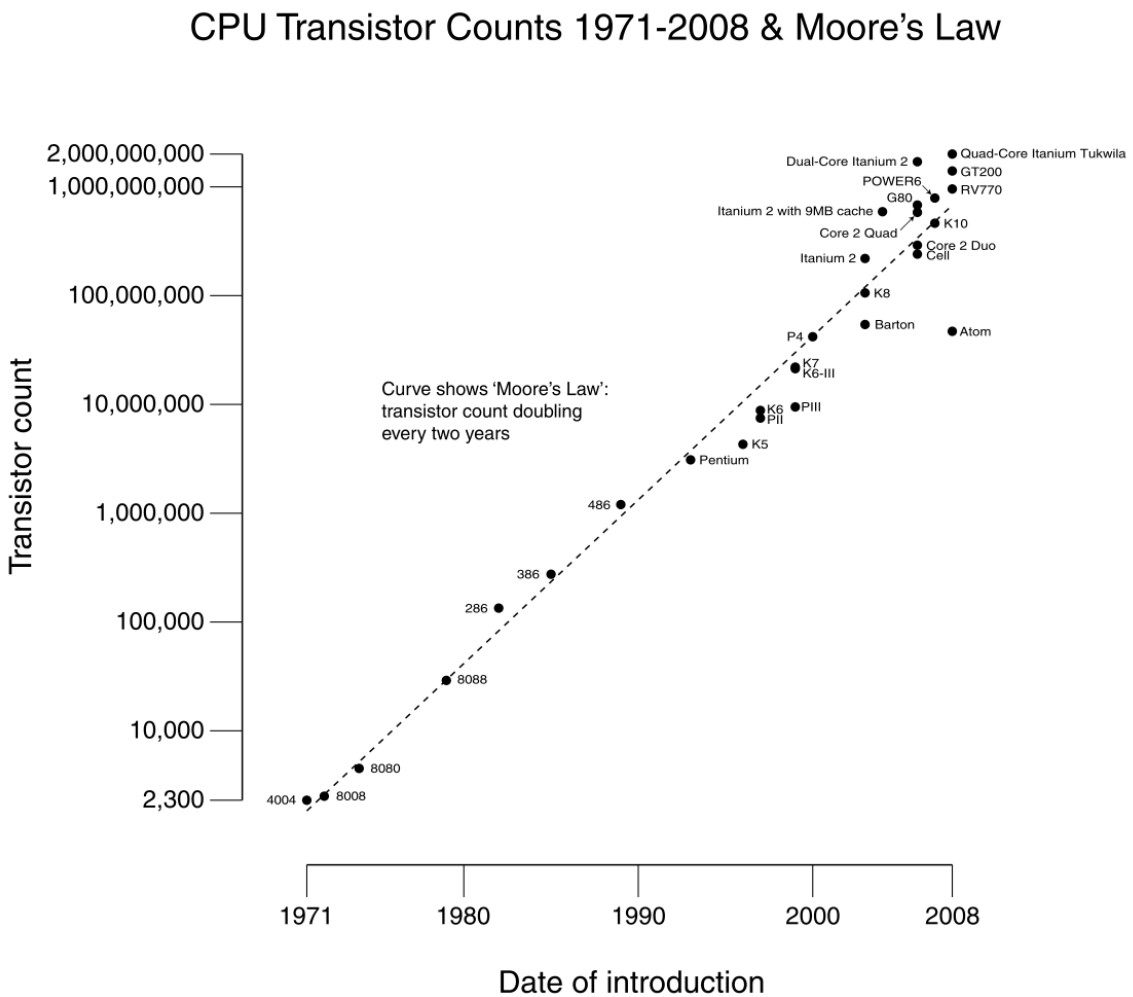
Introduction to High-Performance Computing with R

Dirk Eddelbuettel, Ph.D.
Dirk.Eddelbuettel@R-Project.org
edd@debian.org

Tutorial preceding
R/Finance 2009 Conference
Chicago, IL, USA
April 24, 2009

Outline

Motivation: What describes our current situation?



Source: http://en.wikipedia.org/wiki/Moore's_law

Moore's Law: Computers keep getting *faster and faster*

But at the same time our datasets get *bigger and bigger*.

So we're still *waiting and waiting . . .*

Hence: A need for higher performance computing with R.

Motivation: Presentation Roadmap

We will start by *measuring* how we are doing before looking at ways to improve our computing performance.

We will look at *vectorisation*, as well as various ways to *compile code*.

We will look briefly at *debugging* tools and tricks as well.

In the longer format, this tutorial also covers

- a detailed discussion of several ways to get more things done at the same time by using simple *parallel computing* approaches.
- ways to compute with R *beyond the memory limits* imposed by the R engine.
- ways to *automate and script* running R code.

but we will skip those topics today.

Table of Contents

Outline

Profiling

We need to know where our code spends the time it takes to compute our tasks.

Measuring—using *profiling tools*—is critical.

R already provides the basic tools for performance analysis.

- the `system.time` function for simple measurements.
- the `Rprof` function for profiling R code.
- the `Rprofmem` function for profiling R memory usage.

In addition, the `profr` and `proftools` package on CRAN can be used to visualize `Rprof` data.

We will also look at a script from the R Wiki for additional visualization.

Profiling cont.

The chapter *Tidying and profiling R code* in the *R Extensions* manual is a good first source for documentation on profiling and debugging.

Simon Urbanek has a page on benchmarks (for Macs) at `http://r.research.att.com/benchmarks/`

One can also profile compiled code, either directly (using the `-pg` option to `gcc`) or by using *e.g.* the Google `perftools` library.

RProf example

Consider the problem of repeatedly estimating a linear model, *e.g.* in the context of Monte Carlo simulation.

The `lm()` workhorse function is a natural first choice.

However, its generic nature as well the rich set of return arguments come at a cost. For experienced users, `lm.fit()` provides a more efficient alternative.

But how much more efficient?

We will use both functions on the `longley` data set to measure this.

RProf example cont.

This code runs both approaches 2000 times:

```
data(longley)
Rprof("longley.lm.out")
invisible(replicate(2000,
                    lm(Employed ~ ., data=longley)))
Rprof(NULL)


longleydm <- data.matrix(data.frame(intcp=1, longley))
Rprof("longley.lm.fit.out")
invisible(replicate(2000,
                    lm.fit(longleydm[,-8],
                           longleydm[,8])))
Rprof(NULL)
```

RProf example cont.

We can analyse the output two different ways. First, directly from R into an R object:

```
data <- summaryRprof("longley.lm.out")
print(str(data))
```

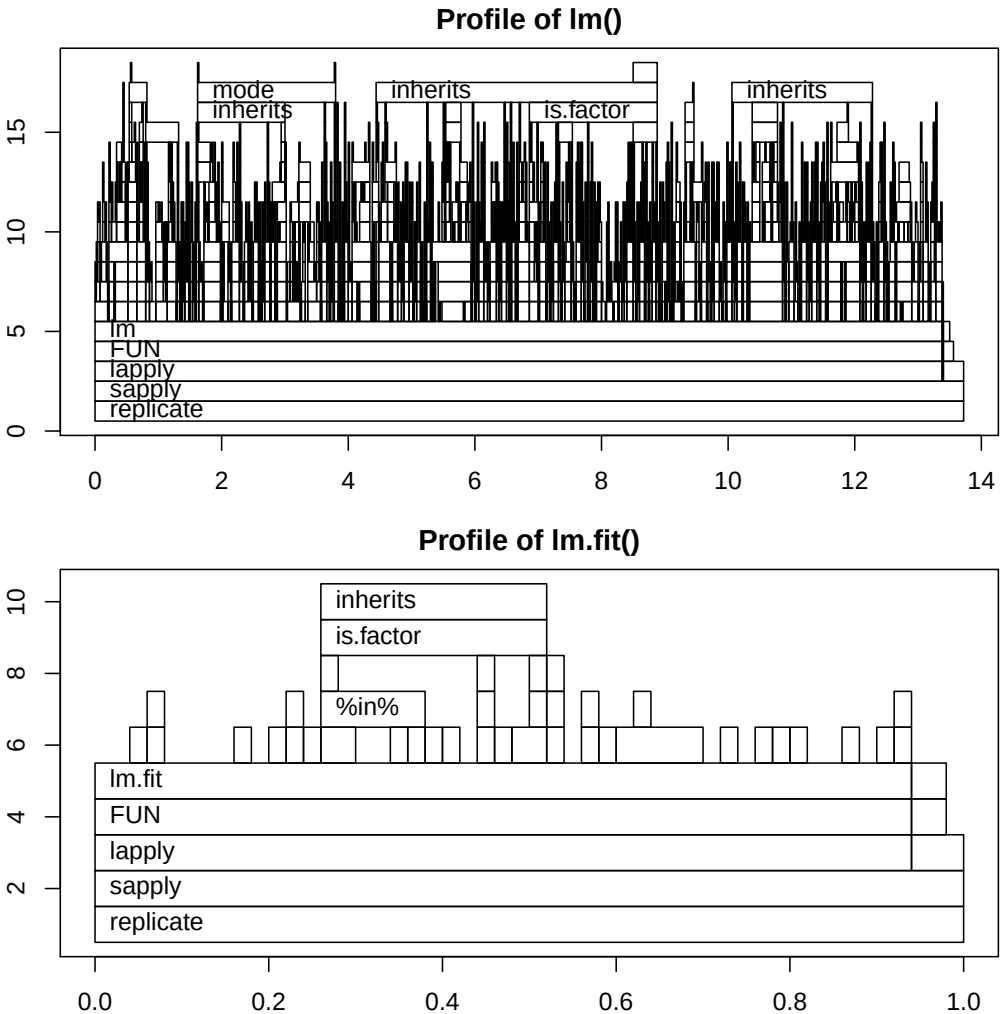
Second, from the command-line (on systems having `Perl`)

```
R CMD Prof longley.lm.out | less
```

The CRAN package / function `profr` by H. Wickham can profile, evaluate, and optionally plot, an expression directly. Or we can use `parse_profr()` to read the previously recorded output:

```
plot(parse_rprof("longley.lm.out"),
                 main="Profile of lm()")
plot(parse_rprof("longley.lm.fit.out"),
                 main="Profile of lm.fit()")
```

RProf example cont.



Source: Our calculations.

We notice the different *x* and *y* axis scales

For the same number of runs, `lm.fit()` is about fourteen times faster as it makes fewer calls to other functions.

RProf example cont.

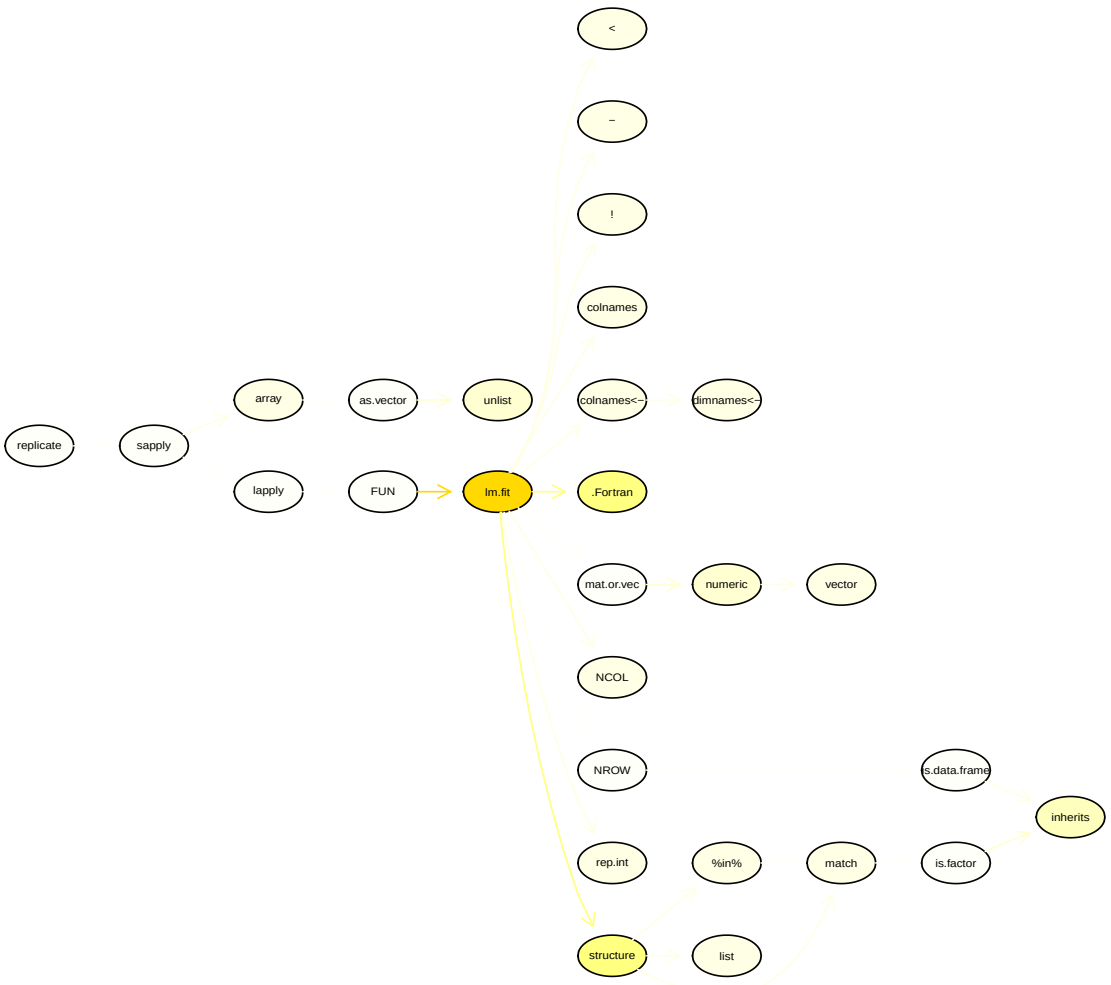
In addition, the `proftools` package by L. Tierney can read profiling data and summarize directly in R.

The `flatProfile` function aggregates the data, optionally with totals.

```
lmfitprod <- readProfileData("longley.lm.fit.out"))
plotProfileCallGraph(lmfitprof)
```

And `plotProfileCallGraph()` can be used to visualize profiling information using the `Rgraphviz` package (which is no longer on CRAN).

RProf example cont.



Color is used to indicate which nodes use the most of amount of time.

Use of color and other aspects can be configured.

Another profiling example

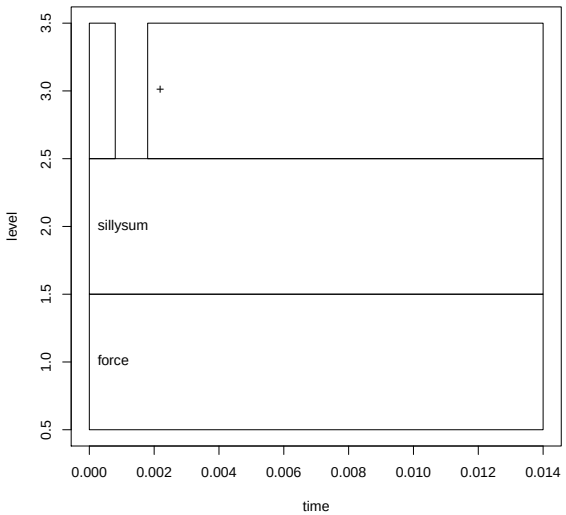
Both packages can be very useful for their quick visualisation of the `RProf` output. Consider this contrived example:

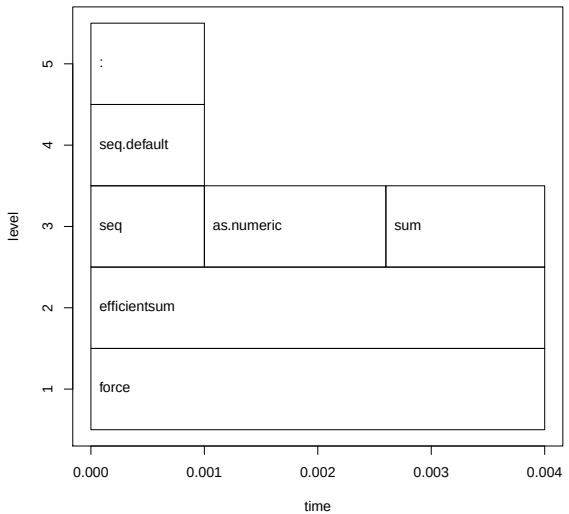
```
sillysum <- function(N) {s <- 0;
      for (i in 1:N) s <- s + i; s}
ival <- 1/5000
plot(profr(a <- sillysum(1e6), ival))
```

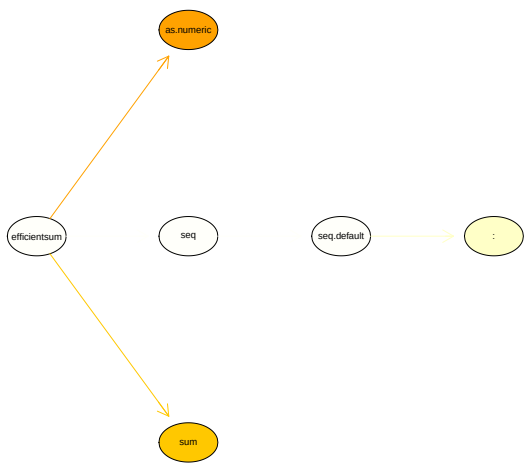
and for a more efficient solution where we use a larger *N*:

```
efficientsum <- function(N) {
sum(as.numeric(seq(1,N))) }
ival <- 1/5000
plot(profr(a <- efficientsum(1e7), ival))
```

Another profiling example (cont.)







`profr` and `proftools` complement each other.

Numerical values in `profr` provide information too.

Choice of colour is useful in `proftools`.

Additional profiling visualizations

Romain Francois has contributed a `Perl` script[1] which can be used to visualize profiling output via the `dot` program (part of `graphviz`):

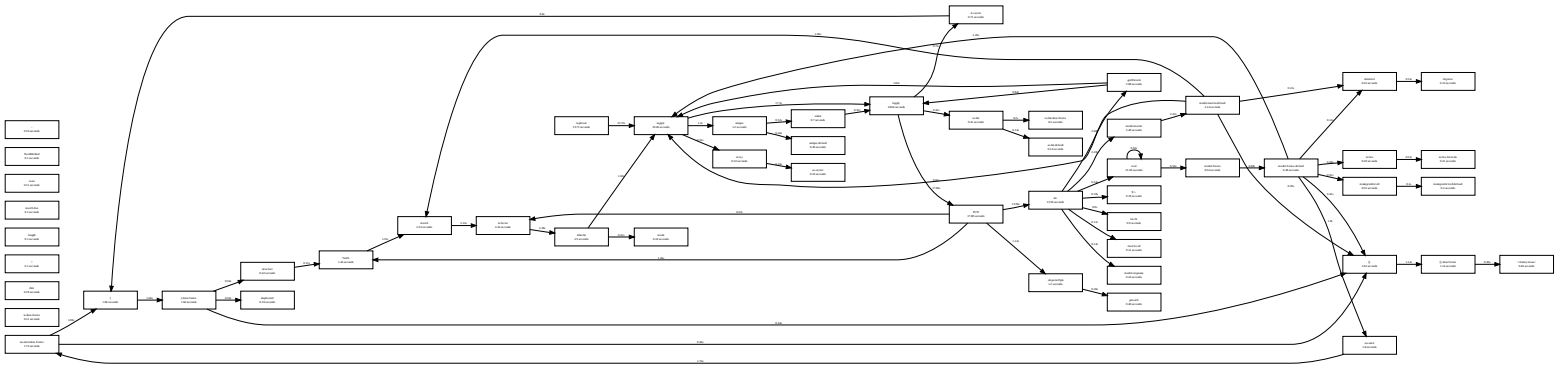
```
./prof2dot.pl longley.lm.out | dot -Tpdf \
              > longley_lm.pdf
./prof2dot.pl longley.lm.fit.out | dot -Tpdf \
              > longley_lmfit.pdf
```

Its key advantages are the ability to include, exclude or restrict functions.

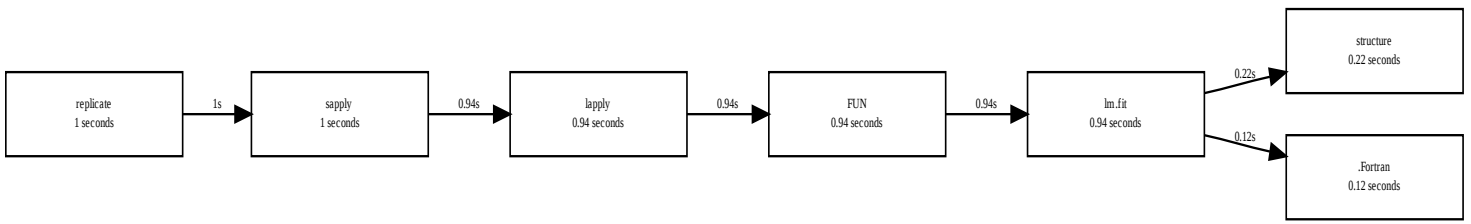
[1] `http://wiki.r-project.org/rwiki/doku.php?id=tips:misc:profiling:current`

Additional profiling visualizations (cont.)

For `lm()`, this yields:



and for `lm.fit()`, this yields:



RProfmem

When R has been built with the `enable-memory-profiling` option, we can also look at use of memory and allocation.

To continue with the *R Extensions* manual example, we issue calls to `Rprofmem` to start and stop logging to a file as we did for `Rprof`. This can be a helpful check for code that is suspected to have an error in its memory allocations.

We also mention in passing that the `tracemem` function can log when copies of a (presumably large) object are being made. Details are in section 3.3.3 of the *R Extensions* manual.

Profiling compiled code

Profiling compiled code typically entails rebuilding the binary and libraries with the `-gp` compiler option. In the case of R, a complete rebuild is required as R itself needs to be compiled with profiling options.

Add-on tools like `valgrind` and `kcachegrind` can be very helpful and may not require rebuilds.

Two other options for Linux are mentioned in the *R Extensions* manual. First, `sprof`, part of the C library, can profile shared libraries. Second, the add-on package `oprofile` provides a daemon that has to be started (stopped) when profiling data collection is to start (end).

A third possibility is the use of the Google Perftools which we will illustrate.

Profiling with Google Perftools

The Google Perftools provide four modes of performance analysis / improvement:

- a thread-caching malloc (memory allocator),
- a heap-checking facility,
- a heap-profiling facility and
- cpu profiling.

Here, we will focus on the last feature.

There are two possible modes of running code with the cpu profiler.

The preferred approach is to link with `-lprofiler`. Alternatively, one can dynamically pre-load the profiler library.

Profiling with Google Perftools (cont.)

```
# turn on profiling and provide a profile log file
LD_PRELOAD="/usr/lib/libprofiler.so.0" \
CPUPROFILE=/tmp/rprof.log \
r profilingSmall.R
```

We can then analyse the profiling output in the file. The profiler (renamed from `pprof` to `google-pprof` on Debian) has a large number of options. Here just use two different formats:

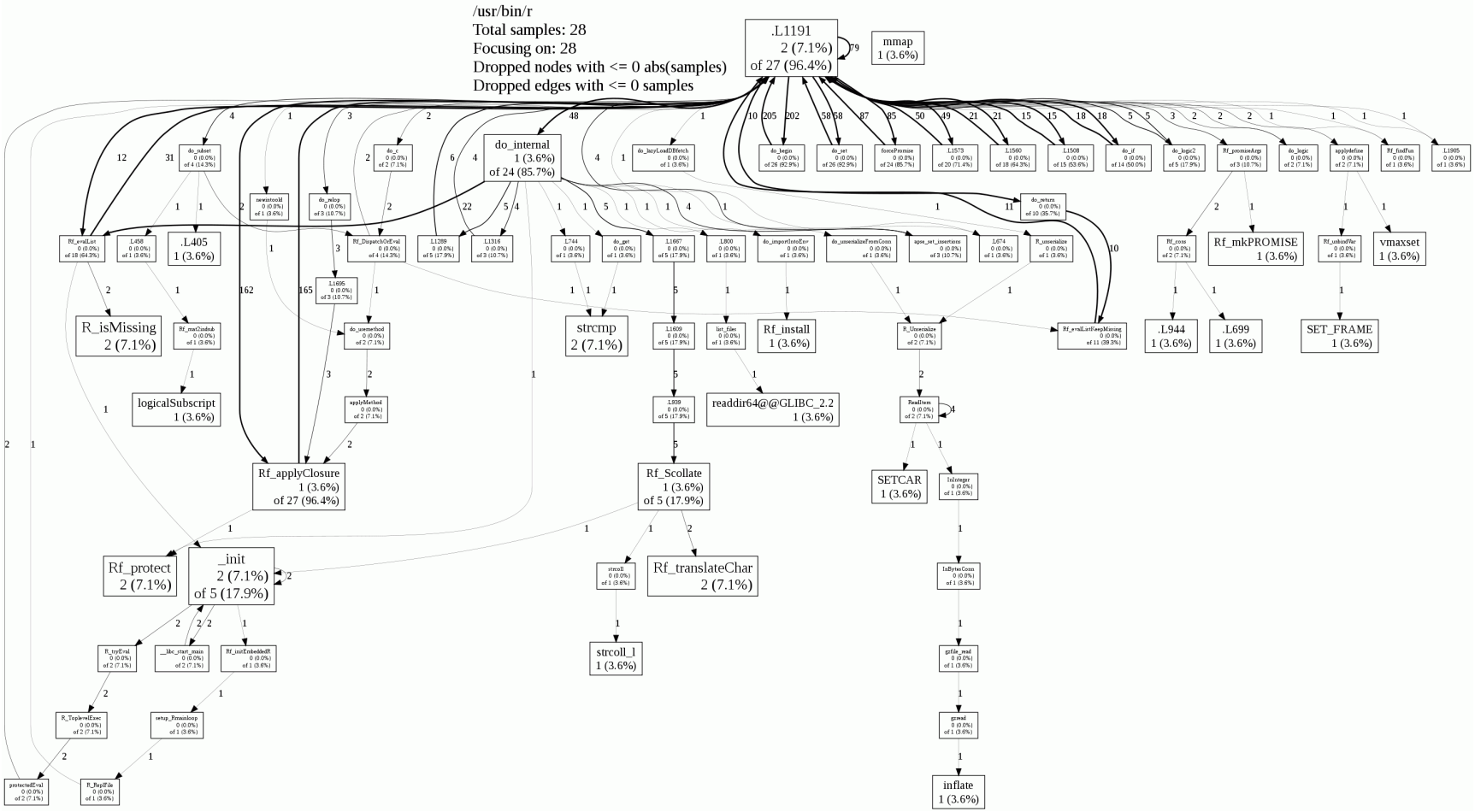
```
# show text output
google-pprof --cum --text \
   /usr/bin/r /tmp/rprof.log | less

# or analyse call graph using gv
google-pprof --gv /usr/bin/r /tmp/rprof.log
```

The shell script `googlePerftools.sh` runs the complete example.

Profiling with Google Perftools

This can generate complete (yet complex) graphs.



Profiling with Google Perftools

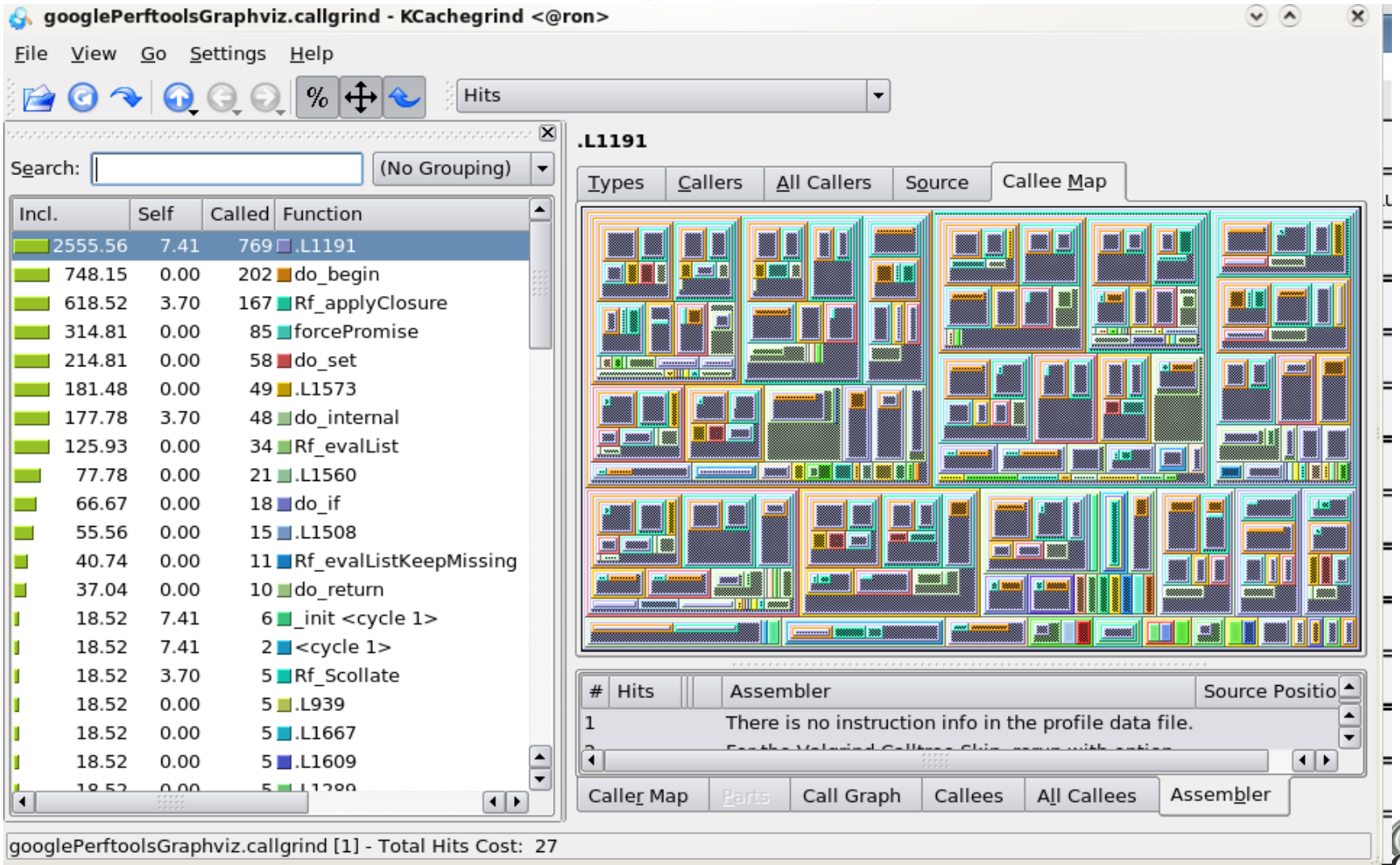
Another output for format is for the *callgrind* analyser that is part of *valgrind*—a frontend to a variety of analysis tools such as *cachegrind* (cache simulator), *callgrind* (call graph tracer), *helpgrind* (race condition analyser), *massif* (heap profiler), and *memcheck* (fine-grained memory checker).

For example, the KDE frontend *kcachegrind* can be used to visualize the profiler output as follows:

```
google-pprof --callgrind \
   /usr/bin/r /tmp/gpProfile.log \
   > googlePerftools.callgrind
kcachegrind googlePerftools.callgrind
```

Profiling with Google Perftools

Kcachegrind running on the the profiling output looks as follows:



Profiling with Google Perftools

One problem with the 'global' approach to profiling is that a large number of internal functions are being reported as well—this may obscure our functions of interest.
An alternative is to re-compile the portion of code that we want to profile, and to bracket the code with

```
ProfilerStart()

// ... code to be profiled here ...

ProfilerEnd()
```

which are defined in `google/profiler.h` which needs to be included. One uses the environment variable `CPUPROFILE` to designate an output file for the profiling information, or designates a file as argument to `ProfilerStart()`.

Outline

Vectorisation

Revisiting our trivial trivial example from the preceding section:

```
> sillysum <- function(N) { s <- 0;
      for (i in 1:N) s <- s + i; return(s) }
> system.time(print(sillysum(1e7)))

[1] 5e+13
   user  system elapsed
 13.617   0.020  13.701
>

> system.time(print(sum(as.numeric(seq(1,1e7)))))

[1] 5e+13
   user  system elapsed
  0.224   0.092   0.315
>
```

Replacing the loop yielded a gain of a factor of more than 40. It really pays to know the corpus of available functions.

Vectorisation cont.

A more interesting example is provided in a case study on the `Ra` (c.f. next section) site and taken from the *S Programming* book:

> *Consider the problem of finding the distribution of the determinant of a 2 x 2 matrix where the entries are independent and uniformly distributed digits 0, 1, . . ., 9. This amounts to finding all possible values of $ac - bd$ where a, b, c and d are digits.*

Vectorisation cont.

The brute-force solution is using explicit loops over all combinations:

```
dd.for.c <- function() {
  val <- NULL
  for (a in 0:9) for (b in 0:9)
      for (d in 0:9) for (e in 0:9)
          val <- c(val, a*b - d*e)
  table(val)
}
```

The naive time is

```
> mean(replicate(10, system.time(dd.for.c())["elapsed"]))
[1] 0.2678
```

Vectorisation cont.

The case study discusses two important points that bear repeating:

- pre-allocating space helps with performance:
  `val <- double(10000)`
  and using `val[i <- i + 1]` as the left-hand side reduces the time to 0.1204
- switching to faster functions can help too as `tabulate` outperforms `table` and reduced the time further to 0.1180.

Vectorisation cont.

However, by far the largest improvement comes from eliminating the four loops with two calls each to `outer`:

```
dd.fast.tabulate <- function() {
  val <- outer(0:9, 0:9, "*")
  val <- outer(val, val, "-")
  tabulate(val)
}
```

The time for the most efficient solution is:

```
> mean(replicate(10,
       system.time(dd.fast.tabulate())["elapsed"]))

[1] 0.0014
```

which is orders of magnitude faster.

All examples can be run via the script `dd.naive.r`.

Outline

Accelerated R with just-in-time compilation

Stephen Milborrow maintains “Ra”, a set of patches to R that allow ’just-in-time compilation’ of loops and arithmetic expressions. Together with his `jit` package on CRAN, this can be used to obtain speedups of standard R operations.

Our trivial example run in `Ra`:

```
library(jit)
sillysum <- function(N) { jit(1); s <- 0;  \
     for (i in 1:N) s <- s + i; return(s) }
```

```
 > system.time(print(sillysum(1e7)))
[1] 5e+13
   user  system elapsed
  1.548   0.028   1.577
```

which gets a speed increase of a factor of five—not bad at all.

Accelerated R with just-in-time compilation

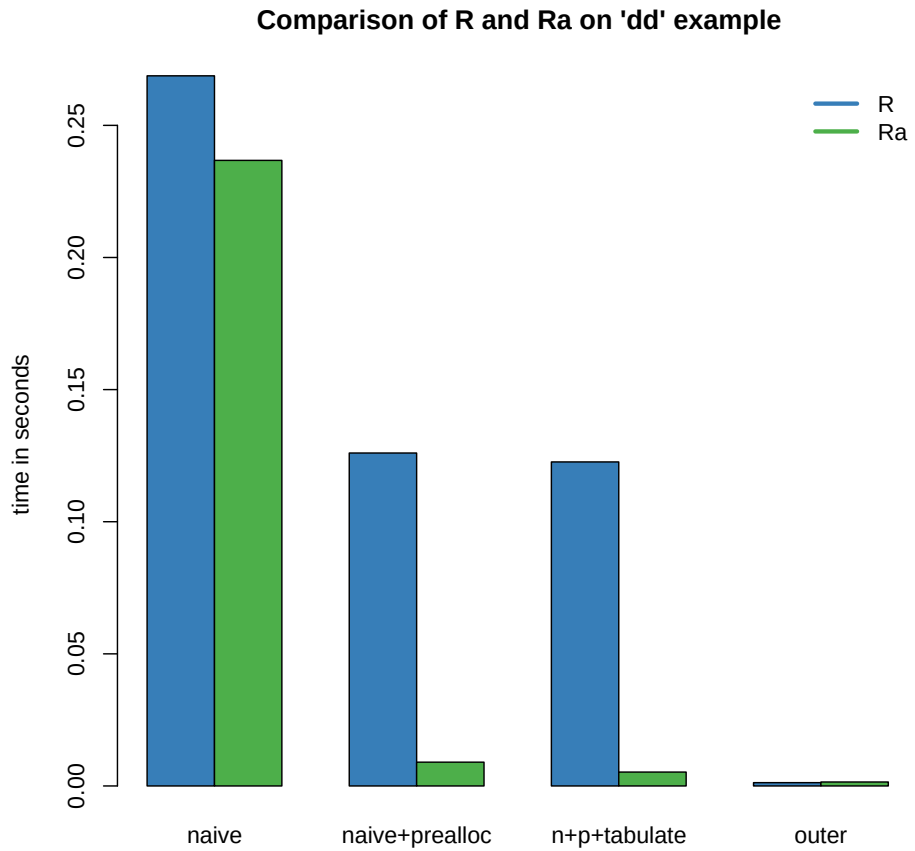
The last looping example can be improved with jit:

```
dd.for.pre.tabulate.jit <- function() {
  jit(1)
  val <- double(10000)
  i <- 0
  for (a in 0:9) for (b in 0:9)
      for (d in 0:9) for (e in 0:9) {
          val[i <- i + 1] <- a*b - d*e
  }
  tabulate(val)
}
```

```
 > mean(replicate(10, system.time(dd.for.pre.tabulate.jit())["elapsed"
[1] 0.0053
```

or only about three to four times slower than the non-looped solution using 'outer'—a rather decent improvement.

Accelerated R with just-in-time compilation



Source: Our calculations

`Ra` achieves very good decreases in total computing time in these examples but cannot improve the efficient solution any further.

`Ra` and `jit` are still fairly new and not widely deployed yet, but readily available in Debian and Ubuntu.

Outline

Optimised Blas

Blas ('basic linear algebra subprogram', see Wikipedia) are standard building blocks for linear algebra. Highly-optimised libraries exist that can provide considerable performance gains.

R can be built using so-called optimised Blas such as Atlas ('free'), Goto (not 'free'), or those from Intel or AMD; see the 'R Admin' manual, section A.3 'Linear Algebra'.

The speed gains can be noticeable. For Debian/Ubuntu, one can simply install on of the `atlas-base-*` packages.

An example from the old README.Atlas, running with a R 2.8.1 on a four-core machine:

Optimised Blas cont.

```
# with Atlas
> mm <- matrix(rnorm(4*10^6), ncol = 2*10^3)
> mean(replicate(10,
      system.time(crossprod(mm))["elapsed"]),trim=0.1)
[1] 2.6465

# with basic. non-optmised Blas,
> mm <- matrix(rnorm(4*10^6), ncol = 2*10^3)
> mean(replicate(10,
      system.time(crossprod(mm))["elapsed"]),trim=0.1)
[1] 16.42813
```

For linear algebra problems, we may get an improvement by an integer factor that may be as large (or even larger) than the number of cores as we benefit from both better code and multithreaded execution. Even higher increases are possibly by 'tuning' the library, see the Atlas documentation.

From Blas to GPUs.

The next frontier for hardware acceleration is computing on GPUs ('graphics programming units', see Wikipedia).

GPUs are essentially hardware that is optimised for both I/O and floating point operations, leading to much faster code execution than standard CPUs on floating-point operations.

Development kits are available (*e.g.* Nvidia CUDA) and the recently announced OpenCL programming specification should make GPU-computing vendor-independent.

Some initial work on integration with R has been undertaken but there appear to no easy-to-install and easy-to-use kits for R – yet.

So this provides a perfect intro for the next subsection on compilation.

Outline

Compiled Code

Beyond smarter code (using *e.g.* vectorised expression and/or just-in-time compilation) or optimised libraries, the most direct speed gain comes from switching to compiled code.

This section covers two possible approaches:

- `inline` for automated wrapping of simple expression
- `Rcpp` for easing the interface between R and C++

A different approach is to keep the core logic 'outside' but to *embed* R into the application. There is some documentation in the 'R Extensions' manual—and the `RInside` package on R-Forge offers C++ classes to automate this. This may still require some familiarity with R internals.

Compiled Code: The Basics

R offers several functions to access compiled code: `.C` and `.Fortran` as well as `.Call` and `.External`. (*R Extensions*, sections 5.2 and 5.9; *Software for Data Analysis*). `.C` and `.Fortran` are older and simpler, but more restrictive in the long run.

The canonical example in the documentation is the convolution function:

```
1  void convolve(double *a, int *na, double *b,
2                int *nb, double *ab)
3  {
4    int i, j, nab = *na + *nb - 1;
5
6    for(i = 0; i < nab; i++)
7      ab[i] = 0.0;
8    for(i = 0; i < *na; i++)
9      for(j = 0; j < *nb; j++)
10       ab[i + j] += a[i] * b[j];
11 }
```

Compiled Code: The Basics cont.

The convolution function is called from R by

```
conv <- function(a, b)
  .C("convolve",
     as.double(a),
     as.integer(length(a)),
     as.double(b),
     as.integer(length(b)),
     ab = double(length(a) + length(b) - 1))$ab
```

As stated in the manual, one must take care to coerce all the arguments to the correct R storage mode before calling `.C` as mistakes in matching the types can lead to wrong results or hard-to-catch errors.

The script `convolve.C.sh` compiles and links the source code, and then calls R to run the example.

Compiled Code: The Basics cont.

Using `.Call`, the example becomes

```
1  #include <R.h>
2  #include <Rdefines.h>
3
4  SEXP convolve2(SEXP a, SEXP b)
5  {
6    int i, j, na, nb, nab;
7    double *xa, *xb, *xab;
8    SEXP ab;
9
10   PROTECT(a = AS_NUMERIC(a));
11   PROTECT(b = AS_NUMERIC(b));
12   na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
13   PROTECT(ab = NEW_NUMERIC(nab));
14   xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
15   xab = NUMERIC_POINTER(ab);
16   for(i = 0; i < nab; i++) xab[i] = 0.0;
17   for(i = 0; i < na; i++)
18     for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
19   UNPROTECT(3);
20   return(ab);
21 }
```

Compiled Code: The Basics cont.

Now the call becomes easier by just using the function name and the vector arguments—all other handling is done at the C/C++ level:

```
conv <- function(a, b) .Call("convolve2", a, b)
```

The script `convolve.Call.sh` compiles and links the source code, and then calls R to run the example.

In summary, we see that

- there are different entry points
- using different calling conventions
- leading to code that may need to do more work at the lower level.

Compiled Code: inline

`inline` is a package by Oleg Sklyar et al that provides the function `cfunction` that can wrap Fortran, C or C++ code.

```
## A simple Fortran example
code <- "
      integer i
      do 1 i=1, n(1)
    1 x(i) = x(i)**3
"
cubefn <- cfunction(signature(n="integer", x="numeric"),
                    code, convention=".Fortran")
x <- as.numeric(1:10)
n <- as.integer(10)
cubefn(n, x)$x
```

`cfunction` takes care of compiling, linking, loading, . . . by placing the resulting dynamically-loadable object code in the per-session temporary directory used by R.

Run this via `cat inline.Fortan.R | R -no-save`.

Compiled Code: inline cont.

`inline` defaults to using the `.Call()` interface:

```
1  ## Use of .Call convention with C code
2  ## Multyplying each image in a stack with a 2D Gaussian at a given position
3  code <- "
4    SEXP res;
5    int nprotect = 0, nx, ny, nz, x, y;
6    PROTECT(res = Rf_duplicate(a)); nprotect++;
7    nx = INTEGER(GET_DIM(a))[0];
8    ny = INTEGER(GET_DIM(a))[1];
9    nz = INTEGER(GET_DIM(a))[2];
10   double sigma2 = REAL(s)[0] * REAL(s)[0], d2 ;
11   double cx = REAL(centre)[0], cy = REAL(centre)[1], *data, *rdata;
12   for (int im = 0; im < nz; im++) {
13     data = &(REAL(a)[im*nx*ny]); rdata = &(REAL(res)[im*nx*ny]);
14     for (x = 0; x < nx; x++)
15       for (y = 0; y < ny; y++) {
16         d2 = (x-cx)*(x-cx) + (y-cy)*(y-cy);
17         rdata[x + y*nx] = data[x + y*nx] * exp(-d2/sigma2);
18       }
19   }
20   UNPROTECT(nprotect);
21   return res;
22 "
23 funx <- cfunction(signature(a="array", s="numeric", centre="numeric"), code)
24
25 x <- array(runif(50*50), c(50,50,1))
26 res <- funx(a=x, s=10, centre=c(25,15))   ## actual call of compiled function
27 if (interactive()) image(res[,,1])
```

Compiled Code: inline cont.

We can revisit the earlier distribution of determinants example.

If we keep it very simple and pre-allocate the temporary vector in R , the example becomes

```
code <- "
  if (isNumeric(vec)) {
    int *pv = INTEGER(vec);
    int n = length(vec);
    if (n = 10000) {
      int i = 0;
      for (int a = 0; a < 9; a++)
        for (int b = 0; b < 9; b++)
          for (int c = 0; c < 9; c++)
            for (int d = 0; d < 9; d++)
              pv[i++] = a*b - c*d;
    }
  }
  return(vec);
"

funx <- cfunction(signature(vec="numeric"), code)
```

Compiled Code: inline cont.

We can use the inlined function in a new function to be timed:

```
dd.inline <- function() {
    x <- integer(10000)
    res <- funx(vec=x)
    tabulate(res)
}
> mean(replicate(100, system.time(dd.inline())["elapsed"]))
[1] 0.00051
```

Even though it uses the simplest algorithm, pre-allocates memory in R and analyses the result in R , it is still more than twice as fast as the previous best solution.

The script `dd.inline.r` runs this example.

Compiled Code: Rcpp

`Rcpp` makes it easier to interface C++ and R code.

Using the `.Call` interface, we can use features of the C++ language to automate the tedious bits of the macro-based C-level interface to R.

One major advantage of using `.Call` is that vectors (or matrices) can be passed directly between R and C++ without the need for explicit passing of dimension arguments. And by using the C++ class layers, we do not need to directly manipulate the SEXP objects.

So let us rewrite the 'distribution of determinant' example one more time.

Rcpp example

The simplest version can be set up as follows:

```
1  #include <Rcpp.hpp>
2
3  RcppExport SEXP dd_rcpp(SEXP v) {
4    SEXP  rl = R_NilValue;              // Use this when nothing is returned
5
6    RcppVector<int> vec(v);             // vec parameter viewed as vector of doubles
7    int n = vec.size(), i = 0;
8
9    for (int a = 0; a < 9; a++)
10     for (int b = 0; b < 9; b++)
11       for (int c = 0; c < 9; c++)
12         for (int d = 0; d < 9; d++)
13           vec(i++) = a*b - c*d;
14
15   RcppResultSet rs;                   // Build result set returned as list to R
16   rs.add("vec", vec);                 // vec as named element with name 'vec'
17   rl = rs.getReturnList();            // Get the list to be returned to R.
18
19   return rl;
20 }
```

but it is actually preferable to use the exception-handling feature of C++ as in the slightly longer next version.

Rcpp example cont.

```
1  #include <Rcpp.hpp>
2
3  RcppExport SEXP dd_rcpp(SEXP v) {
4    SEXP rl = R_NilValue;          // Use this when there is nothing to be returned.
5    char* exceptionMesg = NULL;    // msg var in case of error
6
7    try {
8      RcppVector<int> vec(v);      // vec parameter viewed as vector of doubles.
9      int n = vec.size(), i = 0;
10     for (int a = 0; a < 9; a++)
11       for (int b = 0; b < 9; b++)
12         for (int c = 0; c < 9; c++)
13           for (int d = 0; d < 9; d++)
14             vec(i++) = a*b - c*d;
15
16     RcppResultSet rs;            // Build result set to be returned as a list to R.
17     rs.add("vec", vec);          // vec as named element with name 'vec'
18     rl = rs.getReturnList();     // Get the list to be returned to R.
19   } catch(std::exception& ex) {
20     exceptionMesg = copyMessageToR(ex.what());
21   } catch(...) {
22     exceptionMesg = copyMessageToR("unknown reason");
23   }
24
25   if (exceptionMesg != NULL)
26     error(exceptionMesg);
27
28   return rl;
29 }
```

Rcpp example cont.

We can create a shared library from the source file as follows:

```
PKG_CPPFLAGS=`r -e'Rcpp:::CxxFlags()'` \
     R CMD SHLIB dd.rcpp.cpp \
     `r -e'Rcpp:::LdFlags()'`
```

```
g++ -I/usr/share/R/include \
     -I/usr/lib/R/site-library/Rcpp/lib \
     -fpic -g -O2 \
     -c dd.rcpp.cpp -o dd.rcpp.o
g++ -shared -o dd.rcpp.so dd.rcpp.o \
     -L/usr/lib/R/site-library/Rcpp/lib \
     -lRcpp -Wl,-rpath,/usr/lib/R/site-library/Rcpp/lib \
     -L/usr/lib/R/lib -lR
```

Note how we let the `Rcpp` package tell us where header and library files are stored.

Rcpp example cont.

We can then load the file using `dyn.load` and proceed as in the `inline` example.

```
dyn.load("dd.rcpp.so")

dd.rcpp <- function() {
    x <- integer(10000)
    res <- .Call("dd_rcpp", x)
    tabulate(res$vec)
}

mean(replicate(100,system.time(dd.rcpp())["elapsed"]))
```

```
[1] 0.00047
```

This beats the `inline` example by a neglible amount which is probably due to some overhead the in the easy-to-use inlining.

The file `dd.rcpp.sh` runs the full Rcpp example.

Basic Rcpp usage

Rcpp eases data transfer from R to C++, and back. We always convert to and from SEXP, and return a SEXP to R.

The key is that we can consider this to be a 'variant' type permitting us to extract using appropriate C++ classes. We pass data from R via named lists that may contain different types:

```
list(intnb=42, fltnb=6.78, date=Sys.Date(),
     txt="some thing", bool=FALSE)
```

by initialising a RcppParams object and extracting as in

```
RcppParams param(inputsexp);
int      nmb = param.getIntValue("intnb");
double   dbl = param.getIntValue("fltnb");
string   txt = param.getStringValue("txt");
bool     flg = param.getBoolValue("bool";
RcppDate dt  = param.getDateValue("date");
```

Basic Rcpp usage (cont.)

Similarly, we can constructs vectors and matrics of `double`, `int`, as well as vectors of types `string` and date and datetime. The key is that we *never* have to deal with dimensions and / or memory allocations — all this is shielded by C++ classes.

Similarly, for the return, we declare an object of type `RcppResultSet` and use the `add` methods to insert named elements before coverting this into a list that is assigned to the returned `SEXP`.

Back in R, we access them as elements of a standard R list by position or name.

Another Rcpp example

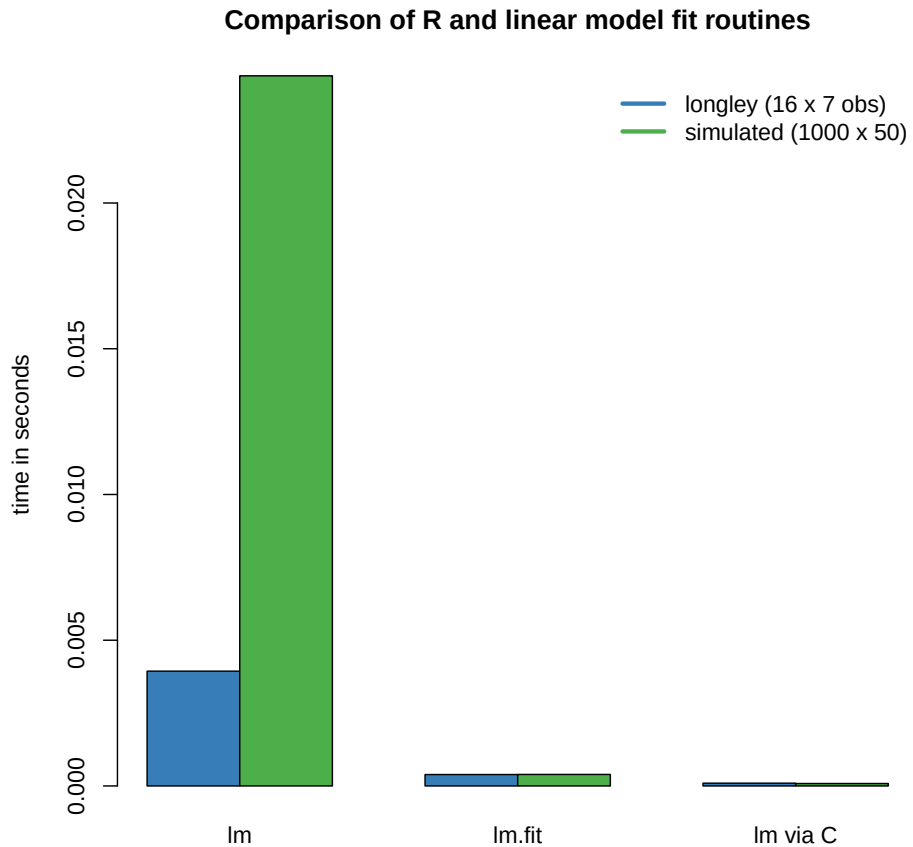
Let us revisit the `lm()` versus `lm.fit()` example. How fast could compiled code be? Let's wrap a GNU GSL function.

```
1  #include <cstdio>
2  extern "C" {
3  #include <gsl/gsl_multifit.h>
4  }
5  #include <Rcpp.h>
6
7  RcppExport SEXP gsl_multifit(SEXP Xsexp, SEXP Ysexp) {
8      SEXP rl=R_NilValue;
9      char *exceptionMesg=NULL;
10
11     try {
12         RcppMatrixView<double> Xr(Xsexp);
13         RcppVectorView<double> Yr(Ysexp);
14
15         int i,j,n = Xr.dim1(), k = Xr.dim2();
16         double chisq;
17
18         gsl_matrix *X = gsl_matrix_alloc (n, k);
19         gsl_vector *y = gsl_vector_alloc (n);
20         gsl_vector *c = gsl_vector_alloc (k);
21         gsl_matrix *cov = gsl_matrix_alloc (k, k);
22         for (i = 0; i < n; i++) {
23             for (j = 0; j < k; j++)
24                 gsl_matrix_set (X, i, j, Xr(i,j));
25             gsl_vector_set (y, i, Yr(i));
26         }
```

Another Rcpp example (cont.)

```
27          gsl_multifit_linear_workspace *work = gsl_multifit_linear_alloc (n, k);
28          gsl_multifit_linear (X, y, c, cov, &chisq, work);
29          gsl_multifit_linear_free (work);
30
31          RcppMatrix<double> CovMat(k, k);
32          RcppVector<double> Coef(k);
33          for (i = 0; i < k; i++) {
34              for (j = 0; j < k; j++)
35                  CovMat(i,j) = gsl_matrix_get(cov,i,j);
36              Coef(i) = gsl_vector_get(c,i);
37          }
38          gsl_matrix_free (X);
39          gsl_vector_free (y);
40          gsl_vector_free (c);
41          gsl_matrix_free (cov);
42
43          RcppResultSet rs;
44          rs.add("coef", Coef);
45          rs.add("covmat", CovMat);
46
47          rl = rs.getReturnList();
48
49      } catch(std::exception& ex) {
50          exceptionMesg = copyMessageToR(ex.what());
51      } catch(...) {
52          exceptionMesg = copyMessageToR("unknown reason");
53      }
54      if(exceptionMesg != NULL)
55          Rf_error(exceptionMesg);
56      return rl;
57  }
```

Another Rcpp example (cont.)

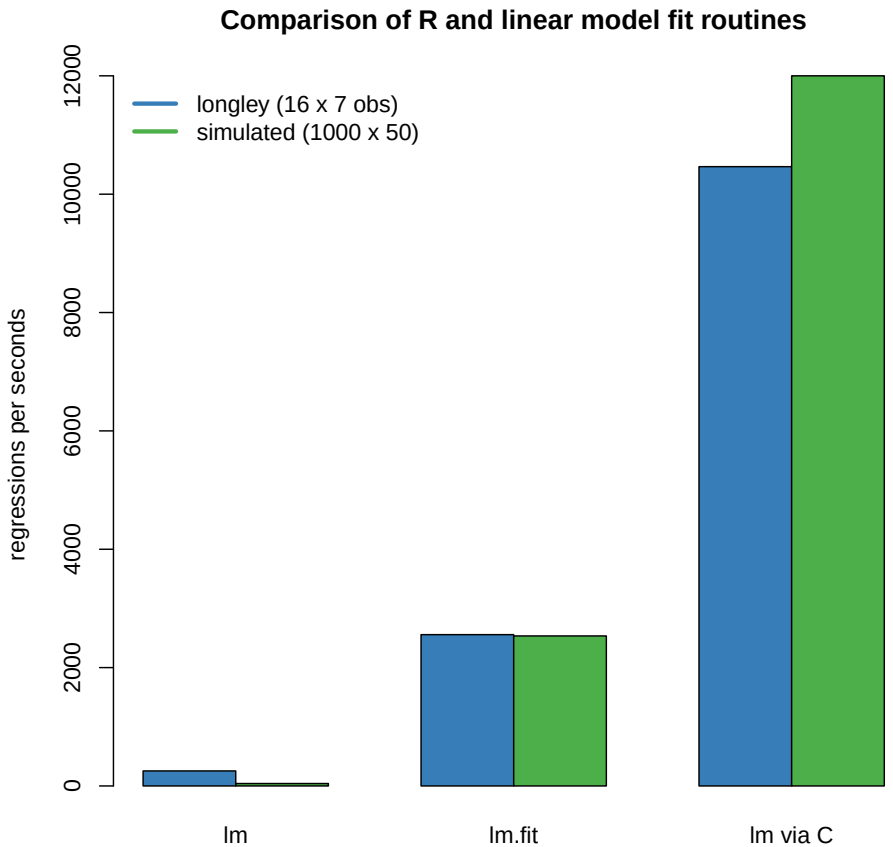


Source: Our calculations

The small `longley` example exhibits less variability between methods, but the larger data set shows the gains more clearly.

The `lm.fit()` approach appears unchanged between `longley` and the larger simulated data set.

Another Rcpp example (cont.)



Source: Our calculations

By inverting the times to see how many 'regressions per second' we can fit, the merits of the compiled code become clearer.

One caveat, measurements depends critically on the size of the data as well as the cpu and libraries that are used.

Revisiting profiling

We can also use the preceding example to illustrate how to profile subroutines.

We can add the following to the top of the function:

```
ProfilerStart("/tmp/ols.profile");
for (unsigned int i=1; i<10000; i++) {
```

and similarly

```
ProfilerStop();
```

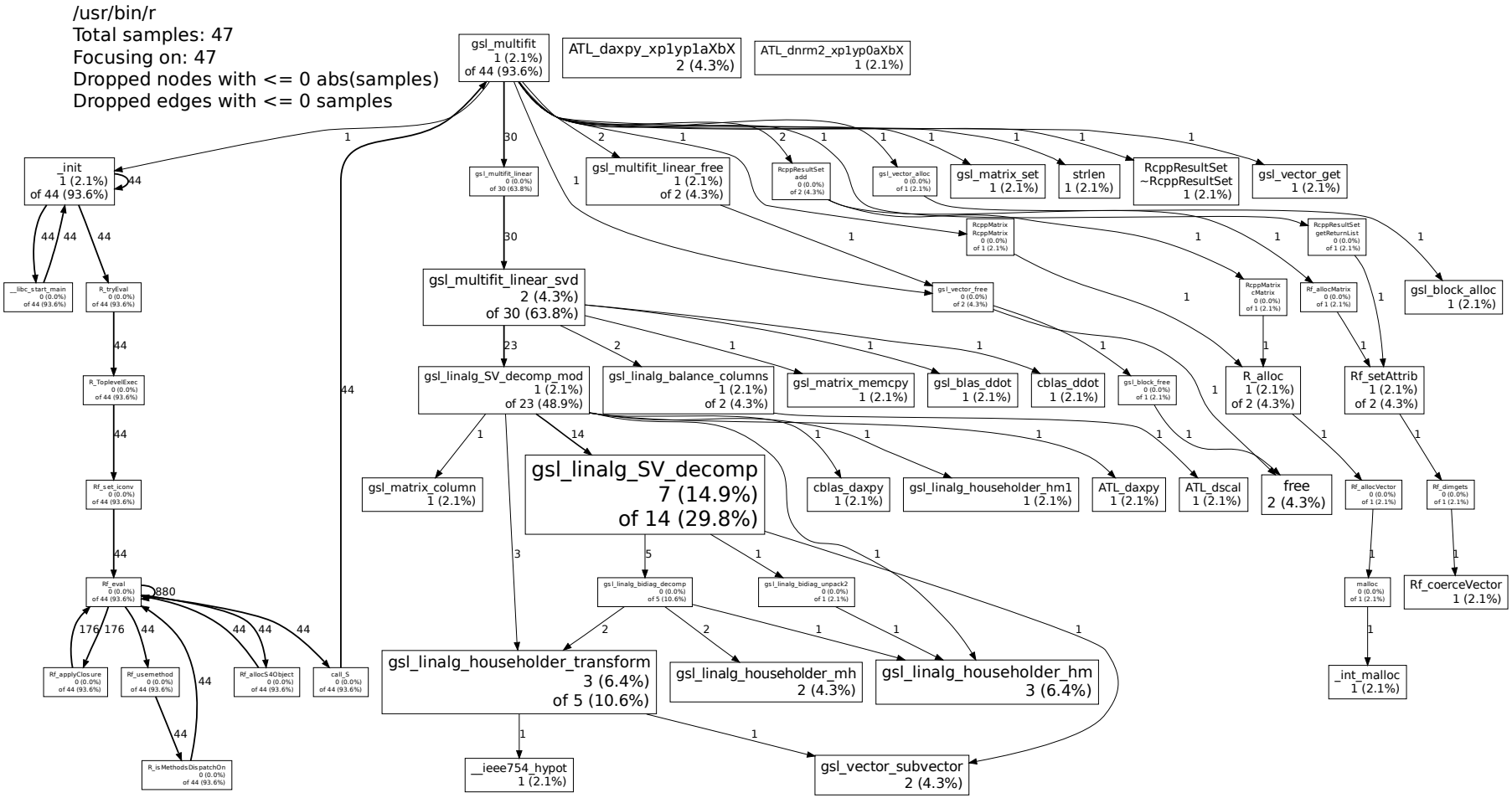
at end before returning. If we then call this function just once from R as in

```
print(system.time(invisible(val <- .Call("gsl_multifit", X, y)
```

we can then call the profiling tools on the output:

```
google-pprof --gv /usr/bin/r /tmp/ols.profile
```

Revisiting profiling



Rcpp and package building

Two tips for easing builds with `Rcpp`:

For command-line use, a shortcut is to copy `Rcpp.h` to `/usr/local/include`, and `libRcpp.so` to `/usr/local/lib`. The earlier example reduces to

```
R CMD SHLIB dd.rcpp.cpp
```

as header and library will be found in the default locations.

For package building, we can have a file `src/Makevars` with

```
# compile flag providing header directory
PKG_CXXFLAGS=`Rscript -e 'Rcpp:::CxxFlags()'`
# link flag providing libary and path
PKG_LIBS=`Rscript -e 'Rcpp:::LdFlags()'`
```

See `help(Rcpp-package)` for more details.

RInside and bringing R to C++

Sometimes we may want to go the other way and add R to and existing C++ project.

This can be simplified using `RInside`:

```
1  #include "RInside.h"                    // for the embedded R via RInside
2  #include "Rcpp.h"                       // for the R / Cpp interface
3
4  int main(int argc, char *argv[]) {
5
6      RInside R(argc, argv);              // create an embedded R instance
7
8      std::string txt = "Hello, world!\n";// assign a standard C++ string to 'txt'
9      R.assign( txt, "txt");              // assign string var to R variable 'txt'
10
11     std::string evalstr = "cat(txt)";
12     R.parseEvalQ(evalstr);              // eval the init string, ignoring any returns
13
14     exit(0);
15 }
```

RInside and bringing R to C++ (cont)

```
#include "RInside.h"                    // for the embedded R via RInside
#include "Rcpp.h"                       // for the R / Cpp interface used for transfer

std::vector< std::vector< double > > createMatrix(const int n) {
    std::vector< std::vector< double > > mat;
    for (int i=0; i<n; i++) {
        std::vector<double> row;
        for (int j=0; j<n; j++) row.push_back((i*10+j));
        mat.push_back(row);
    }
    return(mat);
}

int main(int argc, char *argv[]) {
    const int mdim = 4;
    std::string evalstr = "cat('Running ls()\n'); print(ls()); \
        cat('Showing M\n'); print(M); cat('Showing colSums()\n'); \
        Z <- colSums(M); print(Z); Z";  ## returns Z
    RInside R(argc, argv);
    SEXP ans;
    std::vector< std::vector< double > > myMatrix = createMatrix(mdim);

    R.assign( myMatrix, "M");               // assign STL matrix to R's 'M' var
    R.parseEval(evalstr, ans);              // eval the init string -- Z is now in ans
    RcppVector<double> vec(ans);            // now vec contains Z via ans
    vector<double> v = vec.stlVector();     // convert RcppVector to STL vector

    for (unsigned int i=0; i< v.size(); i++)
        std::cout << "In C++ element " << i << " is " << v[i] << std::endl;
    exit(0);
}
```

Debugging example: valgrind

Analysis of compiled code is mainly undertaken with a debugger like `gdb`, or a graphical frontend like `ddd`.

Another useful tool is `valgrind` which can find memory leaks. We can illustrate its use with a recent real-life example.

`RMySQL` had recently been found to be leaking memory when database connections are being established and closed. Given how `RPostgreSQL` shares a common heritage, it seemed like a good idea to check.

Debugging example: valgrind

We create a small test script which opens and closes a connection to the database in a loop and sends a small 'select' query. We can run this in a way that is close to the suggested use from the 'R Extensions' manual:

```
R -d "valgrind -tool=memcheck -leak-check=full"
-vanilla < valgrindTest.R
```

which creates copious output, including what is on the next slide.

Given the source file and line number, it is fairly straightforward to locate the source of error: a vector of pointers was freed without freeing the individual entries first.

Debugging example: valgrind

The state before the fix:

```
[...]
#==21642== 2,991 bytes in 299 blocks are definitely lost in loss record 34 of 47
#==21642==    at 0x4023D6E: malloc (vg_replace_malloc.c:207)
#==21642==    by 0x6781CAF: RS_DBI_copyString (RS-DBI.c:592)
#==21642==    by 0x6784B91: RS_PostgreSQL_createDataMappings (RS-PostgreSQL.c:400)
#==21642==    by 0x6785191: RS_PostgreSQL_exec (RS-PostgreSQL.c:366)
#==21642==    by 0x40C50BB: (within /usr/lib/R/lib/libR.so)
#==21642==    by 0x40EDD49: Rf_eval (in /usr/lib/R/lib/libR.so)
#==21642==    by 0x40F00DC: (within /usr/lib/R/lib/libR.so)
#==21642==    by 0x40EDA74: Rf_eval (in /usr/lib/R/lib/libR.so)
#==21642==    by 0x40F0186: (within /usr/lib/R/lib/libR.so)
#==21642==    by 0x40EDA74: Rf_eval (in /usr/lib/R/lib/libR.so)
#==21642==    by 0x40F16E6: Rf_applyClosure (in /usr/lib/R/lib/libR.so)
#==21642==    by 0x40ED99A: Rf_eval (in /usr/lib/R/lib/libR.so)
#==21642==
#==21642== LEAK SUMMARY:
#==21642==    definitely lost: 3,063 bytes in 301 blocks.
#==21642==    indirectly lost: 240 bytes in 20 blocks.
#==21642==      possibly lost: 9 bytes in 1 blocks.
#==21642==    still reachable: 13,800,378 bytes in 8,420 blocks.
#==21642==         suppressed: 0 bytes in 0 blocks.
#==21642== Reachable blocks (those to which a pointer was found) are not shown.
#==21642== To see them, rerun with: --leak-check=full --show-reachable=yes
```

Debugging example: valgrind

The state after the fix:

```
[...]
#==3820==
#==3820== 312 (72 direct, 240 indirect) bytes in 2 blocks are definitely lost in loss record 14 of 45
#==3820==    at 0x4023D6E: malloc (vg_replace_malloc.c:207)
#==3820==    by 0x43F1563: nss_parse_service_list (nsswitch.c:530)
#==3820==    by 0x43F1CC3: __nss_database_lookup (nsswitch.c:134)
#==3820==    by 0x445EF4B: ???
#==3820==    by 0x445FCEC: ???
#==3820==    by 0x43AB0F1: getpwuid_r@@GLIBC_2.1.2 (getXXbyYY_r.c:226)
#==3820==    by 0x43AAA76: getpwuid (getXXbyYY.c:116)
#==3820==    by 0x4149412: (within /usr/lib/R/lib/libR.so)
#==3820==    by 0x412779D: (within /usr/lib/R/lib/libR.so)
#==3820==    by 0x40EDA74: Rf_eval (in /usr/lib/R/lib/libR.so)
#==3820==    by 0x40F00DC: (within /usr/lib/R/lib/libR.so)
#==3820==    by 0x40EDA74: Rf_eval (in /usr/lib/R/lib/libR.so)
#==3820==
#==3820== LEAK SUMMARY:
#==3820==    definitely lost: 72 bytes in 2 blocks.
#==3820==    indirectly lost: 240 bytes in 20 blocks.
#==3820==      possibly lost: 0 bytes in 0 blocks.
#==3820==    still reachable: 13,800,378 bytes in 8,420 blocks.
#==3820==         suppressed: 0 bytes in 0 blocks.
#==3820== Reachable blocks (those to which a pointer was found) are not shown.
#==3820== To see them, rerun with: --leak-check=full --show-reachable=yes
```

showing that we recovered 3000 bytes.

Outline

Wrapping up

In this tutorial session, we covered

- *profiling* and tools for *visualising profiling* output
- gaining speed using *vectorisation*
- gaining speed using *Ra* and *just-in-time* compilation
- how to link R to compiled code using tools like *inline* and *Rcpp*
- how to embed R in C++ programs

Wrapping up

Things we have not covered:

- running R code in *parallel* using MPI, nws, snow, ...
- computing with *data beyond the R memory limit* by using biglm, ff or bigmatrix9
- scripting and automation using littler

Wrapping up

Further questions ?

Two good resources are

- the mailing list `r-sig-hpc` on HPC with R,
- and the `HighPerformanceComputing` task view on CRAN.

Scripts are at `http://dirk.eddelbuettel.com/code/hpcR/`.

Lastly, don't hesitate to email me at `edd@debian.org`