

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.

We will have a detailed discussion of several ways to get more things done at the same time by using simple *parallel computing* approaches.

We also look at ways to *automate and script* running R code.



Table of Contents

- 1 Motivation
- 2 Measuring and profiling
- 3 Vectorisation
- 4 Just-in-time compilation
- 5 BLAS and GPUs
- 6 Compiled Code
- 7 Parallel execution: Explicitly and Implicitly
- 8 Automation and scripting
- 9 Summary



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)

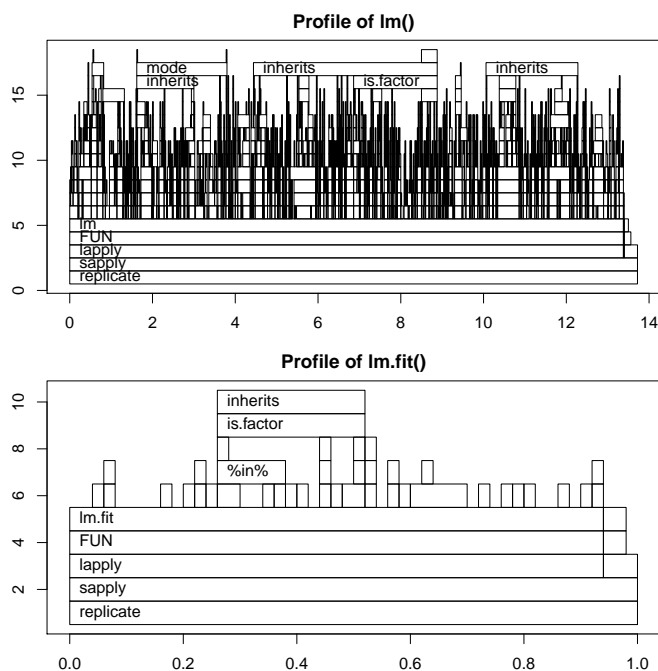
```
R CMD Prof longley.lm.out | less
```

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.



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.

Source: Our calculations.



RProf example cont.

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

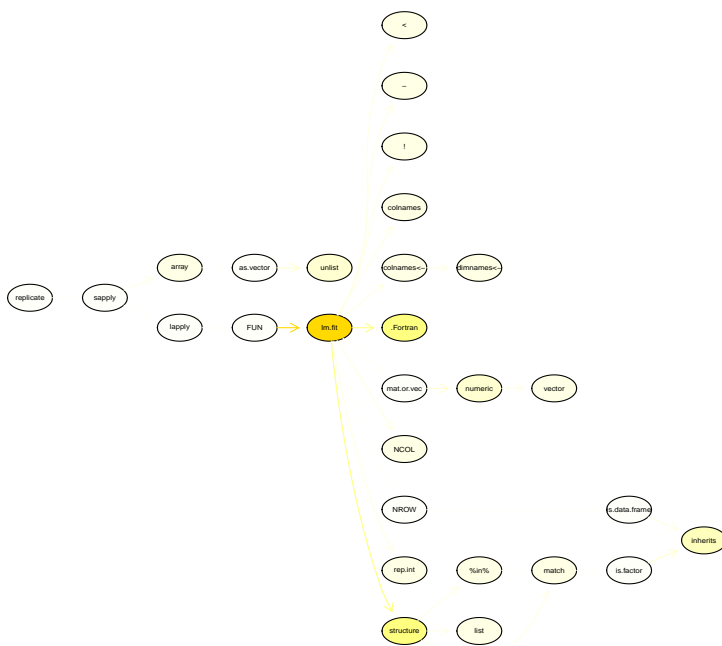
The `flatProfile` function aggregates the data, optionally with totals.

```
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:

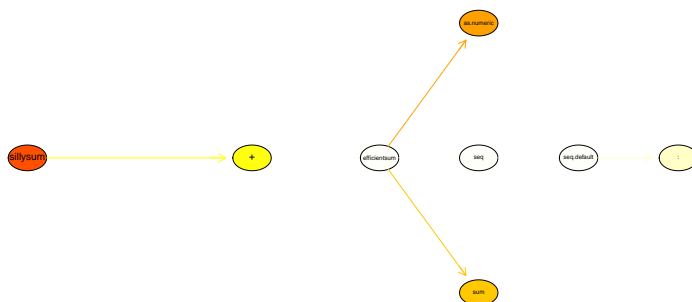
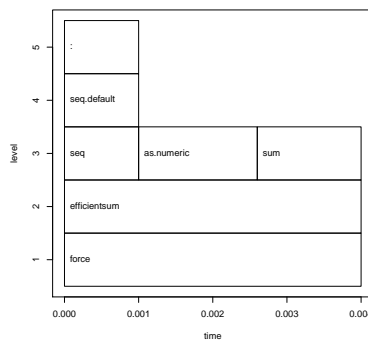
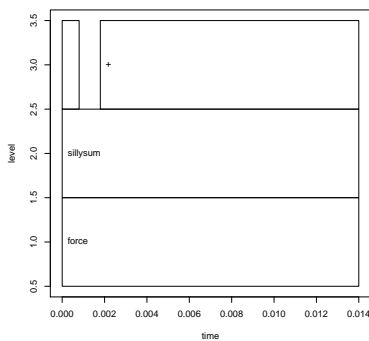
```
sillysum <- function(N) {s <- 0;
  for (i in 1:N) s <- s + i; s}
ival <- 1/5000
plot(profr(a <- sillysum(1e6), ival))
```

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
```

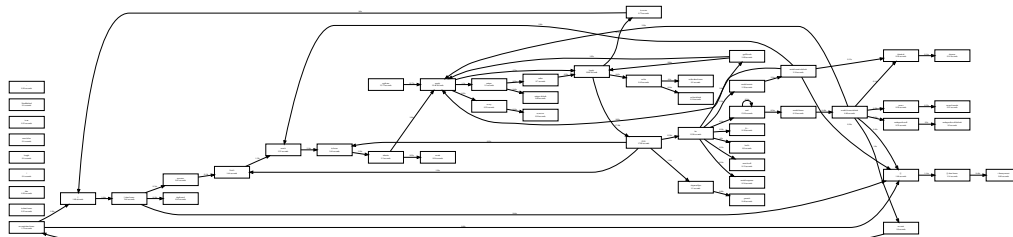
Its key advantages are the ability to include, exclude or restrict functions.

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

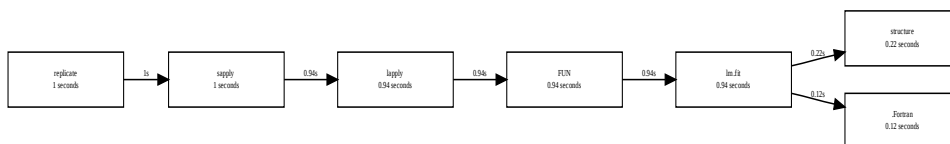


Additional profiling visualizations (cont.)

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 `-pg` 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
```

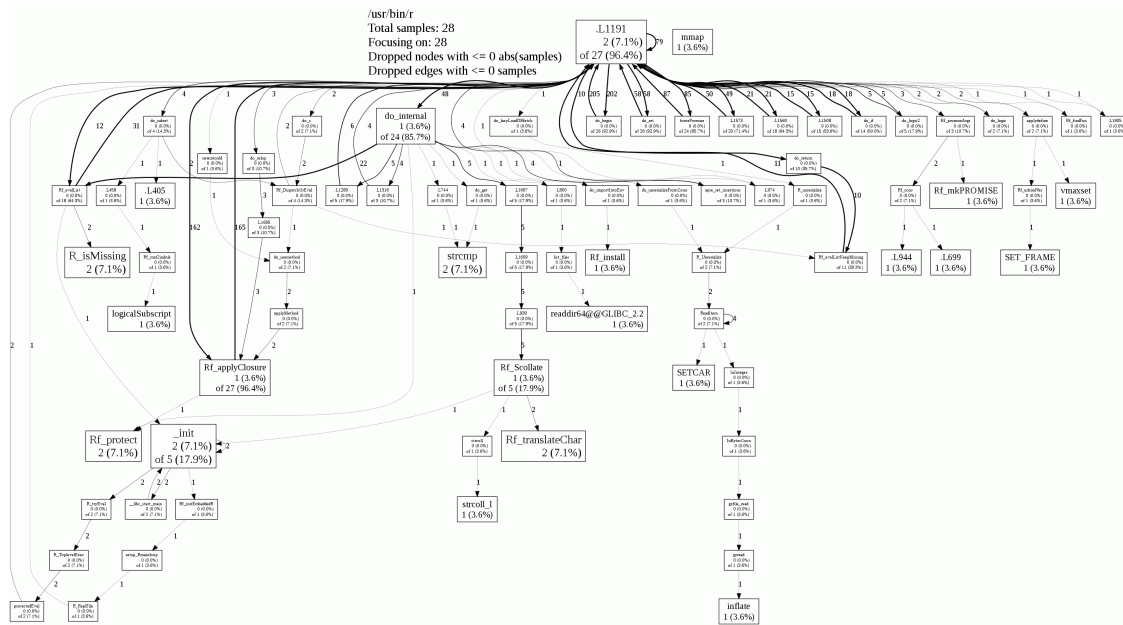
```
# or analyse call graph using gv
google-pprof --gv /usr/bin/r /tmp/rprof.log
```

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 format is used by 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).

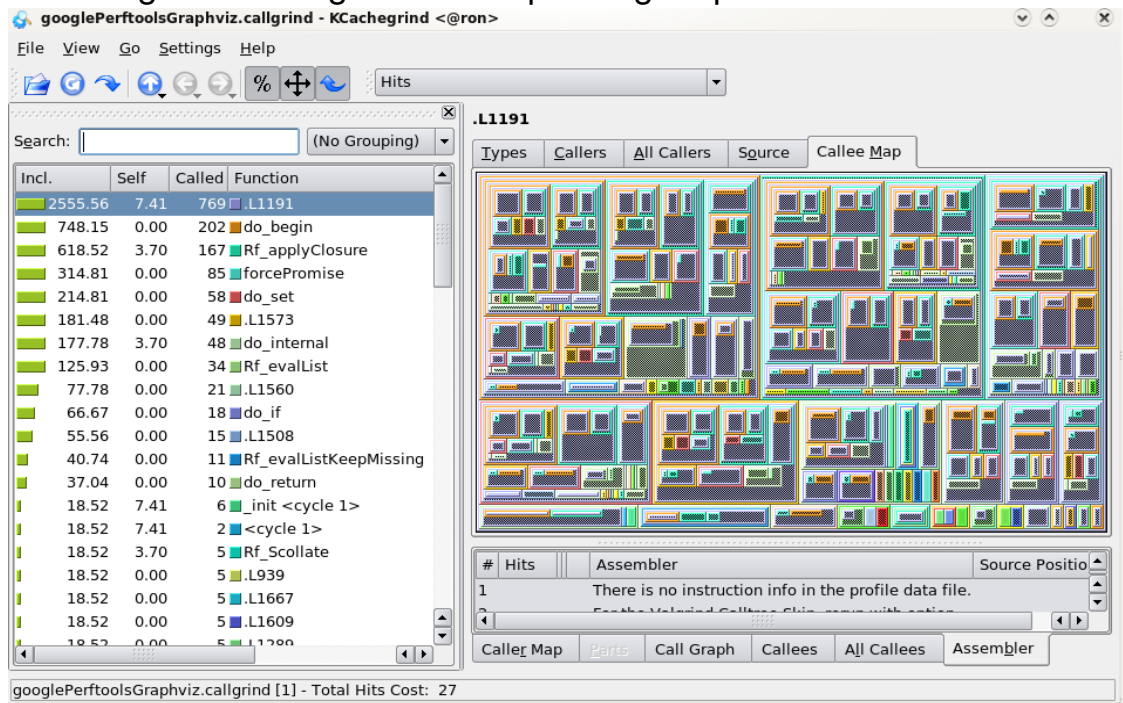
For example, the KDE frontend *kcachegrind* can be used to visualize the profiler output as follows:

```
google-perfprof --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()`.

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 2×2 matrix where the entries are independent and uniformly distributed digits $0, 1, \dots, 9$. 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`.



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:

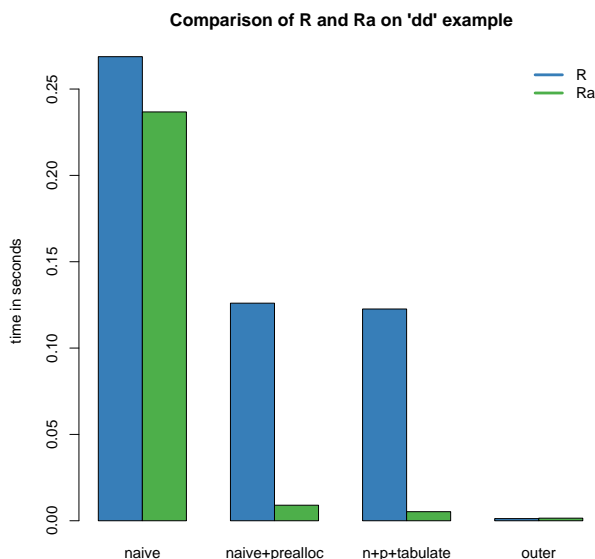
```
dd.for.pre.tabulate.jit <- function() {
  jit(1)
  val <- double(10000)
  i <- 0
  for (a in 0:9) for (b in 0:9)
    for (d in 0:9) for (e in 0:9) {
      val[i <- i + 1] <- a*b - d*e
    }
  tabulate(val)
}

> 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.



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 one 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-optimised 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 be very few easy-to-install and easy-to-use kits for R – the `gputools` packages is a first, providing a few accelerated data-mining functions.

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



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

```

1 conv <- function(a, b)
2   .C("convolve",
3     as.double(a),
4     as.integer(length(a)),
5     as.double(b),
6     as.integer(length(b)),
7     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.

```

1 ## A simple Fortran example
2 code <- "
3     integer i
4     do 1 i=1, n(1)
5     1 x(i) = x(i)**3
6 "
7 cubefn <- cfunction(signature(n="integer", x="numeric"),
8                    code, convention=".Fortran")
9 x <- as.numeric(1:10)
10 n <- as.integer(10)
11 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.Fortran.R | R -no-save`.



Compiled Code: inline cont.

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

```

1 ## Use of .Call convention with C code
2 ## Multiplying 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

```

1 code <- "
2   if (isNumeric(vec)) {
3     int *pv = INTEGER(vec);
4     int n = length(vec);
5     if (n = 10000) {
6       int i = 0;
7       for (int a = 0; a < 9; a++)
8         for (int b = 0; b < 9; b++)
9           for (int c = 0; c < 9; c++)
10            for (int d = 0; d < 9; d++)
11              pv[i++] = a*b - c*d;
12     }
13   }
14   return(vec);
15 "
16
17 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 r1 = 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   r1 = rs.getReturnList();      // Get the list to be returned to R.
18
19   return r1;
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 r1 = 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    r1 = 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 r1;
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 negligible amount which is probably due to some overhead 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.getDoubleValue("fltnb");
string   txt = param.getStringValue("txt");
bool     flg = param.getBoolValue("bool");
RcppDate dt = param.getDateValue("date");
```



Basic Rcpp usage (cont.)

Similarly, we can construct vectors and matrices 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 converting 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 r1=R_NilValue;
9   char *exceptionMsg=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.)

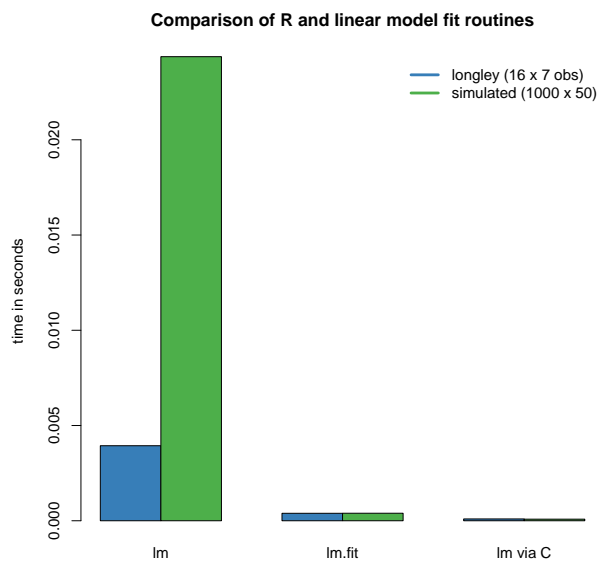
```

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.)



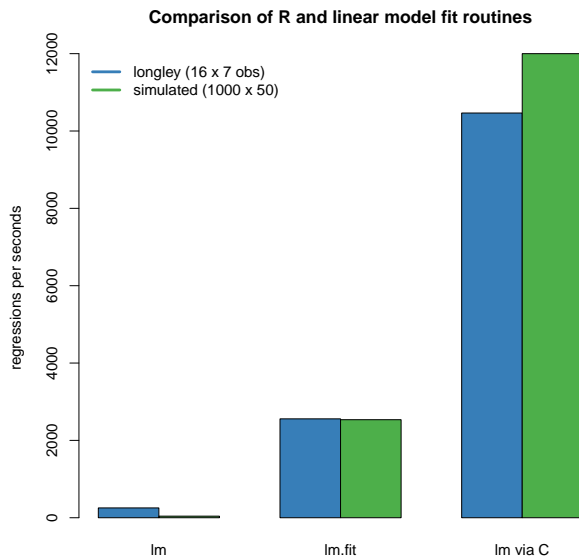
Source: Our calculations

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

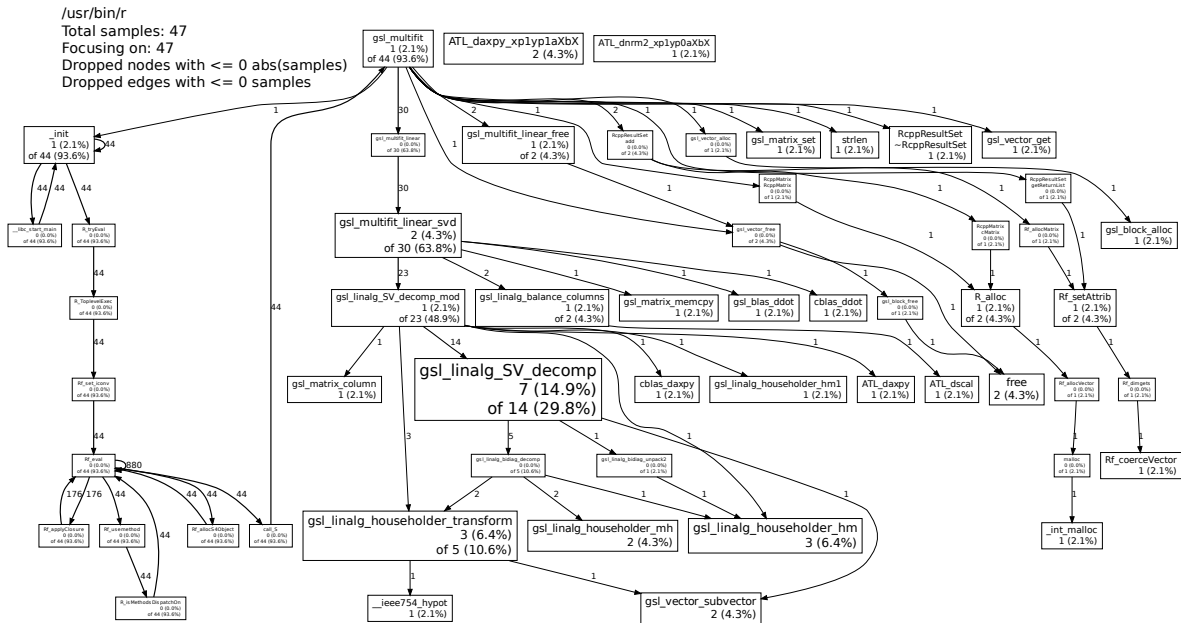
```
print(system.time(invisible(val <- .Call("gsl_multifit", X, y))))
```

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 library and path
PKG_LIBS='Rscript -e 'Rcpp::LdFlags()'`
```

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



RInside and bringing to C++

Sometimes we may want to go the other way and add R to an 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 to C++ (cont)

```

1 #include "RInside.h" // for the embedded R via RInside
2 #include "Rcpp.h" // for the R / Cpp interface used for transfer
3
4 std::vector< std::vector< double > > createMatrix(const int n) {
5     std::vector< std::vector< double > > mat;
6     for (int i=0; i<n; i++) {
7         std::vector<double> row;
8         for (int j=0; j<n; j++) row.push_back((i*10+j));
9         mat.push_back(row);
10    }
11    return(mat);
12 }
13
14 int main(int argc, char *argv[]) {
15     const int mdim = 4;
16     std::string evalstr = "cat('Running ls()\n'); print(ls()); \
17         cat('Showing M\n'); print(M); cat('Showing colSums()\n'); \
18         Z <- colSums(M); print(Z); Z"; ## returns Z
19     RInside R(argc, argv);
20     SEXP ans;
21     std::vector< std::vector< double > > myMatrix = createMatrix(mdim);
22
23     R.assign( myMatrix, "M"); // assign STL matrix to R's 'M' var
24     R.parseEval(evalstr, ans); // eval the init string — Z is now in ans
25     RcppVector<double> vec(ans); // now vec contains Z via ans
26     vector<double> v = vec.stlVector(); // convert RcppVector to STL vector
27
28     for (unsigned int i=0; i< v.size(); i++)
29         std::cout << "In C++ element " << i << " is " << v[i] << std::endl;
30     exit(0);
31 }

```



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.



Embarassingly parallel

Several R packages on CRAN provide the ability to execute code in parallel:

- NWS
- Rmpi
- snow (using MPI, PVM, NWS or sockets)
- papply
- taskPR
- multicore

A recent paper by Schmidberger, Morgan, Edelbuettel, Yu, Tierney and Mansmann (JSS, 2009) provides a survey of this field.



NWS Intro

NWS, or NetWorkSpaces, is an alternative to MPI (which we discuss below). Based on Python, it may be easier to install (in case administrator rights are unavailable) and use than MPI. It is accessible from R, Python and Matlab. It is also cross-platform.

NWS is available via [Sourceforge](#) as well as [CRAN](#). An introductory article (focussing on Python) appeared last summer in [Dr. Dobb's](#).

On Debian and Ubuntu, installing the `python-nwserver` package on at least the server node, and installing `r-cran-nws` on each client is all that is needed. Other system may need to install the `twisted` framework for `Python` first.



NWS data store example

A simple example, adapted from one of the package demos:

```
ws <- netWorkspace('r place') # create a 'value store'
nwsStore(ws, 'x', 1)          # place a value (as fifo)

cat(nwsListVars(ws), "\n")   # we can list
nwsFind(ws, 'x')             # and lookup
nwsStore(ws, 'x', 2)        # and overwrite
cat(nwsListVars(ws), "\n")   # now see two entries

cat(nwsFetch(ws, 'x'), '\n') # we can fetch
cat(nwsFetch(ws, 'x'), '\n') # we can fetch
cat(nwsListVars(ws), '\n')   # and none left

cat(nwsFetchTry(ws, 'x', 'no go'), '\n') # can't fetch
```

The script `nwsVariableStore.r` contains this and a few more commands.



NWS sleigh example

The NWS component sleigh is an R class that makes it very easy to write simple parallel programs. Sleigh uses the master/worker paradigm: The master submits tasks to the workers, who may or may not be on the same machine as the master.

```
# create a sleigh object on two nodes using ssh
s <- sleigh(nodeList=c("joe", "ron"), launch=sshcmd)

# execute a statement on each worker node
eachWorker(s, function() x <- 1)

# get system info from each worker
eachWorker(s, Sys.info)

# run a lapply-style funct. over each list elem.
eachElem(s, function(x) {x+1}, list(1:10))

stopSleigh(s)
```



NWS sleigh cont.

Also of note is the extended `caretNWS` version of `caret` by Max Kuhn, and described in a recent Journal of Statistical Software article.

`caret` (short for 'Classification and Regression Training') provides a consistent interface for dozens of modern regression and classification techniques.

`caretNWS` uses `nws` and `sleigh` to execute embarrassingly parallel tasks: bagging, boosting, cross-validation, bootstrapping, ... This is all done 'behind the scenes' and thus easy to deploy.

Schmidberger et al find NWS to be competitive with the other parallel methods for non-degenerate cases where the ratio between communication and computation is balanced.



Rmpi

`Rmpi` is a CRAN package that provides an interface between R and the Message Passing Interface (MPI), a standard for parallel computing. (c.f. [Wikipedia](#) for more and links to the Open MPI and MPICH2 projects for implementations).

The preferred implementation for MPI is now [Open MPI](#). However, the older LAM implementation can be used on those platforms where Open MPI is unavailable. There is also an alternate implementation called MPICH2. Lastly, we should also mention the similar Parallel Virtual Machine (PVM) tool; see its [Wikipedia](#) page for more.

`Rmpi` allows us to use MPI directly from R and comes with several examples. However, we will focus on the higher-level usage via `snow`.



MPI Example

Let us look at the [MPI](#) variant of the 'Hello, World!' program:

```

1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main(int argc, char** argv)
5 {
6     int rank, size, nameLen;
7     char processorName[MPI_MAX_PROCESSOR_NAME];
8
9     MPI_Init(&argc, &argv);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13    MPI_Get_processor_name(processorName, &nameLen);
14
15    printf("Hello, rank %d, size %d on processor %s\n",
16          rank, size, processorName);
17
18    MPI_Finalize();
19    return 0;
20 }

```



MPI Example: cont.

We can compile the previous example via

```
$ mpicc -o mpiHelloWorld mpiHelloWorld.c
```

If it has been copied across several Open MPI-equipped hosts, we can execute it N times on the M listed hosts via:

```
$ orterun -H ron,joe,tony,mccoy -n 8 /tmp/mpiHelloWorld
```

```

Hello, rank 0, size 8 on processor ron
Hello, rank 4, size 8 on processor ron
Hello, rank 7, size 8 on processor mccoy
Hello, rank 3, size 8 on processor mccoy
Hello, rank 2, size 8 on processor tony
Hello, rank 5, size 8 on processor joe
Hello, rank 6, size 8 on processor tony
Hello, rank 1, size 8 on processor joe

```

Notice how the order of execution is indeterminate.



MPI Example: cont.

Besides `orterun` (which replaces the `mpirun` command used by other MPI implementations), Open MPI also supplies `ompi_info` to query parameter settings.

Open MPI has very fine-grained configuration options that permit e.g. attaching particular jobs to particular cpus or cores.

Detailed documentation is provided at the web site <http://www.openmpi.org>.

We will concentrate on using MPI via the `Rmpi` package.



Rmpi

`Rmpi`, a CRAN package by Hao Yu, wraps many of the MPI API calls for use by R.

The preceding example can be rewritten in R as

```

1 #!/usr/bin/env r
2
3 library(Rmpi) # calls MPI_Init
4
5 rk <- mpi.comm.rank(0)
6 sz <- mpi.comm.size(0)
7 name <- mpi.get.processor.name()
8 cat("Hello, rank", rk, "size", sz, "on", name, "\n")

```



Rmpi: cont.

```
$ orterun -H ron,joe,tony,mccoy -n 8 \
/tmp/mpiHelloWorld.r

Hello, rank 4 size 8 on ron
Hello, rank 0 size 8 on ron
Hello, rank 3 size 8 on mccoy
Hello, rank 7 size 8 on mccoy
Hello, rank Hello, rank 21 size 8 on joe
size 8 on tony
Hello, rank 6 size 8 on tony
Hello, rank 5 size 8 on joe
```



Rmpi: cont.

We can also execute this as a one-liner using `r` (which we discuss later):

```
$ orterun -n 8 -H ron,joe,tony,mccoy \
  r -lRmpi -e'cat("Hello", \
    mpi.comm.rank(0), "of", \
    mpi.comm.size(0), "on", \
    mpi.get.processor.name(), "\n");
    mpi.quit()'
```

```
Hello 4 of 8 on ron
Hello 3 of 8 on mccoy
Hello 7 of 8 on mccoy
Hello 0 of 8 on ron
HelloHello 2 of 8 on tony
  Hello 1 of 8 on joe
Hello 5 of 8 on joe
6 of 8 on tony
```



Rmpi: cont.

`Rmpi` offers a large number functions, mirroring the rich API provided by MPI.

`Rmpi` also offers extensions specific to working with R and its objects, including a set of `apply`-style functions to spread load across the worker nodes.

However, we will use `Rmpi` mostly indirectly via `snow`.



snow

The `snow` package by Tierney et al provides a convenient abstraction directly from R.

It can be used to initialize and use a compute cluster using one of the available methods direct socket connections, MPI, PVM, or (since the most recent release), NWS. We will focus on MPI.

A simple example:

```
cl <- makeCluster(4, "MPI")
print(clusterCall(cl, function() \
  Sys.info()[c("nodename", "machine")]))
stopCluster(cl)
```

which we can as a one-liner as shown on the next slide.



snow: Example

```
$ orterun -n 1 -H ron,joe,tony,mccoy r -lsnow,Rmpi \
  -e'cl <- makeCluster(4, "MPI"); \
    res <- clusterCall(cl, \
      function() Sys.info()["nodename"]); \
    print(do.call(rbind,res)); \
    stopCluster(cl); mpi.quit()'
```

4 slaves are spawned successfully. 0 failed.

```
nodename
[1,] "joe"
[2,] "tony"
[3,] "mccoy"
[4,] "ron"
```

Note that we told `orterun` to start on only one node – as `snow` then starts four instances (which are split evenly over the four given hosts).



snow: Example cont.

The power of `snow` lies in the ability to use the `apply`-style paradigm over a cluster of machines:

```
params <- c("A", "B", "C", "D", "E", "F", "G", "H")
cl <- makeCluster(4, "MPI")
res <- parSapply(cl, params, \
  FUN=function(x) myBigFunction(x))
```

will 'unroll' the parameters `params` one-each over the function argument given, utilising the cluster `cl`. In other words, we will be running four copies of `myBigFunction()` at once.

So the `snow` package provides a unifying framework for parallelly executed `apply` functions.

We will come back to more examples with `snow` below.



papply, biopara and taskPR

We saw that `Rmpi` and `NWS` have `apply`-style functions, and that `snow` provides a unified layer. `papply` is another CRAN package that wraps around `Rmpi` to distribute processing of `apply`-style functions across a cluster.

However, using the Open MPI-based `Rmpi` package, I was not able to get `papply` to actually successfully distribute – and retrieve – results across a cluster. So `snow` remains the preferred wrapper.

`biopara` is another package to distribute load across a cluster using direct socket-based communication. We consider `snow` to be a more general-purpose package for the same task.

`taskPR` uses the `MPI` protocol directly rather than via `Rmpi`. It is however hard-wired to use LAM and failed to launch under the Open MPI-implementation.



slurm resource management and queue system

Once the number of compute nodes increases, it becomes important to be able to allocate and manage resources, and to queue and batch jobs. A suitable tool is `slurm`, an open-source resource manager for Linux clusters.

Paraphrasing from the `slurm` website:

- it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users;
- it provides a framework for starting, executing, and monitoring (typically parallel) work on a set of allocated nodes.
- it arbitrates contention for resources by managing a queue of pending work.

`Slurm` is being developed by a consortium including LLNL, HP, Bull, and Linux Networks.



slurm example

Slurm is rather rich in features; we will only scratch the surface here.

Slurm can use many underlying message passing / communications protocols, and MPI is well supported.

In particular, Open MPI works well with slurm. That is an advantage inasmuch as it permits use of `Rmpi`.



slurm example

A simple example:

```
$ srun -N 2 r -lRmpi -e'cat("Hello", \
    mpi.comm.rank(0), "of", \
    mpi.comm.size(0), "on", \
    mpi.get.processor.name(), "\n")'
```

```
Hello 0 of 1 on ron
```

```
Hello 0 of 1 on joe
```

```
$ srun -n 4 -N 2 -O r -lRmpi -e'cat("Hello", \
    mpi.comm.rank(0), "of", \
    mpi.comm.size(0), "on", \
    mpi.get.processor.name(), "\n")'
```

```
Hello 0 of 1 on ron
```

```
Hello 0 of 1 on ron
```

```
Hello 0 of 1 on joe
```

```
Hello 0 of 1 on joe
```

This shows how to *overcommit* jobs per node, and provides an example where we set the number of worker instances on the command-line.



slurm example

Additional command-line tools of interest are `salloc`, `sbatch`, `scontrol`, `squeue`, `scancel` and `sinfo`. For example, to see the status of a compute cluster:

```
$ sinfo
```

```
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*      up    infinite    2    idle mccoymccoy,ron
```

This shows two idle nodes in a partition with the default name 'debug'.

The `sview` graphical user interface combines the functionality of a few of the command-line tools.

A more complete example will be provided below.



Using all those cores

Multi-core hardware is now a default, and the number of cores per cpus will only increase. It is therefore becoming more important for software to take advantage of these features.

Two recent (and still 'experimental') packages by Luke Tierney are addressing this question:

- `pnmath` uses OpenMP compiler directives for parallel code;
- `pnmath0` uses pthreads and implements the same interface.

They can be found at

<http://www.stat.uiowa.edu/~luke/R/experimental/>

More recently, Simon Urbanek released the 'multicore' package which provides parallel execution of R code on machines with multiple cores or cpu. All jobs share the full state of R when spawned, no data or code needs to be initialized. This makes the actual spawning very fast since no new R instance needs to be started.



pnmath and pnmath0

Both `pnmath` and `pnmath0` provide parallelized vector math functions and support routines.

Upon loading either package, a number of vector math functions are replaced with versions that are parallelized using OpenMP. The functions will be run using multiple threads if their results will be long enough for the parallel overhead to be outweighed by the parallel gains. On load a calibration calculation is carried out to assess the parallel overhead and adjust these thresholds.

Profiling is probably the best way to assess the possible usefulness. As a quick illustration, we compute the `qtukey` function on a eight-core machine:



pnmath and pnmath0 illustration

```
$ r -e'N=1e3;print(system.time(qtukey(seq(1,N)/N,2,2)))'
  user  system elapsed
66.590   0.000  66.649

$ r -lpnmath -e'N=1e3; \
print(system.time(qtukey(seq(1,N)/N,2,2)))'
  user  system elapsed
67.580   0.080   9.938

$ r -lpnmath0 -e'N=1e3; \
print(system.time(qtukey(seq(1,N)/N,2,2)))'
  user  system elapsed
68.230   0.010   9.983
```

The 6.7-fold reduction in 'elapsed' time shows that the multithreaded version takes advantage of the 8 available cores at a sub-linear fashion as some communications overhead is involved.

These improvements will likely be folded into future R versions.



multicore

The multicore package provides two main interfaces:

- `mclapply`, a parallel / multicore version of `lapply`
- the functions `parallel` and `collect` to launch parallel execution and gather results at end

For setups in which a sufficient number of cores is available without requiring network traffic, multicore is likely to be a very compelling package.

Given that future cpu generation will offer 16, 32 or more cores, this package may become increasingly popular.

One thing to note is that 'anything but Windows' is required to take advantage of multicore.



Scripting example for R and slurm

Being able to launch numerous R jobs in a parallel environments is helped by the ability to 'script' R.

Several simple methods existed to start R:

- `R CMD BATCH file.R`
- `echo "commands" | R -no-save`
- `R -no-save < file.R > file.Rout`

These are suitable for one-off scripts, but may be too fragile for distributed computing.



Use scripting with r

The `r` command of the `littler` package (as well as R's `Rscript`) provide more robust alternatives.

`r` can also be used four different ways:

- `r file.R`
- `echo "commands" | r`
- `r -lRmpi -e 'cat("Hello", mpi.get.processor.name())'`
- and *shebang*-style in script files: `#!/usr/bin/r`

It is the last point that is of particular interest in this HPC context. Also of note is the availability of the `getopt` package on CRAN.



slurm and snow

Having introduced `snow`, `slurm` and `r`, we would like to combine them.

However, there is are problems:

- `snow` has a master/worker paradigm yet `slurm` launches its nodes symmetrically,
- `slurm`'s `srun` has limits in spawning jobs
- with `srun`, we cannot communicate the number of nodes 'dynamically' into the script: `snow`'s cluster creation needs a hardwired number of nodes



slurm and snow solution

`snow` solves the master / worker problem by auto-discovery upon startup. The package contains two internal files `RMPISNOW` and `RMPISNOWprofile` that use a combination of shell and R code to determine the node identity allowing it to switch to master or worker functionality.

We can reduce the same problem to this for our R script:

```
mpirank <- mpi.comm.rank(0)
if (mpirank == 0) {                               # are we the master ?
  makeMPIcluster()
} else {                                          # or are we a slave ?
  sink(file="/dev/null")
  slaveLoop(makeMPImaster())
  q()
}
```



slurm and snow solution

For example

```
1 #!/usr/bin/env r
2
3 suppressMessages(library(Rmpi))
4 suppressMessages(library(snow))
5
6 mpirank <- mpi.comm.rank(0)
7 if (mpirank == 0) {
8   cat("Launching master, mpi rank=", mpirank, "\n")
9   makeMPIcluster()
10 } else {                                     # or are we a slave ?
11   cat("Launching slave with, mpi rank=", mpirank, "\n")
12   sink(file="/dev/null")
13   slaveLoop(makeMPImaster())
14   mpi.finalize()
15   q()
16 }
17
18 stopCluster(cl)
```



slurm and snow solution

The example creates

```
$ orterun -H ron,joe,tony,mccoy -n 4 mpiSnowSimple.r
```

```
Launching slave 2
```

```
Launching master 0
```

```
Launching slave 1
```

```
Launching slave 3
```

and we see that $N - 1$ workers are running with one instance running as the coordinating manager node.



salloc for snow

The other important aspect is to switch to `salloc` (which will call `orterun`) instead of `srun`.

We can either supply the hosts used using the `-w` switch, or rely on the `slurm.conf` file.

But importantly, we can govern from the call how many instances we want running (and have neither the `srun` limitation requiring overcommitting nor the hard-coded `snow` cluster-creation size):

```
$ salloc -w ron,mccoy orterun -n 7 mpiSnowSimple.r
```

We ask for a `slurm` allocation on the given hosts, and instruct Open MPI to run seven instances.



salloc for snow

```

1 #!/usr/bin/env r
2
3 suppressMessages(library(Rmpi))
4 suppressMessages(library(snow))
5
6 mpirank <- mpi.comm.rank(0)
7 if (mpirank == 0) {
8     cat("Launching master, mpi rank=", mpirank, "\n")
9     makeMPIcluster()
10 } else {
11     # or are we a slave ?
12     cat("Launching slave with, mpi rank=", mpirank, "\n")
13     sink(file="/dev/null")
14     slaveLoop(makeMPImaster())
15     mpi.finalize()
16     q()
17 }
18 ## a trivial main body, but note how getMPIcluster() learns from the
19 ## launched cluster how many nodes are available
20 cl <- getMPIcluster()
21 clusterEvalQ(cl, options("digits.secs"=3)) ## use millisecond granularity
22 res <- clusterCall(cl, function() paste(format(Sys.time()), Sys.info()["
23     nodename"]))
24 print(do.call(rbind, res))
25 stopCluster(cl)
26 mpi.quit()

```



salloc for snow

```

$ salloc -w ron,joe,tony orterun -n 7 /tmp/mpiSnowSimple.r

salloc: Granted job allocation 39
Launching slave with, mpi rank= 5
Launching slave with, mpi rank= 2
Launching slave with, mpi rank= 6
Launching master, mpi rank= 0
Launching slave with, mpi rank= 3
Launching slave with, mpi rank= 1
Launching slave with, mpi rank= 4
  [,1]
[1,] "2009-06-25 20:51:20.536 joe"
[2,] "2009-06-25 20:51:33.747 tony"
[3,] "2009-06-25 20:51:20.522 ron"
[4,] "2009-06-25 20:51:20.544 joe"
[5,] "2009-06-25 20:51:33.766 tony"
[6,] "2009-06-25 20:51:20.537 ron"
salloc: Relinquishing job allocation 39

```



A complete example

```

cl <- NULL
mpirank <- mpi.comm.rank(0)
if (mpirank == 0) {
  cl <- makeMPIcluster()
} else {
  # or are we a slave?
  sink(file="/dev/null")
  slaveLoop(makeMPImaster())
  mpi.finalize(); q()
}
clusterEvalQ(cl, library(RDieHarder))
res <- parLapply(cl, c("mt19937", "mt19937_1999",
  "mt19937_1998", "R_mersenne_twister"),
  function(x) {
    dieharder(rng=x, test="operm5",
      psamples=100, seed=12345)
  })
stopCluster(cl)
print( do.call(rbind, lapply(res, function(x) { x[[1]] } )))
mpi.quit()

```



A complete example cont.

This uses `RDieHarder` to test four Mersenne-Twister implementations at once.

A simple analysis shows the four charts and prints the four p -values:

```

pdf("/tmp/snowRDH.pdf")
lapply(res, function(x) plot(x))
dev.off()

print( do.call(rbind,
  lapply(res, function(x) { x[[1]] } )))

```



A complete example cont.

```
$ salloc -w ron,joe orterun -n 5 snowRDieharder.r
salloc: Granted job allocation 10
      [,1]
[1,] 0.1443805247
[2,] 0.0022301018
[3,] 0.0001014794
[4,] 0.0061524281
sall: Relinquishing job allocation 10
```



Example summary

We have seen

- how `littler` can help us script R tasks
- how `Rmpi`, `snow` and `slurm` can interact nicely
- a complete example using `RDieHarder` to illustrate these concepts



littler

Both `r` (from the `littler` package) and `Rscript` (included with R) allow us to write simple scripts for repeated tasks.

```
#!/usr/bin/env r
# a simple example to install one or more packages
if (is.null(argv) | length(argv)<1) {
  cat("Usage: install.r pkg1 [pkg2 pkg3 ...]\n")
  q()
}
## adjust as necessary, see help('download.packages')
repos <- "http://cran.us.r-project.org"
lib.loc <- "/usr/local/lib/R/site-library"
install.packages(argv, lib.loc,
                  repos, dependencies=TRUE)
```

If saved as `install.r`, we can call it via

```
$ install.r ff bigmemory
```

The `getopt` package makes it a lot easier for `r` to support command-line options.



Rscript

`Rscript` can be used in a similar fashion.

Previously we had to use

```
$ R --slave < cmdfile.R
$ cat cmdfile.R | R --slave
$ R CMD BATCH cmdfile.R
```

or some shell-script variations around this theme.

By providing `r` and `Rscript`, we can now write 'R scripts' that are executable. This allows for automation in cron jobs, Makefile, job queues, ...



RPy

RPy packages provides access to R from Python:

```

1 from rpy import *
2 set_default_mode(NO_CONVERSION) # avoid automatic conversion
3 r.library("nnet")
4 model = r("Fxy~x+y")
5 df = r.data_frame(x = r.c(0,2,5,10,15)
6                  ,y = r.c(0,2,5,8,10)
7                  ,Fxy = r.c(0,2,5,8,10))
8 NNModel = r.nnet(model, data = df
9                 , size =10, decay =1e-3
10                , lineout=True, skip=True
11                , maxit=1000, Hess =True)
12 XG = r.expand_grid(x = r.seq(0,7,1), y = r.seq(0,7,1))
13 x = r.seq(0,7,1)
14 y = r.seq(0,7,1)
15
16 set_default_mode(BASIC_CONVERSION) # automatic conv. back on
17 fit = r.predict(NNModel,XG)
18 print fit

```



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
- running R code in *parallel* using MPI, nws, snow, ...
- 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

