Analysis of data from aCGH experiments using parallel computing and ff objects

Ramon Diaz-Uriarte[1], Daniel Rico[2], and Oscar M. Rueda[3]

October 15, 2013

1. Department of Biochemistry, Universidad Autonoma de Madrid Instituto de Investigaciones Biomedicas "Alberto Sols" (UAM-CSIC), Madrid (SPAIN). 2. Structural Computational Biology Group. Spanish National Cancer Center (CNIO), Madrid (SPAIN). 3. Cancer Research UK Cambridge Research Institute Cambridge, UK
rdiaz02@gmail.com, drico@cnio.es, Oscar.Rueda@cancer.org.uk

Contents

1 This vignette

This vignette presents the ADaCGH2 package using:

- Three fully commented examples that deal with the usage of the different parallelization options and types of objects (in particular, *ff* objects) available.
- Examples of using ADaCGH2 with CGHregions, Limma, and snapCGH.
- We show some benchmarking results with large data sets.

All of the runnable examples in this vignette use a small toy example (they need to run in a reasonably short of time in a variety of machines). In the vignette called "ADaCGH2-long-examples" we list example calls of all segmentation methods, with different options for methods, as well as different options for type of input object and clustering. That other vignette is provided as both extended help and as a simple way of checking that all the functions can be run and yield identical results regardless of type of input and clustering.

2 Overview:

ADaCGH2 is a package for the analysis of CGH data. The main features of ADaCGH2 are:

- Parallelization of (several of) the main segmentation/calling algorithms currently available, to allow efficient usage of computing clusters. Parallelization can use either *forking* (in Unix-like OSs) or sockets, MPI, etc, as provided by package *snow* (`http://cran.r-project.org/web/packages/snow/index.html`).

  Forking will probably be the fastest approach in multicore machines, whereas MPI or sockets will be used with clusters made of several independent machines with few CPUs/cores each.

- Optional storage of, and access to, data using the *ff* package (`http://cran.r-project.org/web/packages/ff/index.html`), making it possible to analyze data from very large projects and/or use machines with limited memory.

- Parallelization and *ff* can be used simultaneously. WaviCGH Carro et al. (2010) (`http://wavi.bioinfo.cnio.es`), a web-server application for the analysis and visualization of array-CGH data that uses ADaCGH2, consitutes a clear demonstration of the usage of *ff* on a computing cluster with shared storage over NFS.

ADaCGH2 is a major re-write of our former package ADaCGH Diaz-Uriarte and Rueda (2007) and version 2 of ADaCGH2 is, itself, a major rewrite of the version 1.x series. Over time, we have improved the parallelization and, specially, changed completely the data handling routines. The first major rewrite of ADaCGH2 included the usage of the *ff* package, which allows ADaCGH2 to analyze data sets of more than four million probes in machines with no more than 2 GB of RAM. The second major rewrite reimplemented all the reading routines, and much of the analysis, which now allow a wider range of options with increased speed and decreased memory usage, and also allows users to disable the usage of *ff*. Moreover, in the new version, a large part of the reading is parallelized and makes use of temporary *ff* objects and we allow parallelization of analysis (and data reading) using forking.

2.1 Terminology

The following is the meaning of some terms we will use repeatedly.

***ff* object** An object that uses the *ff* package. A tiny part of that object lives in memory, in the R session, but most of the object is stored on the hard drive. The part that lives in memory is just a pointer to the object that resides in the hard drive.

RAM objects The "usual" R objects (in our case, mainly data frames and matrices); these are stored, or live, in memory.

Somewhat similar to what the documentation of the `ff` package does, we refer to these objects, that reside in memory, as RAM objects. Technically, a given data frame, for instance, need not be in RAM in a particular moment (that actual memory page might have been swapped to disk). Regardless, the object is accessed as any other object which resides in memory. Likewise, note that *ff* also have a small part that is in memory, but the data themselves are stored on disk.

forking We copy literally from the vignette of the *parallel* package R Core Team (2013): "Fork is a concept from POSIX operating systems, and should be available on all R platforms except Windows. This creates a new R process by taking a complete copy of the master process, including the workspace and state of the random-number stream. However, the copy will (in any reasonable OS) share memory pages with the master until modified so forking is very fast."

Forking is, thus, a reasonable way of parallelizing jobs in multicore computers. Note, however, that this will not work **across** machines (for instance, across workstations in clusters of workstations).

cluster We use it here to contrast it with *forking*. With *cluster*, tasks are sent to other R processes using, for instance, MPI or any of the other methods provided by package **snow** (e.g., PVM, sockets, or NWS).

For example, MPI (for "Message Passing Interface") is a standardized system for parallel computing, probably the most widely used approach for parallelization with distributed memory machines (such as in clusters of workstations). The package **Rmpi** (and **snow**

on top of **Rmpi**) use MPI. In the examples in this vignette, however, we will use clusters of type *socket*, as these are available in several OSs (including Windows), and do not require installation of MPI.

If we are running Linux, Unix, or other POSIX operating systems, in a single computer with multiple cores we can use both forking and clusters (e.g., MPI or sockets). In most cases forking will be preferable as we will avoid some communication overheads and it will also probably use less total memory. If we are running Windows, however, we will need to use a cluster even in a single multicore machine.

2.2 Suggested usage patterns summary

The following table provides a simple guide of suggested usage patterns with small to moderate data sets:

	Lots of RAM	Little RAM
Single node, many cores/node	RAM objects (?), forking *ff* objects (?), forking	*ff* objects, forking
Many nodes, few cores/node	*ff* objects, cluster	*ff* objects, cluster

The question marks denote not-so-obvious choices, where the best decision will depend on the actual details of number of nodes, size of data sets, speed of communication between nodes, etc. For large data sets, the recommended usage involves always using *ff* objects. Using *ff* objects is slightly more cumbersome, but can allow us to analyze very large data sets in moderate hardware (see examples in section 8). Of course, what is "lots", "many", and "large", will depend on the arrays you analyze and the hardware.

The examples below cover all three possible usage patterns:

RAM objects, forking : section 4.

ff objects, cluster : section 5.

ff objects, forking : section 6.

2.3 Usage: main steps and choices

ADaCGH2 includes functions that use as input, or produce as output, either *ff* objects or RAM R objects. Some functions also allow you to choose between using forking and using other mechanisms for parallelization.

For both interactive and non-interactive executions we will often execute the following in sequence:

1. Check the original data and convert to appropriate objects (e.g., to *ff* objects).
2. Initialize the computing cluster if not using forking.
3. Carry out segmentation and calling
4. Plot the results

We cover each in turn in the remaining of this section and discuss alternative routes.

3 The data for all the examples

We will use a small, fictitious data set for all the examples, with six arrays/subjects and five chromosomes.

The data are available as an RData file

```
> library(ADaCGH2)
> data(inputEx)
> summary(inputEx)
```

```
         ID        chromosome       position              L.1
 Hs.101850:  1   Min.   :1.000   Min.   :  1180411   Min.   :-1.07800
 Hs.1019  :  1   1st Qu.:1.000   1st Qu.: 36030889   1st Qu.:-0.22583
 Hs.105460:  1   Median :2.000   Median : 70805790   Median :-0.01600
 Hs.105656:  1   Mean   :2.284   Mean   : 92600349   Mean   :-0.03548
 Hs.105941:  1   3rd Qu.:3.000   3rd Qu.:149843856   3rd Qu.: 0.16000
 Hs.106674:  1   Max.   :5.000   Max.   :243795357   Max.   : 0.88300
 (Other)  :494                                       NA's   :5
      L.2                  m4                 m5                  L3
 Min.   :-0.795000   Min.   :-0.1867   Min.   :-4.67275   Min.   :-13.273
 1st Qu.:-0.139000   1st Qu.: 1.9790   1st Qu.:-0.02025   1st Qu.:  3.631
 Median :-0.006000   Median : 2.2807   Median : 0.43725   Median :  3.925
 Mean   : 0.007684   Mean   : 3.4504   Mean   : 1.60159   Mean   :  1.981
 3rd Qu.: 0.134000   3rd Qu.: 5.8235   3rd Qu.: 3.04475   3rd Qu.:  4.110
 Max.   : 1.076000   Max.   : 6.6043   Max.   : 9.60425   Max.   :  6.374
 NA's   :15                            NA's   :41         NA's   :9
       m6
 Min.   :-0.7655
 1st Qu.:-0.2260
 Median :-0.0440
 Mean   :-0.0351
 3rd Qu.: 0.1620
 Max.   : 0.7750
 NA's   :203
```

```
> head(inputEx)
```

```
                             ID chromosome position   L.1    L.2      m4
1*1180411*Hs.212680   Hs.212680          1  1180411    NA  0.038 6.22625
1*1188041.5*Hs.129780 Hs.129780          1  1188042    NA  0.028 6.17425
1*1194444*Hs.42806     Hs.42806          1  1194444    NA  0.042 6.17425
1*1332537*Hs.76239     Hs.76239          1  1332537    NA  0.285 5.62425
1*2362211*Hs.40500     Hs.40500          1  2362211    NA  0.058 5.85125
1*2372287*Hs.449936   Hs.449936          1  2372287 0.294 -0.006 5.68525
                            m5    L3 m6
1*1180411*Hs.212680   3.22625 6.038 NA
1*1188041.5*Hs.129780 3.17425 6.028 NA
1*1194444*Hs.42806    3.17425 6.042 NA
1*1332537*Hs.76239    2.62425    NA NA
1*2362211*Hs.40500    2.85125    NA NA
1*2372287*Hs.449936   2.68525    NA NA
```

The data are are also available (in the /data subdirectory of the package) as an ASCII text file in two formats: with columns separated by tabs and with columns separated by spaces[1].

4 Example 1: RAM objects and forking

This is the simplest procedure if you are not under Windows. It will work when data is small (relative to available RAM) and the number of cores/processors in the single computing node is large relative to the number of subjects. However, this will not provide any parallelism under Windows: we use forking, as provided by the `mclapply` function in package `parallel`, and forking is available for POSIX operating systems (and Windows is not one of those).

Using forking can be a good idea because, with fork, creating new process is very fast and lightweight, and all the child process share memory pages until they start modifying the objects, and you do not need to explicitly send those pre-existing objects to the child processes. In contrasts, if we use other types of clusters (e.g., sockets or MPI), we need to make sure packages and R objects are explicitly sent to the child or slave processes.

If you have lots of RAM (ideally all you would need is enough memory to hold one copy of your original CGH data plus the return object), you will also probably use RAM objects and not *ff* objects, as these are less cumbersome to deal with than *ff* objects. But see section 8 for other comments.

The steps for the analysis are:

- Read the input data.
- Carry out the segmentation.

4.1 Reading data and storing as a RAM object (a "usual" R object)

We provide here details on reading data from several different sources. Of course, in any specific case, you only need to use one route.

4.1.1 Data available as a data frame in an RData file

As we said in section 3, the data are available as an R data frame (`inputEx`), which we have saved as an RData file (`inputEx.RData`).

We will use `inputToADaCGH` to produce the three objects needed later for the segmentation, and to carry out some checks for missing values, repeated identifiers and positions, etc.

```
> fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.RData")
> inputToADaCGH(ff.or.RAM = "RAM",
+                           RDatafilename = fnameRdata)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing
```

[1] These two files are used in the example of the help for the `cutFile` function

```
   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1540505 41.2    2403845 64.2  1835812 49.1
Vcells  921083  7.1    1598044 12.2  1445676 11.1

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

We need to provide the path to the RData file, which we stored in the object `fnameRData`. This RData file will contain a single data frame. In this data frame, the first three columns of the data frame are the IDs of the probes, the chromosome number, and the position, and all remaining columns contain the data for the arrays, one column per array. The names of the first three column do not matter, but the order does. Names of the remaining columns will be used if existing; otherwise, fake array names will be created.

Note the usage of `ff.or.RAM = "RAM"`, which is different from that in section 5.2. The output from the call will leave several R objects in the global environment. The name of the objects can be changed with the argument `robjnames`. These are your usual R objects (data frames and vectors); thus, they are RAM objects.

4.1.2 Data available as an R data frame

Instead of accessing the RData file, we will directly use the data frame. This way, we use `inputToADaCGH` basically for its checks. The first three columns of the data frame are the IDs of the probes, the chromosome number, and the position, and all remaining columns contain the data for the arrays, one column per array.

```
> data(inputEx) ## make inputEx available as a data frame with that name
> inputToADaCGH(ff.or.RAM = "RAM",
+                       dataframe = inputEx)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames
```

```
   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1540496 41.2    2403845 64.2  1835812 49.1
Vcells  921045  7.1    1598044 12.2  1445676 11.1

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

Skipping the call to `inputToADaCGH` Since our data are already available as an R data frame, and if we are not interested in the checks provided by `inputToADaCGH`, we do not need to call it. To prepare the data for later usage with `pSegment` we can just do as follows:

```
> data(inputEx)
> cgh.dat <- inputEx[, -c(1, 2, 3)]
> chrom.dat <- as.integer(inputEx[, 2])
> pos.dat <- inputEx[, 3]
```

4.1.3 Using input data from a text file

Our data can also be in a text file, with a format where the first three columns are ID, chromosome, and position, and the remaining columns are arrays[2]. `inputDataToADaCGH` allows this type of input and, inside, uses `read.table.ff`; this way, we can read a very large data set and store it as an *ff* object or a RAM object without exhausting the available RAM.

```
> fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
> tmp <- inputToADaCGH(ff.or.RAM = "RAM",
+                              textfilename = fnametxt)

   ...  textfile reading: reading the ID column

   ...  textfile reading: reading the chrom column

   ...  textfile reading: (parallel) reading of remaining columns

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, ff version
```

[2]If they are not, utilities such as `awk`, `cut`, etc, might be used for this purpose.

```
   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1567507 41.9    2403845 64.2  1835812 49.1
Vcells  946632  7.3    1598044 12.2  1445676 11.1

 Saved objects with names
 cgh.dat chrom.dat pos.dat probenames.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

If you will be using a cluster created with `makeCluster` (see section 5.3) you will not want to use this options. You will need to create *ff* objects because, when using a cluster, and to minimize transferring data and possibly exhausting available RAM, we have written the code so that the slaves do not receive the data itself, but just pointers to the data (i.e., names of *ff* objects) that live in the disk.

Compressed text files The function `inputToADaCGH` will work with both compressed and uncompressed files. However, if you are working with a really large text file, if you start from a compressed file, you will have to add the time it takes to decompress the file; thus, you might want to decompress it, outside R, before you start all of your work if you plan on using this file repeatedly as input.

4.1.4 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

4.1.5 Reading data from a directory

Reading data from a directory is discussed in more detail in section 6.2.6, and it is the preferred approach when we have a lot of data. Since saving the results as a RAM object is not likely to be the way to go in such cases (we would exhaust available RAM), we do not discuss it here any further.

4.2 Carrying out segmentation and calling

Segmentation and calling are carried out with the `pSegment` functions. Here we show just one such example. Many more are available in the second vignette. Setting argument `typeParall` to `fork` is not needed (it is the default), but we set it here explicitly for clarity.

```
> help(pSegment)
```

```
> haar.RAM.fork <- pSegmentHaarSeg(cgh.dat, chrom.dat,
+                                  merging = "MAD",
+                                  typeParall = "fork")
```

Since the input are RAM objects, the output is also a RAM object (a regular R object, in this case a list).

```
> lapply(haar.RAM.fork, head)
```

```
$outSmoothed
         L.1       L.2       m4       m5       L3 m6
1         NA 0.0175353 5.939581 2.929741 6.055182 NA
2         NA 0.0175353 5.939581 2.929741 6.055182 NA
3         NA 0.0175353 5.939581 2.929741 6.055182 NA
4         NA 0.0175353 5.939581 2.929741       NA NA
5         NA 0.0175353 5.939581 2.929741       NA NA
6 0.05851487 0.0175353 5.939581 2.929741       NA NA

$outState
  L.1 L.2 m4 m5 L3 m6
1  NA   0  1  1  1 NA
2  NA   0  1  1  1 NA
3  NA   0  1  1  1 NA
4  NA   0  1  1 NA NA
5  NA   0  1  1 NA NA
6   0   0  1  1 NA NA
```

```
> summary(haar.RAM.fork[[1]])
```

```
      L.1                L.2                 m4               m5
 Min.   :-0.18305   Min.   :-0.080705   Min.   :0.9303   Min.   :-4.0270
 1st Qu.:-0.10712   1st Qu.:-0.004725   1st Qu.:2.0171   1st Qu.: 0.0738
 Median :-0.06615   Median : 0.017535   Median :2.1786   Median : 0.1857
 Mean   :-0.03548   Mean   : 0.007684   Mean   :3.4504   Mean   : 1.6016
 3rd Qu.: 0.05851   3rd Qu.: 0.017535   3rd Qu.:5.9396   3rd Qu.: 2.9014
 Max.   : 0.17439   Max.   : 0.056750   Max.   :5.9396   Max.   : 9.0388
 NA's   :5          NA's   :15                           NA's   :41
       L3                 m6
 Min.   :-12.960   Min.   :-0.20148
 1st Qu.:  3.919   1st Qu.:-0.09948
 Median :  3.995   Median :-0.04680
 Mean   :  1.981   Mean   :-0.03510
 3rd Qu.:  4.008   3rd Qu.: 0.06151
 Max.   :  6.055   Max.   : 0.17410
 NA's   :9         NA's   :203
```

4.3 Plotting the results

Plotting produces PNG files for easier sharing over the Internet. The plotting function takes as main arguments the names of the result from `pSegment` and the input objects to `pSegment` (we will later see, for instance in section 5.5, how to use results stored as *ff* objects). Setting argument `typeParall` to `fork` is not needed (it is the default), but we set it here explicitly for clarity.

```
> pChromPlot(haar.RAM.fork,
+            cghRDataName = cgh.dat,
+            chromRDataName = chrom.dat,
+            posRDataName = pos.dat,
+            probenamesRDataName = probenames.dat,
+            imgheight = 350,
+            typeParall = "fork")
```

5 Example 2: *ff* objects and cluster

This procedure should work even with relatively small amounts of RAM, and it will also work under Windows. However, using a cluster involves additional steps. For both interactive and non-interactive sessions we will often execute the following in sequence:

1. Check the original data and convert to appropriate objects (e.g., to *ff* objects).
2. Initialize the computing cluster.
3. Carry out segmentation and calling
4. Plot the results

Compared to section 4 we introduce here the following new major topics:

- Using *ff* objects.
- Setting up a cluster.

5.1 Choosing a working directory

As we will use *ff* objects, we will read and write quite a few files to the hard drive. The easiest way to organize your work is to create a separate directory for each project. At the end of this example, we will remove this directory. All plot files and ff data will be stored in this new directory.

(Just in case, we check for the existence of the directory first. We also store the current working directory to return to it at the very end.)

```
> originalDir <- getwd()
```

```
> if(!file.exists("ADaCGH2_vignette_tmp_dir"))
+   dir.create("ADaCGH2_vignette_tmp_dir")
> setwd("ADaCGH2_vignette_tmp_dir")
```

It is **very important** to remember that the names of the *ff* objects that are exposed to the user are always the same (i.e., chromData.RData, posData.RData, cghData.RData, probeNames.RData). Therefore, successive invocations of `inputToADaCGH`, if they produce *ff* output (i.e., `ff.or.RAM = "ff"`) will overwrite this objects (and make them point to different binary *ff* files on disk). In this vignette, we keep reusing `inputToADaCGH`, but note that in all the cases we produce as output *ff* files (sections 5.2, 5.2.1, 5.2.2, 5.2.4, 6.2, 6.2.1, 6.2.2, 6.2.4), the data used as input are the same, so there is no problem here (although we will leave binary *ff* objects on disk without a corresponding *ff* RData object on the R session). In particular, note that when we show the usage of Limma and snapCGH objects as input (section 7.1), we are using RAM objects (not *ff* objects) as output, so there is no confusion.

5.2 Reading data and storing as *ff* objects

Converting the original data to *ff* objects can be done either before or after initializing the cluster (section 5.3), as it does not use the computing cluster. The purpose of this step is to write the *ff* files to disk, so they are available for the segmentation and ploting functions.

5.2.1 Data available as a data frame in an RData file

To allow the conversion to be carried out using data from previous sessions, the conversion takes as input the name of an RData that contains plain, "regular" R objects (which, if loaded, would be RAM objects).

```
> fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
+                       full.names = TRUE, pattern = "inputEx.RData")
> inputToADaCGH(ff.or.RAM = "ff",
+                        RDatafilename = fnameRdata)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1580270 42.2    2403845 64.2  1835812 49.1
Vcells  960913  7.4    1757946 13.5  1445676 11.1

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.
```

The first command is used in this example to find the complete path of the example data set. The actual call to the function is the second expression. Note that we used a path to an RData file, and do not just use a RAM object. If you are very short of RAM, you might want to do the conversion in a separate R process that exists once the conversion is done and returns all of the RAM it used to the operating system. This we cover next in section 5.2.2. An alternative approach to try to minimize RAM is available if our data are in a text file, as discussed in section 4.1.3.

5.2.2 Converting from RData to *ff* objects in a separate process

With large data sets, converting from RData to *ff* can be the single step that consumes the most RAM, since we need to load the original data into R. Even if, after the conversion to *ff*, we remove the original data and call `gc()`, R might not return all of the memory to the operating system, and this might be inconvenient in multiuser environments and/or long running processes.

We can try dealing with the above problems by executing the conversion to *ff* in a separate R process that is spawned exclusively just for the conversion. For instance, we could use the `mcparallel` function (from package **parallel**) and do:

```
> mcparallel(inputToADaCGH(ff.or.RAM = "ff",
+                                  RDatafilename = fnameRData),
+                                  silent = FALSE)
>   tableChromArray <- mccollect()
>   if(inherits(tableChromArray, "try-error")) {
+     stop("ERROR in input data conversion")
+   }
```

That way, the *ff* are produced and stored locally in the hard drive, but the R process where the original data was loaded (and the conversion to *ff* carried out) dies immediately after the conversion, freeing back the memory to the operating system.

5.2.3 Data available as an R data frame

Instead of accessing the RData file, we can directly use the data frame, as we did in section 4.1.2.

```
> data(inputEx) ## make inputEx available as a data frame with that name
> inputToADaCGH(ff.or.RAM = "ff",
+                       dataframe = inputEx)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1580267 42.2    2403845 64.2  1835812 49.1
```

```
Vcells  960946  7.4    1757946 13.5  1445676 11.1

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.
```

5.2.4 Using input data from a text file

As in 4.1.3, we can read from a text file. In this case, however, the output will be a set of *ff* objects.

```
> fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
> inputToADaCGH(ff.or.RAM = "ff",
+                       textfilename = fnametxt)
   ...  textfile reading: reading the ID column

   ...  textfile reading: reading the chrom column

   ...  textfile reading: (parallel) reading of remaining columns

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, ff version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1580423 42.3    2403845 64.2  1835812 49.1
Vcells  960942  7.4    1757946 13.5  1445676 11.1

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.
```

5.2.5 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

5.2.6 Reading data from a directory

See section 6.2.6 for further details. This option is the best option with very large data sets. The initial data reading will use forking and, once we have saved the objects as *ff* objects, we can apply all the subsequent analysis steps discussed in the rest of this section.

5.2.7 Moving a set of *ff* objects

This is not specific to ADaCGH2, but since this issue can come up frequently, we explain it here. The paths of the *ff* files are stored in the object. How can we move this R object with all the *ff* files? First, we save the R object and all the *ff* files:

```
ffsave(cghData, file = "savedcghData", rootpath = "./")
```

We then take the resulting RData object (possible a very large object), and load it in the new location, rerooting the path:

```
ffload(file = "pathtofile/savedcghData", rootpath = getwd())
```

5.3 Initializing the computing cluster

Cluster initialization uses the functions provided in `parallel`. In the example we will use a sockect cluster, since this is likely to run under a variety of operating systems and should not need any additional software. Note, however, that MPI can also be used (in fact, that is what we use in our servers). In this example we will use as many nodes as cores can be detected.

```
> number.of.nodes <- detectCores()
> cl2 <- parallel::makeCluster(number.of.nodes, "PSOCK")
> parallel::clusterSetRNGStream(cl2)
> parallel::setDefaultCluster(cl2)
> parallel::clusterEvalQ(NULL, library("ADaCGH2"))

[[1]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[2]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[3]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[4]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[5]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[6]]
```

```
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[7]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[8]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[9]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[10]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[11]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[12]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[13]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[14]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[15]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"

[[16]]
 [1] "ADaCGH2"   "ff"        "bit"       "tools"     "parallel"  "methods"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "base"
```

```
> wdir <- getwd()
> parallel::clusterExport(NULL, "wdir")
> parallel::clusterEvalQ(NULL, setwd(wdir))

[[1]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[2]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2
```

```
[[3]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[4]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[5]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[6]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[7]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[8]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[9]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[10]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[11]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[12]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[13]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[14]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[15]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2

[[16]]
[1] "D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2
```

The first two calls create a cluster and initialize the random number generator[3]. The third expression sets the cluster just created as the default cluster. This is important: to simplify function calls, we do not pass the cluster name around, but rather expect a default cluster to be set up. The fourth line makes the **ADaCGH2** package available in all the nodes of the cluster (notice we did not need to do this with forking, as the child processes shared memory with the parent).

[3]We use the version from package **parallel**, instead of the one from **BiocGenerics**, as the last one is still experimental.

The last three lines make sure the slave processes use the same directory as the master. Because we created the cluster after changing directories (section 5.1) this step is not really needed here. But we make it explicit so as to verify it works, and as a reminder that you will need to do this if you change directories AFTER creating the cluster. If you run on a multinode cluster, you must ensure that the same directory exists in all machines. (In this case, we are running on the localhost).

5.4 Carrying out segmentation and calling

Segmentation and calling are carried out with the `pSegment` functions. Here we show just one such example. Many more are available in the second vignette.

```
> help(pSegment)
```

```
> haar.ff.cluster <- pSegmentHaarSeg("cghData.RData",
+                                    "chromData.RData",
+                                    merging = "MAD",
+                                    typeParall = "cluster")
```

We can take a quick look at the output. We first open the *ff* objects (the output is a list of *ff* objects) and then call `summary` on the list that contains the results of the wavelet smoothing:

```
> lapply(haar.ff.cluster, open)

$outSmoothed
[1] TRUE

$outState
[1] TRUE
```

```
> summary(haar.ff.cluster[[1]][,])
```

```
      L.1                L.2                  m4                m5
 Min.   :-0.18305   Min.   :-0.080705   Min.   :0.9303   Min.   :-4.0270
 1st Qu.:-0.10712   1st Qu.:-0.004725   1st Qu.:2.0171   1st Qu.: 0.0738
 Median :-0.06615   Median : 0.017535   Median :2.1786   Median : 0.1857
 Mean   :-0.03548   Mean   : 0.007684   Mean   :3.4504   Mean   : 1.6016
 3rd Qu.: 0.05851   3rd Qu.: 0.017535   3rd Qu.:5.9396   3rd Qu.: 2.9014
 Max.   : 0.17439   Max.   : 0.056750   Max.   :5.9396   Max.   : 9.0388
 NA's   :5          NA's   :15                           NA's   :41
       L3                 m6
 Min.   :-12.960   Min.   :-0.20148
 1st Qu.:  3.919   1st Qu.:-0.09948
 Median :  3.995   Median :-0.04680
 Mean   :  1.981   Mean   :-0.03510
 3rd Qu.:  4.008   3rd Qu.: 0.06151
 Max.   :  6.055   Max.   : 0.17410
 NA's   :9         NA's   :203
```

5.5 Plotting the results

The call here is the same as in section 4.3, except that we change the values for the arguments. As we are using *ff* objects, we also need to first write to disk the (*ff*) object with the results.

```
> save(haar.ff.cluster, file = "hs_mad.out.RData", compress = FALSE)
> pChromPlot(outRDataName = "hs_mad.out.RData",
+            cghRDataName = "cghData.RData",
+            chromRDataName = "chromData.RData",
+            posRDataName = "posData.RData",
+            probenamesRDataName = "probeNames.RData",
+            imgheight = 350,
+            typeParall = "cluster")
```

Finally, we stop the workers and close the cluster

```
> parallel::stopCluster(cl2)
```

6 Example 3: *ff* objects and forking

This example uses *ff* objects, as in section 5, but it will not use a cluster but forking, as in section 4. Therefore, we will not need to create a cluster, but we will need to read data and convert it to *ff* objects.

Here we introduce no new major topics. Working with *ff* objects was covered in section 5.2 and forking was covered in section 4.2. We simply combine these work-flows.

6.1 Choosing a working directory

As we will use *ff* objects, it will be convenient, as we did in section 5.1, to create a separate directory for each project, to store all plot files and ff data. Since we already did that above (section 5.1) we do not repeat it here. However, for real work, you might want to keep different analyses associated to different working directories.

6.2 Reading data and storing as *ff* objects

We have here the same options as in section 5.2. We repeat them briefly. A key difference with respect to section 5.2 is that we are not creating a cluster, so there will be no need to export the current working directory to slave processes explictly (in contrast to 5.3).

6.2.1 Data available as a data frame in an RData file

```
> fnameRdata <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.RData")
> inputToADaCGH(ff.or.RAM = "ff",
+                           RDatafilename = fnameRdata)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData
```

```
   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1584251 42.4    2403845 64.2  1853422 49.5
Vcells  965807  7.4    1757946 13.5  1729123 13.2

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.

>
```

6.2.2 Converting from RData to *ff* objects in a separate process

Even if we are using forking, we might still want to carry the conversion to *ff* objects in a separate process, as we did in section 5.2.2, since the conversion to *ff* objects might be the step that consumes most RAM in the whole process and we might want to make sure we return that memory to the operating system as soon as possible.

```
> mcparallel(inputToADaCGH(ff.or.RAM = "ff",
+                                  RDatafilename = fnameRdata),
+            silent = FALSE)
> tableChromArray <- collect()
> if(inherits(tableChromArray, "try-error")) {
+   stop("ERROR in input data conversion")
+ }
```

6.2.3 Data available as an R data frame

Instead of accessing the RData file, we can directly use the data frame, as we did in section 5.2.3.

6.2.4 Using input data from a text file

```
> fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                           full.names = TRUE, pattern = "inputEx.txt")
> inputToADaCGH(ff.or.RAM = "ff",
+                        textfilename = fnametxt)

   ...  textfile reading: reading the ID column

   ...  textfile reading: reading the chrom column

   ...  textfile reading: (parallel) reading of remaining columns

   ... done reading; starting checks
```

```
      ... checking identical MidPos

      ... checking need to reorder inputData, ff version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1584395 42.4    2403845 64.2  1853422 49.5
Vcells  965703  7.4    1757946 13.5  1729123 13.2

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.
```

6.2.5 Using data from Limma or snapCGH

You can also use data from snapCGH and Limma. See section 7.1.

6.2.6 Reading data from a directory

This is probably the best option for very large input data. We will read **all** the files in a given directory (except for those you might explicitly specify not to). Even if your original file follows the format of the data file in 6.2.4, you might want to convert it to the format used here (where each column is a file) as the time it takes to convert the file will be more than compensated by the speed ups of reading, in R, each file on its own. With very large files, it is much faster to read the data this way (we avoid having to loop many times over the file to read each column). Reading the data is parallelized, which allows us to speed up the reading process significantly (the parallelization uses forking, and thus you will see no speed gains in Windows). Finally, to maximize speed and minimize memory consumption, we use *ff* objects for intermediate storage.

6.2.7 Cutting the original file into one-column files

We provide a simple function, `cutFile`, to do this job. Here we create a directory where we will place the one-column files (we first check that the directory does not exist[4]). Note that this will probably NOT work under Windows[5] , and thus we skip using `cutFile` under Windows, and use a directory where we have stored the files split by column.

[4]If it exists and contains files, `inputToADaCGH` will probably fail, as it is set to read all the files in the directory.

[5]Under Macs it might or might not work; in all of the Macs we have tried it, it works, but not on the testing machine at BioC.

```
> if( (.Platform$OS.type == "unix") && (Sys.info()['sysname'] != "Darwin") ) {
+   fnametxt <- list.files(path = system.file("data", package = "ADaCGH2"),
+                          full.names = TRUE, pattern = "inputEx.txt")
+   if(file.exists("cuttedFile")) {
+     stop("The cuttedFile directory already exists. ",
+          "Did you run this vignette from this directory before? ",
+          "You will not want to do that, unless you modify the arguments ",
+          "to inputToADaCGH below")
+   } else dir.create("cuttedFile")
+   setwd("cuttedFile")
+   cutFile(fnametxt, 1, 2, 3, sep = "\t")
+   cuttedFile.dir <- getwd()
+   setwd("../")
+ } else {
+   cuttedFile.dir <- system.file("example-datadir",
+                                 package = "ADaCGH2")
+ }
```

We create a new directory and carry out the file cutting there since the upper level directory is already populated with other files we have been creating. If we cut the file in the upper directory, we would later need to specify a lengthy list of files to exclude in the arguments to `inputToADaCGH`. To avoid that, we create a directory, and leave the files in the newly created directory. After cutting, we return to the former level directory, to keep that directory with only the files for input.

It is important to realize that the previous paragraph, which might seem a mess, does not reflect the way you would usually work, which would actually be much simpler, and something like the following:

1. Create a directory for your new project (lets call this directory `d1`).
2. Copy the text file with your big txt file with data to `d1`; lets call this file `afile.txt`.
3. In R, move to `d1` (for example, `setwd(" /d1")`).
4. Use `cutFile`: `cutFile("afile.txt", 1, 2, 3)`.
5. Call `inputToADaCGH`: `inputToADaCGH(ff.or.RAM = "ff", path = getwd(), excludefile = "afile.txt")`

(In this vignette the work flow was not as easy because we are running lots of different examples, with several different work flows.)

`cutFile` will run several jobs in parallel to speed up the cutting process, launching by default as many jobs as cores it can detect, and will produce files with the required naming conventions of `inputToADaCGH`. Note that `cutFile` is unlikely to work under Windows.

If you do not want to use `cutFile` you can use utilities provided by your operating system. The following is a very simple example of using `cut` under bash (which is not unlike what we do internally in `cutFile`) to produce one-column files from a file called `Data.txt`, with 77 arrays/subjects, where cutting the data part is parallelized over four processors:

```
cut -f1 Data.txt > ID.txt
cut -f2 Data.txt > Chrom.txt
cut -f3 Data.txt > Pos.txt

for i in {4..20}; do cut -f$i Data.txt > col_$i.txt; done &
```

```
for i in {21..40}; do cut -f$i Data.txt > col_$i.txt; done &
for i in {41..60}; do cut -f$i Data.txt > col_$i.txt; done &
for i in {61..80}; do cut -f$i Data.txt > col_$i.txt; done &
```

After you have cut the file, each file contains one column of data. Three of the files must be named `"ID.txt"`, `"Chrom.txt"`, and `"Pos.txt"`. The rest of the files contain the data for each one of the arrays or subjects. The name of the rest of the files is irrelevant.

When using `inputDataToADaCGH` with a directory, the output can be either *ff* objects or RAM objects. However, the latter will rarely make sense (it will be slower and we can run into memory contraints); see section 8.

```
> inputToADaCGH(ff.or.RAM = "ff",
+               path = cuttedFile.dir,
+               verbose = TRUE)

 Note: Directory reading: we will be reading  6
 files, not including ID, Chrom, and Pos.
 If this is not the correct number of files,
 stop this process, verify why  (did cutFiles
 work correctly? are you using a directory with
 other files?, etc), and run again.

      These are the files we will try to read:
col_4.txt
col_5.txt
col_6.txt
col_7.txt
col_8.txt
col_9.txt

   ...  directory reading: reading the ID file

   ...  directory reading: reading the chromosome file

   ...  directory reading: reading the Positions file

   ...  directory reading: parallel reading of column names

   ...  directory reading: parallel reading of data columns

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, ff version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData
```

```
   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1584372 42.4    2403845 64.2  1853422 49.5
Vcells  965888  7.4    1757946 13.5  1729123 13.2

 Files saved in current directory
 D:/biocbld/bbs-2.13-bioc/tmpdir/RtmpUtM9Si/Rbuilda38d133e95/ADaCGH2/vignettes/ADaCGH2_vig
 with names :
 chromData.RData, posData.RData, cghData.RData, probeNames.RData.
```

We have used the previously cut files in this example. You can also check the files that live under the "example-datadir" directory and you will see six files with names starting with "col", which are the data files, and the files `"ID.txt"`, `"Chrom.txt"`, and `"Pos.txt"`. (That is the directory we would use as input had we used Windows.)

Note that, to provide additional information on what we are doing we are calling the function with the (non-default) `verbose = TRUE`, which will list all the files we will be reading.

Beware of possible different orderings of files. When reading from a directory, and since each column is a file, the order of the columns (and, thus, subjects or arrays) in the data files that will be created can vary. In particular, the command `list.files` (which we use to list of the files) can produce different output (different order of files) between operating systems and versions of R. What this means is that, say, column three does not necessarily refer to the same subject or array. **Always use the column names to identify unambiguously the data and the results**.

What about performing this step in a separate process? In sections 5.2.2 and 6.2.2 we performed the data preparation in a separate process, to free up RAM to the OS right after the conversion. You can do that too here if you want, but we have not found that necessary, since the memory consumption when reading column by column is often small. See examples with large data sets in section 8.2.

6.3 Carrying out segmentation and calling

The call is similar to the one in 5.4, except for the argument `typeParall`.

```
> haar.ff.fork <- pSegmentHaarSeg("cghData.RData",
+                                 "chromData.RData",
+                                  merging = "MAD",
+                                 typeParall = "fork")
```

6.4 Plotting the results

The call here is the same as in section 5.5, except for argument `typeParall`.

```
> save(haar.ff.fork, file = "haar.ff.fork.RData", compress = FALSE)
> pChromPlot(outRDataName = "haar.ff.fork.RData",
```

```
+            cghRDataName = "cghData.RData",
+            chromRDataName = "chromData.RData",
+            posRDataName = "posData.RData",
+            probenamesRDataName = "probeNames.RData",
+            imgheight = 350,
+            typeParall = "fork")
```

7 Input and output to/from other packages

7.1 Input data from Limma and snapCGH

Many aCGH studies use pre-processing steps similar to those of gene expression data. The `MAList` object, from *Limma* and `SegList` object, from *snapCGH*, are commonly used to store aCGH information. The following examples illustrate the usage of the function `inputToADaCGH` to convert `MAList` and `SegList` data into a format suitable for *ADaCGH2*.

We will start with objects produced by *snapCGH*. The following code is copied from the *snapCGH* vignette (pp. 2 and 3). Please check the original vignette for details. In summary, a set of array files are read, the data are normalized and, finally, averaged over clones. snapCGH uses limma for the initial import of data and, next, with the `read.clonesinfo` function adds additional information such as chromosome and position. The `MA` object created is of class `MAList`, but with added information (compared to a basic, original, limma MAList object). `MA2` is of type `SegList`.

```
> require("limma")
> require("snapCGH")
> datadir <- system.file("testdata", package = "snapCGH")
> targets <- readTargets("targets.txt", path = datadir)
> RG1 <- read.maimages(targets$FileName, path = datadir, source = "genepix")
```

```
Read D:/biocbld/bbs-2.13-bioc/R/library/snapCGH/testdata/10Mbslide28.gpr
Read D:/biocbld/bbs-2.13-bioc/R/library/snapCGH/testdata/10Mbslide29.gpr
```

```
> ## Thi is snapCGH-specific
> RG1 <- read.clonesinfo("cloneinfo.txt", RG1, path = datadir)
> RG1$printer <- getLayout(RG1$genes)
> types <- readSpotTypes("SpotTypes.txt", path = datadir)
> RG1$genes$Status <- controlStatus(types, RG1)
```

```
Matching patterns for: ID Name Chr
Found 21 CTD-
Found 9 CTA-
Found 405 Chrom8
Found 66 Chrom3
Found 75 Chrom1
Setting attributes: values Color
```

```
> RG1$design <- c(-1, -1)
> RG2 <- backgroundCorrect(RG1, method = "minimum") ## class RGList
> MA <- normalizeWithinArrays(RG2, method = "median") ## class MAList
> class(MA)
```

```
[1] "MAList"
attr(,"package")
[1] "limma"
```

```
> ## now obtain an object of class SegList
> MA2 <- processCGH(MA, method.of.averaging = mean, ID = "ID")

Averaging duplicated clones
Processing chromosome  1
Processing chromosome  2
Processing chromosome  3
Processing chromosome  4
Processing chromosome  5
Processing chromosome  6
Processing chromosome  7
Processing chromosome  8
Processing chromosome  9
Processing chromosome  10
Processing chromosome  11
Processing chromosome  12
Processing chromosome  13
Processing chromosome  14
Processing chromosome  15
Processing chromosome  16
Processing chromosome  17
Processing chromosome  18
Processing chromosome  19
Processing chromosome  20
Processing chromosome  21
Processing chromosome  22

> class(MA2)

[1] "SegList"
attr(,"package")
[1] "snapCGH"
```

All the information (intensity ratios and location) is available in the `MA` and `MA2` objects. We can directly convert them to *ADaCGH2* objects (we set `na.omit = TRUE` as the data contain missing values). The first call process the `MAList` and the second the `SegList`.

In this section, we use the argument `robjnames`, to produce as output a set of RAM objects with a different set of names from the default. (Note that we could also have produced *ff* files as output, using the option `ff.or.RAM = "ff"`).

```
> tmp <- inputToADaCGH(MAList = MA,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"))

   ... missing values in Position or Chromosome; removing those rows

   ... done reading; starting checks

      ... checking identical MidPos

 We have identical MidPos!!!

      ... checking need to reorder inputData, data.frame version
```

```
        ... reordering inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1601822 42.8    2403845 64.2  1853422 49.5
Vcells 1035110  7.9    1757946 13.5  1757942 13.5

 Saved objects with names
 cgh-ma.dat chrom-ma.dat pos-ma.dat probenames-ma.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .

> tmp <- inputToADaCGH(MAList = MA2,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"),
+                              minNumPerChrom = 4)

   ... done reading; starting checks

      ... checking identical MidPos

      ... checking need to reorder inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1602270 42.8    2403845 64.2  1853422 49.5
```

```
Vcells 1031232  7.9    1757946 13.5  1757942 13.5

 Saved objects with names
 cgh-ma.dat chrom-ma.dat pos-ma.dat probenames-ma.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

We need to change the argument to `minNumPerChrom` because, after the data processing step in `processCGH`, chromosome 21 has only four observations.

The original `MAList` as produced directly from *limma* do not have chromosome and position information. That is what the `read.clonesinfo` function from *snapCGH* did. To allow using objects directly from *limma* and incorporating position information, we will use an approach to directly mimicks that in *snapCGH*. If you use and `MAList` you can also provide a `cloneinfo` argument; this can be either the full path to a file with the format required by `read.clonesinfo` or, else, the name of an object with (at least) three columns, names `ID`, `Chr`, and `Position`.

We copy from the limma vignette (section 3.2, p.8), changing the names of objets by appending ".limma".

```
> targets.limma <- readTargets("targets.txt", path = datadir)
> RG.limma <- read.maimages(targets.limma, path = datadir,
+                           source="genepix")

Read D:/biocbld/bbs-2.13-bioc/R/library/snapCGH/testdata/10Mbslide28.gpr
Read D:/biocbld/bbs-2.13-bioc/R/library/snapCGH/testdata/10Mbslide29.gpr

> RG.limma <- backgroundCorrect(RG.limma, method="normexp",
+                               offset=50)

Array 1 corrected
Array 2 corrected
Array 1 corrected
Array 2 corrected

> MA.limma <- normalizeWithinArrays(RG.limma)
```

We can add the chromosomal and position information in two different ways. First, as in `read.clonesinfo` or, else, we can provide the name of a file (with the same format as required by `read.clonesinfo`). Note that `fclone` is a path (and, thus, a character vector).

```
> fclone <- list.files(path = system.file("testdata", package = "snapCGH"),
+                      full.names = TRUE, pattern = "cloneinfo.txt")
> fclone

[1] "D:/biocbld/bbs-2.13-bioc/R/library/snapCGH/testdata/cloneinfo.txt"

> tmp <- inputToADaCGH(MAList = MA.limma,
+                              cloneinfo = fclone,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"))
```

```
Assuming cloneinfo is a file (possibly with full path)
   ... missing values in Position or Chromosome; removing those rows

   ... done reading; starting checks

      ... checking identical MidPos

 We have identical MidPos!!!

      ... checking need to reorder inputData, data.frame version

         ... reordering inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1603337 42.9    2403845 64.2  1853422 49.5
Vcells 1068321  8.2    1757946 13.5  1757942 13.5

 Saved objects with names
 cgh-ma.dat chrom-ma.dat pos-ma.dat probenames-ma.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

Alternatively, we can provide the name of an object with the additional information. For illustrative purposes, we can use here the columns of the `MA` object.

```
> acloneinfo <- MA$genes
> tmp <- inputToADaCGH(MAList = MA.limma,
+                              cloneinfo = acloneinfo,
+                              robjnames = c("cgh-ma.dat", "chrom-ma.dat",
+                                            "pos-ma.dat", "probenames-ma.dat"))

Assuming cloneinfo is an R data frame
   ... missing values in Position or Chromosome; removing those rows

   ... done reading; starting checks

      ... checking identical MidPos
```

```
 We have identical MidPos!!!

      ... checking need to reorder inputData, data.frame version

         ... reordering inputData, data.frame version

   ... done with checks; starting writing

   ... done writing/saving probeNames

   ... done writing/saving chromData

   ... done writing/saving posData

   ... done writing/saving cghData


 Calling gc at end

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells 1603293 42.9    2403845 64.2  1853422 49.5
Vcells 1058543  8.1    1757946 13.5  1757942 13.5

 Saved objects with names
 cgh-ma.dat chrom-ma.dat pos-ma.dat probenames-ma.dat
 for CGH data, chromosomal data, position data, and probe names,
 respectively, in environment
 R_GlobalEnv .
```

7.2 Using CGHregions

The CGHregions package Vosse and van de Wiel (2009) is a BioConductor package that implements a well known method van de Wiel and van Wieringen (2007) for dimension reduction for aCGH data (see a review of common regions issues and methods in Rueda and Diaz-Uriarte (2010)).

The `CGHregions` function accepts different type of input, among others a data frame. The function `outputToCGHregions` produces that data frame, ready to be used as input to CGHregions (for the next example, you will need to have the *CGHregions* package installed).

Note: **it is up to you to deal with missing values!!!** In the example below, we do a simple `na.omit`, but note that we are now working with data frames. Extending the usage of this, and other methods, to much larger data sets, using *ff*, and properly dealing with missing values, is beyond the scope of this package.

```
> forcghr <- outputToCGHregions(haar.ff.cluster)
> if(require(CGHregions)) {
+   regions1 <- CGHregions(na.omit(forcghr))
+   regions1
+ }

[1] 1 0 7
[1] 2.00000000 0.08539095 6.00000000
[1] "Tuning on small data set finished...started with entire data set"
[1] 1 0 7 0
```

```
[1] 2.00000000 0.08333333 6.00000000
[1] "c = 1, nr of regions: 7"
[1] "Finished with entire data set."
cghRegions (storageMode: lockedEnvironment)
assayData: 7 features, 6 samples
  element names: regions
protocolData: none
phenoData: none
featureData
  featureNames: 1 2 ... 7 (7 total)
  fvarLabels: Chromosome Start ... AveDist (5 total)
  fvarMetadata: labelDescription
experimentData: use 'experimentData(object)'
Annotation:
```

Please note that `outputToCGHregions` does NOT check if the calls are something that can be meaningfully passed to CGHregions. In particular, you probably do NOT want to use this function when `pSegment` has been called using `merging = "none"`.

```
> ## We are done with the executable code in the vignette.
> ## Restore the directory
> setwd(originalDir)
```

8 (Non-runnable) Examples with large data sets and a comparison of approaches

We discuss here a specific example to give you a rough idea of timings and preferred approaches for analyzing very large data sets.

8.1 Data set and hardware

We will use a simulated data set that contains 6,067,433 rows and 1000 columns of data; these are, thus, data for an array with 6 million probes and data for 1000 subjects. In some examples shown below, we use smaller subsets of the same data (with only 200 or 50 columns) and in some cases we will use a larger data set with 2000 subjects. There are 421,000 missing values per data column. The ASCII file with the data for the 1000 column data is about 96 GB[6] and the RData for those same data is 46 GB (without compression; 41 GB with the standard R compression); in a freshly started R session, loading the RData will use 46 GB (as reported by `gc()`). We will analyze the full data set with 1000 columns, and some smaller versions of that data set (with 200 and 50 data columns).

The examples below were run on a Dell PowerEdge C6145 chasis with two nodes. Each node has 4 AMD Opteron 6276 processors; since each processor has 16 cores, each node has 64 cores. One node has 256 GB RAM and the other 384 GB of RAM. Both nodes are connected by Infiniband (40Gb/s). For the data presented here, when using a single node, the data live on an xfs partition of a RAID0 array of SAS disks (15000 rpm) that are local to the node doing the computations. When using the two nodes, the data live on that same local SAS drive, which is seen by the other node using a simple NFS setup (we have also used distributed file systems, such as FhGFS, but they have tended to be a lot slower in these

[6] All sizes are computed from the reported size in bytes or megabytes, using 1024, or powers of 1024, as denominator.

experiments; your mileage might vary). Therefore, in the table entries below, executions using both nodes will indicate "124 cores" [7]

We will also show some examples run on an HP Z800 workstation, with 2 Intel Xeon E5645 processors (each processor has six cores), and 64 GB of RAM. The data live on an ext4 partition on a SATA disk (7200 rmp). In the tables, this is indicated as "Coleonyx, 12 cores" (table with results for reading data) or by the number 12 in the column for number of cores.

In both systems, the operating system is Debian GNU/Linux (a mixture of Debian testing and Debian unstable). The Dell PowerEdge nodes were running version R-2.15.1 as available from the Debian repository (v. 2.15.1-5) or, later, R-3.0.1, patched (different releases, as available through May and June of 2013), and compiled from sources. The Xeon workstation was running R-2.15.1, patched version (2012-10-02 r60861), compiled from sources or, later R-3.0.1, patched (different releases, as available through May and June of 2013). Open MPI is version 1.4.3-2 (as available from Debian).

As a reference for the size of the data set, the RData object with the 1000 columns, when loaded into R in the PowerEdges, takes 13 minutes to load and uses a total of about 46 GB (45.7 from calls to `gc` before and after loading the object, and adding Ncells and Vcells, or 45.6 as reported by `object.size`). Note that this is not the result of the object being a data frame and having a column with identifiers (a factor), instead of being a matrix; a similarly sized matrix with just the numeric data for the probes (i.e., without the first three columns of ID, chromosome, and location) has a size of 45.2 GB (therefore, the difference of 300 MB due to the first column, ID, being a factor with the identifiers, is minute relative to the size of the matrix).

For the two tables below, the meaning of columns is as follows:

Wall time The "elapsed" entry returned by the command `system.time`. This is the real elapsed time since the function was called.

It is important to understand that these timings can be variable, and we show here a single number.

Max. mem (`gc`) The sum of the two rows of the "max used" column reported by `gc()`, in R, at the end of the execution of the given function. This number cannot reflect all the memory used by the function if the function spawns other R processes.

Max. Σ mem. A simple attempt to measure the memory used by all the processes[8]. Right before starting the execution of our function, we call the operating system command `free` and record the value reported by the "-/+ buffers/cache" row. Then, while the function is executing, we record, every 0.5 seconds, that same quantity. The largest difference between the successive measures and the original one is the largest RAM consumption. Note that this is an approximation. First, if other process start executing, they will lead to an overestimation of RAM usage; this, however, is unlikely to have had serious effects (the systems were basically idle, except for light weight cron jobs). Second, sampling is carried out every 0.5 seconds, so we could miss short peaks in RAM usage (at least one likely example of such underestimation shows in the tables below).

Columns The number of data columns of the data set.

[7] 124 is not a typo; it is 124, even if the total number of cores is $128 = 64 * 2$. This is due to the following documented issue with Open MPI and Infiniband: `http://www.open-mpi.org/community/lists/users/2011/07/17003.php`, and since $128^2 = 16384$, we are hitting the limit, and we have not had a chance to correct this problem yet. Regardless, the penalty we would pay would be a difference of 4 process out of 124.

[8] Just adding the entries given by `top` or `ps` will not do, and will overestimate, sometimes by a huge amount, the total memory used.

Method The analysis method (either HaarSeg or CBS —DNAcopy).

Fork/MPI Whether forking (via `mclapply`) or MPI (using the facilities provided by package **Rmpi**, which are called from package **snow**) are used to parallelize execution. The two "NP" entries refer to non-parallelized execution, using the original packages[9].

Cores Number of cores, and machine. 64: 64 cores in a single node of the Dell PowerEdge C6145. 124: 124 cores, using both nodes of the Dell PowerEdge. 12: 12 cores in the Z800 HP Workstation.

ff/RAM If the data for the analysis had been stored as an *ff* object, or as a data frame inside an RData that was loaded before the analysis.

For the table for reading data, all results, except for the one indicated in the table, correspond to using a single node in the Dell PowerEdge C6145 (64 cores).

[9]The packages are DNAcopy, as available from BioConductor, and the HaarSeg package, available from R-forge: `https://r-forge.r-project.org/projects/haarseg/`; Ben-Yaacov and Eldar (2009)

8.2 Reading data

Table 1: Time and memory usage when reading data

	Columns	Wall time (minutes)	Max. mem. (gc), GB	Max. Σ mem., GB
Directory to *ff*	2000	13.9	1.3	39.1
Directory to *ff*	1000	6.1	1.3	41.21
Directory to *ff*	500	4.6	1.3	39.7
Directory to *ff* (Coleonyx, 12 cores)	1000	88	1.3	8.4
Txt file to *ff*	1000	2630	1.3	NA (not calculated)
RData to *ff*	1000	29.6	169	168.3
Directory to data frame (RAM object)	1000	22 + 2[a]	96	NA. Output unusable for analysis
RData to data frame (RAM object)	1000	22 + 2[b]	139	NA. Output unusable for analysis
Directory to data frame (RAM object)	200	7.7 + 0.4[c]	20	38
Directory to *ff*	200	3.6	1.3	38
Directory to data frame (RAM object)	100	8.7 + 0.3[d]	10.5	30
Directory to *ff*	100	3.6	1.3	32
Directory to *ff*	50	3.2	1.3	28
Directory to data frame (RAM object)	50	5.8 + 0.1[e]	5.8	27

[a] The 2 reflects the time needed to save the resulting data frame to an RData file.
[b] The 2 reflects the time needed to save the resulting data frame to an RData file.
[c] The 0.4 reflects the time needed to save the resulting data frame to an RData file.
[d] The 0.3 reflects the time needed to save the resulting data frame to an RData file.
[e] The 0.1 reflects the time needed to save the resulting data frame to an RData file.

8.3 Analyzing data

Table 2: Time and memory usage of segmentation with default options

Method	Fork /MPI	Cores	ff/RAM	Columns	Wall time (min.)	Max. mem. (gc), GB	Max. Σ mem., GB
CBS	MPI	124	ff	1000	313	0.14	53
CBS	MPI	64	ff	1000	545	0.14	51
CBS	Fork	64	ff	2000	705.3	0.16	45.7
CBS	Fork	12	ff	2000	2414.4	0.15	19.3
CBS	Fork	64	ff	1000	339	0.136	41
CBS	Fork	12	ff	1000	1184.0	0.142	13.8
CBS	Fork	64	ff	500	185.8	0.133	41
HaarSeg	MPI	124	ff	1000	12.3	0.145	38.5
HaarSeg	MPI	64	ff	1000	9.5	0.145	38.5
HaarSeg	MPI	12	ff	1000	36	0.144	7.2
HaarSeg	Fork	64	ff	2000	18.7	0.15	28.4
HaarSeg	Fork	12	ff	2000	57.4	0.16	10.1
HaarSeg	Fork	64	ff	1000	7.9	0.14	27
HaarSeg	Fork	12	ff	1000	32	0.129	7.3
HaarSeg	Fork	64	ff	200	2.3	0.135	28.6
HaarSeg	Fork	64	ff	500	4.6	0.13	26
HaarSeg	Fork	10[f]	ff	1000	33.3	0.142	5.9
HaarSeg	Fork	64	ff	50	0.5	0.133	20.6
HaarSeg	Fork	64	RAM	50	0.7 + 2.5 + 0.9[e]	14.4	140
HaarSeg	MPI	64	ff	50	0.63	0.14	24.6
HaarSeg	Fork	64	RAM	1000	NA	NA	**Cannot allocate memory**
HaarSeg	Fork	64	RAM	200	NA	NA	**Cannot allocate memory**

[d] Since only one process, same as previous column.

[e] 0.7 + 2.5 + 0.9: load data, analyze, and save results.

Table 3: Time and memory usage of segmentation without merging and comparison with non-parallized executions. These examples have all been run on the Dell Power Edges.

Method	Fork /MPI	Cores	ff/RAM	Columns	Wall time (min.)	Max. mem. (gc), GB	Max. Σ mem., GB
HaarSeg	Fork	64	ff	100	1.2	0.13	24.5
HaarSeg	Fork	10	ff	1000	23.5	0.142	5.9
HaarSeg	Fork	20	ff	1000	12.3	0.137	10.0
HaarSeg	Fork	40	ff	1000	7.4	0.142	17.6
HaarSeg	Fork	50	ff	1000	6.7	0.139	21.2
HaarSeg	Fork	64	ff	1000	6.4	0.142	26.9
HaarSeg	Fork	10	ff	2000	49.7	0.16	8.0
HaarSeg	Fork	20	ff	2000	26.3	0.16	11.9
HaarSeg	Fork	40	ff	2000	15.4	0.16	19.5
HaarSeg	Fork	50	ff	2000	13.3	0.16	23.2
HaarSeg	Fork	64	ff	2000	11.9	0.16	28.4
CBS	Fork	64	ff	100	55.9	0.13	35.3
CBS	Fork	10	ff	1000	1855.4	0.135	8.6
CBS	Fork	20	ff	1000	939.8	0.135	14.9
CBS	Fork	40	ff	1000	513.5	0.136	27.0
CBS	Fork	50	ff	1000	438.6	0.142	33.1
CBS	Fork	64	ff	1000	350.3	0.142	41.3
CBS	Fork	10	ff	2000	3770.9	0.15	11.1
CBS	Fork	20	ff	2000	1878.8	0.16	16.5
CBS	Fork	40	ff	2000	1007.0	0.15	28.8
CBS	Fork	50	ff	2000	857.1	0.16	35.3
CBS	Fork	64	ff	2000	717.3	0.163	41.9
HaarSeg	NP	-	RAM	100	0.95 + 22.9 + 1.4[b]	12.5	12.5[d]
CBS	NP	-	RAM	100	0.95 + 1698 + 6.7[c]	40	40[d]

[b] 0.95 + 22.9 + 1.4: load data, analyze, and save results. Since there are missing values in the data, and the original HaarSeg code does not deal with missing values, we are forced to remove NAs array-per-array, and make repeated calls to the function. If there are no missing values in this data set, the total time of analysis (i.e., sending the whole matrix at once and not checking for, nor removing, NAs) is 3.3 minutes.

[c] 0.95 + 1698 + 6.7: load data, analyze, and save results. The analysis involves calling the `CNA` function to create the CNA object (5.3 min), calling the `smooth.CNA` function to smooth the data and detect outlier (83.2 minutes), and segmenting the data with the `segment` function (1609.5 minutes).

8.4 Comments and recommended usage patterns

1. Reading data and trying to save it as a RAM object, a usual in-memory data frame, will quickly exhaust available memory. For these data, we were not able to read data sets of 100 or more columns. Part of the problem lies on the way memory is handled and freed in the slaves, given that we are returning actual lists. In contrast, when saving as *ff* objects, the slaves are only returning tiny objects (just pointers to a file).

2. Saving data as RData objects will also not be an option for large numbers of columns as we will quickly exhaust available memory when trying to analyze them.

3. In a single machine, and for the same number of cores, analyzing data with MPI is often generally slower than using forking, which is not surprising. Note also that with MPI there is an overhead of spawning slaves and loading packages in the slaves (which, in our case, takes about half a minute to a minute).

4. When using two nodes (i.e., almost doubling the number of cores), MPI might, or might not, be faster than using forking on a single node. Two main issues affect the speed differences: inter-process communication and access to files. In our case, the likely bottleneck lies in access to files, which live on a SAS disk which is accessed via NFS. With other hardware/software configurations, access to shared files might be much faster. Regardless, the MPI costs might not be worth if each individual calculation is fast; this is why MPI with HaarSeg does not pay off, but it does pay off with CBS.

5. When using *ff*, the exact same operations in systems with different RAM can lead to different amounts of memory usage, as *ff* tries autotuning when starting. You can tune parameters when you load the **ff** package, but even if you don't (and, by default, we don't), defaults are often sensible and will play in your favor.

6. Even for relatively small examples, using *ff* can be faster than using RAM objects. Using RAM objects incurs overheads of loading and saving the RData objects in memory, but analyses also tend to be slightly slower. The later is somewhat surprising: with forking and RAM objects, the R object that holds the CGH data is accessed only for reading, and thus can be shared between all processes. We expected this to be faster than using *ff*, because access to disk is several orders of magnitude slower than access to memory —note that we made sure that memory was not virtual memory mapped to disk, as we had disabled all swapping. We suspect the main difference lies in bottlenecks and contention problems that result from accessing data in a single data frame simultaneously from multiple processes, compared to loading exactly just one column independently in each process, and/or repeated cache misses.

7. `inputToADaCGH` (i.e., transforming a directory of files into *ff* objects) can be severely affected, of course, by other processes accessing the disk. More generally, since with `inputToADaCGH` several processes can try to access different files at once (we are trying to parallelize the reading of data), issues such as type of file system, configuration and type of RAID, amount of free space, etc, can have an effect on all heavy I/O operations. For example, when reading a set of 2000 files, the time differences in a set of benchmarks varied between 9 and 34 minutes. The only difference between the runs we could find being that in the 34 minute runs, we had added about 60 GB worth of files after creating/copying the text files, and before reading, and the disk was about 64% full. Note also that, as a general rule, it is better if the newly created *ff* files from `inputToADaCGH` are written to an empty directory, and if the working directory for segmentation analysis is another empty directory if you are using `ff` objects.

8. Reordering data takes time (a lot if you do not use *ff* objects) and can use a lot of memory. So it is much better if input data are already ordered (by Chromosome and Position within Chromosome).

References

Ben-Yaacov, E. and Eldar, Y. C. (2009). *HaarSeg: HaarSeg*. R package version 0.0.3/r4.

Carro, A., Rico, D., Rueda, O. M., Diaz-Uriarte, R., and Pisano, D. G. (2010). waviCGH: a web application for the analysis and visualization of genomic copy number alterations. *Nucleic acids research*, 38 Suppl:W182–7.

Diaz-Uriarte, R. and Rueda, O. M. (2007). ADaCGH: A parallelized web-based application and R package for the analysis of aCGH data. *PloS one*, 2(1):e737.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rueda, O. M. and Diaz-Uriarte, R. (2010). Finding Recurrent Copy Number Alteration Regions : A Review of Methods. *Current Bioinformatics*, 5:1–17.

van de Wiel, M. A. and van Wieringen, W. N. (2007). CGHregions: Dimension Reduction for Array CGH Data with Minimal Information Loss. *Cancer informatics*, 3(0):55–63.

Vosse, S. and van de Wiel, M. (2009). *CGHregions: Dimension Reduction for Array CGH Data with Minimal Information Loss*. R package version 1.7.1.