

# Practical experience with the sfCluster/snowfall system while developing the peperr package

Christine Porzelius

Freiburg Center for Data Analysis and Modeling, University of Freiburg  
Institute of Medical Biometry and Medical Informatics, University Medical Center Freiburg  
DFG Forschergruppe FOR 534

cp@fdm.uni-freiburg.de

December 12, 2008



- Situation** Resampling techniques for data based model evaluation, such as cross-validation or the bootstrap, are
- ▶ **essential**, though often not (properly) applied, as not trivial,
  - ▶ very **time consuming**, especially in high-dimensional data settings,
  - ▶ but **predestined for parallelization**.
- Aim** Developing a module that provides
- ▶ **simple** parallelization of the resampling procedure and
  - ▶ **flexibility** to allow evaluation of new model fitting techniques.
- Result** peperr: an R package for **parallelized estimation of prediction error**

# Procedure of prediction error measurement

1. Repeated generation of training and test data sets from the original data
2. For each split:
  - 2.1 **Selection of model complexity** (if desired), using the training data, e.g. number of boosting steps or shrinkage factor
  - 2.2 **Fit** of the model with the selected or a given complexity on the training data
  - 2.3 **Measurement of prediction error** on the corresponding test set
3. Aggregation of the obtained prediction error measurements into one overall measure, e.g. misclassification rate in binary response settings or prediction error curves in survival data

# Function peperr

Minimal call:

```
peperr(response, x,  
       fit.fun=fit.CoxBoost, complexity=complexity.mincv.CoxBoost)
```

response	time-to-event or binary response
x	data set
fit.fun	function to fit predictive model
complexity	function for complexity selection or fixed value

fit.fun and complexity can be user-defined or functions already provided by peperr

# Function peperr

Minimal call:

```
peperr(response, x,  
       fit.fun=fit.CoxBoost, complexity=complexity.mincv.CoxBoost)
```

response	time-to-event or binary response
x	data set
fit.fun	function to fit predictive model
complexity	function for complexity selection or fixed value

fit.fun and complexity can be user-defined or functions already provided by peperr

## WHAT HAPPENS INSIDE?

# Workflow: creating parallel running R programs

1. Identify parts of code that are
  - ▶ **computationally intensive** and
  - ▶ **parallelizable**, i.e., independent of each other, e.g. functions of the apply family, repeated simulations.

# Workflow: creating parallel running R programs

1. Identify parts of code that are
  - ▶ **computationally intensive** and
  - ▶ **parallelizable**, i.e., independent of each other, e.g. functions of the apply family, repeated simulations.
2. Convert code using `snowfall` functions  
Typical structure:
  - A. Initialization of cluster
  - B. Export objects, data and libraries on slaves
  - C. Wrapper function
  - D. Calculation on slaves
  - E. Stop cluster

# Workflow: creating parallel running R programs

1. Identify parts of code that are
  - ▶ **computationally intensive** and
  - ▶ **parallelizable**, i.e., independent of each other, e.g. functions of the apply family, repeated simulations.
2. Convert code using `snowfall` functions  
Typical structure:
  - A. Initialization of cluster
  - B. Export objects, data and libraries on slaves
  - C. Wrapper function
  - D. Calculation on slaves
  - E. Stop cluster
3. Run R code
  - ▶ sequentially (no further installation necessary, e.g. for development),
  - ▶ using *sfCluster* (recommended),
  - ▶ or with socket cluster, MPI, PVM or NWS.

## Model evaluation

```
cplx <- complexity(x, y)
fit <- fit.fun(x, y, cplx)
pred.fun(fit, x.new)
```

x	data set
y	time-to-event or binary response
x.new	new data
cplx	model complexity parameter, optional
complexity	function to determine optimal complexity value(s)
fit	fitted prediction model
fit.fun	function to fit predictive model
pred.fun	function to determine prediction error

## Resampling procedure

```
for (i in 1:n.resample){  
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])  
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)  
  pred.fun(fit, x.test[[i]])  
}
```

n.resample	number of train and test data sets
x.train	list of training data sets obtained by resampling
y.train	corresponding response
cplx.resample	model complexity parameter in given training set
x.test	list of test data sets obtained by resampling
y.test	corresponding response

## Improved resampling

```
resampling.fun <- function(i){  
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])  
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)  
  pred.fun(fit, x.test[[i]])  
}  
  
lapply(1:n.resample, resampling.fun)
```

## Recall: parallelization using `snowfall` functions

- A. **Initialization of cluster**
- B. Export objects, data and libraries on slaves
- C. Wrapper function
- D. Calculation on slaves
- E. Stop cluster

## Initialization and information

- ▶ **Initialization** and setup of cluster:

If R started with *sfCluster*, just call

```
> sfInit()
```

else specify number of cpus and type of cluster, e.g.

```
> sfInit(parallel=TRUE, cpus=3, type="SOCK")
```

Note: change of code not mandatory, also possible via command line

⇒ Issues concerning computation and computing are separated

# Initialization and information

- ▶ **Initialization** and setup of cluster:

If R started with *sfCluster*, just call

```
> sfInit()
```

else specify number of cpus and type of cluster, e.g.

```
> sfInit(parallel=TRUE, cpus=3, type="SOCK")
```

Note: change of code not mandatory, also possible via command line

⇒ Issues concerning computation and computing are separated

- ▶ **Information:** e.g.

```
> sfParallel()
```

```
> TRUE
```

```
> sfCpus()
```

```
> [1] 3
```

```
> sfType()
```

```
> "SOCK"
```

## Recall: parallelization using snowfall functions

- A. Initialization of cluster
- B. **Export objects, data and libraries on slaves**
- C. Wrapper function
- D. Calculation on slaves
- E. Stop cluster

## snowfall functions for convenient computing

Slave R processes are own R instances, so all objects, libraries, etc. required for computation on nodes have to be transferred manually from the master R process:

- ▶ Export of libraries and code:  
`sfLibrary()` and `sfSource()`
- ▶ Export and removal of global and local variables and functions:  
`sfExport()`, `sfExportAll()`,  
`sfRemove()`, `sfRemoveAll()`

## Recall: parallelization using `snowfall` functions

- A. Initialization of cluster
- B. Export objects, data and libraries on slaves
- C. Wrapper function
- D. **Calculation on slaves**
- E. **Stop cluster**

## snowfall functions for computing

- ▶ **Parallel computing** on cluster:  
`sfApply()`, `sfLapply()`, `sfSapply()`,  
`sfClusterApply()`, `sfClusterApplyLB()`, `sfClusterApplySR()`,...
- ▶ **Stop** cluster:  
`sfStop()`

## Load Balancing: function `sfClusterApplyLB`

- ▶ `sfClusterApplyLB` is wrapper around `snow` function `clusterApplyLB`
- ▶ Suppose: number of available CPUs  $n_{CPU_s} <$  number of jobs to execute in parallel
- ▶ Usual procedure: first  $n_{CPU_s}$  jobs are started, after finishing of **all** jobs the next  $n_{CPU_s}$  jobs, and so on
- ▶ Not efficient, if cluster nodes inhomogeneous or computation times of jobs differ
- ▶ Load balancing: immediately after one job completes, the next is placed on that node
- ▶ `sfClusterApplyLB` is advisable for many jobs with potentially different computation time

## Intermediate result saving: function `sfClusterApplySR`

- ▶ Additional function to `snow` functions
- ▶ Not yet compatible with load balancing
- ▶ After each block of  $n_{CPU_S}$  jobs, results are saved automatically
- ▶ Computation can be restarted with restored results after interruption
- ▶ Useful in case of long computations and potential shutdown of nodes
- ▶ Multiple calls possible, as access via specified names
- ▶ If random number generator is involved, use with care
- ▶ Advisable for long computations

## Intermediate result saving: function `sfClusterApplySR`

- ▶ Additional function to snow functions
- ▶ Not yet compatible with load balancing
- ▶ After each block of  $n_{CPUs}$  jobs, results are saved automatically
- ▶ Computation can be restarted with restored results after interruption
- ▶ Useful in case of long computations and potential shutdown of nodes
- ▶ Multiple calls possible, as access via specified names
- ▶ If random number generator is involved, use with care
- ▶ Advisable for long computations

```
> res1 <- sfClusterApplySR(x1, fun1)
> res2 <- sfClusterApplySR(x2, fun2, name="calc2")
> # interruption during second call
```

## Intermediate result saving: function `sfClusterApplySR`

- ▶ Additional function to `snow` functions
- ▶ Not yet compatible with load balancing
- ▶ After each block of  $n_{CPUs}$  jobs, results are saved automatically
- ▶ Computation can be restarted with restored results after interruption
- ▶ Useful in case of long computations and potential shutdown of nodes
- ▶ Multiple calls possible, as access via specified names
- ▶ If random number generator is involved, use with care
- ▶ Advisable for long computations

```
> res1 <- sfClusterApplySR(x1, fun1)
> res2 <- sfClusterApplySR(x2, fun2, name="calc2")
> # interruption during second call
> res1 <- sfClusterApplySR(x1, fun1, restore=TRUE)
> # complete result loaded
```

## Intermediate result saving: function `sfClusterApplySR`

- ▶ Additional function to snow functions
- ▶ Not yet compatible with load balancing
- ▶ After each block of  $n_{CPUs}$  jobs, results are saved automatically
- ▶ Computation can be restarted with restored results after interruption
- ▶ Useful in case of long computations and potential shutdown of nodes
- ▶ Multiple calls possible, as access via specified names
- ▶ If random number generator is involved, use with care
- ▶ Advisable for long computations

```
> res1 <- sfClusterApplySR(x1, fun1)
> res2 <- sfClusterApplySR(x2, fun2, name="calc2")
> # interruption during second call
> res1 <- sfClusterApplySR(x1, fun1, restore=TRUE)
> # complete result loaded
> res2 <- sfClusterApplySR(x2, fun2, restore=TRUE, name="calc2")
> # restart at point of interruption
```

- ▶ Note: Change of code ('`restore=TRUE`') can easily be avoided.

# Parallelization using snowfall functions

- A. Initialization of cluster  
`sfInit()`

## Parallelization using snowfall functions

### A. Initialization of cluster

```
sfInit()
```

### B. Export objects, data and libraries on slaves

```
sfExport("x.train", "y.train", "x.test", "y.test",  
        "complexity", "fit.fun", "pred.fun")
```

## Parallelization using snowfall functions

### A. Initialization of cluster

```
sfInit()
```

### B. Export objects, data and libraries on slaves

```
sfExport("x.train", "y.train", "x.test", "y.test",  
        "complexity", "fit.fun", "pred.fun")
```

### C. Wrapper function

```
resampling.fun <- function(i){  
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])  
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)  
  pred.fun(fit, x.test[[i]])  
}
```

## Parallelization using snowfall functions

### A. Initialization of cluster

```
sfInit()
```

### B. Export objects, data and libraries on slaves

```
sfExport("x.train", "y.train", "x.test", "y.test",  
        "complexity", "fit.fun", "pred.fun")
```

### C. Wrapper function

```
resampling.fun <- function(i){  
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])  
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)  
  pred.fun(fit, x.test[[i]])  
}
```

### D. Calculation on slaves

```
sfClusterApplyLB(1:n.resample, resampling.fun)
```

## Parallelization using snowfall functions

### A. Initialization of cluster

```
sfInit()
```

### B. Export objects, data and libraries on slaves

```
sfExport("x.train", "y.train", "x.test", "y.test",  
        "complexity", "fit.fun", "pred.fun")
```

### C. Wrapper function

```
resampling.fun <- function(i){  
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])  
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)  
  pred.fun(fit, x.test[[i]])  
}
```

### D. Calculation on slaves

```
sfClusterApplyLB(1:n.resample, resampling.fun)
```

### E. Stop cluster

```
sfStop()
```

## Parallelization using snowfall functions

```
sfInit()

sfExport("x.train", "y.train", "x.test", "y.test",
        "complexity", "fit.fun", "pred.fun")

resampling.fun <- function(i){
  cplx.resample <- complexity(x.train[[i]], y.train[[i]])
  fit <- fit.fun(x.train[[i]], y.train[[i]], cplx.resample)
  pred.fun(fit, x.test[[i]])
}

sfClusterApplyLB(1:n.resample, resampling.fun)

sfStop()
```

To export all resampled data sets to all nodes is not efficient.  
⇒ Export whole data set plus a list of indices obtained by `peperr` function `resample.indices`

## peperr function resample.indices

```
indices <- resample.indices(n, sample.n)

sfInit()

sfExport("x", "y", "indices", "complexity", "fit.fun", "pred.fun")

resampling.fun <- function(i){
  cplx.resample <- complexity(x[indices[[i]]$train,],
    y[indices[[i]]$train])
  fit <- fit.fun(x[indices[[i]]$train,], y[indices[[i]]$train],
    cplx.resample)
  pred.fun(fit, x[indices[[i]]$test])
}

sfClusterApplyLB(1:n.resample, resampling.fun)

sfStop()
```

## Wrap into a function

```
peperr <- function(x, y, indices, complexity, fit.fun, ...){
  sfInit()
  sfExport("x", "y", "indices", "complexity", "fit.fun")
  resampling.fun <- function(i){
    cplx.resample <- complexity(x[indices[[i]]$train,],
      y[indices[[i]]$train])
    fit <- fit.fun(x[indices[[i]]$train,], y[indices[[i]]$train,],
      cplx.resample)
    pred.fun(fit, x[indices[[i]]$test])
  }
  res <- sfClusterApplyLB(1:n.resample, resampling.fun)
  sfStop()
  res
}
```

Note: strongly simplified for illustration

## Wrap into a function

```
peperr <- function(x, y, indices, complexity, fit.fun, ...){  
  sfInit()  
  sfExport("x", "y", "indices", "complexity", "fit.fun")  
  resampling.fun <- function(i){  
    cplx.resample <- complexity(x[indices[[i]]$train,],  
      y[indices[[i]]$train])  
    fit <- fit.fun(x[indices[[i]]$train,], y[indices[[i]]$train],  
      cplx.resample)  
    pred.fun(fit, x[indices[[i]]$test])  
  }  
  res <- sfClusterApplyLB(1:n.resample, resampling.fun)  
  sfStop()  
  res  
}
```

# Operating modes

snowfall functions work

- ▶ sequential, i.e. without compute cluster and even without snow
- ▶ with socket, MPI, PVM, NWS cluster and
- ▶ with *sfCluster* (recommended)

⇒ peperr function needs additional arguments:

```
peperr(..., parallel=NULL, cpus=2,  
        clustertype=NULL, clusterhosts=NULL, ...){  
  
    sfInit(parallel=parallel, cpus=cpus, type=clustertype,  
           socketHosts=clusterhosts)  
  
    ...  
}
```

Default setting corresponds to sequential use or use with *sfCluster*.

## peperr function

```
peperr <- function(x, y, indices, complexity, fit.fun, ...){
  sfInit()
  sfExport("x", "y", "indices", "complexity", "fit.fun")
  resampling.fun <- function(i){
    cplx.resample <- complexity(x[indices[[i]]$train,],
      y[indices[[i]]$train,])
    fit <- fit.fun(x[indices[[i]]$train,], y[indices[[i]]$train,],
      cplx.resample)
    pred.fun(fit, x[indices[[i]]$test])
  }
  sfClusterApplyLB(1:n.resample, resampling.fun)
  sfStop()
}
```

`sfExport` allows to export not only global (as snow function `clusterExport`) but also local variables!

## Additionally required libraries and objects

```
peperr <- function(x, y, indices, complexity, fit.fun, ...){
  sfInit()
  sfExport("x", "y", "indices", "complexity", "fit.fun")
  resampling.fun <- function(i){
    cplx.resample <- complexity(x.sample[[i]]$train,
      y.sample[[i]]$train)
    fit <- fit.fun(x[indices[[i]]$train,], y[indices[[i]]$train,]
      cplx.resample)
    pred.fun(fit, x[indices[[i]]$test])
  }
  sfClusterApplyLB(1:n.resample, resampling.fun)
  sfStop()
}
```

**Problem:** Functions `complexity` and `fit.fun` can be user defined.  
What if additional libraries, functions or global variables are required?

## Export required libraries and objects

peperr offers different possibilities:

- ▶ Argument `load.list`:

Default: automatically extraction of required libraries and objects via  
peperr function `extract.fun`:

```
peperr (... , load.list=extract.fun(complexity, fit.fun), ...)
```

Alternatively: user-defined `load.list`

## Export required libraries and objects

peperr offers different possibilities:

- ▶ Argument `load.list`:

Default: automatically extraction of required libraries and objects via peperr function `extract.fun`:

```
peperr (... , load.list=extract.fun(complexity, fit.fun), ...)
```

Alternatively: user-defined `load.list`

- ▶ Argument `load.all`:

Export of the whole global environment of the master and all loaded libraries:

```
peperr (... , load.all=TRUE, ...)
```

## Export required libraries and objects

peperr offers different possibilities:

- ▶ Argument `load.list`:

Default: automatically extraction of required libraries and objects via `peperr` function `extract.fun`:

```
peperr (... , load.list=extract.fun(complexity, fit.fun), ...)
```

Alternatively: user-defined `load.list`

- ▶ Argument `load.all`:

Export of the whole global environment of the master and all loaded libraries:

```
peperr (... , load.all=TRUE, ...)
```

- ▶ Argument `noclusterstart`:

Initialization of cluster is skipped, allowing to work on previous initialized cluster:

```
sfInit()
```

```
sfLibrary(RequiredLibrary)
```

```
sfExport("RequiredVariable1", "RequiredVariable2")
```

```
peperr (... , noclusterstart=TRUE, ...)
```

# Reproducibility

- ▶ To assure reproducibility of results, a `set.seed` call on the master R process is not sufficient, as it does not affect the slave R processes.
- ▶ The number of slaves as well as their order might differ from run to run.
- ▶ If computation on slaves involves random number generation, `peperr` assures reproducibility of results
  - ▶ by function `sfClusterSetupSPRNG` or `sfClusterSetupRNG` (argument `RNG="SPRNG"`, or `RNG="RNG"`) and specified argument `seed`, producing independent parallel random number streams
  - ▶ or alternatively by argument `RNG="fixed"` and specified `seed`:  
At each sample run on a slave process, a seed is set.  
Advantage: allows easier reproducibility if cluster was interrupted.

## Multiple parallelization instances

- ▶ Consecutive parallelization, e.g. more than one `sfClusterApply` call in a row: possible
  - ▶ Use `sfRemove/sfRemoveAll`, if necessary
- ▶ Nested parallelization, e.g. parallelized cross-validation to determine complexity parameter within `peperr`: not possible/reasonable
  - ▶ No further speed-up obtainable
  - ▶ In running cluster no further subcluster can be set up

If parallelizable function is used within already parallelized environment, use `sfInit(nostart=TRUE)` (corresponding to `peperr` argument `noclusterstart=TRUE`), i.e. force sequential execution as cluster is already set up.

## Experience and availability

*sfCluster* and *snowfall* at our institute:

After solving initial problems and pitfalls it is working well for us, used

- ▶ almost daily
- ▶ by several users
- ▶ for more than a year now.

### **Availability:**

- ▶ *snowfall* and *peperr* on CRAN
- ▶ *sfCluster* together with extensive documentation:

**[www.imbi.uni-freiburg.de/parallel](http://www.imbi.uni-freiburg.de/parallel)**

## References concerning peperr

- ▶ Binder, H. and Schumacher, M. (2008a): Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples, *Statistical Application in Genetics and Molecular Biology*, **7**(1), 12.
- ▶ Gerds, T. and Schumacher, M. (2007): Efron-type measures of prediction error for survival analysis, *Biometrics*, **63**, 1283–1287.
- ▶ Schumacher, M. *et al.* (2007): Assessment of survival prediction models based on microarray data, *Bioinformatics*, **23**, 1768–1774.