

Tutorial:
Parallel computing using R package **snowfall**

Jochen Knaus, Christine Porzelius
Institute of Medical Biometry and Medical Informatics, Freiburg

June 29, 2009
Statistical Computing 2009, Reimsburg

Exercises

If you have not worked with `snow` or `snowfall` yet, we suggest to go through Section 2.2 first. An overview of the most important `snowfall` functions for this tasks is given in appendix A.

Please choose the exercises you want to do. The examples are artificial and kept very simple and short, but are designed to show some principles and problems encountered in real-world problems. If you are completely new to parallel computing, you probably want to stick with the [\[green\]](#) exercises, if you know what parallelization is all about or consider yourself an “R pro”, feel free to skip those.

Hints, solutions and additional information are given on the last pages. The working materials and programs can be downloaded under:

<http://imbi.uni-freiburg.de/parallel/reisensburg>

1. Getting started: initialization [\[short, understanding\]](#)

Initialize a cluster using any of the possible types (if you are working on a laptop, only use parallel mode “SOCK” and sequential mode). See Section 2.2 for help.

- Use functions `sfParallel`, `sfCpus`, `sfGetCluster`, `sfType`, `sfSession` and `sfSocketHosts` to get information about your cluster.
- Change these parameters through the commandline on R startup.

2. Parallelization principles: loop and code refactoring [\[short, easy\]](#)

Basic principle of most R parallelization packages are the “data parallel” (or “vectorized”) calls like `lapply` and `sapply`. Rewriting of program loops or sequential parts is mostly required to use parallel computing. Section 2.2.3 offers help.

- Given are four matrices a_1 , b_1 , c_1 , d_1 . All of them are multiplied with a constant matrix `constMat`. Write a program that can handle these multiplications in parallel.
- Suppose you have several lists containing matrices which need to be multiplied with that constant matrix. Expand your program for this case. Think about the fastest possibility of transporting of `constMat` (see Section 2.2.2).

3. Parallelization principles: scaling and depth [\[short, but maybe tricky\]](#)

Adding numbers (or any other associative problem) can be done in parallel as well. But there is – like in many real-world situations – a general limit of the benefit of parallelization.

- Think about this limit.
- Write a program that adds the numbers from 1 to 100 on multiple CPUs (in other words: replace `sum(1:100)` with a parallel program). Hint: You need several parallel calls.

4. **Parallelization principles: code parallelization** [short, tricky]

Suppose you are using time consuming functions, which cannot be parallelized itself. However, there may be several simulation scenarios using that functions or other independent tasks.

Write a program which runs four different functions at the same time: take the `sum`, find the `min`, `max` and `mean` of the numbers from 1 to 100.

5. **Parallel random numbers** [short, understanding]

Try out the three possibilities to assure reproducibility of parallel processes described in Section 3.1, as there are

- setting the same seed on each worker process,
- using independent random number streams and
- using a fixed seed for each computation on the worker processes.

6. **Real life situations: load balancing, network traffic** [medium, understanding]

In the tutorial materials there is a short program for calculation of the Mandelbrot set. On its parallelization, some side effects of parallelization become visible.

- *Modify the program to work on several CPUs.*¹
- *Performance can vary depending of the choice of the parallelization. You can create parallel versions, which run slower than a sequential run or nearly scale linear with the amount of CPU cores. What is happening in these cases? If it takes too long on your laptop: a table of runtimes can be found in Section 3.6.*
- *Load Balancing works very well with this problem (see Section 3.5). Why?*

7. **Intermediate result saving** [understanding]

Try out function `sfClusterApplySR`, using one of the wrapper functions given in Section 2.3, by interrupting the cluster manually. Does restoring also work, if computation was finished already?

¹There is an old programmer tale each man has to create a program for drawing the Mandelbrot set before he dies. If you haven't done this before in your life, you are doing this now – in a parallel version! Btw., we are thinking this tale is valid for women as well.

Contents

1	Introduction	3
1.1	How to parallelize problems?	3
2	Getting started	3
2.1	Installation	3
2.2	Writing parallel programs with <code>snowfall</code>	4
2.2.1	Initialization	4
2.2.2	Data exchange of objects and loading of packages	5
2.2.3	Wrapper functions	6
2.2.4	Stop cluster	8
2.3	Typical examples	8
3	snowfall advanced	10
3.1	Parallel random numbers	10
3.2	Intermediate result saving	11
3.3	Using <code>snowfall</code> within functions/packages	12
3.4	Multiple parallelization	13
3.5	Load balancing	13
3.6	Using <code>snowfall</code> with <code>sfCluster</code>	14
A	Overview: Main <code>snowfall</code> functions	15
A.1	Initialization and shutdown of the cluster	15
A.2	Parallel calculation functions (excerpt)	15
A.3	Cluster tools (excerpt)	15

1 Introduction

1.1 How to parallelize problems?

Ordinary computer programs run sequential, that means a program command is executed if the former command is finished. A computer program can be seen as a single worker who performs tasks step by step. On turning to parallel computing, you have a whole team of workers, whom can be feed with either the same task on another area (“data”) or you give them different tasks (“code”).

Imagine the building of a wall. A single worker place the bricks one after another. A team of workers can place several bricks at the same time. But there is a limit: the wall will not be done faster if adding a huge number of workers, there is a limit – in this case the amount of bricks per wall row (which is called “scalability” of the problem, which basically describes the benefit of adding additional processors).

For our goal, the parallelization of R programs, there is bad and good news: the bad one is R features no build-in support for any parallelization technique yet. The good news is that there are packages which enable parallel computing and, even better, R helps to develop certain parallel programs by relying deeply on data parallelism on the language level.

In this tutorial, we stick to `snowfall`, which is basically an extension above the package `snow` (by Luke Tierney, Toni Rossini et al.). It adds some common functions and many extensions for usability. Please note: the basic idea to enable parallel computing is mainly the same for all R packages which enable parallel computing. They all rely on the “data parallel” functions like `lapply` or `sapply`.

2 Getting started

2.1 Installation

In this tutorial we will be faced with several laptops with different operating systems and installations. Therefore we will only describe the basic usage of parallel computing using the package `snowfall`.

Beside an R version you need to install `snowfall` and `snow` first (as `snowfall` is build upon `snow`, both are needed). Just type on the R shell:

```
install.packages(c("snow", "snowfall"))
```

If you do not have internet access, ask us about the tarballs for the packages and install them directly:

```
install.packages("snow.tar.gz", rep=NULL)  
install.packages("snowfall.tar.gz", rep=NULL)
```

For the tutorial, it is sufficient to work with a so called socket cluster, which communicates via TCP/IP sockets. It works without installation of any other software besides the R package and is also working on Microsoft Windows.

If you like, you can install additional cluster software on your laptop for the exercises (like an MPI implementation (e.g. OpenMPI) or PVM), please use your local package manager for this (if running Linux). If running Windows, we strongly recommend sticking to the socket cluster for the tutorial, as you probably need some time to set up the required software.

2.2 Writing parallel programs with snowfall

This section is a short guide for the first steps using `snowfall`, including the typical parts when writing parallel programs.

2.2.1 Initialization

In a very short description, cluster computing with `snowfall` works this way: a “master” R instance is spawning other R processes, probably across a cluster of machines (means: not only on the local machine). These are called “workers” then (or politically incorrect “slaves”). The master is basically the R instance you started yourself. This master is not participating in parallel calculations (these are exclusively done on the workers), but does all the sequential parts (like setup) and transfers data to and from the workers. Keep this in mind, as in some constellations, this is very important (see Exercise 6).

First, the cluster has to be initialized by specifying the type and desired number of CPUs using function `sfInit`. Arguments can also be passed from commandline. `snowfall` really starts to shine if used with `sfCluster`, a small cluster management tool. But it can also be used without. All types of cluster need some installation first, see Section 2.1, except the socket cluster. So the simplest possibility to initialize a cluster is e.g.

```
sfInit(parallel=TRUE, cpus=4, type="SOCK")
```

This initialization is straight forward: request a socket cluster (`type="SOCK"`) with 4 cpus and enable parallel computing (`parallel=TRUE`).

As this kind of initialization would require yourself to change your program if e.g. you want more CPUs, you can also use its plain form:

```
sfInit()
```

In this case, `snowfall` searches on the commandline for arguments to setup the cluster. You can inject these parameters on the command line by adding arguments to the R call:

```
R --no-save ---no-restore --args --parallel --cpus=4 --type=SOCK
```

After R start, you see the commandline arguments are used:

```

> library(snowfall)
Loading required package: snow
> sfInit()
snowfall 1.70 initialized: parallel execution on 4 CPUs.

```

The following table shows the arguments:

Argument	Description	Notes
<code>--parallel</code>	Enable cluster usage	
<code>--cpus=X</code>	Amount of CPUs in parallel mode	
<code>--type=X</code>	Type of cluster	MPI, PVM, NWS and SOCK allowed
<code>--restoreSR</code>	Enable restoring on <code>sfClusterApplySR</code> calls	see Section 3.2
<code>--session=X</code>	Session-ID	Only set by <code>sfCluster</code>
<code>--hosts=X</code>	Hostlist for SOCK and NWS clusters	If you want to use many computers with socket and NetworkSpaces clusters, you need to specify the hosts. For example: <code>--hosts=localhost:3,other:7</code> spawns 3 workers on <code>localhost</code> and 7 on <code>other</code> . Note this overwrites <code>--cpus</code> . If you do not specify hosts for these cluster types, only <code>localhost</code> ² is used.

Probably you wondered about the possibility to disable parallel computing within the package `snowfall`. Didn't you installed it to do some parallel computing? Yes, sure. But probably you are writing a package and give it to others. Probably the users do not want to enable parallel computing all the time or even having other parallel solutions implemented. In this case, `snowfall` behaves like a normal sequential program without any code changes (except the `parallel=FALSE` switch on initialization). The so called "sequential mode" is also sometimes handy for development and debugging purposes.

2.2.2 Data exchange of objects and loading of packages

With the former `sfInit(...)` call, you have spawned a certain number of worker processes. Objects on the master process, which is the process, you are working on interactively at the moment, are not known on the worker processes. This is also true for libraries (loading a library on the master does not load them on the workers!).

This can be illustrated via `sfClusterEval`, which evaluates expressions on each worker process and returns a list with one entry per worker process:

```
a <- 3
sfClusterEval(ls())
```

Any object that is required for parallel computation on the worker processes, has to be exported, this means – transferred – to the workers. This is done via function `sfExport`:

```
sfExport("a")
sfClusterEval(ls())
```

Now, `a` can be used on the worker processes.

Alternatively you can export objects implicitly in a more functional style:

```
# Returns 4,5,6
sfSapply(1:3, sum, a)
```

Explicit export via `sfExport` or `sfExportAll` should be used for objects which are needed in one or more parallel calls on each worker (for example your simulation is working on a large dataset, which is processed many times). Also if you need to export many objects, it is quite handy to export them with these functions instead of adding dozens of arguments to your function (this is for example happening on the rewriting of existing programs to parallel programs). For all other cases, you want to prefer the functional style.

Loading libraries or sources on all workers can be done through `sfLibrary` and `sfSource` which have the same syntax than the R build-in counterparts.

Beware of the following: If you transfer any object to the workers and change it afterwards on the master, the values differ and you need to reexport these objects.

2.2.3 Wrapper functions

How does parallelization work in your program? It is mainly done via functions of the “apply” family. For example, `lapply` is working as a data parallel function: For any list element a given function is called and the result is collected as list again. That means: Data changes, the code does not change. As these calls are – without being really evil – secure your program from sideeffects³, the plain idea is to execute the given function on different worker processes with each independent data. That is the basic idea of parallel computing with R. It is illustrated in Figure 1.

Following our previous example, we have just exported object `a`, so for example `sfClusterEval(2*a)` works now, but is useless. Different computations on each worker process and more than just one at each are possible using `sfLapply`, which is a parallelized version of `lapply`:

```
sfLapply(seq(0, 100, 10), function(x) x*a)
```

³Like changing the calling list in the called function.

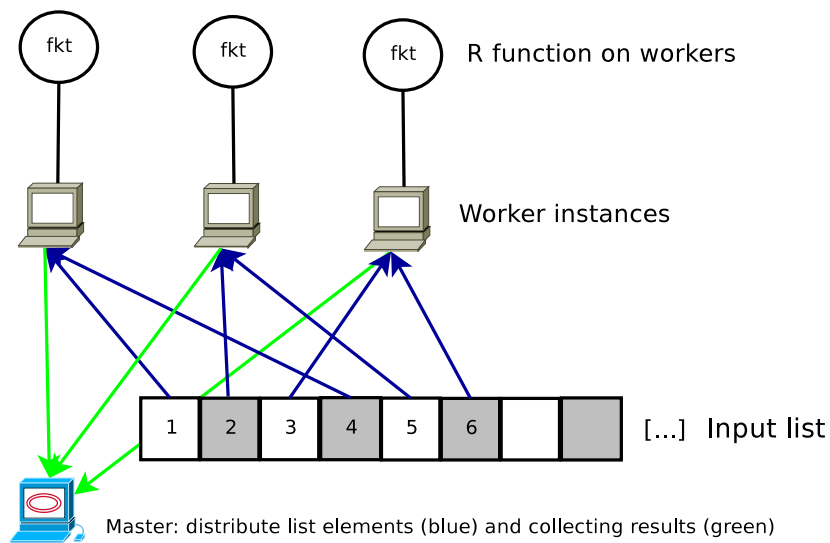


Figure 1: Parallel distribution of work using apply calls.

The example shows, that the code, we want to run on the worker processes, has to be wrapped into a function, the wrapper function. Usually, it would be assigned separately, as possibly longer than in the example above, e.g.

```

wrapper <- function(x){
  x*a
}
values <- seq(0, 100, 10)
sfLapply(values, wrapper)

```

Assume, your cluster has four worker processes. `sfLapply` calls function `wrapper` on the worker processes with the first four elements of the passed vector `values` each. After finishing calculation on all worker processes, results are returned, and the next four elements are passed and so on. Especially, if computation time varies or machines with different speeds are used, a lot of time is lost with this behavior. To use load balancing is more appropriate then, which means, that immediately after one worker process has finished its job, the next is passed. Function `sfClusterApplyLB` can be used for that, which takes the same arguments as `sfLapply`.

All parallel functions work in this way, that means they are supporting parallel data, but not parallel code. These is often the case when you have independent functions with same or different data and execute them concurrently (the same is true e.g. for running different simulation settings at the same time). If you want to execute code parallel programs, you need some tricks. If you want to do it on your own, see Exercise 4 or just take a look in the solutions. Btw. future `snowfall` function will have better support for code parallel problems.

2.2.4 Stop cluster

The cluster is stopped with `sfStop()`. Also, another `sfInit` call would cause an `sfStop` call automatically.

2.3 Typical examples

The following example illustrates, how an out-of-bag error estimate based on bootstrap samples can be obtained using parallelization. For the sake of clarity, all examples are simplified as much as possible, as we focus only on typical structure of R code and not on statistical details. We think, that the examples can easily be adapted to lots of statistical applications.

```
library(snowfall)

sfInit()
sfLibrary(randomForest)

data(iris)

boot.n <- 10
boot.index <- t(sapply(1:boot.n,function(b){sample(1:nrow(iris),replace=TRUE)}))
not.in.sample <- list()
for (i in 1:boot.n){
  not.in.sample[[i]] <- (1:nrow(iris))[-unique(boot.index[i,])]
}

boot.fun <- function(actual.sample){
  sample.fit <- randomForest(Species ~ ., data=iris[boot.index[actual.sample, ],])
  prediction <- predict(sample.fit, newdata=iris[not.in.sample[[actual.sample]],1:4])
  sum(prediction != iris[not.in.sample[[actual.sample]],5])/length(prediction)
}

sfExport("iris", "boot.index", "not.in.sample")

res <- sfClusterApplyLB(1:boot.n, boot.fun)

error <- mean(unlist(res))
```

Reproducibility of the results is not addressed in this example. See Section 3.1 for this. Note that the possibly large trainings and test data samples are not passed to the worker processes each time, but the whole data set is exported only once. Using the generated indices of trainings and test data, they are determined on the worker processes.

Alternatively, determination of the data can also be switched to the worker processes completely. Consider a simulation study, where similar or identical data simulation is repeated several times. The wrapper function could look like that:

```
sim.fun <- function(p, n){
  x <- matrix(rnorm(n*p), n, p)
```

```

  y <- as.factor(ifelse(x[,1] > 0.5, 0, 1))
  randomForest(y~., data=data.frame(x, y))$importance
}

```

and can be called for example via

```

sfLibrary(randomForest)
sfClusterApplyLB(c(5, 10, 20), sim.fun, n=100)

```

Parameter p varies here, while n is fixed for all simulations.

Typically, you might return more than the result of one calculation. In the example from above e.g.:

```

sim.fun <- function(p, n){
  x <- matrix(rnorm(n*p), n, p)
  y <- as.factor(ifelse(x[,1] > 0.5, 0, 1))
  varimp <- randomForest(y~., data=data.frame(x, y))$importance
  list(p=p, n=n, x=x, y=y, varimp=varimp)
}

```

Now, a list is returned by

```

res.list <- sfClusterApplyLB(c(5, 10, 20), sim.fun, n=100)

```

Results can be rearranged using something like

```

all.y <- lapply(res.list, function(arg) arg$y)

```

or

```

all.p <- unlist(lapply(res.list, function(arg) arg$p))

```

In most of the examples, only a vector is passed at the `sfClusterApplyLB` call. This is not mandatory, though reasonable as avoiding intense data traffic between the master and the worker processes. However, the following example illustrates, how more than one varying variable can be passed to the worker processes.

```

wrapper.list <- function(data){
  randomForest(data$y~., data=data.frame(data$x, data$y))$importance
}
x1 <- matrix(rnorm(n*p), n, p)
y1 <- as.factor(ifelse(x1[,1] > 0.5, 0, 1))
x2 <- matrix(rnorm(n*p), n, p)
y2 <- as.factor(ifelse(x2[,1] > 0.5, 0, 1))
datalist <- list(list(x=x1,y=y1), list(x=x2, y=y2))
sfClusterApplyLB(datalist, wrapper.list)

```

3 snowfall advanced

3.1 Parallel random numbers

Many statistical methods such as cross-validation or the bootstrap, which are also incorporated in several applications, as well as data simulations involve the use of random numbers. Reproducibility of results is essential, not only for debugging, i.e. for oneself, but also to reveal the research for others. R provides several (pseudo-)random number generators. Parallel execution deserves special attention.

Worker processes are own R instances, so by default, pseudo-random numbers are initialized via the system clock on each worker process. As they are spawned at the same time, some processes might have the same random number streams by coincidence, some not. If one aims for identical random number streams for each process, function `set.seed` with the same seed should be called on each process. Though, usually independent pseudo-random number streams are desirable, which are provided by function `sfClusterSetupRNGstream`.

This approach works perfectly, but suffers from some limitations: If load-balancing is used, which means that it is possible that one job is computed on a different worker process in another run, or if the number of worker processes differs in different runs, for example if more CPUs are available later on, or even if sequential execution is required, e.g. for debugging, reproducibility is not warranted any more, even with the above mentioned methods. One solution is to avoid totally any randomness in computation on the worker processes. For example, if random samples of the data are required, they can be determined on the master process and then be passed to or accessed from the worker processes in the particular call. Possibly, this approach requires the storage or passing of large data sets. Furthermore, many statistical methods involve random procedures, making that strategy impracticable.

A simple, but efficient approach is to use a fixed random seed for each run on the worker processes. This reduces the ‘one stream per process’ paradigm applied when using `sfClusterSetupRNGstream` to ‘one stream per run’. For example, use the following structure for the wrapper function:

```
prn <- function(seed){
  set.seed(seed)
  # computation, e.g.
  rnorm(2)
}
```

A vector `seeds` containing one seed for each run, e.g. for ten runs `1:10` or `rnorm(10)` is passed when calling:

```
sfLapply(seeds, prn)
```

To allow reproducibility, no more information than only this vector has to be stored. Although there is no theoretical support for this approach and random number streams are not guaranteed to be independent, we think that the advantages outweigh the disadvantages, as it proved to be very convenient in practice due to its simplicity and flexibility. For example, computation can also be carried out in parts or be extended, i.e. calling `sfLapply` with vector

seeds equal to 1:8 and 9:10 later on leads to the same results as calling with 1:10 once, likewise, if the cluster was interrupted in between. It is even load balancing compatible, as this might change the order of the jobs on the worker processes, but results are returned in the same way as without load balancing.

Note that a `set.seed` call on the master is required anyway, whatever method is used to allow reproducibility on the slaves.

3.2 Intermediate result saving

Computing on a cluster usually implies that computations are quite long. Additionally, the use of several machines makes interruption of computation due to the failure or maintenance of only one machine more likely. If one wants to avoid the loss of complete calculation, the intermediate result saving via function `sfClusterApplySR` is advisable.

Consider any wrapper function `wrapper.fun`, which is called with e.g. 1:100. Thus, the call simply is

```
res.sr <- sfClusterApplySR(1:100, wrapper.fun)
```

Assume you have 10 CPUs available for computation. Now, after any 10 finished runs, intermediate results are saved. If e.g. computation is interrupted while results 51 to 60 are computed, results 1 to 50 can be restored and computation is continued via

```
res.sr <- sfClusterApplySR(1:100, wrapper.fun, restore=TRUE)
```

Even if this requires a small change in the code, it prevents the user from restoring results unknowingly or unwanted. Using `sfCluster`, it is sufficient to change `sfCluster` option `--restore`.

`sfClusterApplySR` can be called several times for different calculations, if argument `name` is specified. In our example, we can e.g. use

```
res.sr2 <- sfClusterApplySR(1:87, wrapper.fun2, name="anothercalculation")
```

in the same or another program without overwriting results from the first call. Restoring works equivalent, specifying argument `name`.

Apart from that elaborate approach, note that most `snowfall` calculation functions return lists, so calculations can simply be split and results be put together afterwards. For example, if one does not want to use `sfClusterApplySR` as it does not feature load balancing, which is possibly faster when having lots of short computations of potentially different length,

```
res1 <- sfClusterApplyLB(1:500, wrapper.fun)
save(res1, file="part1.Rdata")
res2 <- sfClusterApplyLB(501:1000, wrapper.fun)
save(res2, file="part2.Rdata")
res.LB <- c(res1, res2)
```

results in exactly the same as having called

```
res.LB <- sfClusterApplyLB(1:1000, wrapper.fun)
```

but is more save.

3.3 Using snowfall within functions/packages

Package `snowfall` might be used within a function or even in packages to allow parallelization without too much effort for the user. For example, it is incorporated in package `peperr` already. This Section addresses some features of `snowfall` that might be useful concerning this issue.

Using `snowfall` requires an `sfInit` call first. Relieving this from the user, involves the call within the function, that shall contain parallel executable parts. At the same time, all arguments of `sfInit` should be able to change. So, a function `myfun` could look like that:

```
myfun <- function{args.init=list()}{  
  require(snowfall)  
  do.call("sfInit", args.init)  
  # some (parallel) calculation  
}
```

This structure causes that `sfInit` is called with its default arguments, except the ones given as list in argument `args.init`. The advantage is that not every single argument of `sfInit` has to be specified explicitly. Additionally, it is also independent from possible changes in the arguments. The “...” argument provides the same functionality, but can be used only once and might be needed for another function called within `myfun`.

Argument `nostart` of `sfInit` is especially introduced for using within functions. Setting it to `TRUE`, skips cluster setup. It is useful, if variables or libraries were already exported to the cluster (`sfExport` or `sfLibrary`) before calling `myfun`, or if no cluster set up is wanted as using in already parallelized environment. However, our initial aim was to relieve the user from such tasks. One could simply call `sfExportAll()` within function `myfun` to make all (local) objects on the master available on the slaves, though not advisable if data is large and possibly unnecessary on each node. Analog to that, all libraries that are loaded on the master can be loaded on the nodes via

```
for (i in 1:length(.packages())){  
  eval(call("sfLibrary", (.packages()[i]), character.only=TRUE))  
}
```

At the end of our function, `sfStop()` might be called. Here, argument `nostop` was introduced to have the possibility to skip shutdown of cluster. Then, the worker processes remain accessible for further calls.

Again, using `snowfall` functions forces the user only to install the library, not to work on a compute cluster. All functions run in sequential execution without any modification.

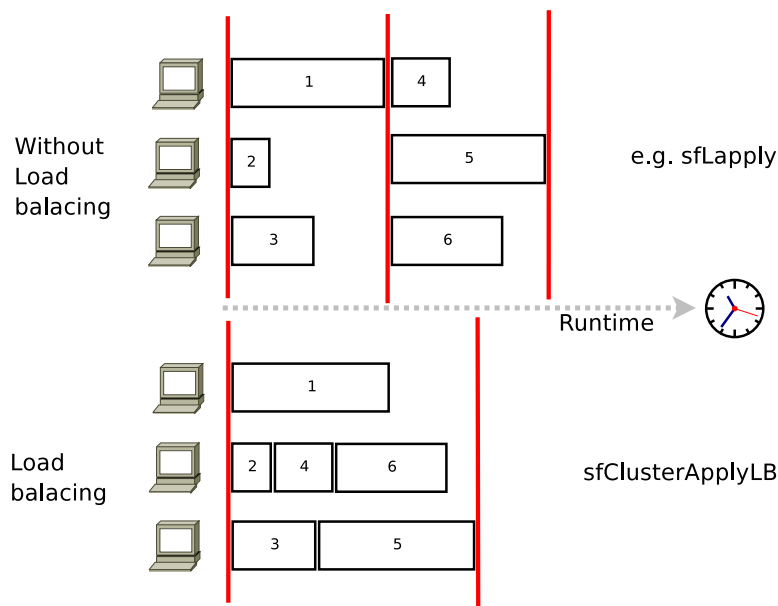


Figure 2: Distribution of work in load balanced calls compared with “ordinary” list calls.

3.4 Multiple parallelization

Consecutive parallelization, i.e. several `sfLapply/sfClusterApplyLB` etc. calls in a row are possible without any restriction. `sfRemove/sfRemoveAll` calls might be reasonable as long as the same worker processes are used, i.e. if cluster is not shut down and set up again using `sfStop/sfInit`. Using `sfCluster` (and MPI), it is even possible to set up several clusters, i.e. to have more than one cluster independently running at the same time on the same machines.

To use nested parallelization, i.e. to parallelize within already parallelized code, is not possible though. Apart from that it is not possible to set up a subcluster within a cluster, it is not reasonable, as no further speed-up would be obtained, assuming that the available resources are optimally used already. If a parallelizable function (see Section 3.3) is used within parallelizable code, `sfInit` argument `nostart` has to be set to `TRUE`, to force sequential execution as the cluster is already set up.

3.5 Load balancing

The distribution of the list elements in parallel calls like `sfLapply` are working in a simple manner: The first node will get index 1, the second index 2. This will continue until the number of used CPUs is reached. Then, the master waits for all workers to return their result and afterwards they will be fed with the next elements. While this works very good in most cases, it causes trouble if the runtime is varying on the individual list elements or you are using computers with different power – some workers finish work early and then sleep until the last worker finishes working.

`snow` and therefore `snowfall` feature a so called “load balanced” call, where the workers

immediately receive new work when they have done their work. Figure ?? is illustrating the different approaches to distribute work.

After realizing this principle, the question arises why “normal” distribution like in `sfLapply` is needed anymore, as load balancing seems to be a superior method. Well, there are reasons not to use load balancing: As the elements are shuffled in a unreproducible manner (consider a cluster with different computers), e.g. the described random number generators (Section 3.1) will not work anymore. You have to take care of random numbers yourself in a load balanced run. Also, in some parallel programs, on using few processors and very short runtime on the workers, the overall runtime may collapse, as the master is overstrained by dispatching the data – in this special cases not using load balancing will result in better runtime. Exercise 6 shows such an example.

3.6 Using snowfall with sfCluster

As `sfCluster` is only needed if several people work on the same machines and requires some Unix knowledge to install, we will not give further information in this tutorial. Please refer to our R-News article (found in the tutorial materials) or ask Jochen to show or help you with it.

A Overview: Main snowfall functions

This section provides an overview of the most important `snowfall` functions. For more details, see the help pages in the package.

A.1 Initialization and shutdown of the cluster

Function	Description
<code>sfInit</code>	Initializes the cluster. Is called automatically with default arguments, when calling any of the functions in A.2 or A.3.
<code>sfStop</code>	Shuts down the cluster. Is called automatically, if <code>sfInit</code> is called without calling <code>sfStop</code> before.

A.2 Parallel calculation functions (excerpt)

Function	Description
<code>sfLapply</code>	Parallel version of function <code>lapply</code> .
<code>sfSapply</code>	Parallel version of function <code>sapply</code> .
<code>sfClusterApplyLB</code>	Load balanced version of function <code>sfLapply</code> .
<code>sfClusterApplySR</code>	As <code>sfLapply</code> , but intermediate results can be restored if necessary. See Section 3.2.
<code>sfMM</code>	Parallel matrix multiplication.

A.3 Cluster tools (excerpt)

Function	Description
<code>sfLibrary</code>	Loads R package on master and all worker processes. Parallel version of function <code>library</code> .
<code>sfSource</code>	Sources code on master and all worker processes. Parallel version of function <code>source</code> .
<code>sfExport</code>	Exports given objects to worker processes.
<code>sfExportAll</code>	Exports all objects to worker processes.
<code>sfRemove</code>	Removes given objects from worker processes.
<code>sfRemoveAll</code>	Removes all objects from worker processes.
<code>sfClusterCall</code>	Calls a function on each worker process.
<code>sfClusterEval</code>	Evaluates a literal expression on each worker process.
<code>sfClusterSetupRNGstream</code>	Initializes a random number generator.

Detailed information about `snowfall` and `sfCluster` including download can also be found on our webpage:

www.imbi.uni-freiburg.de/parallel

Hints and solutions for the exercises

Please note: For any of the programs `snowfall` (and therewith the usage of parallel computing) has to be initialized first.

The following header enables parallel computing on the local machine with 2 CPUs and a socket cluster, which runs “out of the box” on most operating systems. The exercise programs have such a short runtime that you can start the programs even on a laptop with a single CPU.

```
library(snowfall)

sfInit(parallel=TRUE, cpus=2, type="SOCK")
```

After the program you should tell `snowfall` to stop the cluster:

```
sfStop()
```

1. Initialization

See Section 2.2.1.

2. Parallelization principles: loop and code refactoring

```
matMultiply <- function(m, const){
  m %*% const
}

## Part 1
a1 <- matrix(runif(9), nrow=3)
b1 <- matrix(runif(9), nrow=3)
c1 <- matrix(runif(9), nrow=3)
d1 <- matrix(runif(9), nrow=3)

constMat <- matrix(runif(9), nrow=3)

matList <- list(a1, b1, c1, d1)

l <- sfLapply(matList, matMultiply, constMat)

a2 <- l[[1]]
b2 <- l[[2]]
c2 <- l[[3]]
d2 <- l[[4]]

## Part 2/3:
list1 <- list(a1, b1, c1, d1)
list2 <- list(a2, b2, c2, d2)
```

```

mainList <- list(list1, list2)

## The wrapper now only features the matrix to multiply.
## constMat is a global variable through sfExport()
matMul <- function(x){
  x %*% constMat
}

## Exporting is much better here, as accessed in any
## loop step.
sfExport("constMat")

for(ml in 1:length(mainList)){
  mainList[[ml]] <- sfLapply(mainList[[ml]], matMul)
}

print(mainList)

```

3. Parallelization principles: scaling and depth

As on any step the amount of additions is halved, there is a solution in $\log_2(n)$ steps, where n is the size of the data (amount of numbers to be added, in our example 100). This is called the “depth” of the problem. Also, there are $n - 1$ additions to be done, which is called “work number”. In the given case, the depth is reduced through parallelization (in sequential it is 99, as `sum(1:100)` requires 99 additions), but the work number being the same for both approaches.

Of course you do not want to start adding numbers in parallel, but this type of solution indeed can be used for several real-life problems.

Two possible approaches:

- (a) The idea is to sum up the numbers pairwise and repeat this until only one number is left - the sum of all numbers. Requires explicit export, a wrapper function and some unneeded data is transferred to the workers.

```

sumup <- function(index, separator){
  numbers[index]+numbers[separator+index]
}

numbers <- 1:100

while(length(numbers)>1){
  ## Changed this vector: resubmit to nodes.
  sfExport("numbers")

  ## if odd, preserve last element
  addOdd <- NULL
  if(length(numbers) %% 2) addOdd <- numbers[length(numbers)]
}

```

```

    idx <- floor(length(numbers)/2)

    ## if is.null(add) nothing is added to vector
    numbers <- c(sfSapply(1:idx, sumup, separator=idx), addOdd)
  }

stopifnot(sum(1:100) == numbers)

numbers

```

- (b) Going the direct way: Transfer bundles of two numbers on the workers and use function `sum` directly. Therefore, there is no need of a wrapper function nor explicit export (as only data really required on the worker are transferred).

```

numbers <- 1:100

while(length(numbers) > 1){
  distribute <- list()

  ## Build packages of two numbers and add these to list.
  ## (Last element if odd length: NA).
  for(i in seq(1, length(numbers), 2))
    distribute[[i/2]] = numbers[c(i,i+1)]

  ## Call sum for each two-number vector.
  ## On odd cases we are doing one addition too much
  ## (last element + NA) to preserve the last element.
  numbers <- sfSapply(distribute, sum, na.rm=TRUE)
}

stopifnot(sum(1:100) == numbers)

numbers

```

4. Parallelization principles: code parallelization

The basic problem is of course that all the R parallel packages are *data-parallel* and now we have a *code-parallel* problem.

- (a) The first solution is working with a hard coded wrapper function, which is handy for dispatching only few function calls:

```

numbers <- 1:100

## "Dispatch" between different functions.
## Note you can add any function calls with additional
## arguments as well.
wrapper <- function(type, data){
  switch(type,

```

```

        sum = sum(data),
        prod = prod(data),
        mean = mean(data),
        min = min(data),
        max = max(data))
    }

res <- unlist(sfClusterApplyLB(c('sum', 'mean', 'min', 'max'),
                             wrapper, 1:100))

res

stopifnot(res == c(sum(1:100), mean(1:100), min(1:100), max(1:100)))

```

- (b) The second solution can handle variable functions and argument lists. It is more flexible, although a bit more effort to create (and in this form far from being perfect):

```

numbers <- 1:100

wrapper <- function(fkt, ...){
  options <- list()

  for(i in 2:length(fkt))
    options[[i-1]] <- fkt[[i]]

  do.call(fkt[[1]], options, ...)
}

## For each parallel call, a list with a function name and
## any number of arguments is provided.
callList <- list(list("sum", numbers[-(1:20)], numbers[1:20]),
                list("min", numbers),
                list("max", numbers),
                list("mean", numbers))

sfLapply(callList, wrapper)

res

stopifnot(res == c(sum(1:100), mean(1:100), min(1:100), max(1:100)))

```

5. Parallel random numbers

For example:

- Same seed on each worker process:

```

sfClusterEval(set.seed(123))
sfClusterEval(rnorm(2))

```

- Independent random number streams:

```
sfClusterSetupRNGstream(123)
sfClusterEval(rnorm(2))
```

- Fixed seed for each run: see example in Section 3.1

You can test the behavior on load balanced settings on your own: if using two or more processors, the results are different in most runs:

```
library(snowfall)

sfInit(parallel=TRUE, cpus=2, type="SOCK")

## Get some numbers to compare for load balancing.
i <- sfClusterSetupRNGstream(123)
rList2cpus <- unlist(sfClusterApplyLB(1:10,function(x)return(sum(runif(100))))))

sfStop()

## Restart cluster with a different amount of CPUs.
sfInit(parallel=TRUE, cpus=3, type="SOCK")

i <- sfClusterSetupRNGstream(123)
rList3cpus <- unlist(sfClusterApplyLB(1:10,function(x)return(sum(runif(100))))))

print("Ordinary: different sequences. If both are equal, it is luck. Rerun.")
print(rList2cpus)
print(rList3cpus)

sfStop()
```

6. Real life situations: load balancing, network traffic

This program is a bit tricky. Even after thinking twice, the inner loop with the iteration cannot be taken into parallel and must be executed sequentially (as the result of $z^2 + c$ is directly used as parameter again).

So the decision is: Take the outer loop with the lines or the middle loop with the single points. Both work. If you tried the point version first, probably the runtime was shocking. Due to the nature of the iteration in the inner loop, many calls return almost immediately. Organizing the parallel calls on the master takes far more time than the calculation itself. If you have several cores in your machine: you see spawned workers are sleeping most of the time. Only the master process, which organizes the dispatching of the calls to the workers, is fully loaded.

The best choice is to let workers each work on complete rows or columns of the resulting matrix. There still are several hundred steps (means: rows or columns in this case, depending on `xPoints` or `yPoints`) for the workers to do.

Load balancing really shines on this problem: as the runtime of the inner loops vary a lot, some areas of the picture are calculated very fast, others need a lot of time. Load balancing keeps all the CPUs busy, where normal parallel calls wait on certain points for all the workers to be finished.

Runtimes for different methods (R 2.9.0, 8 core machine, socket cluster). Bold numbers are used where the parallel calculation significantly failed. The sequential run is done without any parallelization, whereas the 1 core parallel run means it was running in parallel mode with a single worker:

Cores	Point		Line	
	sfSapply	sfClusterApplyLB	sfSapply	sfClusterApplyLB
1 (sequential)	14.65	14.67	14.65	14.67
1 (parallel)	105.00	94.76	15.52	14.74
2	28.17	68.57	8.60	7.27
3	7.60	59.85	7.31	4.90
4	5.86	57.69	6.33	3.69
7	4.65	69.61	3.95	2.22

The following program both implement point and line-based parallelization, just uncomment the calls to try both types.

```
## Wrapper for parallelization on point level.
calcPoint <- function(y, x=0, maxIterations=0){
  c <- complex(real=x, imaginary=y)
  z <- complex(real=0, imaginary=0)
  iteration <- 0

  while((Mod(z) <= 2) && (iteration < maxIterations)){
    z <- z*z + c

    iteration <- iteration + 1
  }

  if(iteration == maxIterations)
    iteration <- 0

  return(iteration)
}

# Wrapper for parallelization on line level.
calcLine <- function(x=0, maxIterations=0){
```

```

line <- rep(length(ySequence), 0)

for(y in 1:length(ySequence)){
  ## You can use the function "calcPoint" instead of copying
  ## the point calculation again.
  ## Note: in this case, you need to call sfExport("calcPoint")
  ## before the start of the calculation.
  c <- complex(real=x, imaginary=ySequence[y])
  z <- complex(real=0, imaginary=0)
  iteration <- 0

  while((Mod(z) <= 2) && (iteration < maxIterations)){
    z <- z*z + c

    iteration <- iteration + 1
  }

  if(iteration == maxIterations)
    iteration <- 0

  line[y] <- iteration
}

return(line)
}

mandelbrotSet <- function(
  xPoints=500, yPoints=400, maxIterations=32,
  viewBox=list(xmin=-2.0, ymin=-1.0, xmax=1.0, ymax=1.0)){

  xStepSize <- (viewBox$xmax - viewBox$xmin) / xPoints
  yStepSize <- (viewBox$ymax - viewBox$ymin) / yPoints
  xSequence <- seq(viewBox$xmin, viewBox$xmax, xStepSize)
  ySequence <- seq(viewBox$ymin, viewBox$ymax, yStepSize)

  ## Point level parallelization (load balanced)
  # mandel <- matrix(ncol=length(xSequence), nrow=length(ySequence))
  # for(x in 1:length(xSequence))
  # mandel[,x] = unlist(sfClusterApplyLB(ySequence, calcPoint, xSequence[x], maxIterations))

  ## Line level parallelization (load balanced)
  sfExport("ySequence", local=TRUE) ## Needed many times!
  mandel <- matrix(unlist(sfClusterApplyLB(xSequence, calcLine, maxIterations=maxIterations)
    ncol=length(xSequence))

  cols <- gray(c(0,seq(0.2,1,by=0.8/(maxIterations-1))))

  return(list(x = xSequence, y = ySequence, z = t(mandel),
    colors = cols))
}

```

```
}  
  
system.time(mandelSet <- mandelbrotSet())  
  
image(mandelSet,  
      col=mandelSet$colors,  
      xlab="x", ylab="y")
```

7. Intermediate result saving

See Section 3.2.

KTHXBYE