ## checks (and load required packages here for parallel computing later on)
4      w ← varWgts[[as.character(d)]]
5      stopifnot(require(copula), # load 'copula'
6                sapply(list(w, alpha, tau, d), is.numeric)) # sanity checks
7
8      ## simulate risk-factor changes (if defined outside doOne(), use
9      ## doOne ← local({...}) construction as in some of simsalapar's demos)
10     simRFC ← function(n, d, qF, family, tau) {
11         ## define the copula of the risk factor changes
12         theta ← getAcop(family)@iTau(tau) # determine copula parameter
13         cop ← onacopulaL(family, list(theta, 1:d)) # define the copula
14         ## sample the meta-copula-model for the risk-factor changes X
15         qF(rCopula(n, cop)) # simulate via Sklar's Theorem
16     }
17     X ← simRFC(n, d=d, qF=qF[["qF"]], family=family, tau=tau) # simulate X
18
19     ## compute the losses and estimate VaR_alpha(L)
20     L ← -rowSums(expm1(X) * matrix(rep(w, length.out=d),
21                                    nrow=n, ncol=d, byrow=TRUE)) # losses
22     quantile(L, probs=alpha, names=names) # empirical quantile as VaR estimate
23 }
```

Black boxes:



`doOne()`: Computing a value (for one set of variables; numeric array)

`doCallWE()`: Catching warnings, errors, run time (via `tryCatch.W.E()`)

`subjob()`:
- Computes a sub-job (= line in virtual grid)
- Seeding: see `?subjob`; defaults to `1:n.sim` (same for each row in physical grid); L'Ecuyer-CMRG is available

`do*()`:
- Calls `subjob()` sequentially or in parallel (nodes/cores),
- Calls `saveSim()` which tries to create (via `mkAL()`) an array of 4-/5-lists and saves it in an `.rds`.

Compute and extract results (value, error, warning, time, (seed)):

```
> res  <- doClusterApply(varList, sfile="res.rds", doOne=doOne, names=TRUE) # results
> val  <- getArray(res) # array of values
> err  <- getArray(res, "error") # array of error indicators
> warn <- getArray(res, "warning") # array of warning indicators
> time <- getArray(res, "time") # array of user times in ms
```

A quick look at the errors:

```
> ftable(100 * err, row.vars=c("family", "d"), col.vars=c("tau", "n")) # % of errors
```

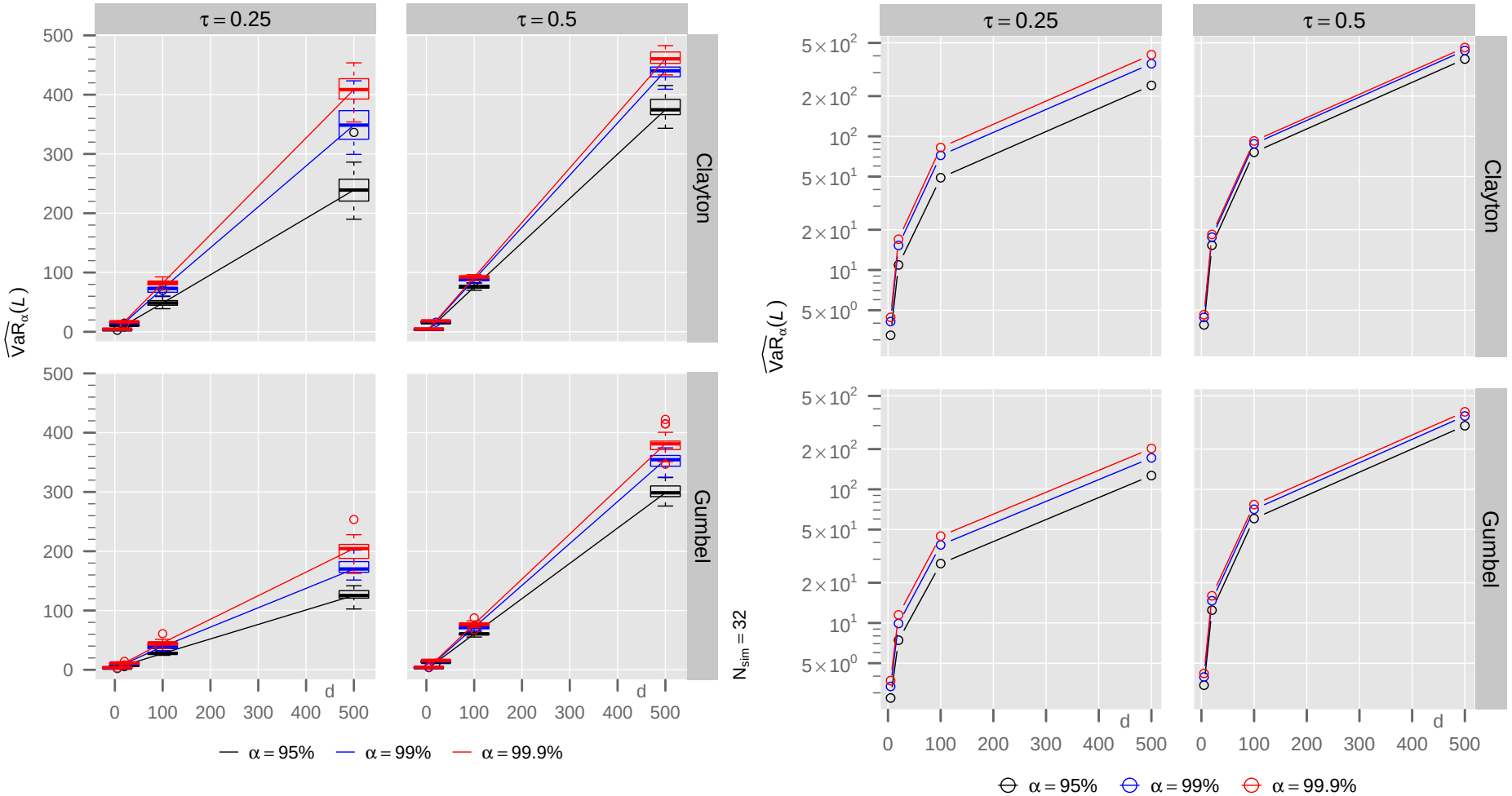
```
            tau 0.25     0.50
            n     64 256   64 256
family  d
Clayton 5          0   0    0   0
        20         0   0    0   0
        100        0   0    0   0
        500        0   0    0   0
Gumbel  5          0   0    0   0
        20         0   0    0   0
        100        0   0    0   0
        500        0   0    0   0
```

(Much) more sophisticated tools...

```
ft ← ftable(<formatted res>, row.vars=c("family","n","d"), col.vars=c("tau","alpha"))
tabL ← toLatex(ft, vList = varList, fontsize = "scriptsize", caption = "...")
```

		τ	0.25			0.50		
C	n	$d \mid \alpha$	95%	99%	99.9%	95%	99%	99.9%
Clayton	64	5	3.1 (0.4)	3.8 (0.4)	4.0 (0.5)	3.6 (0.3)	4.2 (0.2)	4.4 (0.2)
		20	10.6 (1.4)	13.5 (1.5)	14.8 (2.2)	14.2 (1.6)	16.7 (1.0)	17.4 (1.0)
		100	46.1 (9.1)	63.5 (11.6)	68.5 (13.6)	70.7 (8.6)	83.7 (3.9)	86.7 (4.2)
		500	224.8 (50.6)	307.8 (61.5)	336.0 (66.8)	350.0 (40.5)	418.6 (22.3)	434.0 (21.4)
	256	5	3.2 (0.2)	4.1 (0.2)	4.4 (0.2)	3.9 (0.2)	4.4 (0.1)	4.6 (0.1)
		20	10.9 (1.0)	15.3 (1.2)	17.0 (0.9)	15.3 (0.7)	17.6 (0.5)	18.5 (0.6)
		100	49.0 (5.5)	72.1 (7.7)	82.5 (4.8)	76.0 (3.4)	87.9 (2.7)	92.3 (3.0)
		500	240.4 (27.0)	349.7 (35.3)	408.5 (24.3)	378.8 (17.4)	439.4 (12.7)	461.7 (14.2)
Gumbel	64	5	2.7 (0.3)	3.3 (0.4)	3.4 (0.5)	3.3 (0.3)	3.8 (0.3)	4.0 (0.2)
		20	7.3 (1.1)	9.4 (1.2)	10.1 (1.5)	12.2 (0.6)	14.0 (1.2)	14.6 (1.2)
		100	26.0 (4.2)	35.8 (4.7)	38.5 (5.6)	57.7 (5.1)	67.7 (4.8)	70.3 (5.4)
		500	117.2 (12.5)	154.4 (19.0)	167.5 (18.2)	288.2 (18.0)	333.7 (23.0)	347.9 (20.7)
	256	5	2.7 (0.2)	3.3 (0.2)	3.7 (0.2)	3.4 (0.2)	3.9 (0.1)	4.2 (0.1)
		20	7.4 (0.5)	9.9 (0.8)	11.5 (0.9)	12.5 (0.4)	14.7 (0.7)	16.0 (0.6)
		100	27.8 (2.8)	38.4 (3.1)	44.7 (3.2)	60.4 (2.3)	70.9 (2.5)	76.9 (3.5)
		500	126.8 (10.3)	171.9 (11.2)	202.3 (13.5)	299.1 (13.7)	353.8 (13.2)	380.0 (9.7)

Graphics with `mayplot()`:



- $\mathrm{VaR}_\alpha(L)$ is increasing in d although $\mathbb{E}[L] = d(1 - \sqrt{e}) \approx -0.65d \downarrow$
- There could be done more on the graphical side...

To summarize

- Proceed as follows:

  Step 1: Determine the input variables.

  Step 2: Determine their types;
  Create the variable list with `varlist()`.

  Step 3: Implement (and test) the problem-specific function `doOne()`

  Step 4: Run the simulation sequentially with `doLapply()` or in parallel with `doMclapply()` or `doClusterApply()`. This can involve replicates via a variable of type "N" as our `n.sim` (N_{sim}).

  Step 5: Analyze the results (= values, warnings, errors, run time and possibly `.Random.seed`) (see `toLatex()` and `mayplot()`).

- For computing worst VaR under no information about the copula, use ARA() from `qrmtools`.

The/A future: wearable devices

- `demo(VaRsuperadd)`
- run on a Nexus 5...
- ... on two cores (!)