Benchmarking ADaCGH2 and comparison with previous versions

Ramon Diaz-Uriarte[1]

28-December-2013

1. Department of Biochemistry, Universidad Autonoma de Madrid Instituto de Investigaciones Biomedicas "Alberto Sols" (UAM-CSIC), Madrid (SPAIN).
rdiaz02@gmail.com

Contents

List of Tables

List of Figures

1 Introduction

I provide here comparisons with the former version of ADaCGH2 (v. 1.10, as available in BioConductor v. 2.12) as well as some comparisons against non-parallelized executions and, finally, some details about recommended patterns of usage. The benchmarks shown here total more than 2047 hours of wall time (more than 85 days) and correspond to over 400 runs for reading and 370 for analysis. The purpose of these benchmarks is to show the differences in performance between the new and old versions, as well as to illustrate the effects of changing several parameters in the new version. Before showing the results, we provide information about the hardware and data sets.

It should be noted that the very first version of ADaCGH (the one documented in Diaz-Uriarte and Rueda (2007)) will no longer run in current versions of R without tweaks, as it depends on a package (papply) that no longer installs in current versions of R (it had problems at least since v. 2.15.0 of R). The web-based application documented in that paper still works because I did perform those tweaks, and because in some servers we are still running version 2.9.0 of R. However, for the final user, comparisons against that initial version are therefore of little interest. Thus, the benchmarks that I show provide comparisons against the version available from release 2.12 of BioConductor; this "old" version (v. 1.10) still runs, but already incorporates several major advantages over the initial one documented in the PLoS ONE paper, mainly:

- Clusters are not restricted to MPI cluster, whereas the initial version only allowed MPI clusters; so, for instance socket clusters (much easier to use in Windows than MPI clusters) were not available.
- There are no dependencies on deprecated or orphaned packages so the package will install in current versions of R.
- *ff* objects start to be used. The very first version always required the complete data set to be in memory in the master process for the duration of the analysis, and MPI moved around actual columns of the data frame, and not just pointers to *ff* objects. As a consequence, for the initial version, memory requirements for analysis were actually higher than the memory requirements of reading data of the "old" version; thus the largest possible data for analysis were smaller than the ones for v. 1.10, and analyses were also slower.
- Two algorithms, ACE and PSW, were eliminated because of little usage, and a new and very fast one, HaarSeg, included.
- Input from, and output to, other BioConductor packages was added.

Therefore, in what follows, "old" refers to v. 1.10, as available from BioConductor 2.12, and "new" to versions *ge*2.3, and those are the versions that will be compared.

1.1 Data set and hardware

We will use a simulated data set that contains 6,067,433 rows and up to 4000 columns of data; these are, thus, data for an array with 6 million probes and data for up to 4000 subjects. Many of the examples shown below will use smaller subsets of the data, smaller in terms of the number of subjects or samples (columns). There are 421,000 missing values per data column.

To give an idea of sizes, the ASCII file with the data for the 1000 column data is about 96 GB[1]. The directory with the data for 2000 columns occupies about 198 GB and, when archived and compressed with bzip2, occupies 78 GB. The RData for the 1000 columns 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()`). 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).

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 one of the xfs partitions, 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 experiments; your mileage might vary). Therefore, in the table entries below, executions using both nodes will indicate "124 cores" [2]

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 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).

1.2 Segmentation methods used

The methods available in ADaCGH2 are (see further details in the help of function `pSegment`):

- The popular Circular Binary Segmentation approach, described in its current implementation in Venkatraman and Olshen (2007) and implemented in package `DNAcopy`.
- A wavelet-based method, proposed in Hsu et al. (2005). This is called `pSegementWavelets` in ADaCGH2.
- Another wavelet-based method, HaarSeg, published in Ben-Yaacov and Eldar (2008). This was later made available as an R-package as Ben-Yaacov and Eldar (2009).

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

[2] 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.

- HMM, as described by Fridlyand et al. (2004) and implemented in package Fridlyand and Dimitrov (2010).
- BioHMM, a non-homogeneous HMM, described in Marioni et al. (2006) and implemented in package Smith et al. (2009).
- The CGHseg method described in Picard et al. (2005). An implementation of **part** of this method is available in the R package Huber et al. (2006), but ADaCGH2 is the first R implementation of the full description of the CGHseg procedure (see comments in the help of function `pSegmentCGHseg`).
- GLAD, a method first described in Hupe et al. (2004) and implemented in Hupe (2011).

1.2.1 Difficulties of using some methods with large data sets

The tables below only show benchmarks for methods HMM, BioHMM, HaarSeg (referred as Haar) and CBS. CGHseg and the wavelet approach described in Hsu et al. (2005) cannot be used when any chromosome has a large number of probes because of their memory use. With CGHseg the problem arises in the underlying `tilingArray` package, in the internal step of computing the "costMatrix", a function called by the function `segment` in `tilingArray`. When analyzing the first chromosome (for a single subject), the request is for 1493 GB. For the wavelet approach, the problem shows up in the clustering step, when function `pam` (from package `cluster`) is called. For instance, the memory requirements for a chromosome of 350000 probes would exceeded 400 GB (the request is for a vector of doubles of size $1 + (n * (n - 1)))$. It must be emphasized that, in both cases, it is not the complete 6 million probes, nor using multiple subjects, which causes the problems: neither of the methods is capable of analyzing the first chromosome for a single subject. GLAD seems capable of dealing with large data sets in terms of memory usage, but it is extremely slow. After more than four days, the method had not been able to finish the analysis of the 50-column data set in the machines with 64 cores; on closer inspection, the problem lies in function `OptimBkpFindCluster`, a C function internal to the package, and is not attributable, therefore, to the initial segmentation method (we were using, anyway, the recommended fast function, which uses HaarSeg). Finally, to run method BioHMM we often had to increase the `ulimit` (stack limit), by using `ulimit -u`, from the shell.

1.3 Tables: column name explanation

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

Wall time (min.) The "elapsed" entry returned by the command `system.time`. This is the real elapsed time, the wall time, in minutes, since the function was called.

It is important to understand that these timings can be variable. In many cases, we show repeated executions with the exact same settings, that will help show the variability in those numbers.

Memory (GB) The memory used by the master R process. This is 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 (via MPI or forking, for example).

Σ **Memory (GB)** A simple attempt to measure the memory used by all the processes[3]. Right before starting the execution of our function, we call the operating system com-

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

mand `free` and record the value reported by the "-/+ buffers/cache" row. Then, while the function is executing, we record, every 0.1 seconds (or every 0.05 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), though a few results in the tables suggest this happened in a few instances (related to backup processes). Second, sampling is carried out every 0.5 seconds, so we could miss short peaks in RAM usage but, again, this is unlikely to lead to a serious underestimation.

Finally, note that for cases where we know that there is a single R process (e.g., reading with the old version), there is an excellent agreement between the "Memory (GB)" (whose value is reported from R itself) and "Σ Memory (GB)".

Columns The number of data columns of the data set; the same as the number of arrays or the number of samples.

Method The analysis method. "Haar" for HaarSeg, "CBS" for Circular Binary Segmentation (from package `DNAcopy`), "HMM" for the HMM approach in package `aCGH`, "BioHMM" for the non-homogeneous HMM method in package `snapCGH`, and "GLAD" for the method with the same name in package `GLAD`. See section 1.2.1 for why other methods are not shown in the tables.

MPI/Fork Whether forking (via `mclapply`) or explicitly using an MPI cluster (using the facilities provided by package **Rmpi**, which are called from package **snow**) are used to parallelize execution.

The "NP" entries in table 8 refer to non-parallelized execution, using the original packages[4].

The entries marked as "-LB" correspond to the load-balanced options with the new version of ADaCGH2 (setting `loadBalance = TRUE`, in v. $\geq$ 2.3.4).

Cores Number of cores used. In most cases, when running in the AMD Opteron machines we used all 64 cores, and when running on the Intel Xeon machine we used all 12 cores, but not always, to show the effects of changing the number of cores used. When running over both AMD Opterons we used 124 cores (see above).

Procs. per node When using MPI, the total number of R processes that can run in a node; this is the parameter `npernode` passed to `mpirun` (from Open MPI). When running on a single node, that is the number of R slaves + 1.

Universe size The number of slave nodes in the MPI universe (over all nodes in the universe). This is the parameter `count` passed to `makeMPIcluster` in R.

Version The version of ADaCGH2. For simplicity, "Old" means version 1.10 and "New" versions 2.1 and larger.

The post-fix "-noNA" means the new version was run using option `certain_noNA = TRUE`; note that the old version of ADaCGH2 assumes there are no missing values in the data. Thus, `certain_noNA = TRUE` is the closest to what the old version assumes.

ff/RAM Where applicable in the tables, 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.

[4]The packages are DNAcopy, as available from BioConductor, and the HaarSeg package, available from R-forge: `https://r-forge.r-project.org/projects/haarseg/`

2 Comparison with v. 1.10 of ADaCGH2

2.1 Main differences between the old (v. 1.10) and new versions (v. $\geq$ 2.3.4) of ADaCGH2

The code for the new version of ADaCGH2 represent a major rewrite of most of the code in the former version. Listed here are some of the major advantages of the new version[5]; they are shown in approximately decreasing order of importance from the user's point of view.

Reading of large data sets The new version of ADaCGH2 can read data sets much larger than the old one (see section 2.3). In a machine with 64 GB RAM the old version cannot read data sets with 500 columns (each with 6 million probes —see section 1.1), whereas data sets with 4000 columns can be read with the new version (see table 1) and the scaling of the memory consumption with number of columns suggests that much larger data sets could be read. Likewise, in machines with 256 and 384 GB of RAM (tables 2 and 3) data sets of 2000 columns could not be read with the old version of ADaCGH2, but data sets of 4000 columns are read with the new version and, again, the scaling of memory consumption with number of columns suggests (see Figure 1) that much larger data sets could be read and, even for the sizes of data that can be read by the old version, reading is much faster with the new version because of the parallelized reading, which can make much better usage of available hardware (e.g., RAID arrays for disks).

Missing value handling The old version of ADaCGH2 used row-wise deletion of missing values when reading data: a probe would be deleted from the data if it had one missing value in any subject/column. Analysis could be speed up, as no checks or provisions had to be taken for dealing with NAs, and all procedures are simplified, as the data are then known to be complete. However, row-wise deletion of missing values is probably not an appropriate approach, especially as the number of samples increases (because the probability that a given probe will then be left out of the analysis increases). The new version of ADaCGH2 deals with missing values column by column, so for each column (or subject) all available data (or probes) are used in the segmentation. Nevertheless, the new version incorporates a setting to provide speed ups when the user is certain that there are no missing values (`certain_noNA = TRUE`).

Analysis of large data sets The old version of ADaCGH2 cannot analyze large data sets, as it cannot read them (and it cannot use data read by the new version since the old version assumes there are no missing values in the data after reading).

In addition, although time increases, obviously, with number of samples to analyze, the scaling of memory consumption is modest and well below the memory available for the systems.

Forking and clusters The new version of ADaCGH2 allows for the usage of forking or an explicit cluster (e.g., MPI, sockets, etc) to parallelize reading and analysis. In POSIX operating systems (including Unix, GNU/Linux, and Mac OS), forking can be faster, less memory consuming, and much easier to use than using a cluster.

Speed of analysis The new version can be slightly faster than the old one for the default options. Further speed improvements can be achieved in some cases, for instance by not using load balancing with certain methods (e.g., HaarSeg).

[5]Most of the new capabilities were already available in version 2.1.3; however, the vignettes have suffered major changes and there have been some changes in the code and help files. Thus, these comments all do apply to version $\geq$ 2.3.4.

Flexibility of reading data The new version of ADaCGH2 has not removed the mechanisms of reading data available in the old version. Thus, when data are small or memory is plentiful, reading data from a single RData is an available option. But the new version adds new mechanisms, mainly reading from a text file and from a directory of text files that, as discussed above, allow for reading much larger data sets.

Usage of data read from the other version The new version of ADaCGH2 can accept data read by the old version. However, the old version of ADaCGH2 cannot accept data from read by the new version unless the original data contained no missing values at all: the old version of ADaCGH2 assumes that data that have been read contain no missing values.

Dependencies The old version depends on package `snowfall` for parallelization, whereas the new version depends only on `multicore`. This makes the new version less likely to break in the future, as `multicore` is one of the core packages distributed with R (whereas, for instance, there were some problems with `snowfall` not building with the development versions of v. 3 of R around February 2013).

More flexible options for load balancing The old version of ADaCGH2 forced load balancing. Whether or not load-balancing is the best approach depends on the size and number of jobs relative to the number of cores. As shown in the tables (see tables 4, 6, 5), not using load balancing can sometimes lead to speed improvements. The new version of ADaCGH2 allows not to use load balancing with the argument `loadBalance = FALSE`.

Limiting memory consumption Memory usage is generally well below the available memory of the system. However, if it were necessary to limit memory usage during reading and analysis this is simple with the new version of ADaCGH2: limit the number of processes that are allowed to run simultaneously. This is not possible with the old version of ADaCGH2.

Method availability Two of the methods available, HMM and BioHMM, depend on packages `aCGH` and `snapCGH`. These two packages haven not been updated since 2010 and 2009, respectively, and aCGH will no longer be maintained (personal communication from the authors). There is code in the ADaCGH2 repositories (including both C and R code), taken and modified from those packages, that can be readily uncommented to make these two methods available in ADaCGH2 if either of those packages were not to pass checks in future versions of BioConductor.

2.2 Data and code availability

All scripts, data, and results from these benchmarks are available from `http://www2.iib.uam.es/rduriarte_lab/ADaCGH2-v2-suppl-files/public`. The scripts include the R code to obtain the tables and figures shown here. All of the code, scripts, and benchmark results are also available from my personal web site at `http://ligarto.org/rdiaz/Papers/ADaCGH2-v2-suppl-files/` (for space restriction reasons, the more than 140 GB of data in the form of RData and txt files are only available from the previous site.)

2.3 Reading data

The next three tables show time and memory consumption when reading data using the recommended approach with each version of the package (an RData object for the old version, a directory of single-column txt files for the new version). Some of the major patterns and results are:

Size limits for old version Table 1 **cannot** show reading benchmarks for the old version with data sets of sizes 500, 1000, 2000, or 4000, as those could not be read with the old version (R run out of memory). Likewise, tables 2 and 3 show reading benchmarks for the old version with up to 1000 columns, because the AMD Opteron machines with $\geq$ 256 GB RAM could not read data sets of sizes 2000 or 4000, as R could not allocate the necessary memory.

In contrast, the new version is capable of reading data sets of 4000 columns in all machines, without getting anywhere near the memory limits of the machines. Moreover, the **scaling** (see also figure 1) shows that the total number of columns could be increased to much larger numbers and, in addition, that the total memory used can be limited by reducing the number of cores used (with little effects on speed —see next point).

Speed of new and old version Reading is much faster with the new version. These differences are most likely inconsequential for small sized data sets (where differences are by a factor of about 2x), but can have large effects with a large number of columns. As data sets grow larger in size, reading speeds are much faster with the new version than the old by factors of about 10x (this can only be verified in the machines with larger memory, as the old version will not read data sets of 500 columns or more in the smaller machine). There can, nevertheless, be quite a bit of variation in reading speeds of small data sets, specially in less capable machines; these variations, however, are most likely of little practical relevance.

Speed and number of cores in new version Reading speed does not always increase with the number of cores. In fact, for a range of number of cores, reading speeds show little variation with number of cores, as the most likely limiting factor is I/O, which is related to the number of spindles and the speed of the drives. Increasing the number of cores used, however, tends to make the system less responsive (higher loads) and thus using a reasonably small number of cores is recommended and the default option.

If the reading operation is to be performed many times, or on very large set of data, it would pay off to experiment with the number of cores used for reading, which can be done with the option `mc.cores` to the function `inputToADaCGH`.

Table 1: Reading benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
1	3.0	1.33	3.2	50	4	New
2	3.0	1.33	3.2	50	4	New
3	5.7	1.33	4.5	50	6	New
4	2.7	1.33	4.5	50	6	New
5	2.7	1.33	4.5	50	6	New
6	2.4	1.33	5.6	50	8	New
7	2.4	1.33	6.9	50	10	New
8	2.4	1.33	8.0	50	12	New
9	6.0	1.33	8.1	50	12	New
10	2.4	1.33	8.1	50	12	New
11	5.9	1.33	8.1	50	12	New
12	2.4	1.33	8.1	50	12	New
13	2.4	1.33	8.1	50	12	New
14	4.2	9.05	9.0	50		Old
15	4.6	9.05	8.9	50		Old
16	4.5	9.05	8.6	50		Old
17	4.2	9.05	9.0	50		Old
18	4.1	9.05	9.0	50		Old
19	12.4	1.33	1.3	100	1	New
20	10.2	1.33	1.3	100	1	New
21	6.2	1.33	2.0	100	2	New
22	6.2	1.33	1.9	100	2	New
23	3.9	1.33	3.3	100	4	New
24	4.2	1.33	3.2	100	4	New
25	3.9	1.33	3.4	100	4	New
26	4.1	1.33	3.4	100	4	New
27	3.5	1.33	4.4	100	6	New
28	3.3	1.33	4.4	100	6	New
29	3.3	1.33	4.5	100	6	New
30	3.4	1.33	4.5	100	6	New
31	3.3	1.33	5.0	100	7	New
32	3.3	1.33	5.0	100	7	New
33	2.9	1.33	5.7	100	8	New
34	2.9	1.33	5.7	100	8	New
35	2.8	1.33	6.2	100	9	New
36	2.8	1.33	6.2	100	9	New
37	2.9	1.33	7.0	100	10	New
38	2.9	1.33	6.9	100	10	New
39	2.7	1.33	7.5	100	11	New
40	2.7	1.33	7.5	100	11	New
41	2.7	1.33	8.1	100	12	New
42	2.6	1.33	8.1	100	12	New
43	10.3	1.33	8.1	100	12	New
44	2.6	1.33	8.1	100	12	New
45	10.0	1.33	8.1	100	12	New
46	2.6	1.33	8.4	100	12	New
47	2.8	1.33	8.1	100	12	New
48	7.1	16.93	17.0	100		Old

Table 1: *(Reading benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
49	5.9	16.93	16.4	100		Old
50	5.5	16.93	16.6	100		Old
51	5.6	16.93	17.0	100		Old
52	6.1	1.33	3.5	200	4	New
53	6.3	1.33	3.5	200	4	New
54	17.4	1.33	4.7	200	6	New
55	5.0	1.33	4.6	200	6	New
56	4.9	1.33	4.6	200	6	New
57	4.8	1.33	4.8	200	6	New
58	5.6	1.33	5.1	200	7	New
59	4.6	1.33	5.0	200	7	New
60	4.2	1.33	5.9	200	8	New
61	4.2	1.33	5.9	200	8	New
62	4.1	1.33	6.2	200	9	New
63	4.1	1.33	6.3	200	9	New
64	3.9	1.33	7.2	200	10	New
65	3.8	1.33	7.0	200	10	New
66	3.8	1.33	7.5	200	11	New
67	3.6	1.33	7.5	200	11	New
68	3.5	1.33	8.1	200	12	New
69	3.6	1.33	8.2	200	12	New
70	18.3	1.33	8.2	200	12	New
71	3.4	1.33	8.1	200	12	New
72	12.8	1.33	8.4	200	12	New
73	3.6	1.33	8.1	200	12	New
74	3.6	1.33	8.3	200	12	New
75	9.2	30.01	30.0	200		Old
76	9.1	31.23	31.1	200		Old
77	9.1	31.02	31.1	200		Old
78	9.1	31.02	31.2	200		Old
79	42.2	1.33	3.4	500	4	New
80	41.1	1.33	4.4	500	6	New
81	42.4	1.33	6.1	500	8	New
82	42.7	1.33	7.3	500	10	New
83	44.6	1.33	8.5	500	12	New
84	44.5	1.33	8.7	500	12	New
85	42.4	1.33	8.5	500	12	New
86	82.3	1.33	3.6	1000	4	New
87	81.7	1.33	4.4	1000	6	New
88	85.8	1.33	6.1	1000	8	New
89	87.7	1.33	7.3	1000	10	New
90	96.2	1.33	8.6	1000	12	New
91	171.2	1.33	3.6	2000	4	New
92	173.8	1.33	4.5	2000	6	New
93	162.3	1.33	4.7	2000	6	New
94	173.0	1.33	6.1	2000	8	New
95	181.6	1.33	7.4	2000	10	New

Table 1: *(Reading benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
96	188.2	1.33	8.7	2000	12	New
97	349.3	1.32	3.0	4000	4	New
98	339.7	1.32	4.4	4000	6	New
99	333.8	1.32	4.4	4000	6	New
100	347.5	1.32	6.1	4000	8	New
101	366.6	1.32	7.4	4000	10	New
102	373.4	1.32	8.6	4000	12	New

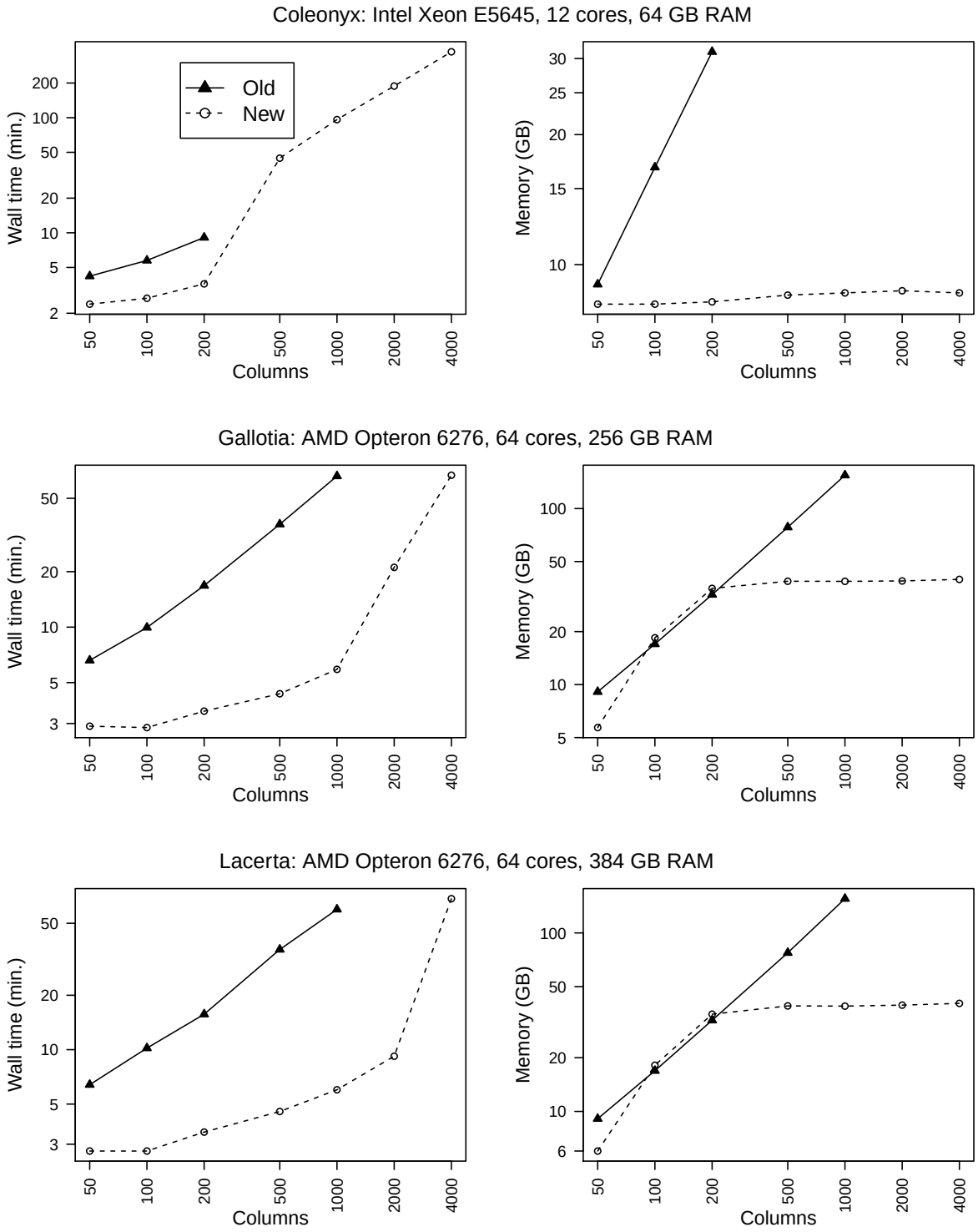

Figure 1: Comparison between old and new versions in wall time and total memory usage (over all spawned processes) when reading data as a function of number of columns (or arrays or samples). Both axes shown in log scale. The figure shows the benchmarks using 12 cores in the Intel Xeon machine and 64 cores in the AMD Opterons; note that for some scenarios better speeds (and lower memory usage) can be achieved by decreasing the number of cores used (see tables). When more than one benchmark is available for a scenario, the median is shown.

Table 2: Reading benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
1	3.7	1.33	3.8	50	5	New
2	3.8	1.33	3.8	50	5	New
3	3.5	1.33	6.9	50	10	New
4	3.2	1.33	6.9	50	10	New
5	3.4	1.33	6.7	50	10	New
6	2.9	1.33	11.5	50	20	New
7	2.9	1.33	11.6	50	20	New
8	2.8	1.33	17.0	50	30	New
9	3.3	1.33	16.5	50	30	New
10	2.8	1.33	9.9	50	40	New
11	2.8	1.33	10.0	50	40	New
12	2.7	1.33	5.8	50	50	New
13	3.1	1.33	5.8	50	50	New
14	2.8	1.33	6.1	50	55	New
15	2.6	1.33	5.7	50	55	New
16	2.9	1.33	6.2	50	60	New
17	2.9	1.33	5.4	50	60	New
18	2.9	1.33	5.6	50	64	New
19	2.7	1.33	6.3	50	64	New
20	2.9	1.33	5.7	50	64	New
21	2.9	1.33	5.7	50	64	New
22	6.6	9.05	9.1	50		Old
23	7.3	9.05	9.1	50		Old
24	6.5	9.05	9.1	50		Old
25	7.1	9.05	9.1	50		Old
26	6.5	9.05	9.1	50		Old
27	5.0	1.33	4.0	100	5	New
28	5.0	1.33	4.0	100	5	New
29	4.3	1.33	6.9	100	10	New
30	3.8	1.33	7.1	100	10	New
31	4.1	1.33	6.8	100	10	New
32	3.2	1.33	13.1	100	20	New
33	3.4	1.33	13.1	100	20	New
34	3.0	1.33	17.1	100	30	New
35	3.6	1.33	16.9	100	30	New
36	2.8	1.33	21.5	100	40	New
37	3.3	1.33	21.0	100	40	New
38	3.0	1.33	24.9	100	50	New
39	3.1	1.33	24.9	100	50	New
40	3.2	1.33	21.9	100	55	New
41	3.1	1.33	23.0	100	55	New
42	3.0	1.33	18.7	100	60	New
43	2.9	1.33	19.3	100	60	New
44	2.7	1.33	18.8	100	64	New
45	3.4	1.33	16.8	100	64	New
46	2.7	1.33	18.3	100	64	New
47	3.0	1.33	18.6	100	64	New
48	9.8	16.93	17.0	100		Old

Table 2: *(Reading benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
49	10.9	16.93	17.0	100		Old
50	10.1	16.93	17.0	100		Old
51	9.8	16.93	16.9	100		Old
52	7.5	1.33	4.1	200	5	New
53	7.7	1.33	4.0	200	5	New
54	5.5	1.33	6.8	200	10	New
55	5.2	1.33	7.4	200	10	New
56	5.7	1.33	7.1	200	10	New
57	3.6	1.33	13.3	200	20	New
58	3.9	1.33	13.3	200	20	New
59	3.5	1.33	19.2	200	30	New
60	4.8	1.33	18.7	200	30	New
61	3.8	1.33	24.2	200	40	New
62	3.7	1.33	24.2	200	40	New
63	3.3	1.33	28.0	200	50	New
64	3.7	1.33	28.2	200	50	New
65	3.5	1.33	30.4	200	55	New
66	3.2	1.33	30.1	200	55	New
67	3.6	1.33	32.8	200	60	New
68	3.5	1.33	32.4	200	60	New
69	3.5	1.33	35.2	200	64	New
70	3.5	1.33	35.2	200	64	New
71	3.5	1.33	35.1	200	64	New
72	3.5	1.33	35.2	200	64	New
73	16.6	32.24	32.6	200		Old
74	18.9	32.24	32.4	200		Old
75	17.0	32.24	32.4	200		Old
76	15.9	32.24	32.5	200		Old
77	15.1	1.33	4.3	500	5	New
78	15.1	1.33	4.3	500	5	New
79	10.1	1.33	7.1	500	10	New
80	9.0	1.33	7.5	500	10	New
81	9.9	1.33	7.3	500	10	New
82	5.8	1.33	13.6	500	20	New
83	5.6	1.33	13.7	500	20	New
84	4.9	1.33	18.7	500	30	New
85	8.7	1.33	18.5	500	30	New
86	4.9	1.33	24.3	500	40	New
87	4.7	1.33	24.5	500	40	New
88	4.2	1.33	30.1	500	50	New
89	4.7	1.33	30.3	500	50	New
90	4.8	1.33	33.2	500	55	New
91	4.6	1.33	33.4	500	55	New
92	4.4	1.33	36.0	500	60	New
93	4.5	1.33	36.2	500	60	New
94	4.4	1.33	38.7	500	64	New
95	4.3	1.33	38.4	500	64	New

Table 2: *(Reading benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
96	4.3	1.33	38.7	500	64	New
97	4.6	1.33	38.6	500	64	New
98	36.5	77.67	78.3	500		Old
99	37.7	77.62	79.3	500		Old
100	35.8	77.67	77.8	500		Old
101	35.7	77.67	78.4	500		Old
102	28.1	1.33	4.8	1000	5	New
103	26.7	1.33	4.8	1000	5	New
104	17.9	1.33	7.5	1000	10	New
105	14.9	1.33	7.8	1000	10	New
106	16.6	1.33	7.6	1000	10	New
107	8.9	1.33	13.9	1000	20	New
108	8.5	1.33	13.9	1000	20	New
109	6.8	1.33	19.3	1000	30	New
110	9.9	1.33	18.3	1000	30	New
111	6.6	1.33	24.6	1000	40	New
112	6.7	1.33	24.7	1000	40	New
113	6.2	1.33	30.5	1000	50	New
114	6.3	1.33	30.2	1000	50	New
115	6.4	1.33	33.5	1000	55	New
116	6.5	1.33	33.2	1000	55	New
117	6.0	1.33	36.4	1000	60	New
118	6.2	1.33	36.4	1000	60	New
119	5.6	1.33	39.0	1000	64	New
120	5.8	1.33	38.7	1000	64	New
121	5.9	1.33	38.4	1000	64	New
122	6.0	1.33	38.6	1000	64	New
123	5.9	1.33	38.3	1000	64	New
124	64.8	153.58	154.2	1000		Old
125	67.2	153.58	155.0	1000		Old
126	55.4	1.33	5.8	2000	5	New
127	56.7	1.33	5.7	2000	5	New
128	30.3	1.33	7.9	2000	10	New
129	32.4	1.33	8.7	2000	10	New
130	32.1	1.33	8.4	2000	10	New
131	30.3	1.33	8.8	2000	10	New
132	23.6	1.33	14.5	2000	20	New
133	21.1	1.33	14.5	2000	20	New
134	29.6	1.33	14.4	2000	20	New
135	19.4	1.33	20.0	2000	30	New
136	31.3	1.33	19.5	2000	30	New
137	22.3	1.33	25.6	2000	40	New
138	22.5	1.33	25.2	2000	40	New
139	21.9	1.33	25.5	2000	40	New
140	26.2	1.33	30.1	2000	50	New
141	21.2	1.33	30.5	2000	50	New
142	23.0	1.33	33.5	2000	55	New

Table 2: *(Reading benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
143	23.0	1.33	33.7	2000	55	New
144	21.8	1.33	36.3	2000	60	New
145	21.1	1.33	36.2	2000	60	New
146	21.1	1.33	38.8	2000	64	New
147	21.8	1.33	38.4	2000	64	New
148	19.7	1.33	38.8	2000	64	New
149	110.2	1.32	6.2	4000	5	New
150	60.9	1.32	9.2	4000	10	New
151	62.6	1.32	9.5	4000	10	New
152	58.0	1.32	15.0	4000	20	New
153	59.1	1.32	15.0	4000	20	New
154	60.1	1.32	18.8	4000	30	New
155	59.2	1.32	19.8	4000	30	New
156	63.8	1.32	26.0	4000	40	New
157	62.1	1.32	26.0	4000	40	New
158	63.2	1.32	31.6	4000	50	New
159	65.0	1.32	31.9	4000	50	New
160	65.8	1.32	34.9	4000	55	New
161	64.5	1.32	34.3	4000	55	New
162	65.4	1.32	37.0	4000	60	New
163	65.7	1.32	37.1	4000	60	New
164	66.7	1.32	39.2	4000	64	New
165	65.8	1.32	39.7	4000	64	New
166	67.4	1.32	39.6	4000	64	New

Table 3: Reading benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
1	3.5	1.33	3.8	50	5	New
2	3.5	1.33	3.8	50	5	New
3	3.2	1.33	6.8	50	10	New
4	3.3	1.33	6.9	50	10	New
5	2.8	1.33	11.5	50	20	New
6	3.3	1.33	11.6	50	20	New
7	2.9	1.33	17.0	50	30	New
8	3.3	1.33	16.7	50	30	New
9	2.8	1.33	9.2	50	40	New
10	2.8	1.33	9.3	50	40	New
11	2.8	1.33	5.5	50	50	New
12	2.9	1.33	5.9	50	50	New
13	2.8	1.33	5.8	50	55	New
14	3.0	1.33	5.7	50	55	New
15	2.8	1.33	5.8	50	60	New
16	2.6	1.33	5.9	50	60	New
17	2.8	1.33	6.1	50	64	New
18	2.4	1.33	5.9	50	64	New
19	2.7	1.33	6.4	50	64	New
20	2.9	1.33	5.6	50	64	New
21	6.4	9.05	9.0	50		Old
22	6.2	9.05	9.1	50		Old
23	6.9	9.05	9.1	50		Old
24	4.9	1.33	4.0	100	5	New
25	5.1	1.33	4.0	100	5	New
26	4.0	1.33	6.8	100	10	New
27	3.5	1.33	7.1	100	10	New
28	3.1	1.33	13.1	100	20	New
29	3.6	1.33	12.8	100	20	New
30	3.0	1.33	17.4	100	30	New
31	3.6	1.33	16.9	100	30	New
32	3.0	1.33	21.5	100	40	New
33	3.7	1.33	21.3	100	40	New
34	2.9	1.33	24.8	100	50	New
35	3.4	1.33	24.2	100	50	New
36	3.1	1.33	22.6	100	55	New
37	3.1	1.33	23.2	100	55	New
38	2.9	1.33	20.3	100	60	New
39	3.4	1.33	17.4	100	60	New
40	2.8	1.33	17.0	100	64	New
41	2.6	1.33	18.0	100	64	New
42	2.7	1.33	18.7	100	64	New
43	2.9	1.33	18.3	100	64	New
44	10.2	16.93	16.9	100		Old
45	9.2	16.93	16.9	100		Old
46	10.2	16.93	17.1	100		Old
47	7.3	1.33	4.0	200	5	New
48	7.7	1.33	4.1	200	5	New

Table 3: *(Reading benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
49	5.4	1.33	6.9	200	10	New
50	4.9	1.33	7.3	200	10	New
51	4.0	1.33	13.3	200	20	New
52	4.6	1.33	12.7	200	20	New
53	3.5	1.33	19.1	200	30	New
54	3.9	1.33	18.4	200	30	New
55	3.5	1.33	24.1	200	40	New
56	4.1	1.33	24.1	200	40	New
57	3.4	1.33	28.6	200	50	New
58	3.8	1.33	28.1	200	50	New
59	3.4	1.33	30.4	200	55	New
60	3.6	1.33	30.4	200	55	New
61	3.4	1.33	32.9	200	60	New
62	3.7	1.33	33.0	200	60	New
63	3.2	1.33	35.1	200	64	New
64	3.5	1.33	35.2	200	64	New
65	3.7	1.33	35.0	200	64	New
66	3.5	1.33	34.4	200	64	New
67	16.6	32.24	32.1	200		Old
68	15.5	32.24	32.4	200		Old
69	15.7	32.24	32.6	200		Old
70	14.8	1.33	4.4	500	5	New
71	15.4	1.33	4.4	500	5	New
72	9.6	1.33	7.5	500	10	New
73	8.4	1.33	7.6	500	10	New
74	5.3	1.33	13.7	500	20	New
75	8.1	1.33	12.8	500	20	New
76	4.8	1.33	18.9	500	30	New
77	5.4	1.33	18.7	500	30	New
78	5.4	1.33	24.6	500	40	New
79	4.8	1.33	24.4	500	40	New
80	4.8	1.33	30.4	500	50	New
81	4.6	1.33	30.3	500	50	New
82	4.8	1.33	33.2	500	55	New
83	4.6	1.33	33.3	500	55	New
84	5.1	1.33	36.2	500	60	New
85	4.4	1.33	36.6	500	60	New
86	4.3	1.33	39.3	500	64	New
87	4.6	1.33	39.0	500	64	New
88	4.5	1.33	39.0	500	64	New
89	4.6	1.33	38.6	500	64	New
90	35.2	77.67	77.4	500		Old
91	35.8	77.67	77.6	500		Old
92	36.7	77.67	77.2	500		Old
93	27.0	1.33	4.9	1000	5	New
94	27.9	1.33	4.8	1000	5	New
95	17.1	1.33	8.2	1000	10	New

Table 3: *(Reading benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
96	14.9	1.33	7.8	1000	10	New
97	9.1	1.33	13.8	1000	20	New
98	9.1	1.33	13.8	1000	20	New
99	7.5	1.33	18.7	1000	30	New
100	7.3	1.33	19.6	1000	30	New
101	7.0	1.33	24.6	1000	40	New
102	6.7	1.33	24.8	1000	40	New
103	6.3	1.33	30.2	1000	50	New
104	6.3	1.33	30.6	1000	50	New
105	6.1	1.33	33.1	1000	55	New
106	6.1	1.33	33.3	1000	55	New
107	5.9	1.33	36.1	1000	60	New
108	5.9	1.33	36.4	1000	60	New
109	6.1	1.33	39.4	1000	64	New
110	5.9	1.33	38.9	1000	64	New
111	6.1	1.33	38.6	1000	64	New
112	6.0	1.33	38.9	1000	64	New
113	5.8	1.33	38.4	1000	64	New
114	64.9	153.58	154.1	1000		Old
115	59.7	153.58	155.5	1000		Old
116	59.1	153.58	155.5	1000		Old
117	52.1	1.33	6.0	2000	5	New
118	54.5	1.33	6.1	2000	5	New
119	29.5	1.33	8.2	2000	10	New
120	26.7	1.33	8.9	2000	10	New
121	31.7	1.33	9.6	2000	10	New
122	15.5	1.33	14.9	2000	20	New
123	15.5	1.33	14.7	2000	20	New
124	25.5	1.33	14.9	2000	20	New
125	13.2	1.33	20.2	2000	30	New
126	11.7	1.33	20.7	2000	30	New
127	17.1	1.33	25.5	2000	40	New
128	10.9	1.33	25.7	2000	40	New
129	11.1	1.33	25.8	2000	40	New
130	12.8	1.33	31.4	2000	50	New
131	10.2	1.33	31.8	2000	50	New
132	10.8	1.33	34.1	2000	55	New
133	9.7	1.33	34.5	2000	55	New
134	10.1	1.33	37.7	2000	60	New
135	9.1	1.33	37.1	2000	60	New
136	9.0	1.33	39.4	2000	64	New
137	12.1	1.33	39.2	2000	64	New
138	9.2	1.33	39.9	2000	64	New
139	121.5	1.32	7.7	4000	5	New
140	65.1	1.32	10.6	4000	10	New
141	57.4	1.32	16.2	4000	20	New
142	62.1	1.32	16.2	4000	20	New

Table 3: *(Reading benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Columns	Cores	Version
143	45.3	1.32	22.6	4000	30	New
144	47.0	1.32	19.5	4000	30	New
145	58.3	1.32	26.9	4000	40	New
146	73.1	1.32	27.4	4000	40	New
147	61.6	1.32	32.6	4000	50	New
148	72.1	1.32	31.5	4000	50	New
149	64.6	1.32	35.6	4000	55	New
150	69.9	1.32	35.4	4000	55	New
151	65.1	1.32	38.1	4000	60	New
152	71.6	1.32	38.5	4000	60	New
153	66.3	1.32	40.3	4000	64	New
154	68.4	1.32	40.1	4000	64	New
155	72.4	1.32	40.5	4000	64	New

2.4 Analyzing data

The next four tables show time and memory consumption when analyzing data. For the old version, the largest data sets analyzed are of 200 columns for the Intel Xeon machine with 64 GB of RAM, and 1000 columns for the AMD Opteron machines (see details in section 2.3). In these benchmarks, runs were not allowed to run for more than 36 hours (2160 minutes) except for a few cases that were allowed to run for longer to either compare between methods (e.g., HMM in Coleonyx) or to verify that the method is definitely not suitable for very large data, such as in the case of GLAD, where two processes were allowed to run for four days (see section 1.2.1). Finally, note that we do not compute the time it takes to set up the MPI environment (with the old version or with the new version, when using MPI), but only the time of the call for the segmentation itself; setting up the cluster takes about half a minute to a minute.

Some of the major patterns and results shown in tables 4 to 7 (see also Figure2) are:

Version comparison There are small speed differences between the old and new versions, generally favoring the new version, specially with HaarSeg and CBS. The new version generally also uses less memory than the old version. The main difference, however, is that the new version can analyze much larger data sets, as the old version is limited by the size of the data sets that can be read (see section 2.3).

Load balancing Load balancing is generally a good choice, but not with HaarSeg on a single multicore machine, because the individual analysis of HaarSeg are so fast that they rarely make it worth it the increased communication and processing overheads of load balancing.

MPI vs. forking Forking is faster than MPI when running on a single node, which is to be expected, and in some cases (e.g., HMM) the differences can be very large.

Running over several nodes Even with fast communication between nodes (as in this case) duplicating the number of cores might not result in significant decreases in wall time for the fastest methods. In particular, Wall time for HaarSeg is actually larger when run over two nodes. For CBS there is a slight advantage of running over two nodes. Running over more than one node to increase the number of cores/CPUs is, however, advantageous for the slower methods (e.g., HMM).

Note that these results are **highly hardware dependent**: slower communication between nodes or slower I/O from shared storage will make running over several nodes less worth it. However, increasing the available number of cores/CPUs by larger factors (e.g., 4x or 8x) might make it worth it to use them even for fast methods such as HaarSeg.

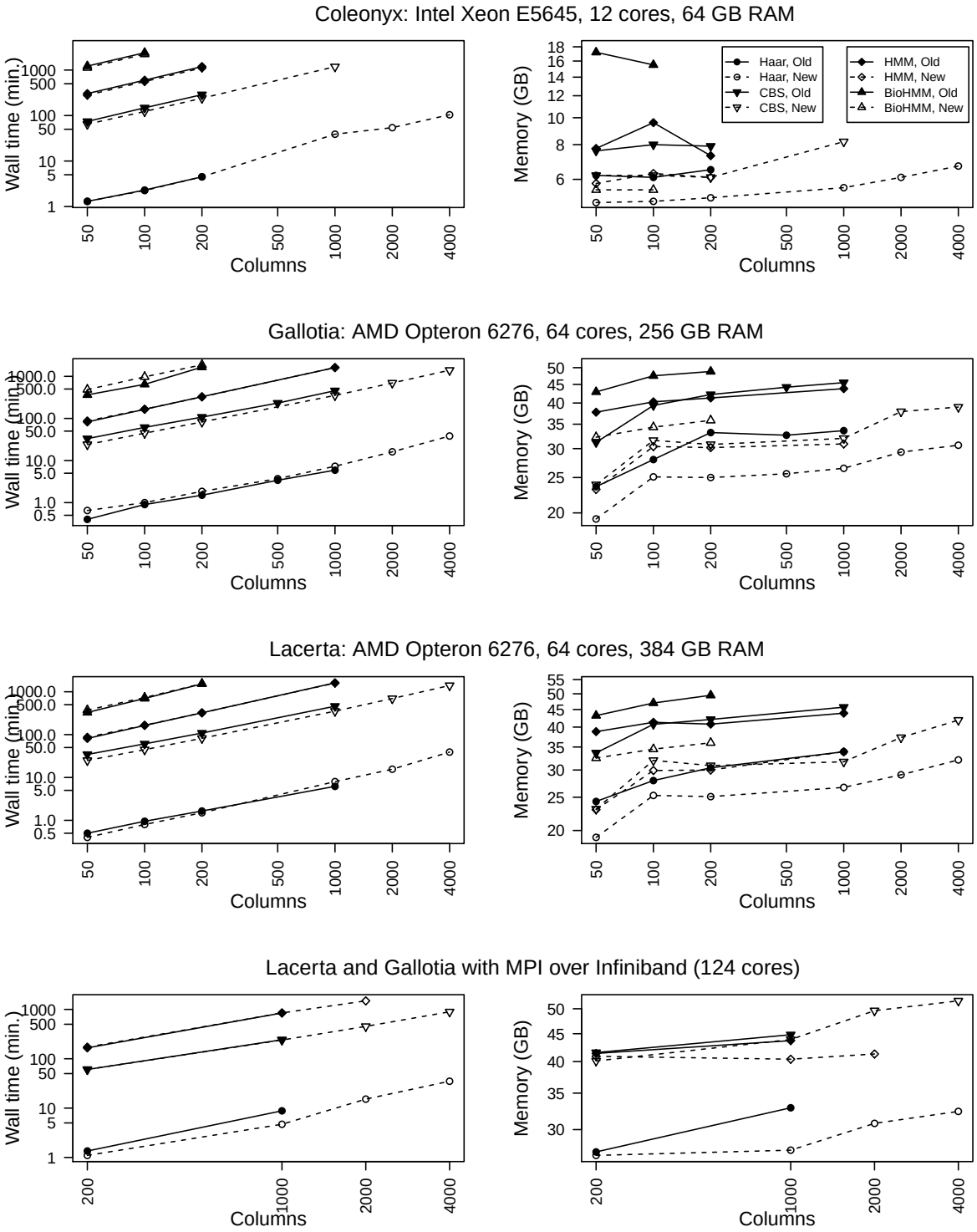

Figure 2: Comparison between old and new version in wall time and total memory usage (over all spawned processes) as a function of number of columns (or arrays or samples). Both axes shown in log scale. The figure shows the default use cases: using 12 cores in the Intel Xeon machine and 64 cores in the AMD Opterons. Since the old - version assumes no missing data, when possible (i.e., when data read by the old version are available) the data without missing values have been used with option `certain_noNA = TRUE`; these correspond to rows labeled "New-noNA" in tables 4 to 7. When more than one benchmark is available for a scenario, the median is shown.

Table 4: Analysis benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
1	2.0	0.13	5.3	Haar	50	Fork	12			New
2	1.8	0.13	5.2	Haar	50	Fork-LB	12			New
3	1.4	0.13	5.0	Haar	50	Fork	12			New-noNA
4	1.2	0.13	4.9	Haar	50	Fork	12			New-noNA
5	1.6	0.13	4.9	Haar	50	Fork-LB	12			New-noNA
6	1.4	0.16	5.8	Haar	50	MPI		12	11	Old
7	1.3	0.16	6.2	Haar	50	MPI		13	12	Old
8	1.2	0.16	6.4	Haar	50	MPI		13	12	Old
9	3.1	0.14	5.2	Haar	100	Fork	12			New
10	3.2	0.13	5.1	Haar	100	Fork-LB	12			New
11	2.8	0.13	3.2	Haar	100	MPI		12	11	New
12	3.1	0.13	5.2	Haar	100	MPI-LB		12	11	New
13	2.3	0.13	5.0	Haar	100	Fork	12			New-noNA
14	2.2	0.13	5.0	Haar	100	Fork	12			New-noNA
15	2.7	0.13	5.0	Haar	100	Fork-LB	12			New-noNA
16	2.6	0.17	5.9	Haar	100	MPI		12	11	Old
17	2.3	0.17	6.1	Haar	100	MPI		13	12	Old
18	2.3	0.17	6.1	Haar	100	MPI		13	12	Old
19	5.8	0.13	5.2	Haar	200	Fork	12			New
20	5.3	0.13	5.4	Haar	200	Fork-LB	12			New
21	7.9	0.13	5.3	Haar	200	MPI		12	11	New
22	5.7	0.13	5.6	Haar	200	MPI-LB		12	11	New
23	4.8	0.13	5.3	Haar	200	Fork	12			New-noNA
24	4.2	0.13	5.0	Haar	200	Fork	12			New-noNA
25	5.3	0.13	5.0	Haar	200	Fork-LB	12			New-noNA
26	6.7	0.13	5.2	Haar	200	MPI		12	11	New-noNA
27	5.2	0.13	5.1	Haar	200	MPI-LB		12	11	New-noNA
28	4.8	0.18	6.0	Haar	200	MPI		12	11	Old
29	4.5	0.18	6.5	Haar	200	MPI		13	12	Old

Table 4: *(Analysis benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
30	4.3	0.18	6.5	Haar	200	MPI		13	12	Old
31	38.9	0.14	5.6	Haar	1000	Fork	12			New
32	29.8	0.14	5.7	Haar	1000	Fork-LB	12			New
33	38.7	0.14	5.7	Haar	1000	MPI		12	11	New
34	29.8	0.14	10.0	Haar	1000	MPI-LB		12	11	New
35	54.2	0.15	6.1	Haar	2000	Fork	12			New
36	58.6	0.15	6.1	Haar	2000	Fork-LB	12			New
37	104.1	0.18	6.7	Haar	4000	Fork	12			New
38	117.3	0.18	6.3	Haar	4000	Fork-LB	12			New
39	69.7	0.13	7.5	CBS	50	Fork	12			New
40	64.5	0.13	7.1	CBS	50	Fork-LB	12			New
41	70.0	0.13	6.8	CBS	50	Fork	12			New-noNA
42	68.3	0.13	6.6	CBS	50	Fork	12			New-noNA
43	70.9	0.13	6.8	CBS	50	Fork	12			New-noNA
44	64.6	0.13	6.2	CBS	50	Fork-LB	12			New-noNA
45	74.2	0.13	7.6	CBS	50	MPI		12	11	Old
46	74.1	0.13	7.6	CBS	50	MPI		12	11	Old
47	77.3	0.13	8.1	CBS	50	MPI		13	12	Old
48	133.5	0.13	9.8	CBS	100	Fork	12			New
49	124.2	0.13	7.2	CBS	100	Fork-LB	12			New
50	132.4	0.13	8.4	CBS	100	Fork	12			New-noNA
51	131.2	0.13	7.2	CBS	100	Fork	12			New-noNA
52	122.7	0.13	6.2	CBS	100	Fork-LB	12			New-noNA
53	146.2	0.13	8.0	CBS	100	MPI		12	11	Old
54	147.9	0.13	7.7	CBS	100	MPI		12	11	Old
55	150.0	0.13	15.5	CBS	100	MPI		13	12	Old
56	250.8	0.13	8.2	CBS	200	Fork	12			New
57	241.4	0.13	7.5	CBS	200	Fork-LB	12			New
58	380.0	0.13	9.0	CBS	200	MPI		12	11	New

Table 4: *(Analysis benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
59	307.9	0.13	8.4	CBS	200	MPI-LB		12	11	New
60	255.2	0.13	14.2	CBS	200	Fork	12			New-noNA
61	250.7	0.13	7.3	CBS	200	Fork	12			New-noNA
62	240.3	0.13	6.1	CBS	200	Fork-LB	12			New-noNA
63	372.4	0.13	8.7	CBS	200	MPI		12	11	New-noNA
64	307.8	0.13	7.9	CBS	200	MPI-LB		12	11	New-noNA
65	287.7	0.13	7.9	CBS	200	MPI		12	11	Old
66	287.2	0.13	7.2	CBS	200	MPI		12	11	Old
67	294.6	0.13	10.6	CBS	200	MPI		13	12	Old
68	1246.0	0.14	14.8	CBS	1000	Fork	12			New
69	1185.2	0.14	8.2	CBS	1000	Fork-LB	12			New
70	1804.2	0.14	10.0	CBS	1000	MPI		12	11	New
71	1512.4	0.14	8.9	CBS	1000	MPI-LB		12	11	New
72	311.2	0.18	6.4	HMM	50	Fork	11			New
73	309.5	0.18	10.5	HMM	50	Fork	11			New
74	282.1	0.18	7.0	HMM	50	Fork	12			New
75	286.4	0.18	7.1	HMM	50	Fork	12			New
76	288.4	0.18	9.6	HMM	50	Fork	12			New
77	281.0	0.18	6.3	HMM	50	Fork-LB	12			New
78	310.9	0.18	9.1	HMM	50	Fork	12			New-noNA
79	284.0	0.18	11.5	HMM	50	Fork	12			New-noNA
80	280.1	0.18	5.8	HMM	50	Fork-LB	12			New-noNA
81	322.5	0.18	7.2	HMM	50	MPI		11	10	Old
82	324.4	0.18	7.2	HMM	50	MPI		11	10	Old
83	298.2	0.18	7.3	HMM	50	MPI		12	11	Old
84	302.2	0.18	10.2	HMM	50	MPI		12	11	Old
85	299.9	0.18	8.2	HMM	50	MPI		13	12	Old
86	305.8	0.18	9.8	HMM	50	MPI		13	12	Old
87	574.2	0.18	9.5	HMM	100	Fork	12			New

Table 4: *(Analysis benchmarks for Coleonyx: Intel Xeon E5645, 12 cores, 64 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
88	552.2	0.18	6.4	HMM	100	Fork-LB	12			New
89	560.4	0.18	11.3	HMM	100	Fork	12			New-noNA
90	559.1	0.18	6.3	HMM	100	Fork-LB	12			New-noNA
91	603.0	0.18	10.7	HMM	100	MPI		12	11	Old
92	595.4	0.18	8.5	HMM	100	MPI		13	12	Old
93	1117.5	0.18	6.1	HMM	200	Fork-LB	12			New
94	3188.2	0.20	8.7	HMM	200	MPI		12	11	New
95	1213.7	0.20	8.1	HMM	200	MPI-LB		12	11	New
96	1108.7	0.18	7.9	HMM	200	Fork	12			New-noNA
97	1122.8	0.18	13.5	HMM	200	Fork	12			New-noNA
98	1112.3	0.18	6.1	HMM	200	Fork-LB	12			New-noNA
99	3240.6	0.20	9.1	HMM	200	MPI		12	11	New-noNA
100	1205.0	0.20	8.4	HMM	200	MPI-LB		12	11	New-noNA
101	1192.3	0.20	7.3	HMM	200	MPI		12	11	Old
102	1169.4	0.18	10.5	BioHMM	50	Fork	12			New-noNA
103	1122.9	0.18	5.5	BioHMM	50	Fork-LB	12			New-noNA
104	1222.2	0.18	17.2	BioHMM	50	MPI		13	12	Old
105	2339.5	0.18	11.0	BioHMM	100	Fork	12			New-noNA
106	2235.4	0.18	5.5	BioHMM	100	Fork-LB	12			New-noNA
107	2401.8	0.18	15.5	BioHMM	100	MPI		13	12	Old

Table 5: Analysis benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
1	0.7	0.13	20.2	Haar	50	Fork	64			New
2	1.4	0.13	18.6	Haar	50	Fork-LB	64			New
3	0.6	0.13	19.2	Haar	50	Fork	64			New-noNA
4	0.7	0.13	19.3	Haar	50	Fork	64			New-noNA
5	0.6	0.13	19.2	Haar	50	Fork-LB	64			New-noNA
6	0.4	0.16	23.6	Haar	50	MPI		64	63	Old
7	1.2	0.13	25.7	Haar	100	Fork	64			New
8	1.9	0.13	23.6	Haar	100	Fork-LB	64			New
9	1.0	0.13	25.2	Haar	100	Fork	64			New-noNA
10	1.0	0.13	25.0	Haar	100	Fork	64			New-noNA
11	1.1	0.13	24.5	Haar	100	Fork-LB	64			New-noNA
12	0.8	0.17	28.1	Haar	100	MPI		64	63	Old
13	0.9	0.17	28.0	Haar	100	MPI		65	64	Old
14	0.9	0.17	27.8	Haar	100	MPI		65	64	Old
15	2.4	0.13	25.8	Haar	200	Fork	64			New
16	2.8	0.13	25.2	Haar	200	Fork-LB	64			New
17	2.6	0.14	28.3	Haar	200	MPI		64	63	New
18	1.9	0.14	28.1	Haar	200	MPI-LB		64	63	New
19	1.9	0.14	25.0	Haar	200	Fork	64			New-noNA
20	1.8	0.14	25.0	Haar	200	Fork	64			New-noNA
21	1.9	0.13	24.7	Haar	200	Fork-LB	64			New-noNA
22	2.7	0.14	27.1	Haar	200	MPI		64	63	New-noNA
23	1.8	0.14	26.7	Haar	200	MPI-LB		64	63	New-noNA
24	1.4	0.18	34.2	Haar	200	MPI		64	63	Old
25	1.6	0.18	33.2	Haar	200	MPI		65	64	Old
26	1.5	0.18	32.6	Haar	200	MPI		65	64	Old
27	3.7	0.13	27.0	Haar	500	Fork	64			New
28	3.7	0.14	25.6	Haar	500	Fork	64			New-noNA
29	3.8	0.18	31.3	Haar	500	MPI		64	63	Old

Table 5: *(Analysis benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
30	3.0	0.18	34.0	Haar	500	MPI		64	63	Old
31	7.0	0.14	27.7	Haar	1000	Fork	64			New
32	10.1	0.14	27.5	Haar	1000	Fork	64			New
33	13.0	0.14	26.4	Haar	1000	Fork-LB	64			New
34	10.2	0.14	27.4	Haar	1000	Fork-LB	64			New
35	12.4	0.14	29.3	Haar	1000	MPI		64	63	New
36	8.9	0.14	28.6	Haar	1000	MPI-LB		64	63	New
37	6.5	0.14	26.3	Haar	1000	Fork	64			New-noNA
38	8.9	0.14	26.6	Haar	1000	Fork	64			New-noNA
39	7.3	0.14	26.5	Haar	1000	Fork	64			New-noNA
40	7.6	0.14	25.9	Haar	1000	Fork-LB	64			New-noNA
41	8.7	0.14	25.8	Haar	1000	Fork-LB	64			New-noNA
42	6.2	0.14	28.7	Haar	1000	MPI		64	63	New-noNA
43	6.2	0.14	28.5	Haar	1000	MPI-LB		64	63	New-noNA
44	5.9	0.18	33.6	Haar	1000	MPI		64	63	Old
45	15.3	0.15	28.9	Haar	2000	Fork	64			New
46	16.9	0.15	29.8	Haar	2000	Fork	64			New
47	21.8	0.15	28.4	Haar	2000	Fork-LB	64			New
48	19.9	0.15	29.1	Haar	2000	Fork-LB	64			New
49	35.6	0.18	30.7	Haar	4000	Fork	64			New
50	40.8	0.18	30.6	Haar	4000	Fork	64			New
51	43.2	0.18	29.1	Haar	4000	Fork-LB	64			New
52	39.9	0.18	30.2	Haar	4000	Fork-LB	64			New
53	25.1	0.13	29.2	CBS	50	Fork	64			New
54	24.3	0.14	27.9	CBS	50	Fork-LB	64			New
55	25.1	0.13	23.8	CBS	50	Fork	64			New-noNA
56	24.7	0.13	23.8	CBS	50	Fork	64			New-noNA
57	24.0	0.13	23.9	CBS	50	Fork-LB	64			New-noNA
58	33.1	0.13	31.2	CBS	50	MPI		64	63	Old

Table 5: *(Analysis benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
59	47.7	0.13	37.1	CBS	100	Fork	64			New
60	45.1	0.14	36.3	CBS	100	Fork-LB	64			New
61	48.2	0.13	31.1	CBS	100	Fork	64			New-noNA
62	47.0	0.14	30.4	CBS	100	Fork	64			New-noNA
63	44.6	0.13	31.6	CBS	100	Fork-LB	64			New-noNA
64	60.3	0.13	38.8	CBS	100	MPI		64	63	Old
65	61.2	0.13	39.4	CBS	100	MPI		65	64	Old
66	61.6	0.13	39.7	CBS	100	MPI		65	64	Old
67	87.5	0.13	38.7	CBS	200	Fork	64			New
68	82.9	0.14	37.4	CBS	200	Fork-LB	64			New
69	134.6	0.14	44.2	CBS	200	MPI		64	63	New
70	104.1	0.14	44.3	CBS	200	MPI-LB		64	63	New
71	88.9	0.13	34.1	CBS	200	Fork	64			New-noNA
72	87.0	0.13	34.4	CBS	200	Fork	64			New-noNA
73	82.5	0.13	30.8	CBS	200	Fork-LB	64			New-noNA
74	135.0	0.14	41.6	CBS	200	MPI		64	63	New-noNA
75	104.7	0.14	41.9	CBS	200	MPI-LB		64	63	New-noNA
76	106.6	0.13	41.7	CBS	200	MPI		64	63	Old
77	108.7	0.13	43.0	CBS	200	MPI		65	64	Old
78	107.5	0.13	42.2	CBS	200	MPI		65	64	Old
79	184.6	0.13	40.7	CBS	500	Fork	64			New
80	182.6	0.14	36.8	CBS	500	Fork	64			New-noNA
81	231.6	0.14	44.2	CBS	500	MPI		64	63	Old
82	362.5	0.14	42.1	CBS	1000	Fork	64			New
83	355.1	0.14	36.3	CBS	1000	Fork-LB	64			New
84	356.6	0.14	36.8	CBS	1000	Fork-LB	64			New
85	563.2	0.14	50.9	CBS	1000	MPI		64	63	New
86	447.6	0.14	49.1	CBS	1000	MPI-LB		64	63	New
87	358.9	0.14	39.3	CBS	1000	Fork	64			New-noNA

Table 5: *(Analysis benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
88	352.1	0.14	32.0	CBS	1000	Fork-LB	64			New-noNA
89	446.7	0.14	45.6	CBS	1000	MPI-LB		64	63	New-noNA
90	455.5	0.14	45.5	CBS	1000	MPI		64	63	Old
91	722.7	0.16	43.8	CBS	2000	Fork	64			New
92	696.4	0.16	37.9	CBS	2000	Fork-LB	64			New
93	1499.6	0.18	44.2	CBS	4000	Fork	64			New
94	1396.8	0.18	39.0	CBS	4000	Fork-LB	64			New
95	87.6	0.18	25.8	HMM	50	Fork-LB	64			New
96	105.9	0.17	30.0	HMM	50	Fork	64			New-noNA
97	94.5	0.17	30.0	HMM	50	Fork	64			New-noNA
98	87.2	0.18	23.2	HMM	50	Fork-LB	64			New-noNA
99	83.0	0.18	37.9	HMM	50	MPI		64	63	Old
100	82.2	0.18	37.6	HMM	50	MPI		64	63	Old
101	175.7	0.18	33.6	HMM	100	Fork	63			New
102	174.2	0.18	33.4	HMM	100	Fork	63			New
103	165.4	0.18	34.5	HMM	100	Fork	64			New
104	166.3	0.18	34.4	HMM	100	Fork	64			New
105	179.2	0.18	32.8	HMM	100	Fork-LB	64			New
106	166.9	0.18	31.4	HMM	100	Fork	64			New-noNA
107	168.0	0.18	31.6	HMM	100	Fork	64			New-noNA
108	166.2	0.18	30.4	HMM	100	Fork-LB	64			New-noNA
109	164.0	0.18	38.9	HMM	100	MPI		63	62	Old
110	164.6	0.18	39.6	HMM	100	MPI		63	62	Old
111	160.9	0.18	40.7	HMM	100	MPI		64	63	Old
112	163.0	0.18	39.9	HMM	100	MPI		64	63	Old
113	168.8	0.18	41.1	HMM	100	MPI		65	64	Old
114	167.9	0.18	41.4	HMM	100	MPI		65	64	Old
115	333.8	0.18	34.4	HMM	200	Fork	63			New
116	336.5	0.18	34.8	HMM	200	Fork	63			New

Table 5: *(Analysis benchmarks for Gallotia: AMD Opteron 6276, 64 cores, 256 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
117	331.5	0.18	36.1	HMM	200	Fork	64			New
118	333.3	0.18	36.1	HMM	200	Fork	64			New
119	359.6	0.19	33.1	HMM	200	Fork-LB	64			New
120	1110.5	0.20	40.5	HMM	200	MPI		64	63	New
121	329.7	0.20	41.5	HMM	200	MPI-LB		64	63	New
122	328.7	0.18	33.5	HMM	200	Fork	64			New-noNA
123	328.9	0.18	33.6	HMM	200	Fork	64			New-noNA
124	325.1	0.18	30.2	HMM	200	Fork-LB	64			New-noNA
125	1109.4	0.20	43.1	HMM	200	MPI		64	63	New-noNA
126	326.9	0.20	42.8	HMM	200	MPI-LB		64	63	New-noNA
127	331.3	0.20	40.8	HMM	200	MPI		63	62	Old
128	330.2	0.20	41.1	HMM	200	MPI		63	62	Old
129	319.5	0.20	41.7	HMM	200	MPI		64	63	Old
130	325.8	0.20	41.5	HMM	200	MPI		64	63	Old
131	1616.9	0.21	40.7	HMM	1000	Fork	64			New-noNA
132	1618.0	0.22	30.9	HMM	1000	Fork-LB	64			New-noNA
133	1612.1	0.26	43.8	HMM	1000	MPI		64	63	Old
134	469.8	0.18	32.2	BioHMM	50	Fork	64			New-noNA
135	499.2	0.18	32.3	BioHMM	50	Fork	64			New-noNA
136	363.3	0.18	42.9	BioHMM	50	MPI		64	63	Old
137	964.7	0.18	34.5	BioHMM	100	Fork	64			New-noNA
138	979.8	0.18	34.2	BioHMM	100	Fork	64			New-noNA
139	645.5	0.18	47.5	BioHMM	100	MPI		64	63	Old
140	1813.8	0.18	36.1	BioHMM	200	Fork	64			New-noNA
141	1939.3	0.18	35.6	BioHMM	200	Fork	64			New-noNA
142	1652.7	0.20	48.8	BioHMM	200	MPI		64	63	Old

Table 6: Analysis benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
1	0.7	0.13	20.2	Haar	50	Fork	64			New
2	0.6	0.13	20.2	Haar	50	Fork-LB	64			New
3	0.4	0.13	19.1	Haar	50	Fork	64			New-noNA
4	0.6	0.13	19.2	Haar	50	Fork-LB	64			New-noNA
5	0.5	0.16	24.3	Haar	50	MPI		65	64	Old
6	0.5	0.16	24.3	Haar	50	MPI		65	64	Old
7	1.3	0.13	25.7	Haar	100	Fork	64			New
8	1.2	0.13	25.7	Haar	100	Fork-LB	64			New
9	0.8	0.13	25.3	Haar	100	Fork	64			New-noNA
10	1.1	0.13	24.5	Haar	100	Fork-LB	64			New-noNA
11	0.8	0.17	27.8	Haar	100	MPI		64	63	Old
12	1.1	0.17	28.1	Haar	100	MPI		65	64	Old
13	2.5	0.13	26.8	Haar	200	Fork	64			New
14	1.8	0.13	26.9	Haar	200	Fork-LB	64			New
15	2.6	0.13	28.2	Haar	200	MPI		64	63	New
16	2.1	0.13	28.1	Haar	200	MPI-LB		64	63	New
17	1.5	0.14	25.1	Haar	200	Fork	64			New-noNA
18	2.0	0.13	24.6	Haar	200	Fork-LB	64			New-noNA
19	2.4	0.13	27.1	Haar	200	MPI		64	63	New-noNA
20	1.5	0.13	27.1	Haar	200	MPI-LB		64	63	New-noNA
21	1.4	0.18	31.9	Haar	200	MPI		64	63	Old
22	1.9	0.18	29.0	Haar	200	MPI		65	64	Old
23	10.5	0.14	27.9	Haar	1000	Fork	64			New
24	8.1	0.14	27.7	Haar	1000	Fork	64			New
25	11.4	0.14	26.7	Haar	1000	Fork-LB	64			New
26	10.5	0.14	28.2	Haar	1000	Fork-LB	64			New
27	11.6	0.14	29.8	Haar	1000	MPI		64	63	New
28	10.0	0.14	29.2	Haar	1000	MPI-LB		64	63	New
29	8.0	0.14	26.3	Haar	1000	Fork	64			New-noNA

Table 6: *(Analysis benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
30	8.8	0.14	26.7	Haar	1000	Fork	64			New-noNA
31	7.1	0.14	26.7	Haar	1000	Fork	64			New-noNA
32	9.2	0.14	26.5	Haar	1000	Fork-LB	64			New-noNA
33	7.0	0.14	26.2	Haar	1000	Fork-LB	64			New-noNA
34	6.1	0.14	28.5	Haar	1000	MPI		64	63	New-noNA
35	6.1	0.14	28.5	Haar	1000	MPI-LB		64	63	New-noNA
36	6.2	0.18	32.9	Haar	1000	MPI		65	64	Old
37	6.2	0.18	34.9	Haar	1000	MPI		65	64	Old
38	14.8	0.14	29.0	Haar	2000	Fork	64			New
39	16.4	0.14	29.1	Haar	2000	Fork	64			New
40	21.7	0.14	27.1	Haar	2000	Fork-LB	64			New
41	20.2	0.14	28.4	Haar	2000	Fork-LB	64			New
42	36.1	0.18	32.3	Haar	4000	Fork	64			New
43	42.2	0.18	31.9	Haar	4000	Fork	64			New
44	45.0	0.18	31.8	Haar	4000	Fork-LB	64			New
45	44.9	0.18	31.7	Haar	4000	Fork-LB	64			New
46	25.5	0.13	26.2	CBS	50	Fork-LB	64			New
47	24.6	0.13	24.4	CBS	50	Fork	64			New-noNA
48	25.0	0.14	23.1	CBS	50	Fork-LB	64			New-noNA
49	34.0	0.13	35.0	CBS	50	MPI		65	64	Old
50	34.9	0.13	32.3	CBS	50	MPI		65	64	Old
51	44.7	0.13	34.0	CBS	100	Fork-LB	64			New
52	46.2	0.13	31.1	CBS	100	Fork	64			New-noNA
53	44.5	0.14	32.0	CBS	100	Fork-LB	64			New-noNA
54	61.0	0.13	40.5	CBS	100	MPI		64	63	Old
55	60.6	0.13	41.0	CBS	100	MPI		65	64	Old
56	83.6	0.13	35.7	CBS	200	Fork-LB	64			New
57	131.5	0.13	46.1	CBS	200	MPI		64	63	New
58	103.3	0.13	43.9	CBS	200	MPI-LB		64	63	New

Table 6: *(Analysis benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
59	88.1	0.14	34.1	CBS	200	Fork	64			New-noNA
60	82.1	0.14	30.9	CBS	200	Fork-LB	64			New-noNA
61	134.9	0.13	43.1	CBS	200	MPI		64	63	New-noNA
62	102.8	0.13	42.0	CBS	200	MPI-LB		64	63	New-noNA
63	107.8	0.13	42.5	CBS	200	MPI		64	63	Old
64	109.7	0.13	41.7	CBS	200	MPI		65	64	Old
65	353.7	0.14	36.7	CBS	1000	Fork-LB	64			New
66	353.4	0.14	36.4	CBS	1000	Fork-LB	64			New
67	556.8	0.14	51.0	CBS	1000	MPI		64	63	New
68	451.2	0.14	47.9	CBS	1000	MPI-LB		64	63	New
69	359.9	0.14	38.8	CBS	1000	Fork	64			New-noNA
70	352.3	0.14	31.7	CBS	1000	Fork-LB	64			New-noNA
71	443.2	0.14	45.0	CBS	1000	MPI-LB		64	63	New-noNA
72	459.6	0.14	45.7	CBS	1000	MPI		65	64	Old
73	722.2	0.15	44.1	CBS	2000	Fork	64			New
74	695.2	0.16	37.3	CBS	2000	Fork-LB	64			New
75	1430.5	0.18	46.4	CBS	4000	Fork	64			New
76	1399.4	0.18	41.9	CBS	4000	Fork-LB	64			New
77	94.5	0.18	25.8	HMM	50	Fork	64			New
78	88.0	0.18	25.7	HMM	50	Fork-LB	64			New
79	94.0	0.17	29.9	HMM	50	Fork	64			New-noNA
80	86.8	0.18	23.0	HMM	50	Fork-LB	64			New-noNA
81	81.3	0.18	38.8	HMM	50	MPI		64	63	Old
82	166.1	0.18	32.5	HMM	100	Fork-LB	64			New
83	167.1	0.18	31.7	HMM	100	Fork	64			New-noNA
84	165.6	0.18	29.9	HMM	100	Fork-LB	64			New-noNA
85	160.3	0.18	41.6	HMM	100	MPI		64	63	Old
86	166.5	0.18	41.0	HMM	100	MPI		65	64	Old
87	324.9	0.18	32.9	HMM	200	Fork-LB	64			New

Table 6: *(Analysis benchmarks for Lacerta: AMD Opteron 6276, 64 cores, 384 GB RAM, continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
88	1113.2	0.20	41.6	HMM	200	MPI		64	63	New
89	326.7	0.20	40.3	HMM	200	MPI-LB		64	63	New
90	323.8	0.18	33.0	HMM	200	Fork	64			New-noNA
91	323.1	0.19	30.0	HMM	200	Fork-LB	64			New-noNA
92	1123.3	0.20	42.7	HMM	200	MPI		64	63	New-noNA
93	324.5	0.20	42.7	HMM	200	MPI-LB		64	63	New-noNA
94	322.3	0.20	40.8	HMM	200	MPI		64	63	Old
95	1612.9	0.21	40.0	HMM	1000	Fork	64			New-noNA
96	1597.5	0.22	33.9	HMM	1000	Fork-LB	64			New-noNA
97	1613.6	0.27	43.9	HMM	1000	MPI		64	63	Old
98	458.4	0.18	28.3	BioHMM	50	Fork	64			New
99	373.7	0.17	32.5	BioHMM	50	Fork	64			New-noNA
100	328.9	0.18	43.2	BioHMM	50	MPI		64	63	Old
101	731.2	0.18	34.5	BioHMM	100	Fork	64			New-noNA
102	697.5	0.18	47.0	BioHMM	100	MPI		64	63	Old
103	1543.5	0.18	36.0	BioHMM	200	Fork	64			New-noNA
104	1525.2	0.20	49.5	BioHMM	200	MPI		64	63	Old

Table 7: Analysis benchmarks using Lacerta and Gallotia with MPI over Infiniband (total 124 cores)

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
1	1.4	0.13	27.3	Haar	200	MPI		63	124	New-noNA
2	1.3	0.13	26.9	Haar	200	MPI		62	124	New-noNA
3	1.3	0.13	27.1	Haar	200	MPI		62	124	New-noNA
4	1.3	0.13	27.3	Haar	200	MPI		63	124	New-noNA
5	1.1	0.13	27.2	Haar	200	MPI-LB		63	124	New-noNA
6	1.1	0.13	26.7	Haar	200	MPI-LB		62	124	New-noNA
7	1.0	0.13	26.7	Haar	200	MPI-LB		62	124	New-noNA
8	1.2	0.13	27.1	Haar	200	MPI-LB		63	124	New-noNA
9	1.4	0.18	27.4	Haar	200	MPI		63	124	Old
10	1.3	0.18	27.2	Haar	200	MPI		62	124	Old
11	7.9	0.14	28.5	Haar	1000	MPI		62	124	New-noNA
12	4.7	0.14	27.5	Haar	1000	MPI-LB		62	124	New-noNA
13	8.8	0.18	32.9	Haar	1000	MPI		62	124	Old
14	27.7	0.15	31.6	Haar	2000	MPI		62	124	New
15	15.2	0.15	30.8	Haar	2000	MPI-LB		62	124	New
16	59.0	0.18	31.8	Haar	4000	MPI		62	124	New
17	35.0	0.18	32.4	Haar	4000	MPI-LB		62	124	New
18	70.1	0.13	42.1	CBS	200	MPI		63	124	New-noNA
19	69.6	0.13	41.4	CBS	200	MPI		62	124	New-noNA
20	60.6	0.13	40.1	CBS	200	MPI-LB		62	124	New-noNA
21	60.0	0.13	42.0	CBS	200	MPI		63	124	Old
22	61.2	0.13	41.1	CBS	200	MPI		62	124	Old
23	324.1	0.14	44.7	CBS	1000	MPI		62	124	New-noNA
24	239.1	0.14	43.9	CBS	1000	MPI-LB		62	124	New-noNA
25	242.5	0.14	44.8	CBS	1000	MPI		62	124	Old
26	612.7	0.15	51.5	CBS	2000	MPI		62	124	New
27	452.6	0.15	49.6	CBS	2000	MPI-LB		62	124	New
28	1237.0	0.18	53.8	CBS	4000	MPI		62	124	New
29	893.3	0.18	51.7	CBS	4000	MPI-LB		62	124	New

Table 7: *(Analysis benchmarks using Lacerta and Gallotia with MPI over Infiniband (total 124 cores), continued)*

	Wall time (min.)	Memory (GB)	Σ Memory (GB)	Method	Columns	MPI/Fork	Cores	Procs. per node	Universe size	Version
30	584.2	0.20	41.8	HMM	200	MPI		62	124	New-noNA
31	172.1	0.20	40.9	HMM	200	MPI-LB		62	124	New-noNA
32	165.7	0.20	41.8	HMM	200	MPI		63	124	Old
33	166.4	0.20	41.0	HMM	200	MPI		62	124	Old
34	847.6	0.27	40.4	HMM	1000	MPI-LB		62	124	New-noNA
35	846.4	0.27	43.7	HMM	1000	MPI		62	124	Old
36	1489.2	0.35	41.3	HMM	2000	MPI-LB		62	124	New

3 Other comparisons

3.1 Comparison with non-parallelized executions

ADaCGH2 was run without merging, to compare it to the canonical, non-parallelized, implementations of CBS and Haar. Note that, as 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.

Table 8: 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, except for the last two, run on the Intel machine (on the Intel machine non-parallelized runs with 1000 columns cannot be attempted as R runs out of memory loading the data).

	Method	MPI /Fork	Cores	ff/ RAM	Columns	Wall time (min.)	Memory (GB)	Σ Memory (GB)
1	Haar	Fork	64	ff	100	1.2	0.13	24.5
2	Haar	Fork	10	ff	1000	23.5	0.142	5.9
3	Haar	Fork	20	ff	1000	12.3	0.137	10.0
4	Haar	Fork	40	ff	1000	7.4	0.142	17.6
5	Haar	Fork	50	ff	1000	6.7	0.139	21.2
6	Haar	Fork	64	ff	1000	6.4	0.142	26.9
7	Haar	Fork	10	ff	2000	49.7	0.16	8.0
8	Haar	Fork	20	ff	2000	26.3	0.16	11.9
9	Haar	Fork	40	ff	2000	15.4	0.16	19.5
10	Haar	Fork	50	ff	2000	13.3	0.16	23.2
11	Haar	Fork	64	ff	2000	11.9	0.16	28.4
12	CBS	Fork	64	ff	100	55.9	0.13	35.3
13	CBS	Fork	10	ff	1000	1855.4	0.135	8.6
14	CBS	Fork	20	ff	1000	939.8	0.135	14.9
15	CBS	Fork	40	ff	1000	513.5	0.136	27.0
16	CBS	Fork	50	ff	1000	438.6	0.142	33.1
17	CBS	Fork	64	ff	1000	350.3	0.142	41.3
18	CBS	Fork	10	ff	2000	3770.9	0.15	11.1
19	CBS	Fork	20	ff	2000	1878.8	0.16	16.5
20	CBS	Fork	40	ff	2000	1007.0	0.15	28.8
21	CBS	Fork	50	ff	2000	857.1	0.16	35.3
22	CBS	Fork	64	ff	2000	717.3	0.163	41.9
23	Haar	NP	-	RAM	100	25.2[a]	12.5	12.5
24	CBS	NP	-	RAM	100	1706[b]	40	40
25	Haar	NP	-	RAM	1000	198.3[c]	***Cannot allocate memory***	
26	CBS	NP	-	RAM	1000	***Cannot allocate memory***		
27	Haar	NP	-	RAM	100	15.1[d]	14.1	14.1
28	CBS	NP	-	RAM	100	1112[e]	38.3	38.3

[a] 25.25 = 0.95 + 22.9 + 1.4: load data, analyze, and save results. 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.

[b] 1706 = 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).

[c] The analysis uses 113 GB, but results cannot be saved. This was in the machine with 256 GB of RAM.

[d] 15.1 = 0.65 + 13.5 + 0.9: load data, analyze, and save results.

[e] 1112 = 0.65 + 1106 + 4.9: load data, analyze, and save results. The analysis involves calling the `CNA` function to create the CNA object (0.95 min), calling the `smooth.CNA` function to smooth the data and detect outlier (20.2 minutes), and segmenting the data with the `segment` function (1085 minutes).

3.2 Reading from a directory of files vs. other options

Here we show time and memory usage of options that are not the recommended approach with large data set (using *ff* objects and reading from a directory of single-column files). All these benchmarks have been carried out in the AMD Opteron machines. These data show the patterns discussed in the main vignette: with large data sets the best approach is to read from a directory of single-column files and store as *ff* objects. Wall time is much smaller when reading from a directory of single column files (see also table 1 for a comparison with former versions of ADaCGH2, where original data where stored as RData and then read to *ff* objects). Moreover, storing as a RAM object, even when possible, might result in a RAM object that can then not be successfully used for analysis (see section 3.3).

Table 9: Time and memory usage when reading data

	Reading operation	Columns	Wall time (min)	Memory (GB)	Σ Memory (GB)
1	Txt file to *ff*	1000	2630	1.3	NA
2	RData to *ff*	1000	29.6	169	168.3
3	Directory to data frame (RAM object)	1000	22 + 2[a]	96	NA. Output unusable for analysis. See table 3.3.
4	RData to data frame (RAM object)	1000	22 + 2[b]	139	NA. Output unusable for analysis. See table 3.3.
5	Directory to data frame (RAM object)	200	7.7 + 0.4[c]	20	38
6	Directory to data frame (RAM object)	100	8.7 + 0.3[d]	10.5	30
7	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.

3.3 Analyzing large data with RAM objects

Here we present some results from attempting to analyze large data sets, with the new version of ADaCGH2, using RAM objects. Even moderately size data sets (200 columns) cannot be analyzed when using RAM objects in a machine with 384 GB of RAM; memory usage is already very large (140 GB) with just 50 columns. Time of analysis is also much larger for the case shown than for similarly sized problems when using *ff* objects (see tables 5 and 6).

Table 10: Time and memory usage of segmentation with default options

	Method	MPI /Fork	Cores	ff/RAM	Columns	Wall time (min.)	Memory (GB)	Σ Memory (GB)
1	Haar	Fork	64	RAM	50	0.7 + 2.5 + 0.9[a]	14.4	140
2	Haar	Fork	64	RAM	200	NA	NA	**Cannot allocate memory**
3	Haar	Fork	64	RAM	1000	NA	NA	**Cannot allocate memory**

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

4 Comments and recommended usage patterns

4.1 Recommended usage: summary

1. For data analysis, use the defaults when running on a single multicore computer:
   (a) *ff* objects for input and output.
   (b) Forking (instead of explicit clusters).
   (c) Load balancing, except when using HaarSeg.
2. If you have multiple machines available for analysis, try using them with explicit clusters (e.g., MPI). However, especially for methods such as HaarSeg, the gains could be modest unless you add many machines to the cluster. Use load balancing for all methods (i.e., override the defult if using HaarSeg).
3. When reading data, the fastest and least memory consuming is using a directory of single-column files. The best number of cores is likely to be strongly hardware (and possibly file system) dependent. The default `mc.cores` has been set to half the number of cores, but this is not necessarily a sensible default.

4.2 Recommended usage: details

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 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 an array of disks that 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 HMM and is borderline 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, number of spindles, quality of the RAID card, amount of free space, etc, etc, etc, can have an effect on all heavy I/O operations. 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.

   `inputToADaCGH` accepts an argument to reduce the number of cores used, which can help with contention issues related to I/O. A multicore machine (say, 12 cores) with a single SATA drive might actually complete the reading faster if you use fewer than 12 cores for the reading. But your mileage might vary. See also comments and full tables in section 2.3.

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. (2008). A fast and flexible method for the segmentation of aCGH data. *Bioinformatics (Oxford, England)*, 24(16):i139–i145.

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

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.

Fridlyand, J. and Dimitrov, P. (2010). *aCGH: Classes and functions for Array Comparative Genomic Hybridization data.* R package version 1.41.2.

Fridlyand, J., Snijders, A. M., Pinkel, D., and Albertson, D. G. (2004). Hidden Markov models approach to the analysis of array CGH data. . *Journal of Multivariate Analysis*, 90:132–153.

Hsu, L., Self, S. G., Grove, D., Randolph, T., Wang, K., Delrow, J. J., Loo, L., and Porter, P. (2005). Denoising array-based comparative genomic hybridization data using wavelets. *Biostatistics*, 6(2):211–226.

Huber, W., Toedling, J., and Steinmetz, L. M. (2006). Transcript mapping with high-density oligonucleotide tiling arrays. *Bioinformatics*, 22:1963–1970.

Hupe, P. (2011). *GLAD: Gain and Loss Analysis of DNA*. R package version 2.27.0.

Hupe, P., Stransky, N., Thiery, J. P., Radvanyi, F., and Barillot, E. (2004). Analysis of array cgh data: from signal ratio to gain and loss of dna regions. *Bioinformatics*, pages 3413–3422.

Marioni, J. C., Thorne, N. P., and Tavaré, S. (2006). BioHMM: a heterogeneous hidden Markov model for segmenting array CGH data. *Bioinformatics*, 22(9):1144–1146.

Picard, F., Robin, S., Lavielle, M., Vaisse, C., and Daudin, J. J. (2005). A statistical approach for array CGH data analysis. *BMC Bioinformatics*, 6.

Smith, M. L., Marioni, J. C., McKinney, S., Hardcastle, T., and Thorne, N. P. (2009). *snapCGH: Segmentation, normalisation and processing of aCGH data.* R package version 1.33.0.

Venkatraman, E. S. and Olshen, A. B. (2007). A faster circular binary segmentation algorithm for the analysis of array CGH data. *Bioinformatics (Oxford, England)*, 23(6):657–663.