

Scalable BFAST: R package optimizations and array-based data management

March 8, 2017

Marius Appel
marius.appel@uni-muenster.de



Part I: BFAST R package optimizations

Part II: Scalable EO data management with SciDB

Part III: Hands-on with SciDB, Landsat, and BFAST

1. SciDB installation (with Docker)
2. Data ingestion
3. Analysis (practical part)

Online Material

Slides, reports, and tutorial available at:

- <https://github.com/appelmar/scalbf-wur/>
- <https://appelmar.github.io/scalbf-wur/>

Overview

- Change detection and monitoring are computationally intensive operations:
 - Example from bfastSpatial package with the TURA dataset [1,2] (148 x 143 pixels and 166 images, approx. 4x4 km, 7 MB) takes around 20 mins on this computer
- Change detection / monitoring on national scale?
- Optimize R package and enable scalable distributed processing of large areas

[1] DeVries, B., Verbesselt, J., Kooistra, L., & Herold, M. (2015). Robust monitoring of small-scale forest disturbances in a tropical montane forest using Landsat time series. *Remote Sensing of Environment*, 161, 107-121.

[2] Dutrieux, L. & DeVries, B. (2014), bfastSpatial: Set of utilities and wrappers to perform change detection on satellite image time-series.

BFAST R package

`bfast()`

Main input:

- time series of response variable
- seasonal model
- **type of fluctuation process**
- maximum number of breaks / minimum segment size between breaks

Main output:

- time, number, and magnitude of changes

`bfastmonitor()`

Main input:

- time series of response variable (and optional regressors)
- start of the monitoring period
- model formula
- **type of monitoring process:** OLS-CUSUM, OLS-MOSUM, RE, ME
- **model for identifying a stable history** (Reverse CUSUM, Bai and Perron, all, ...)

Main output:

- change magnitude, change date (if any)

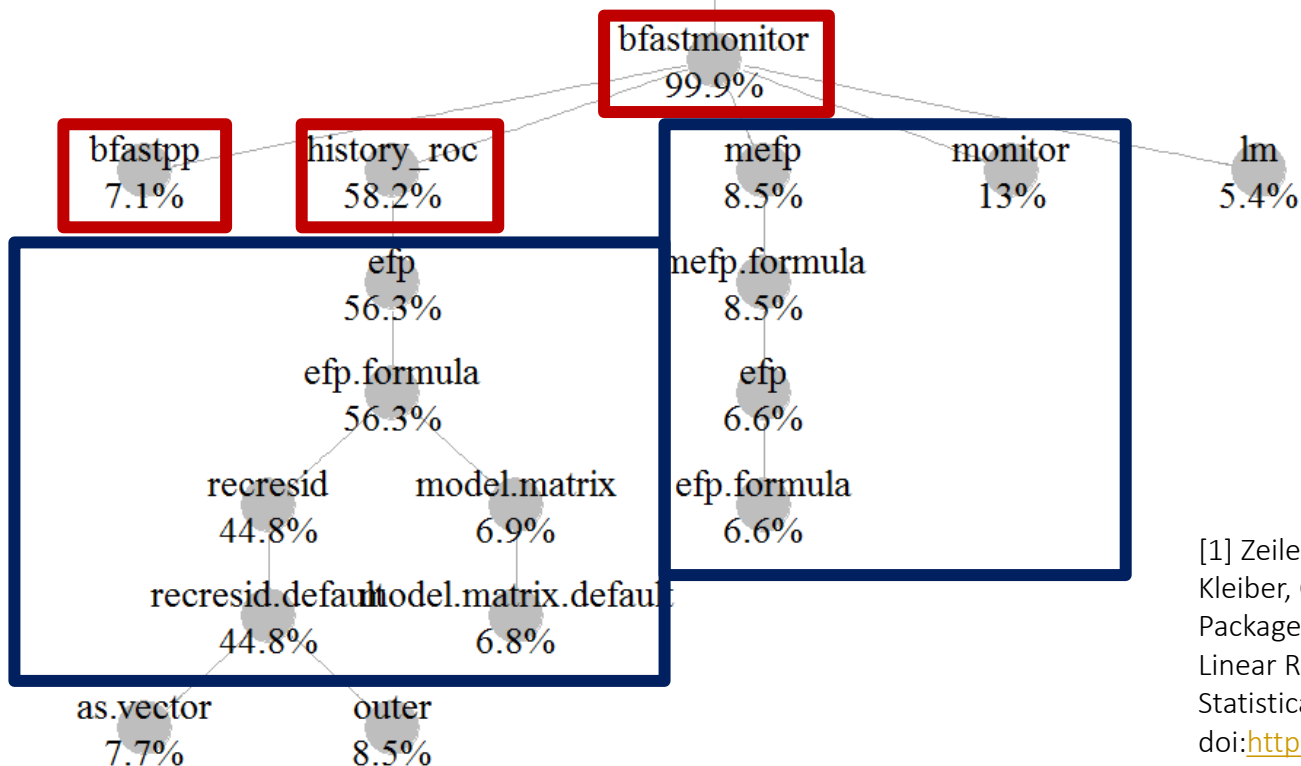
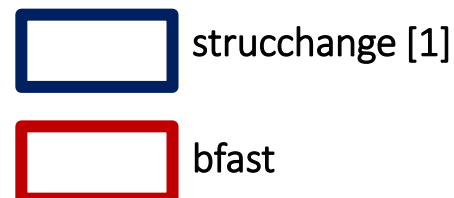
Identifying computational bottlenecks with profiling

- Profiling = executing R expressions and regularly checking which function is currently being evaluated (e.g. every 5 ms)
- Counting how often the execution is within a specific function allows to estimate how much of the overall time that function takes
- Profiling in R: Rprof, lineprof, profr

Profiling: example for bfastmonitor()

Example: profiling result of running bfastmonitor with ROC history model. Functions with relative execution time < 5% are omitted.

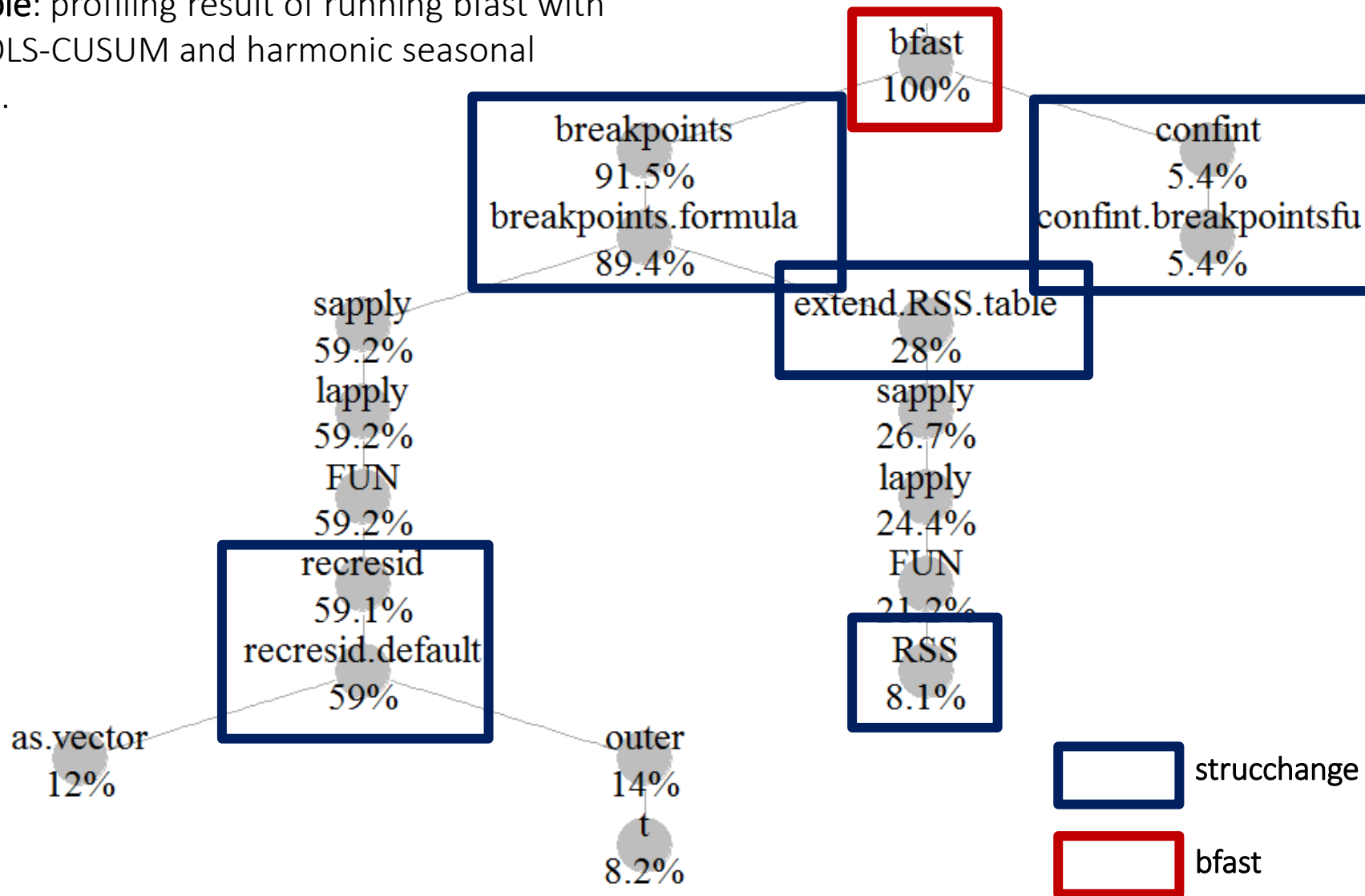
replicate
100%
sapply
100%
lapply
100%
FUN
99.9%



[1] Zeileis, A., Leisch, F., Hornik, K., & Kleiber, C. (2002). strucchange: An R Package for Testing for Structural Change in Linear Regression Models. Journal of Statistical Software, 7(2), 1 - 38.
doi:<http://dx.doi.org/10.18637/jss.v007.i02>

Profiling: example bfast()

Example: profiling result of running bfast with type OLS-CUSUM and harmonic seasonal model.



Package optimizations (overview)

- Move operations to C++ with Rcpp and RcppArmadillo [1,2] if possible
 - `recresid`, `extend_rss_table`, `efp_process_me`, `efp_process_re`
- Use design matrix and response vector instead of `data.frame` and formula and `lm.fit()` instead of `lm()`
 - `bfastpp`, `bfastmonitor`, `bfast`
 - `breakpoints`, `efp`, `mefp`, `monitor`,
- `bfastts`: avoid `as.ts(zoo(...))`

[1] Conrad Sanderson and Ryan Curtin. *Armadillo: a template-based C++ library for linear algebra*. Journal of Open Source Software, Vol. 1, pp. 26, 2016.

[2] Dirk Eddelbuettel, Conrad Sanderson (2014). RcppArmadillo: Accelerating R with high-performance C++ linear algebra. Computational Statistics and Data Analysis, Volume 71, March 2014, pages 1054-1063. URL <http://dx.doi.org/10.1016/j.csda.2013.02.005>

Moving computations to C++: recursive residuals

Recursive residuals in C++ with RcppArmadillo

```
arma::vec sc_cpp_recesid_arma(const mat& X, const vec& y, unsigned int
start, unsigned int end, const double& tol, const double& rcond_min) {
  if(!start > X.n_cols || !end <= X.n_rows) stop("Invalid end");
  if(!end >= start || !start <= X.n_cols || !end <= X.n_rows) stop("Invalid end");
  --start;
  --end;
  int n=end;
  int q=start-1;
  int k = X.n_cols;
  vec rval = vec(n-q, fill::zeros);
  vec cur_y = y.subvec(0, q);
  mat cur_X = X.submat(0, 0, q, k-1);
  mat X1;
  colvec cur_coef_full;
  solve(cur_coef_full, cur_X, cur_y, solve_opts::no_approx);
  inv_sympd(X1, trans(cur_X) * cur_X);
  colvec cur_coef = cur_coef_full;
  mat xr = X.row(q+1);
  vec fr = (1 + xr * X1 * trans(xr));
  rval(0) = as_scalar((y(q+1) - xr * cur_coef)/sqrt(fr));
  bool check = true;
  if((q+1) < n)
  {
    for(int r=q+2; r<=n; ++r) {
      X1 -= (X1 * trans(xr) * xr * X1)/as_scalar(fr);
      cur_coef += X1 * trans(xr) * rval(r-q-2) * sqrt(fr);
      if (check) {
        cur_y = y.subvec(0, r-1);
        cur_X = X.submat(0, 0, r-1, k-1);

        solve(cur_coef_full, cur_X, cur_y, solve_opts::no_approx);
        inv_sympd(X1, trans(cur_X) * cur_X);
        bool nona = is_finite( cur_coef) && is_finite( cur_coef_full );
        if(nona && approx_equal(cur_coef_full,cur_coef, "absdiff", tol)) {
          check = false;
        }
        cur_coef = cur_coef_full;
      }
      xr = X.row(r); // This a a row
      fr = (1 + xr * X1 * trans(xr));
      rval(r-q-1) = as_scalar((y(r) - xr * cur_coef)/sqrt(fr));
    }
  }
  return rval;
}
```

vec sc_cpp_recesid_arma(const mat& X, const vec& y, ...)

cur_coef += X1 * trans(xr) * rval(r-q-2) * sqrt(fr);

Recursive residuals in R

```
recesid.default <- function(x, y, start = ncol(x) + 1, end = nrow(x),
tol = sqrt(.Machine$double.eps)/ncol(x), ...)
{
  # checks and data dimensions
  if(start > ncol(x) & start <= nrow(x))
  stopifnot(end >= start & end <= nrow(x))
  n <- end
  q <- start - 1
  k <- ncol(x)
  rval <- rep(0, n - q)
  y1 <- y[1:q]
  fr <- lm.fit(x[1:(q+1), , drop = FALSE], y1)
  X1 <- .Xinv0(fm)
  betar <- .coef0(fm)
  xr <- as.vector(x[q+1,])
  fr <- as.vector((1 + (t(xr) %*% X1 %*% xr)))
  rval[1] <- (y[q+1] - t(xr) %*% betar)/sqrt(fr)
  check <- TRUE
  if((q+1) < n)
  {
    for(r in (q+2):n)
    {
      nona <- !is.na(fm$coefficients)
      X1 <- X1 - (X1 %*% outer(xr, xr) %*% X1)/fr
      betar <- betar + X1 %*% xr * rval[r-q-1] * sqrt(fr)
      if(check) {
        y1 <- y[1:(r-1)]
        fm <- lm.fit(x[1:(r-1), , drop = FALSE], y1)
        nona <- nona & all(!is.na(betar)) & all(!is.na(fm$coefficients))
        if(nona && isTRUE(all.equal(as.vector(fm$coefficients), as.vector(betar),
tol = tol))) {
          check <- FALSE
        }
      }
      X1 <- .Xinv0(fm)
      betar <- .coef0(fm)
    }
    xr <- as.vector(x[r,])
    fr <- as.vector((1 + (t(xr) %*% X1 %*% xr)))
    rval[r-q] <- (y[r] - sum(xr * betar, na.rm = TRUE))/sqrt(fr)
  }
  return(rval)
}
```

betar <- betar + X1 %*% xr * rval[r-q-1] * sqrt(fr)

Moving computations to C++: recursive residuals

- fits linear models in the first few iterations, until recursive model parameter updates are stable
- for ill-conditioned systems (e.g. dummy variables), R's `lm.fit` is used, otherwise Armadillo's `solve`
- speedup varies with length of time series, number of explanatory variables, and stability of the system, mostly around 15-20 on this machine
- changes have effect on performance of both `bfast` and `bfastmonitor` (unless `history = „all“`)

bfastpp

```
bfastpp(data, order = 3,  
  lag = NULL, slag = NULL, na.action = na.omit,  
  stl = c("none", "trend", "seasonal", "both"),  
  formula = NULL)
```

- Derives variables that occur in the formula only
- Instead of a data.frame, modified output is a list with elements
 - `y`: response vector
 - `X`: design matrix
 - `t`: vector of dates (same as `time(data)`)

Avoiding formulas and data.frames

S3 methods for matrix input:

- `breakpoints`, `efp`, `mefp`, `monitor`

```
breakpoints.formula(formula, h = 0.15, breaks = NULL,  
  data = list(), hpc = c("none", "foreach"), ...)
```



```
breakpoints.matrix(x, y, h = 0.15, breaks = NULL,  
  hpc = c("none", "foreach"), ...)
```

Matrix methods

- avoid `model.frame()` and `model.matrix()`
- use `lm.fit()` instead of `lm()`

bfasts modifications

- `bfasts()` not directly called from `bfast()` or `bfastmonitor()` but often used before (e.g. in `bfmSpatial()`)
- takes up to 30% of computation time of `bfastSpatial` example with the TURA dataset
- modification does not use `zoo`
- around 2 -3 times faster

Using package modifications

1. Install strucchange and bfast from github (Rtools needed for Windows)

```
library(devtools)
install_github("appelmar/strucchange")
install_github("appelmar/bfast")
```

2. Load the package and enable modifications

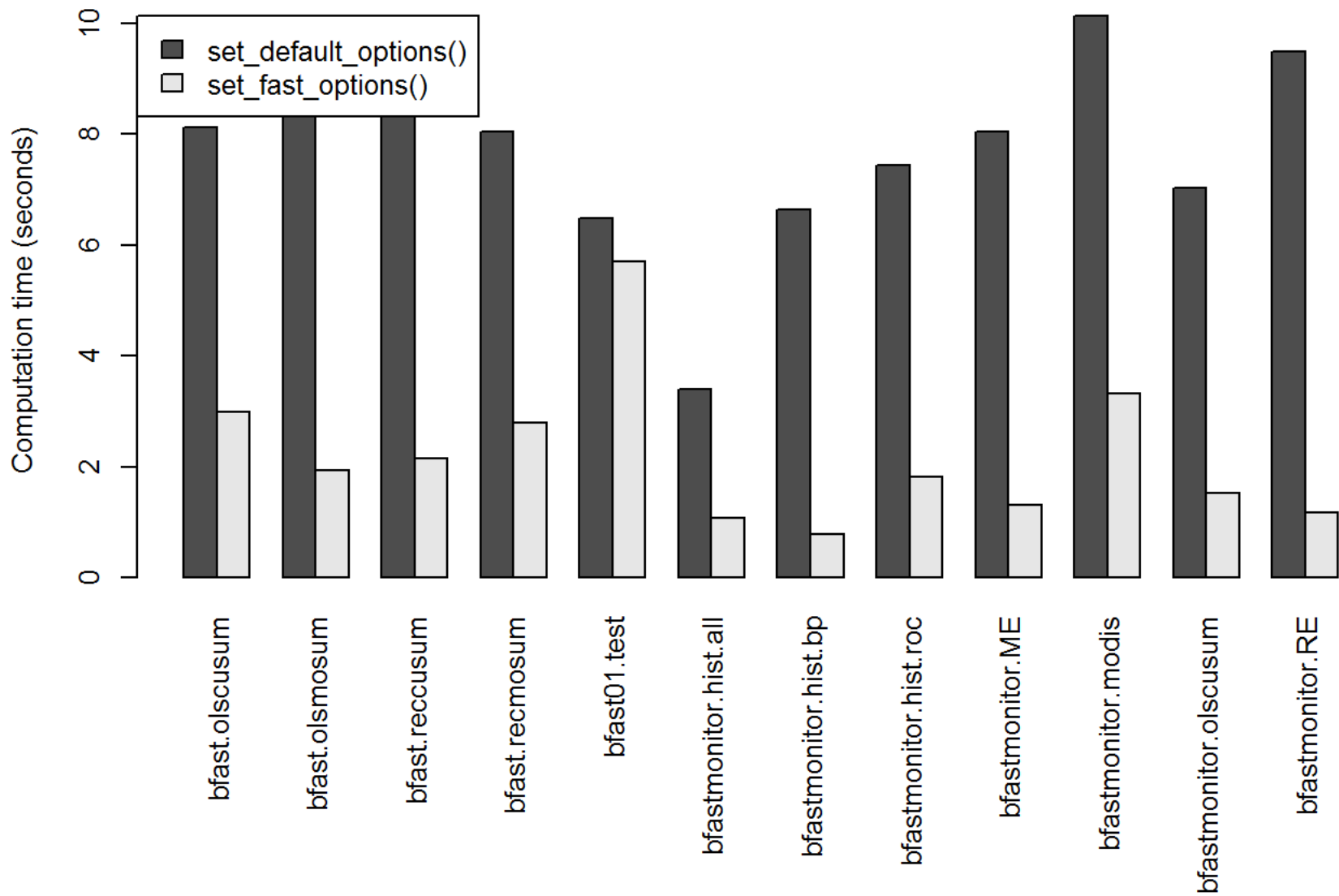
```
library(bfast)
set_fast_options()      # use modifications
...
set_default_options()  # use default implementation
...
```

see <https://github.com/appelmar/bfast>

Package Options

Package option	Description
<code>strucchange.use_armadillo</code>	Defines whether or not C++ functions should be used if available
<code>strucchange.armadillo_rcond_min</code>	For ill-conditioned systems in the recursive residual computation, the minimum reciprocal conditioning number to use armadillo solve instead of column pivoting QR from R
<code>bfast.prefer_matrix_methods</code>	Defines whether or not bfastpp generates a data frame or a matrix
<code>bfast.use_bfastts_modifications</code>	Defines whether or not bfastts modifications should be used

Results: speedup



see <https://appelmar.github.io/scalbf-wur/reports/report.benchmark.html>

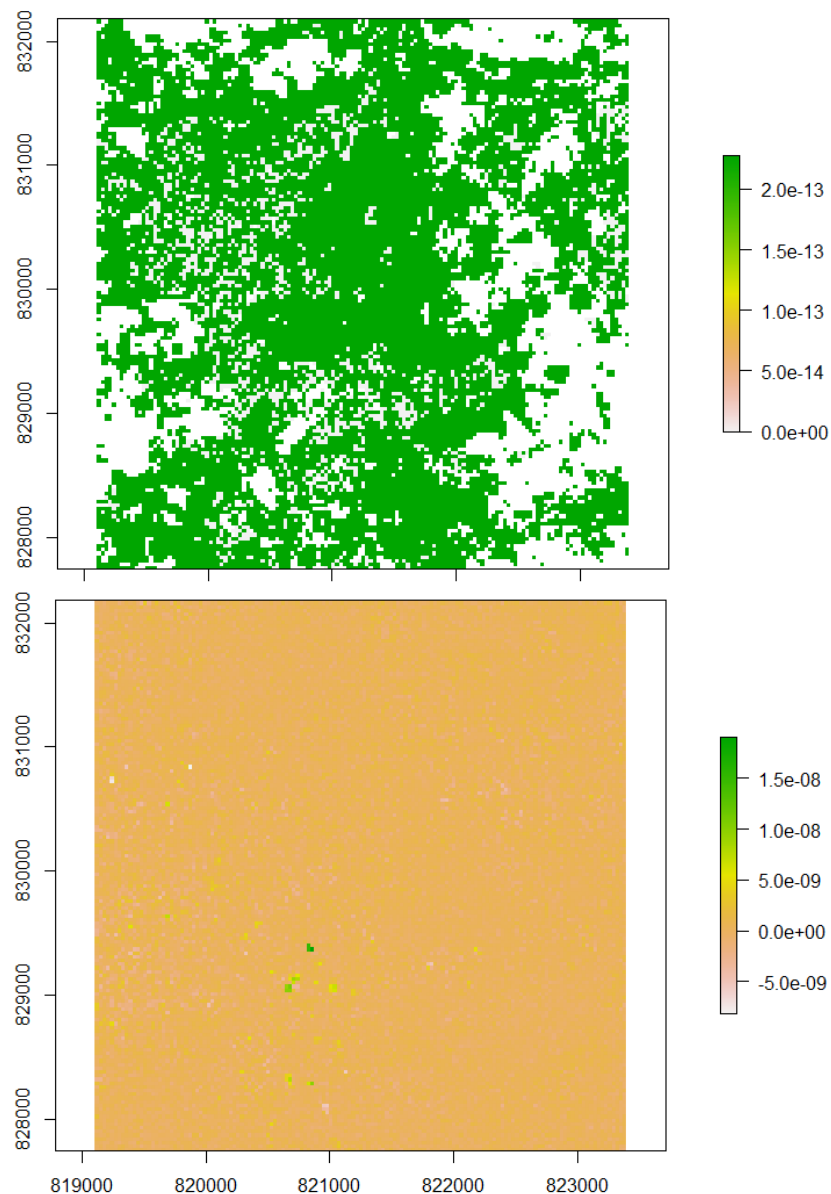
Example: bfastSpatial with tura dataset

```
library(bfastSpatial)
data(tura)

set_default_options()
system.time(bfm.tura.new <-
  bfmSpatial(tura, start=c(2009, 1),
    history = "ROC")) # 1318.52 s

set_fast_options()
system.time(bfm.tura.new <-
  bfmSpatial(tura, start=c(2009, 1),
    history = "ROC")) # 288.75 s

plot(bfm.tura.new$breakpoint -
  bfm.tura.reference$breakpoint)
plot(bfm.tura.new$magnitude -
  bfm.tura.reference$magnitude)
```



Package reports

The package comes with some R Markdown reports to test, benchmark, and profile the modifications.

```
library(rmarkdown)

render(system.file("reports/report.test.Rmd",
  package = "bfast"), output_file = "report.test.html")

render(system.file("reports/report.benchmark.Rmd",
  package = "bfast"), output_file = "report.benchmark.html")

render(system.file("reports/report.profiling.Rmd",
  package = "bfast"), output_file = "report.profiling.html")
```

Examples:

- <https://appelmar.github.io/scalbf-wur/reports/report.test.html>
- <https://appelmar.github.io/scalbf-wur/reports/report.benchmark.html>
- <https://appelmar.github.io/scalbf-wur/reports/report.profiling.html>

Summary and conclusions

- speedup of bfastmonitor and bfast varies between 2 and 10, depending on computations
- There is no single computational bottleneck, probably difficult to employ e.g. GPUs
- Strongest speedup in bfastmonitor with `(type="RE", Bai and Perron history (history = "BP"))`
- Optimizations also work with bfastSpatial even with parallelization
- bfast and strucchange now need to be compiled and linked to Rcpp and RcppArmadillo respectively

Thank you

Questions?

- please try your own analyses with the modifications
- any ideas for improvements are welcome