Introduction to High-Performance Computing with R

Tutorial at *useR! 2010*

Dirk Eddelbuettel, Ph.D.

Dirk.Eddelbuettel@R-Project.org

edd@debian.org

useR! 2010

National Institute of Standards and Technology (NIST)

Gaithersburg, Maryland, USA

Outline

1 Motivation

2 Automation and scripting

3 Measuring and profiling

4 Speeding up

5 Compiled Code

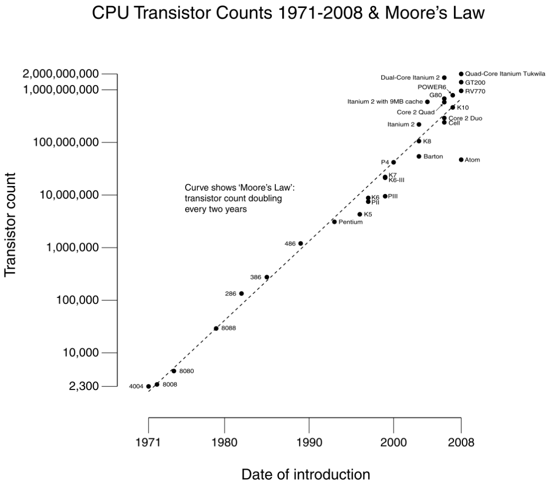
6 Implicitly Parallel

7 Explicitly Parallel

8 Out-of-memory processing

9 Summary

Motivation: What describes our current situation?



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

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

Yet our datasets get *bigger and bigger* and an even faster rate.

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

Result: An urgent need for *high(er) performance computing* with R.

Motivation: Data sets keep growing

There are a number of reasons behind 'big data':

- *more collection*: from faster DNA sequencing to larger experiments to per-item RFID scanning to complex social networks — our ability to *originate* data keeps increasing
- *more networking*: (internet) capacity, transmission speeds and usage keep growing leading to easier ways to assemble data sets from different sources
- *more storage* as what used to be disk capacity is now provided by USB keychains, while data warehousing / data marts are aiming beyond petabytes

Of course, not all large data sets are suitable for R, and data is frequently pruned, filtered or condensed down to *manageable* size (and the meaning of manageable will vary by user).

Motivation: Presentation Roadmap

We look at ways to *'script'* running R code which is helpful for both automation and debugging.

We will *measure* using profiling tools to analyse and visualize performance; we will also glance at debugging tools and tricks.

We will look at *vectorisation*, a key method for speed as well as various ways to *compile and use code* before a brief discussion and example of GPU computing.

Next, we will discuss several ways to get more things done at the same time by using simple *parallel computing* approaches.

We will then look at computations *beyond the memory limits*.

A discussion and question sesssion finishes.

Typographics conventions

R itself is highlighted, packages like `Rmpi` get a different color.

External links to e.g. Wikipedia are clickable in the pdf file.

R input and output in different colors, and usually set flush-left so that can show long lines:

```
cat("Hello\n")
```

```
Hello
```

Source code listings are boxed and with lines numbers

```
cubed <- function(n) {
  m <- n^3
  return(m)
}
```

Resources

This tutorial has been given at useR! 2008 (Dortmund, Germany) and useR! 2009 (Rennes, France).

It has also been adapated to full-day invited tutorials / workshops at the Bank of Canada (Ottawa, Canada) and the Institute for Statistical Mathematics (Tokyo, Japan).

Shorter one-hour versions were presented at R/Finance 2009 and R/Finance 2010, both held in Chicago, USA.

Past (and possible future) presentation slides can be found at `http://dirk.eddelbuettel.com/presentations.html`

Outline

1 Motivation

2 Automation and scripting

- Overview
- littler
- Rscript

3 Measuring and profiling

4 Speeding up

5 Compiled Code

6 Implicitly Parallel

7 Explicitly Parallel

8 Out-of-memory processing

9 Summary

Tools: Using R in batch mode

Non-interactive use of R is possible:

- Using R in batch mode:

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

- Using R in here documents is awkward:

```
#!/bin/sh
cat << EOF | R --slave
   a <- 1.23; b <- 4.56
   cat("a times b is", a*b, "\n")
EOF
```

These approaches feels cumbersome. Variable expansion by the shell may interfere as well.

Tools: littler

The `r` frontend provided by the littler package was released by Horner and Eddelbuettel in September 2006 based on Horner's work on rapache.

- execute scripts:
  ```
  $ r somefile.R
  ```
- run Unix pipelines:
  ```
  $ echo 'cat(pi^2, "\n")' | r
  ```
- use arguments:
  ```
  $ r -lboot -e'example(boot.ci)'
  ```
- write Shebang scripts such as `install.r` (see next slide)

littler ’Shebang’ example

Consider the following code from the `littler` examples directory:

```
#!/usr/bin/env r
# a simple example to install one or more packages
if (is.null(argv) | length(argv)<1) {
  cat("Usage: installr.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)
```

Tools: littler cont.

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

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

The `getopt` and `optparse` packages make it easy for `r` and `Rscript` to support command-line options.

For simple use and debugging, direct evalualtions are useful:

```
r --package pkgA,pkgB --eval "someFunction(1,2)"
```

We will use a combination of these commands throughout the tutorial.

Tools: littler cont.

A simple example about using pipes:

```
$ du -csk /usr/local/lib/R/site-library/* | \
     awk '!/total$/ {print $1}' | \
     ~/svn/littler/examples/fsizes.r
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      4     218     540     864     972    3620

  The decimal point is 3 digit(s) to the right of the |

  0 | 0112335689
  1 | 079
  2 |
  3 | 6
```

This shows that I have a number of small packages installed, as well as one larger one.

Tools: Rscript

`Rscript`, which was first released with R 2.5.0 in April 2007, can be used in a similar fashion.

Due to implementation details, `r` starts up faster than `Rscript`.

On the other hand, `Rscript` is also available on Windows whereas `r` is limited to Linux and OS X.

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

Outline

1. Motivation
2. Automation and scripting
3. Measuring and profiling
   - Overview
   - RProf
   - RProfmem
   - Profiling Compiled Code
   - Summary
4. Speeding up
5. Compiled Code
6. Implicitly Parallel
7. Explicitly Parallel
8. Out-of-memory processing
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. `rbenchmark` is useful for comparisons.

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 `gcc` option `-pg`) 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.

RProf example cont.

This code runs both approaches 2000 times:

```
data(longley)

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

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

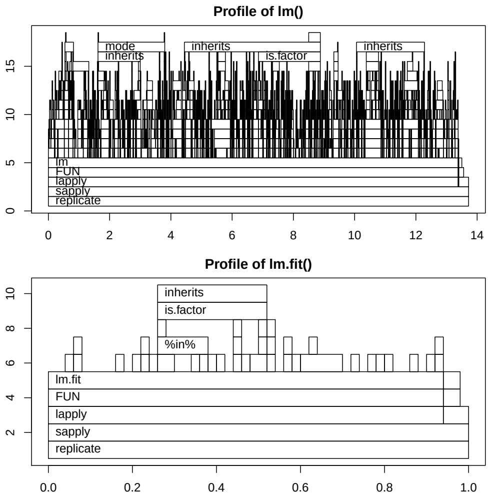
Second, from the command-line (on systems having `Perl`)

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

The CRAN package / function `profr` by Hadley 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.

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.

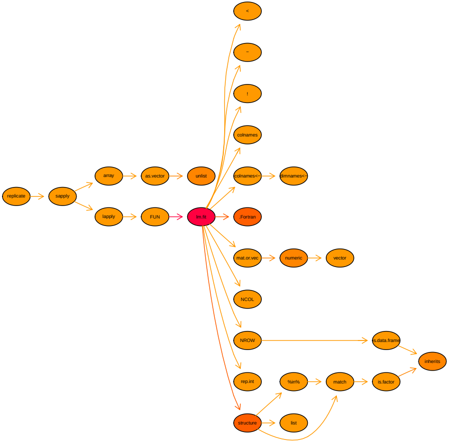
In addition, the `proftools` package by Luke Tierney can read profiling data and summarize directly in R.

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.

(The deprecated `Rgraphviz` is needed for the visualization.)

Another profiling example

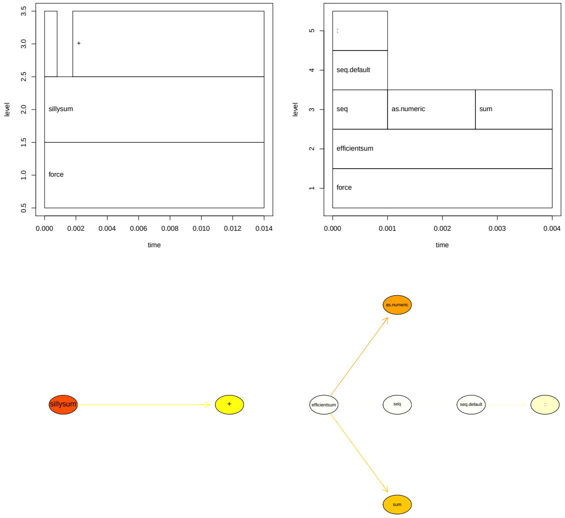
Both packages can be very useful for their quick visualisation of the RProf output. Consider this contrived example:

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

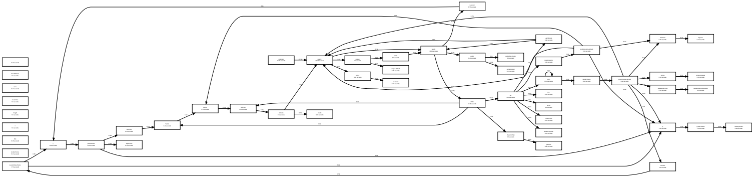
```
./prof2dot.pl longley.lm.out | dot -Tpdf \
             > longley_lm.pdf
./prof2dot.pl longley.lm.fit.out | dot -Tpdf \
             > longley_lmfit.pdf
```

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

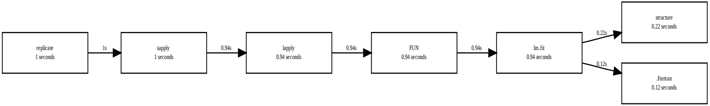
[1] http://wiki.r-project.org/rwiki/doku.php?id=tips:misc:profiling:current

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
```

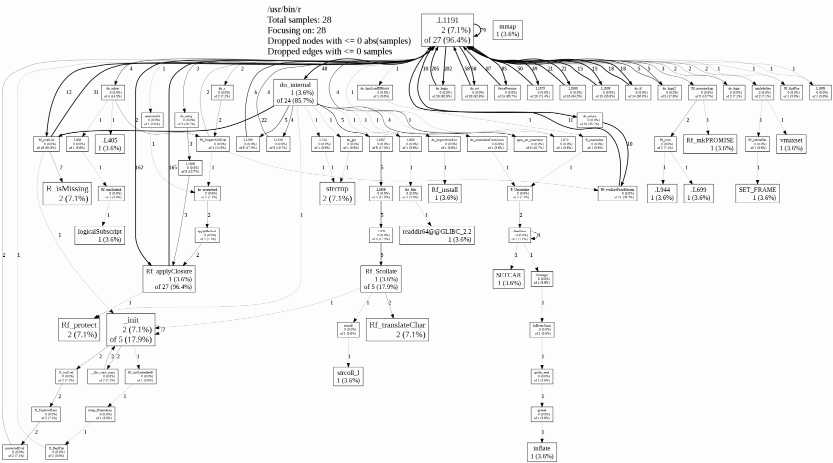
We can then analyse the profiling output in the file. The profiler (renamed from `pprof` to `google-pprof` on Debian / Ubuntu) has a large number of options. Here just use two different formats:

```
# 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.


/usr/bin/r
Total samples: 28
Focusing on: 28
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
.L1191 2 (7.1%) of 27 (96.4%)
mmap 1 (3.6%)
do_subset 0 (0.0%) of 4 (14.3%)
do_c 0 (0.0%) of 2 (7.1%)
do_internal 1 (3.6%) of 24 (85.7%)
do_lazyLoadDBfetch 0 (0.0%) of 1 (3.6%)
do_begin 0 (0.0%) of 26 (92.9%)
do_set 0 (0.0%) of 26 (92.9%)
forcePromise 0 (0.0%) of 24 (85.7%)
.L1573 0 (0.0%) of 20 (71.4%)
.L1560 0 (0.0%) of 18 (64.3%)
.L1508 0 (0.0%) of 15 (53.6%)
do_if 0 (0.0%) of 14 (50.0%)
do_logic2 0 (0.0%) of 5 (17.9%)
Rf_promiseArgs 0 (0.0%) of 3 (10.7%)
do_logic 0 (0.0%) of 2 (7.1%)
applydefine 0 (0.0%) of 2 (7.1%)
Rf_findFun 0 (0.0%) of 1 (3.6%)
.L1905 0 (0.0%) of 1 (3.6%)
newintoold 0 (0.0%) of 1 (3.6%)
do_relop 0 (0.0%) of 3 (10.7%)
do_return 0 (0.0%) of 10 (35.7%)
Rf_evalList 0 (0.0%) of 18 (64.3%)
.L458 0 (0.0%) of 1 (3.6%)
.L405 1 (3.6%)
Rf_DispatchOrEval 0 (0.0%) of 4 (14.3%)
.L1289 0 (0.0%) of 5 (17.9%)
.L1316 0 (0.0%) of 3 (10.7%)
.L744 0 (0.0%) of 1 (3.6%)
do_get 0 (0.0%) of 1 (3.6%)
.L1667 0 (0.0%) of 5 (17.9%)
.L800 0 (0.0%) of 1 (3.6%)
do_importIntoEnv 0 (0.0%) of 1 (3.6%)
do_unserializeFromConn 0 (0.0%) of 1 (3.6%)
apse_set_insertions 0 (0.0%) of 3 (10.7%)
.L674 0 (0.0%) of 1 (3.6%)
R_unserialize 0 (0.0%) of 1 (3.6%)
Rf_cons 0 (0.0%) of 2 (7.1%)
Rf_mkPROMISE 1 (3.6%)
Rf_unbindVar 0 (0.0%) of 1 (3.6%)
vmaxset 1 (3.6%)
.L1695 0 (0.0%) of 3 (10.7%)
R_isMissing 2 (7.1%)
Rf_mat2indsub 0 (0.0%) of 1 (3.6%)
do_usemethod 0 (0.0%) of 2 (7.1%)
strcmp 2 (7.1%)
.L1609 0 (0.0%) of 5 (17.9%)
list_files 0 (0.0%) of 1 (3.6%)
Rf_install 1 (3.6%)
R_Unserialize 0 (0.0%) of 2 (7.1%)
Rf_evalListKeepMissing 0 (0.0%) of 11 (39.3%)
.L944 1 (3.6%)
.L699 1 (3.6%)
SET_FRAME 1 (3.6%)
logicalSubscript 1 (3.6%)
applyMethod 0 (0.0%) of 2 (7.1%)
.L939 0 (0.0%) of 5 (17.9%)
readdir64@@GLIBC_2.2 1 (3.6%)
ReadItem 0 (0.0%) of 2 (7.1%)
Rf_applyClosure 1 (3.6%) of 27 (96.4%)
Rf_Scollate 1 (3.6%) of 5 (17.9%)
SETCAR 1 (3.6%)
InInteger 0 (0.0%) of 1 (3.6%)
Rf_protect 2 (7.1%)
_init 2 (7.1%) of 5 (17.9%)
strcoll 0 (0.0%) of 1 (3.6%)
Rf_translateChar 2 (7.1%)
InBytesConn 0 (0.0%) of 1 (3.6%)
R_tryEval 0 (0.0%) of 2 (7.1%)
__libc_start_main 0 (0.0%) of 2 (7.1%)
Rf_initEmbeddedR 0 (0.0%) of 1 (3.6%)
strcoll_l 1 (3.6%)
gzfile_read 0 (0.0%) of 1 (3.6%)
R_ToplevelExec 0 (0.0%) of 2 (7.1%)
setup_Rmainloop 0 (0.0%) of 1 (3.6%)
gzread 0 (0.0%) of 1 (3.6%)
protectedEval 0 (0.0%) of 2 (7.1%)
R_ReplFile 0 (0.0%) of 1 (3.6%)
inflate 1 (3.6%)

Profiling with Google Perftools

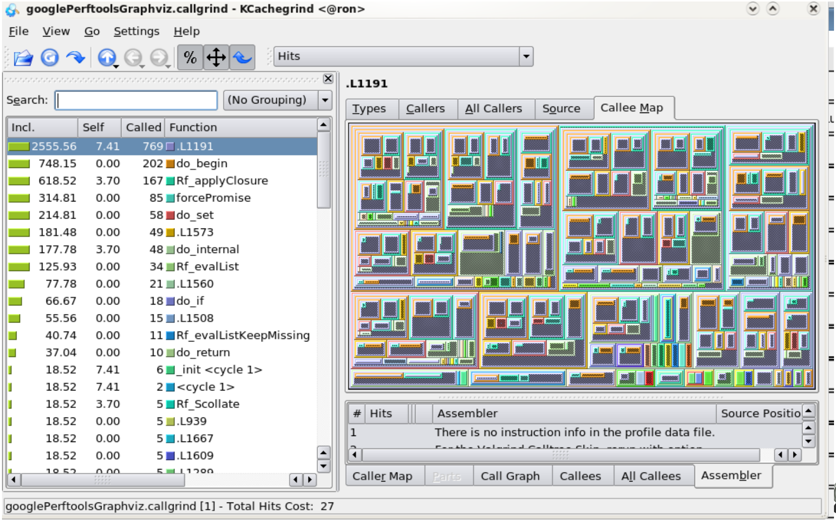
Another output format is used by the *callgrind* analyser that is part of *valgrind*—a frontend to a variety of analysis tools such as *cachegrind* (cache simulator), *callgrind* (call graph tracer), *helpgrind* (race condition analyser), *massif* (heap profiler), and *memcheck* (fine-grained memory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

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

```
google-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:



googlePerftoolsGraphviz.callgrind - KCachegrind <@ron>
File View Go Settings Help
Hits
Search:
(No Grouping)
.L1191
Types Callers All Callers Source Callee Map
Caller Map Parts Call Graph Callees All Callees Assembler

Incl.	Self	Called	Function
2555.56	7.41	769	.L1191
748.15	0.00	202	do_begin
618.52	3.70	167	Rf_applyClosure
314.81	0.00	85	forcePromise
214.81	0.00	58	do_set
181.48	0.00	49	.L1573
177.78	3.70	48	do_internal
125.93	0.00	34	Rf_evalList
77.78	0.00	21	.L1560
66.67	0.00	18	do_if
55.56	0.00	15	.L1508
40.74	0.00	11	Rf_evalListKeepMissing
37.04	0.00	10	do_return
18.52	7.41	6	_init <cycle 1>
18.52	7.41	2	<cycle 1>
18.52	3.70	5	Rf_Scollate
18.52	0.00	5	.L939
18.52	0.00	5	.L1667
18.52	0.00	5	.L1609
18.52	0.00	5	.L1289

#	Hits		Assembler	Source Positio
1			There is no instruction info in the profile data file.	
2			For the Valgrind Calltree Skin, rerun with option	

googlePerftoolsGraphviz.callgrind [1] - Total Hits Cost: 27

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()`.

Section Summary

We covered

- basic profiling functions in R : `Rprof`
- CRAN packages `profr` and `proftools` for visualization along with a contributed script
- memory profiling
- profiling compiled code in general and using Google's perftools
- the valgrind front-end kcachegrind for visualization / exploration

Outline

1. Motivation
2. Automation and scripting
3. Measuring and profiling
4. Speeding up
   - Vectorisation
   - Just-in-time compilation
   - BLAS
   - GPUs
   - Summary
5. Compiled Code
6. Implicitly Parallel
7. Explicitly Parallel
8. Out-of-memory processing
9. Summary

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 fourty.

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, or less than half.
- switching to faster functions can help as well 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 than the initial naive approach.

Accelerated R with just-in-time compilation

Stephen Milborrow created “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

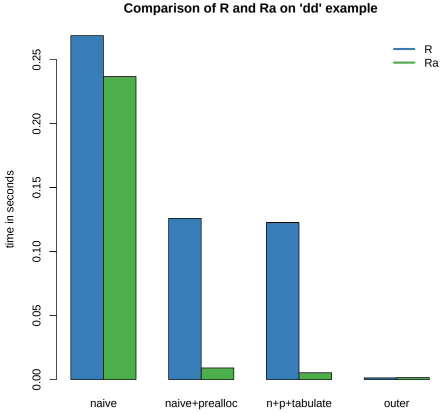
The last looping example can be improved with jit:

```
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 not widely deployed (and no longer updated since R 2.9.0) but available in Debian and Ubuntu.

Optimised BLAS

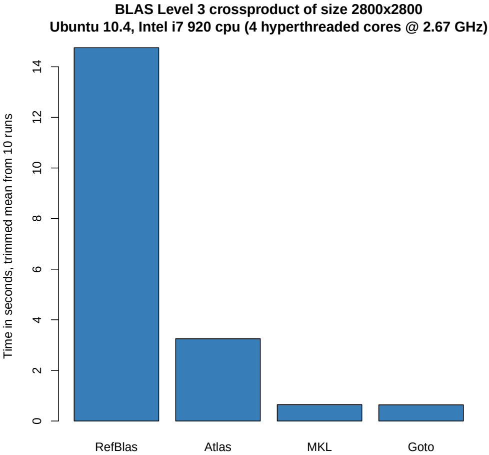
BLAS ('basic linear algebra subprogram') are standard building blocks for linear algebra. Highly-optimised libraries exist that can provide considerable performance gains.

R can be built using so-called optimised BLAS such as Atlas (open source), Goto (not 'free'), or the Intel MKL or AMD AMCL; 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 follows, making use of MKL libraries made available by Revolution for Ubuntu 9.10, and the Goto BLAS from U Texas.

Optimised BLAS Benchmarking

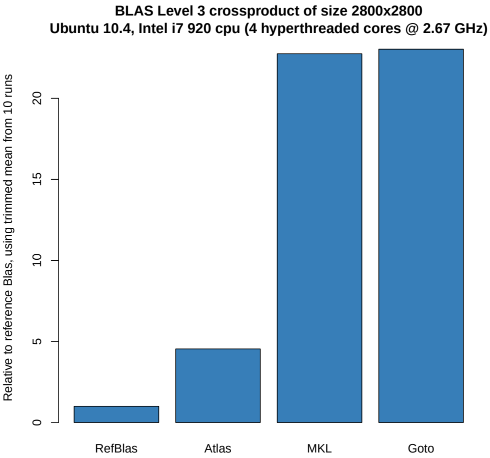


Source: Our calculations.

The Intel MKL and Goto BLAS perform well on multi-core machines.

Atlas is the current Ubuntu default (3.6.*) and is single-core.

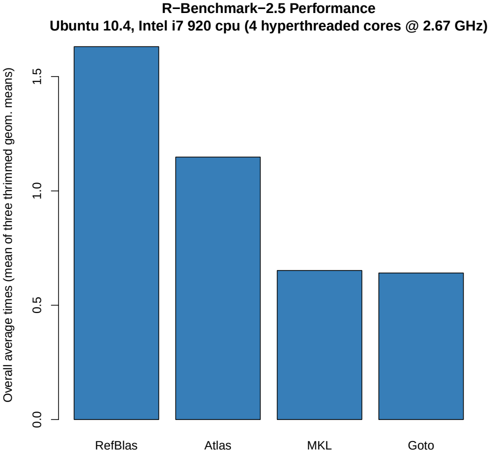
Optimised BLAS Benchmarking



Relative performance is equally impressive – a speed-up of factor twenty-three.

Source: Our calculations.

Optimised BLAS Benchmarking



Source: Our calculations.

But on a more well-rounded benchmark, the performance difference is nowhere near as dramatic.

Real work is rarely confined to BLAS Level operations.

From Blas to GPUs.

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

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

The key development environments that are available are

- Nvidia CUDA (Compute Unified Device Architecture) introduced in 2007 and provides C-like programming
- OpenCL (Open Computing Language) introduced in 2009 provides a vendor-independent interface to GPU hardware.

GPU resources

These are some of the resources and libraries for GPU programming:

- Vendor-specific:
  - CUDA for NVidia hardware
  - ATI Stream SDL for AMD hardware
- Vendor-independent: OpenCL
- For CUDA / NVividia:
  - BLAS on GPUs: Magma for Multicore/GPU
  - STL-alike containers: Thrust
  - Commercial CUDA libraries: CULAtools

CUDA Example

Consider a simple vector multiplication. In C, we write

```
void vecMult_h(int *A, int *B, unsigned long long N) {
   for (unsigned long long i=0;i<N;i++) {
     B[i] = A[i]*2;
   }
}

// which gets called as ...
a_h = (int *)malloc(sizeof(int)*n);
b_h = (int *)malloc(sizeof(int)*n);
// ... fill a_h
vecMult_h(a_h, b_h, n);
```

CUDA Example

With CUDA, we create so-called *kernels* which access the data in parallel using multiple threads. The equivalent function is

```
__global__ void vecMult_d(int *A, int *B, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if (i<N) {
      B[i] = A[i]*2;
    }
}

// which gets called as ...
cudaMalloc((void **)&a_d, n*sizeof(int));   // alloc. on device
cudaMalloc((void **)&b_d, n*sizeof(int));
dim3 dimBlock( blocksize );
dim3 dimGrid( ceil(float(n)/float(dimBlock.x)) );
cudaMemcpy(a_d, a_h, n*sizeof(int),cudaMemcpyHostToDevice);
vecMult_d<<<dimGrid,dimBlock>>>(a_d,b_d,n);
cudaThreadSynchronize();
cudaMemcpy(b_h, b_d, n*sizeof(int),cudaMemcpyDeviceToHost);
```

GPU programming for R

Currently, two packages provide GPU computing for R:

- gputools by Josh Buckner and Mark Seligman provides a number of basic routines (among them are e.g. `gpuCor`, `gpuDistClust`, `gpuFastICA`, `gpuGranger`, `gpuHclust`, `gpuLm`, `gpuMatMult`, `gpuSolve`, `gpuSvd`, `gpuSvmPredict`, `gpuSvmTrain`).
- cudaBayesreg by Adelino Ferreira da Silva reimplements Bayesian multilevel modeling for fMRI data.

Both use the CUDA toolchain for NVidia hardware.

GPU performance with R

A simple example, using a matrix of size 720 x 98 containing almost three years of daily returns data on the SP100:

```
# using R
> system.time(cor(X, method="kendall"))

   user  system elapsed
 59.220   0.000  59.224

# using GPU
> system.time(gpuCor(X, method="kendall"))

   user  system elapsed
  8.350   0.070   8.434
```

This correspond to about a *seven-fold* increase in speed.

GPU performance with R

Now let’s redo the example using a matrix of size 1206 x 477 containing almost five years of daily returns data on the SP500:

```
# using R
> system.time(cor(X, method="kendall"))
```

```
    user   system  elapsed
3925.730    0.010 3925.735
```

```
# using GPU
> system.time(gpuCor(X, method="kendall"))
```

```
   user  system elapsed
148.650   0.070 148.716
```

This correspond to about a *twenty-six-fold* increase in speed!

How is GPU programming different?

As R or C/C++ programmers on modern hardware, our life is relatively easy: flat and large memory spaces, little direct consideration of hardware representation.

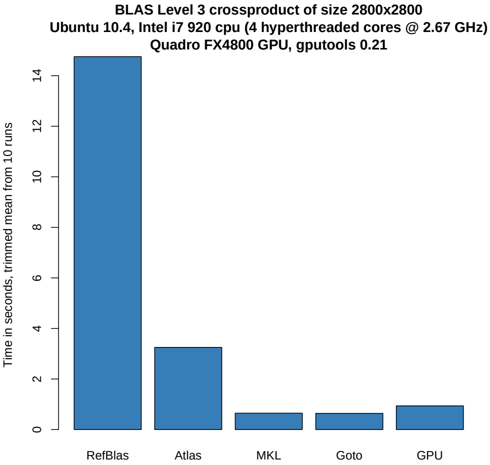
This makes for a nice level of abstraction.

With GPU, this abstraction goes away and we have to worry (again) about memory layout, access, ...

Also, the communication versus computation trade-off is critical: the GPU cam compute really fast, but it takes additional time to get results to the CPU.

So while there is a clear promise of increased performance, there is clearly ’No Free Lunch’.

Another GPU Example

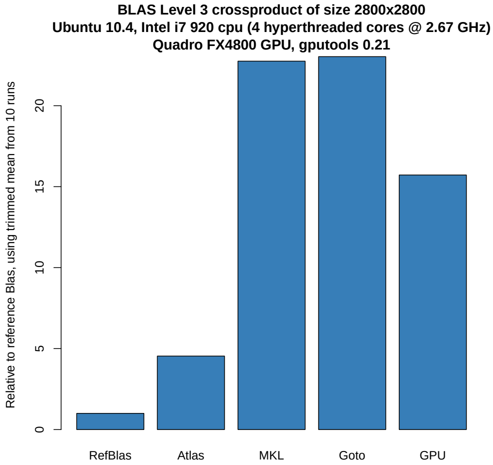


Source: Our calculations.

At 2800x2800, GPUs do not yet outperform (all competetitors) due to the relatively high cost of communication.

However, at 4000x4000 the GPU solution beats the Goto BLAS by a factor of 4.8 (at 9.459 sec versus 1.939 sec).

Optimised BLAS Benchmarking



Source: Our calculations.

At 2800x2800, GPUs do not yet outperform (all competetitors) due to the relatively high cost of communication.

However, at 4000x4000 the GPU solution beats the Goto BLAS by a factor of 4.8 (at 9.459 sec versus 1.939 sec).

Section Summary

In this section, we looked at

- vectorisation is the first thing to look for performance improvements,
- the 'distribution of determinants' example to measure different approaches,
- just-in-time compilation via jit and Ra
- BLAS and the performance gains from different implementations
- GPU computing and some simple example

Outline

1. Motivation
2. Automation and scripting
3. Measuring and profiling
4. Speeding up
5. Compiled Code
   - Overview
   - Inline
   - Rcpp
   - RInside
   - Summary
6. Implicitly Parallel
7. Explicitly Parallel
8. Out-of-memory processing
9. Summary

Compiled Code

Beyond smarter code (using *e.g.* vectorised expression and/or just-in-time compilation), hardware-driven acceleration 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 offers C++ classes to automate this.

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:

```c
void convolve(double *a, int *na, double *b,
              int *nb, double *ab)
{
  int i, j, nab = *na + *nb - 1;

  for(i = 0; i < nab; i++)
    ab[i] = 0.0;
  for(i = 0; i < *na; i++)
    for(j = 0; j < *nb; j++)
      ab[i + j] += a[i] * b[j];
}
```

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

```c
#include <R.h>
#include <Rdefines.h>

extern "C" SEXP convolve2(SEXP a, SEXP b)
{
  int i, j, na, nb, nab;
  double *xa, *xb, *xab;
  SEXP ab;

  PROTECT(a = AS_NUMERIC(a));
  PROTECT(b = AS_NUMERIC(b));
  na = LENGTH(a); nb = LENGTH(b); nab = na + nb - 1;
  PROTECT(ab = NEW_NUMERIC(nab));
  xa = NUMERIC_POINTER(a); xb = NUMERIC_POINTER(b);
  xab = NUMERIC_POINTER(ab);
  for(i = 0; i < nab; i++) xab[i] = 0.0;
  for(i = 0; i < na; i++)
    for(j = 0; j < nb; j++) xab[i + j] += xa[i] * xb[j];
  UNPROTECT(3);
  return(ab);
}
```

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.

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 Overview

The `Rcpp` package facilitates integration of C++ and R code.

It features a set of C++ classes (`Rcpp::IntegerVector`, `Rcpp::Function`, `Rcpp::Environment`, ...) that makes it easier to manipulate R objects of matching types (integer vectors, functions, environments, etc ...).

`Rcpp` takes advantage of C++ language features such as the explicit constructor/destructor lifecycle of objects to manage garbage collection automatically and transparently.

Users generally do not need to manage memory directly (via calls to new / delete or malloc / free) as this is done by the `Rcpp` classes or the corresponding STL containers.

Compiled Code: Rcpp API

Rcpp has two APIs: the 'classic' and a 'new' set of classes.

Classes of the new Rcpp API belong to the Rcpp namespace. Each class is associated to a given SEXP type.

SEXP type	Rcpp class
INTSXP	Rcpp::IntegerVector
REALSXP	Rcpp::NumericVector
RAWSXP	Rcpp::RawVector
LGLSXP	Rcpp::LogicalVector
CPLXSXP	Rcpp::ComplexVector
STRSXP	Rcpp::CharacterVector
VECSXP	Rcpp::List
EXPRSXP	Rcpp::ExpressionVector
ENVSXP	Rcpp::Environment
SYMSXP	Rcpp::Symbol
BUILTINSXP	Rcpp::Function
LANGSXP	Rcpp::Language
LISTSXP	Rcpp::Pairlist
S4SXP	Rcpp::S4
PROMSXP	Rcpp::Promise
WEAKREFSXP	Rcpp::WeakReference
EXTPTRSXP	template <typename T> Rcpp::XPtr

Some SEXP types do not have dedicated Rcpp classes : NILSXP, DOTSXP, ANYSXP, BCODESXP and CHARSXP.

Compiled Code: Rcpp Data Interchange

Data interchange between R and C++ is managed by extensible and powerful yet simple mechanisms.

Conversion of an R object to a C++ object is managed by the `Rcpp::as<T>` template which can handle:

- primitive types (`double`, `int`, ...)
- `std::string`, `const char*`
- STL containers such as `std::vector<T>` or `std::map<std::string, T>`

Compiled Code: Rcpp Data Interchange cont.

Conversion of C++ objects to R is managed by the template function `Rcpp::wrap`. This function currently manages :

- primitive types : `int`, `double`, `bool`, `float`, `Rbyte`, ...
- `std::string`, `const char*`
- STL containers such as `std::vector<T>` and STL maps such as `std::map< std::string, T>` provided that the template type T is wrappable
- any class that can be implicitly converted to `SEXP`, through `operator SEXP()`

Compiled Code: Rcpp Data Interchange cont.

Rcpp::wrap and Rcpp::as are often used implicitly. For example, when assigning objects to an environment:

```
// grab the global environment
Rcpp::Environment global = Rcpp::Environment::global_env() ;
std::deque<bool> z( 3 ); z[0] = false; z[1] = true; z[3] = false ;

global["x"] = 2 ;                    // implicit call of wrap<int>
global["y"] = "foo";                 // implicit call of wrap<char*>
global["z"] = z ;                    // impl. call of wrap<std::deque<bool>>

int x = global["x"] ;                // implicit call of as<int>
std::string y = global["y"]          // implicit call of as<std::string>
std::vector<bool> z1 = global["z"] ; // impl. call of as<std::vector<bool>>
```

Rcpp contains several examples that illustrate wrap and as.

Compiled Code: Rcpp Data Interchange cont.

Separate packages illustrate how to extend this `Rcpp` conversion mechanisms to third-party types:

- `RcppArmadillo` for conversion of types from the Armadillo C++ library.
- `RcppGSL` (on R-Forge) for conversion of types from the GNU Scientific Library.

We will see more of this below.

`Rcpp` is also used for data interchange by the `RInside` package which provides and easy way of embedding an R instance inside of C++ programs.

Compiled Code: Rcpp and inline

Inline has been extended to work with `Rcpp` using the `.Call` interface. The `cxxfunction` function is the primary interface.

This allows quick prototyping of compiled code. All unit tests are based on this and can serve as examples of how to use the mechanism.

For example, this function defines from R a C++ (simplified) version of lapply:

```
## create a compiled function cpp_lapply using cppfunction
cpp_lapply <- cxxfunction(signature(x = "list", g = "function" ),
              'Function fun(g) ;
               List input(x) ;
               List output( input.size() ) ;
               std::transform( input.begin(), input.end(), output.begin(), fun ) ;
               output.names() = input.names() ;
               return output ;
              ', plugin="Rcpp")
## call cpp_lapply on the iris data with the R function summary
cpp_lapply( iris, summary )
```

Rcpp example

So let us rewrite the 'distribution of determinant' example one more time. The simplest version can be set up as follows:

```
#include <Rcpp.h>

RcppExport SEXP dd_rcpp(SEXP v) {
  try {
    Rcpp::NumericVector vec(v); // vec parameter viewed as vector of doubles.
    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++)
            vec[i++] = a*b - c*d;

    return Rcpp::wrap(vec);      // return updated vector

  } catch(std::exception &ex) {
    forward_exception_to_r(ex);
  } catch(...) {
    ::Rf_error("c++ exception (unknown reason)");
  }
  return R_NilValue;
}
```

Rcpp example cont.

Using `inline` and `Rcpp`, we can create a compiled version:

```
suppressMessages(library(inline))
src <- '
    Rcpp::NumericVector vec(v); // param. as numeric vector
    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++)
            vec[i++] = a*b - c*d;

    return Rcpp::wrap(vec); // return updated vector
'
fun <- cxxfunction(signature(v="numeric"), src, plugin="Rcpp")
```

The `try ... catch()` block is automatically added by `inline`.

Rcpp example cont.

We can now use our new inlined function:

```
dd.rcpp.inline <- function() {
    x <- integer(10000)
    res <- fun(x)
    tabulate(res$vec)
}

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

```
[1] 0.00047
```

This beats the earlier (plain C) `inline` example by a small amount.

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

Another Rcpp example

Let us revisit the `lm()` example. How fast could compiled code be? We use the Armadillo library to find out.

```
lmArmadillo <- function() {
  src <- '

  Rcpp::NumericVector yr(Ysexp),  Xr(Xsexp);
  std::vector<int> dims = Xr.attr("dim") ;
  int n = dims[0], k = dims[1];

  arma::mat X(Xr.begin(), n, k, false);       // use armadillo constructors
  arma::colvec y(yr.begin(), yr.size());

  arma::colvec coef = solve(X, y);            // fit model y ~ X

  arma::colvec resid = y - X*coef;            // to compute std. err of coef.
  double sig2 = arma::as_scalar(trans(resid)*resid)/(n-k);
  arma::mat covmat = sig2 * arma::inv(arma::trans(X)*X);

  Rcpp::NumericVector coefr(k), stderrestr(k);
  for (int i=0; i<k; i++) {
      coefr[i]      = coef[i];
      stderrestr[i] = sqrt(covmat(i,i));
  }

  return Rcpp::List::create( Rcpp::Named( "coefficients", coefr),
                             Rcpp::Named( "stderr", stderrestr));
  '
```

Another Rcpp example (cont.)

We continue the function with the compilation:

```
    ## turn into a function that R can call
    fun <- cxxfunction(signature(Ysexp="numeric", Xsexp="numeric"),
                       src, plugin="RcppArmadillo")
}
```

We run the example code via

```
## generate X and y
N <- 100
mean(replicate(N, system.time(val <-
     lmArmadillo(y, X))["elapsed"]),trim=0.05)
```

Introducing RcppArmadillo

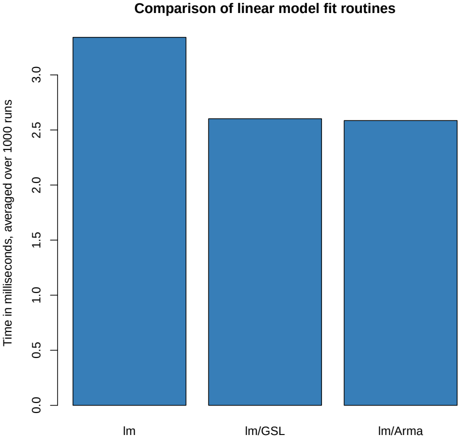
```
extern "C" SEXP fastLm(SEXP ys, SEXP Xs) {

  try {
    Rcpp::NumericVector yr(ys);                   // creates Rcpp vector from SEXP
    Rcpp::NumericMatrix Xr(Xs);                   // creates Rcpp matrix from SEXP
    int n = Xr.nrow(), k = Xr.ncol();

    arma::mat X(Xr.begin(), n, k, false);         // reuses memory, avoids extra copy
    arma::colvec y(yr.begin(), yr.size(), false);

    arma::colvec coef = arma::solve(X, y);        // fit model y ~ X
    arma::colvec res = y - X*coef;                // residuals

    double s2 = std::inner_product(res.begin(),res.end(),res.begin(),double())/(n-k);
                                                     // std.errors of coefficients
    arma::colvec std_err = arma::sqrt(s2*arma::diagvec(arma::inv(arma::trans(X)*X)));

    return Rcpp::List::create(Rcpp::Named("coefficients") = coef,
                              Rcpp::Named("stderr")       = std_err,
                              Rcpp::Named("df")           = n - k
                              );

  } catch( std::exception &ex ) {
    forward_exception_to_r( ex );
  } catch(...) {
    ::Rf_error( "c++ exception (unknown reason)" );
  }
  return R_NilValue; // -Wall
}
```

Introducing RcppArmadillo (cont.)

RcppArmadillo is a CRAN package that

- grew out of the work on these 'faster lm()' operations
- makes it even easier to use Armadillo from C++ for R
- by adding just a few lines of glue code to automagically pass date between the Rcpp classes (that eases access from R) and the Armadillo classes
- as Armadillo uses clever template meta-programming (TMP), algebra operations are fast yet concise and easy to use
- We also have a matching RcppGSL package on R-Forge.
- Both provide a standard formula-based interface as well as a bare-bones one.

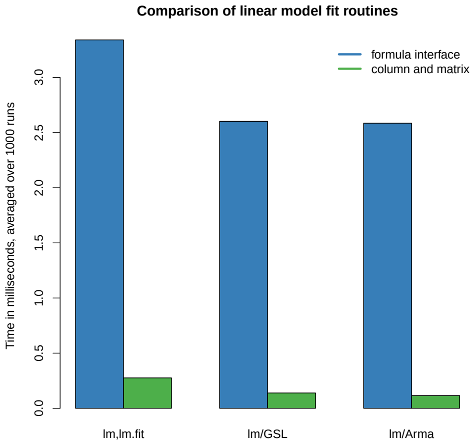
Another Rcpp example (cont.)



We get (relatively) small speed improvements using the fastLm functions from the RcppGSL and RcppArmadillo packages.

Source: See the examples/FastLM/ directory in Rcpp.

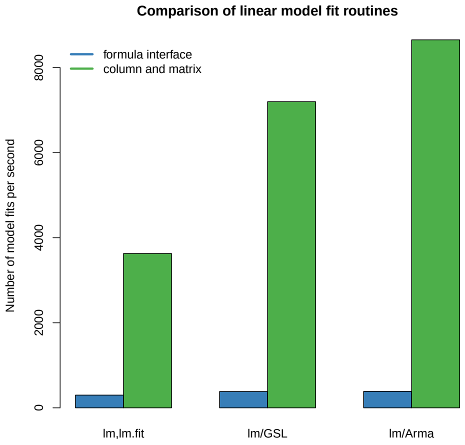
Another Rcpp example (cont.)



By switching to `lm.fit` and `fastLmPure` (i.e. foregoing the formula interfaces), we get more significant speed increases.

Source: See the `examples/FastLM/` directory in `Rcpp`.

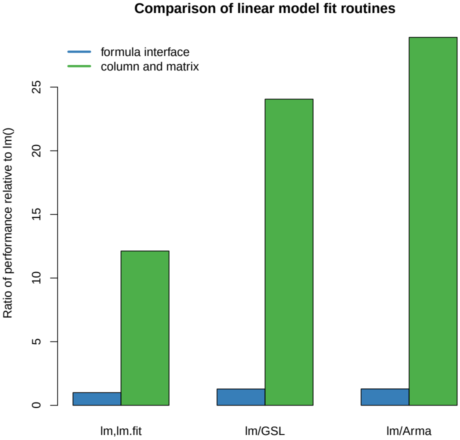
Another Rcpp example (cont.)



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

Source: See the `examples/FastLM/` directory in Rcpp.

Another Rcpp example (cont.)



We can also normalise by looking at the performance relative to `lm()`.

Source: See the `examples/FastLM/` directory in `Rcpp`.

RInside and bringing R 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`:

```
#include <RInside.h>                    // for the embedded R via RInside

int main(int argc, char *argv[]) {

    RInside R(argc, argv);              // create an embedded R instance

    R["txt"] = "Hello, world!\n";       // assign a char* (string) to 'txt'

    R.parseEvalQ("cat(txt)");           // eval the init string, ignoring any returns

    exit(0);
}
```

RInside and bringing R to C++ (cont)

```
#include <RInside.h>                    // for the embedded R via RInside
#include <iomanip>
int main(int argc, char *argv[]) {
  RInside R(argc, argv);              // create an embedded R instance
  std::string txt =
      "suppressMessages(library(fPortfolio)); "
      "lppData <- 100 * LPP2005.RET[, 1:6]; "
      "ewSpec <- portfolioSpec(); "
      "nAssets <- ncol(lppData); ";
  R.parseEvalQ(txt);                  // prepare problem

  const double dvec[6] = { 0.1, 0.1, 0.1, 0.1, 0.3, 0.3 }; // choose any weights
  const std::vector<double> w(dvec, &dvec[6]);
  R["weightsvec"] = w;                // assign weights

  R.parseEvalQ("setWeights(ewSpec) <- weightsvec");       // evaluate assignment

  txt = "ewPf <- feasiblePortfolio(data=lppData,spec=ewSpec,"
        "                         constraints=\"LongOnly\");"
        "print(ewPf); "
        "vec <- getCovRiskBudgets(ewPf@portfolio)";
  Rcpp::NumericVector   V(     (SEXP) R.parseEval(txt) );
  Rcpp::CharacterVector names( (SEXP) R.parseEval("names(vec)"));

  std::cout << "\n\nAnd now from C++\n\n";
  for (int i=0; i<names.size(); i++) {
    std::cout << std::setw(16) << names[i] << "\t" << std::setw(11) << V[i] << "\n";
  }
  exit(0);
}
```

RInside and bringing R to C++ (cont)

```
edd@max:~/svn/rinside/pkg/inst/examples/standard$ ./rinside_sample4

Using the GLPK callable library version 4.37
Title:
 MV Feasible Portfolio
 Estimator:         covEstimator
 Solver:            solveRquadprog
 Optimize:          minRisk
 Constraints:       LongOnly

Portfolio Weights:
SBI SPI SII LMI MPI ALT
0.1 0.1 0.1 0.1 0.3 0.3

Covariance Risk Budgets:
    SBI     SPI     SII     LMI     MPI     ALT
-0.0038  0.1423  0.0125 -0.0058  0.4862  0.3686

Target Return and Risks:
  mean     mu    Cov  Sigma   CVaR    VaR
0.0548 0.0548 0.4371 0.4371 1.0751 0.6609

Description:
 Mon Jul  5 12:37:33 2010 by user:

And now from C++
             SBI        -0.00380065
             SPI           0.142261
             SII          0.0125242
             LMI        -0.00576251
             MPI           0.486228
             ALT           0.368551
```

Section Summary

We covered how

- `inline` eases compiling, linking and loading C, C++ or Fortran into R
- `Rcpp` allows us to transfer R objects to C++ and back, and to create or alter R objects at the C++ level
- `RcppArmadillo` gives us linear algebra at C++ speed
- `RInside` permits us to add R to existing C++ applications

Things we did not cover

- debugging memory leaks in code: the previous tutorials had a short valgrind example
- more advanced `Rcpp` use: extensions, modules, sugar. But *"There is a vignette for that!"*

Outline

1. Motivation
2. Automation and scripting
3. Measuring and profiling
4. Speeding up
5. Compiled Code
6. Implicitly Parallel
   - pnmath and pnmath0
   - OpenMP
   - multicore
   - Summary
7. Explicitly Parallel
8. Out-of-memory processing
9. Summary

Using all those cores

Multi-core hardware is now a default, and the number of cores per cpus continues to increase. We want to advantage of these cores.

Two (no longer recent but still 'experimental') packages by Luke Tierney are addressing this question:

- `pnmath` uses OpenMP compiler directives;
- `pnmath0` uses pthreads to implements the same interface.

See `http://www.stat.uiowa.edu/~luke/R/experimental/`

Other related approaches are `multicore` discussed below as well as GPU computing.

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. 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 asses 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.

OpenMP for shared-memory parallelism

Citing from Wikipedia on OpenMP:

> *OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran [...]. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.*
>
> *[...] OpenMP is a portable, scalable model [...] for developing parallel applications for platforms ranging from the desktop to the supercomputer.*
>
> *An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), [...].*

OpenMP example

A C++ example from the aforementioned Wikipedia entry:

```
#include <omp.h>
#include <iostream>
#include <sstream>

int main (int argc, char *argv[]) {

    int th_id, nthreads;

#pragma omp parallel private(th_id)
    {
        th_id = omp_get_thread_num();
        std::ostringstream ss;
        ss << "Hello World from thread " << th_id << std::endl;
        std::cout << ss.str();
#pragma omp barrier
#pragma omp master
        {
            nthreads = omp_get_num_threads();
            std::cout << "There are " << nthreads << " threads" << std::endl;
        }
    }
    return 0;
}
```

OpenMP example (cont.)

We can build and run the example:

```
$ g++ -fopenmp -o hello_OpenMP hello_OpenMP.cpp -lgomp
$ OMP_NUM_THREADS=4 ./hello_OpenMP
```

```
Hello World from thread 2
Hello World from thread 3
Hello World from thread 0
Hello World from thread 1
There are 4 threads
```

The environment variable `OMP_NUM_THREAD` can be used to select the number of threads; otherwise the number of available cores is used.

multicore

The multicore package by Simon Urbanek provides a convenient interface to *locally* running parallel computations in R on machines with multiple cores or CPUs. Jobs can share the entire initial workspace.

This is implemented using the fork system call available for POSIX-compliant system (*i.e.* Linux and OS X but not Windows).

All jobs launched by multicore share the full state of R when spawned, no data or code needs to be initialized. This make the actual spawning very fast since no new R instance needs to be started.

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 16 (hyper-threaded) cores are now a reality, and that 32 or more cores are on the horizon, this package is very useful.

One thing to note is that 'anything but Windows' is required to take advantage of multicore (though Revolution offers a commercial closed-source alternative doSMPI).

multicore cont.

We can illustrate the `mclapply` function with a simple example:

```
R> system("pgrep R")
```

```
28352
```

```
R> mclapply(1:2,
+> FUN=function(x) system("pgrep R", intern=TRUE))
```

```
[[1]]
[1] "28352" "31512" "31513"

[[2]]
[1] "28352" "31512" "31513"
```

So two new R processes were started by `multicore`.

Section Summary

We looked at

- OpenMP for parallel processing via compiler-generated directives, and the `pnmath` package
- And `pnmath0` (using threads) is an alternative
- `multicore` for letting R schedule code on several available cores on one computer is very compelling and easy

Outline

1 Motivation

2 Automation and scripting

3 Measuring and profiling

4 Speeding up

5 Compiled Code

6 Implicitly Parallel

7 Explicitly Parallel
- Overview
- NWS
- Rmpi
- snow
- iterators, foreach and dopar
- Summary
- Some general tips

8 Out-of-memory processing

9 Summary

Embarassingly parallel

Several CRAN (or R-Forge) packages provide the ability to execute R code in parallel:

- `NWS`
- `Rmpi`
- `snow` (using MPI, PVM, NWS or sockets), also `snowFT` and `snowfall`
- `multicore` (see previous section)
- `foreach` with `doMC`, `doSNOW`, `doMPI`, `doRedis`,
- plus others (`rpvm`, `papply`, `taskPR` ...)

The survey paper by Schmidberger, Morgan, Eddelbuettel, Yu, Tierney and Mansmann (JSS, 2009) is a useful resource.

NWS Intro

NWS ("NetWorkSpaces") is a simple alternative to MPI (see below). It is based on Python and cross-platform, and originates with one of the predecessor companies to Revolution Analytics. NWS is accessible from R, Python, Matlab, Ruby, and other languages.

NWS is available via Sourceforge and CRAN. An introductory article appeared in Dr. Dobb's.

On Debian/Ubuntu, one installs the `python-nwsserver` package on the server node, and installs `r-cran-nws` on each client. Other systems may need to install the `twisted` framework for `Python` first.

A new implementation 'swn' (for 'Shared Workspace Neighborhood', or 'Swn wasn't NWS', or ...) may be forthcoming.

NWS data store example

A simple example, adapted from `demo(nwsExample)`

```
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
```

NWS sleigh example

The NWS component sleigh is an R class that makes it 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)
```

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. It can also be used as a building block for higher-level suage via snow or doMPI/foreach.

MPI Example

Let us look at the MPI variant of the 'Hello, World!' program:

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv)
{
    int rank, size, nameLen;
    char processorName[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Get_processor_name(processorName, &nameLen);

    printf("Hello, rank %d, size %d on processor %s\n",
           rank, size, processorName);

    MPI_Finalize();
    return 0;
}
```

MPI Example: cont.

We can compile the previous example via

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

If it 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

```r
#!/usr/bin/env r

library(Rmpi) # calls MPI_Init

rk <- mpi.comm.rank(0)
sz <- mpi.comm.size(0)
name <- mpi.get.processor.name()
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 exectute 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, or via the new doMPI package.

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 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.

Iterators, foreach and dopar

Revolution Analytics (and/or its predecessor companies) released several packages to CRAN to elegantly work with serial or parallel loops:

- `iterators` generalized the concepts of iteration control
- `foreach` offers to switch between serial and parallel execution using `%dopar%`
- available backends for `%dopar%` are
  - `doMC` (for multicore)
  - `doMPI` (for MPI)
  - `doSNOW` (for `snow`)

Also of note is the compatible `doRedis` package (for `rredis`, a clever NoSQL backend) by Bryan Lewis.

iterators

`iterators` provides an object that offers one data element at a time by calling a method `nextElem`

`iterators` can be created using the `iter` method on `list`, `vector`, `matrix`, or `data.frame` objects

`iterators` resemble the Java and Python constructs of the same name.

`iterators` are memory-friendly: one element at a time whereas sequences gets enumerated fully.

foreach

The foreach package provides a new looping construct which scan switch transparently between serial and parallel modes.

It can be seen a mix of for loops and lapply-style functional operation, and similar to foreach operators in other programming languages.

We can switch foreach to execute in parallel leaning on the existing snow or multicore (and soon Rmpi) backends

It works like lapply, but without the need for a function:

```
x <- foreach(i=1:10) %do% {
   sqrt(i)
}
```

and we can switch to %dopar% for parallel execution.

Why is this interesting?

Objects used in the body of `foreach` are automatically exported to remote nodes easing parallel programming:

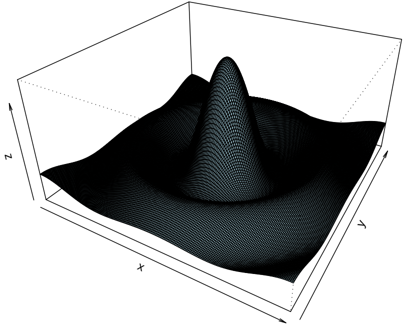
```
m <- matrix(rnorm(16), 4, 4)
foreach(i=1:ncol(m)) %dopar% {
   mean(m[,i])    # makes m available on nodes
}
```

We can nest this using the `:` operator:

```
foreach (i=1:3, .combine=cbind) %:%
   foreach (j=1:3, .combine=c) %dopar%
       (i+j)
```

See the vignette Nesting Foreach Loops for details.

foreach example: demo(sincSEQ)



Rendered via

```
library(foreach)
demo(sincSEQ)
```

foreach example: demo(sincSEQ)

```r
library(foreach)
# function that creates an iterator that returns subvectors
ivector <- function(x, chunks) {
  n <- length(x); i <- 1
  nextEl <- function() {
    if (chunks <= 0 || n <= 0) stop('StopIteration')
    m <- ceiling(n / chunks); r <- seq(i, length=m)
    i <<- i + m; n <<- n - m; chunks <<- chunks - 1; x[r]
  }
  obj <- list(nextElem=nextEl)
  class(obj) <- c('abstractiter', 'iter');   obj
}
x <- seq(-10, 10, by=0.1) # Define coordinate grid
cat('Running sequentially\n'); ntasks <- 4
# Compute the value of the sinc function at each grid point
z <- foreach(y=ivector(x, ntasks), .combine=cbind) %do% {
  y <- rep(y, each=length(x));  r <- sqrt(x ^ 2 + y ^ 2)
  matrix(10 * sin(r) / r, length(x))
}
# Plot the results as a perspective plot
persp(x,x,z,ylab='y',theta=30,phi=30,expand=0.5,col="lightblue")
```

foreach example: demo(sincSEQ) cont.

The key in the `foreach` demo was the line

```
z <- foreach(y=ivector(x,ntasks),.combine=cbind) %do% {
   y <- rep(y, each=length(x))
   r <- sqrt(x ^ 2 + y ^ 2)
   matrix(10 * sin(r) / r, length(x))
}
```

where *z* is computed in a `foreach` loop using a custom `ivector` iterator over the grid *x* with a given number of task; results are recombined using `cbind`.

The actual work is being done in the code block following `%do%`.

foreach example: demo(sincMC)

In order to run this code in parallel using multicore, we simply use

```
library(doMC)
registerDoMC()
[...]
nw <- getDoParWorkers()
cat(sprintf('Running with %d worker(s)\n', nw))
[...]
z <- foreach(y=ivector(x, nw), \
            .combine=cbind) %dopar% {
[...]
```

as can be seen via demo(sincMC).

foreach example: demo(sincMPI)

Similarly, in order to run this code in parallel using `Rmpi`, we simply use the `doMPI` package (on R-Forge, soon on CRAN):

```
library(doMPI)

# create and register a doMPI cluster
cl <- startMPIcluster(count=2)
registerDoMPI(cl)
[...]
# compute the sinc function in parallel
v <- foreach(y=x, .combine="cbind") %dopar% {
  r <- sqrt(x^2 + y^2) + .Machine$double.eps
  sin(r) / r
}
[...]
closeCluster(cl)
```

as can be seen via `demo(sincMPI)`.

Section Summary

We looked at

- NWS, a simple cross-platform toolkit that can also be used for parallel computing
- Rmpi as well as MPI, the standard for parallel computing via message passing
- snow, a wrapper around sockets, MPI, PVM or NWS for easy parallel computing with R
- iterators and foreach as another alternative.

Things we did not cover

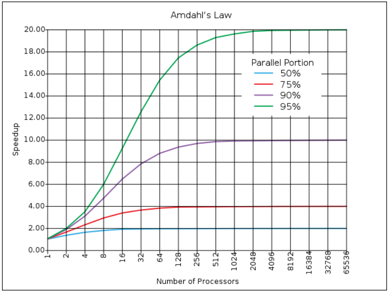
- Hadoop and packages like RHIPE
- different parallel computing approaches like Rdsm using distributed shared memory

Amdahl's Law: An upper bound to speed gains

An upper bound to expected gains by parallelization is provided by Amdahl's law which relates the *proportion P* of total running time which can realize a *speedup S* due to parallelization (using S nodes) to the expected net speedup:

$$\frac{1}{(1-P)+\frac{P}{S}}$$

e.g. for $P = 0.75$ and $S = 128$ we expect a net speedup of up to 3.9.



Source: `http://en.wikipedia.org/wiki/Amdahl's_law`

Best practices for Parallel Computing with R

Quoting from the Schmidberger et al pager:

- Communication is much slower than computation; minimize data transfer to and from workers, maximize remote computation.
- Random number generators require extra care. Special-purpose packages `rsprng` and `rlecuyer` are available; `snow` provides an integrated interface.
- R's lexical scoping, serializing functions and the environments they are defined in require care to avoid transmitting unnecessary data. Functions used in `apply`-like calls should be defined in the global environment, or in a package name space. `forever` can be helpful too,

Outline

1 Motivation

2 Automation and scripting

3 Measuring and profiling

4 Speeding up

5 Compiled Code

6 Implicitly Parallel

7 Explicitly Parallel

8 Out-of-memory processing
- overview
- biglm
- ff
- bigmemory
- Example
- Summary

9 Summary

Extending physical RAM limits

Two CRAN packages ease the analysis of *large* datasets.

- `ff` which maps R objects to files and is therefore only bound by the available filesystem space
- `bigmemory` which maps R objects to dynamic memory objects not managed by R

Both packages can use the `biglm` package for out-of-memory (generalized) linear models.

Also worth mentioning are the older packages `g.data` for delayed data assignment from disk, `filehash` which takes a slightly more database-alike view by 'attaching' objects that are still saved on disk, and `R.huge` which also uses the disk to store the data.

biglm

The `biglm` package operates on ’larger-than-memory’ datasets by operating on ’chunks’ of data at a time.

```
make.data <- function ... # see ’help(bigglm)’
dataurl <-
   "http://faculty.washington.edu/tlumley/NO2.dat"
airpoll <- make.data(dataurl, chunksize=150, \
   col.names=c("logno2","logcars","temp",\
   "windsp","tempgrad","winddir","hour","day"))
b <- bigglm(exp(logno2)~logcars+temp+windsp, \
            data=airpoll, family=Gamma(log), \
            start=c(2,0,0,0),maxit=10)
summary(b)
```

`lm()` and `glm()` models can be estimated (and updated).

ff: Large Objects

ff won the *UseR! 2007* 'large datasets' competition. It has since undergone a complete rewrite for versions 2.0 and 2.1.

ff provide memory-efficient storage of R objects on disk, and fast access functions that transparently map these in pagesize chunks to main memory. Many native data types are supported.

ff is complex package with numerous options that offer data access that can be tailored to be extremely memory-efficient.

ff: Large Objects cont.

As a small example, consider

```
b <- 1000
n <- 100000
k <- 3
x <- ff(vmode="double", dim=c(b*n,k), \
        dimnames=list(NULL, LETTERS[1:k]))
lsos()
```

```
              Type  Size  Rows Columns
x        ff_matrix  2088 1e+08       3
b          numeric    32 1e+00      NA
k          numeric    32 1e+00      NA
n          numeric    32 1e+00      NA
```

We see the matrix *x* has 100 million elements and three columns, yet occupies only 2088 bytes (essentially an external pointer and some meta-data).

ff: Large Objects cont.

We can use `ff` along with `biglm`:

```
ffrowapply({
   l <- i2 - i1 + 1
   z <- rnorm(l)
   for (i in 1:k)  x[i1:i2,i] <- z + rnorm(l)
}, X=x, VERBOSE=TRUE, BATCHSIZE=n)

form <- A ~ B + C
first <- TRUE
ffrowapply({
 if (first){
  first <- FALSE
  fit <- biglm(form,as.data.frame(x[i1:i2,,drop=FALSE]))
 } else
  fit <- update(fit,as.data.frame(x[i1:i2,,drop=FALSE]))
}, X=x, VERBOSE=TRUE, BATCHSIZE=n)
```

bigmemory

The bigmemory project comprises several packages: bigmemory, bigtabulate, biganalytics, bigalgebra as well as synchronicity.

Michael Kane is the 2010 winner of the Chambers price for his work on bigmemory.

bigmemory has undergone several rewrites and is now at version 4.*. The package is similar to ff as it allows allocation and access to memory managed by the operating system but 'outside' of the view of R (and optionally mapped to disk).

bigmemory cont.

bigmemory implements locking and sharing which allows multiple R sessions on the same host to access a common (large) object managed by bigmemory.

```
> object.size( big.matrix(1000,1000, "double") )

[1] 372

> object.size( matrix(double(1000*1000), ncol=1000) )

[1] 8000112
```

To R, a big.matrix of 1000 × 1000 elements occupies only 372 bytes of memory. The actual size of 8 mb is allocated by the operating system, and R interfaces it via an 'external pointer' object.

bigmemory cont.

We can illustrate bigmemory use of biglm:

```
x <- matrix(unlist(iris), ncol=5)
colnames(x) <- names(iris)
x <- as.big.matrix(x)

silly.biglm <- biglm.big.matrix(Sepal.Length ~ \
          Sepal.Width + Species, data=x, fc="Species")
summary(silly.biglm)
```

As before, the memory use of the new ’out-of-memory’ object is smaller than the actual dataset as the ’real’ storage is outside of what the R memory manager sees.

This can of course be generalized to really large datasets and ’chunked’ access.

Example

The recent ASA dataviz competition asked for a graphical summary of a huge dataset.

We are going to look at the entry by Jay Emerson and his student Michael Kane as it covers several of the packages we looked at here.

The data contains flight arrival and departure data for almost all commercial flights within the USA from October 1987 to April 2008.

There are almost 120 million records and 29 variables, with some recoding done by Emerson and Kane.

Example: Sequential data access

Task: For every plane, find the month of its earliest flight in the data set.

```r
# Take one: Sequential
#
date()
numplanes <- length(unique(x[,"TailNum"])) - 1
planeStart <- rep(0, numplanes)
for (i in theseflights) {  ## theseflights is a sample
  y <- x[mwhich(x, "TailNum", i, 'eq'),
         c("Year", "Month"), drop=FALSE]  # Note this.
  minYear <- min(y[,"Year"], na.rm=TRUE)
  these <- which(y[,"Year"]==minYear)
  minMonth <- min(y[these,"Month"], na.rm=TRUE)
  planeStart[i] <- 12*minYear + minMonth
  cat("TailNum",i,minYear,minMonth,nrow(y),planeStart[i],"\n")
}
planeStart[planeStart!=0]
date()  ## approximately 9 hours on the Yale cluster
```

Example: Sequential data access

```
# Take two: foreach(), sequential:
#
require(foreach)
date()
planeStart <- foreach(i=theseflights, .combine=c) %dopar% {
  y <- x[mwhich(x, "TailNum", i, 'eq'),
           c("Year", "Month"), drop=FALSE]   # Note this.
  minYear <- min(y[,"Year"], na.rm=TRUE)
  these <- which(y[,"Year"]==minYear)
  minMonth <- min(y[these,"Month"], na.rm=TRUE)
  cat("TailNum",i, minYear, minMonth, nrow(y), planeStart[i], "\n")
  12*minYear + minMonth
}
planeStart
date()  ## time ?
```

Example: Sequential data access

```
# Take three: foreach() and multicore
#
# Master and four workers
#
library(doMC)
registerDoMC()
date()
planeStart <- foreach(i=theseflights, .combine=c) %dopar% {
  x <- attach.big.matrix(xdesc)
  y <- x[mwhich(x, "TailNum", i, 'eq'),
         c("Year", "Month"), drop=FALSE]  # Note this.
  minYear <- min(y[,"Year"], na.rm=TRUE)
  these <- which(y[,"Year"]==minYear)
  minMonth <- min(y[these,"Month"], na.rm=TRUE)
  rm(x); gc()
  12*minYear + minMonth
}
planeStart
date() ## now about 2.5 hours
```

Example: Sequential data access

```r
# Take four: foreach() and snow / SOCK
#
# Master and three workers
#
library(doSNOW)
cl <- makeSOCKcluster(3)
registerDoSNOW(cl)
date()
planeStart <- foreach(i=theseflights, .combine=c) %dopar% {
  require(bigmemory)
  x <- attach.big.matrix(xdesc)
  y <- x[mwhich(x, "TailNum", i, 'eq'),
         c("Year", "Month"), drop=FALSE]  # Note this.
  minYear <- min(y[,"Year"], na.rm=TRUE)
  these <- which(y[,"Year"]==minYear)
  minMonth <- min(y[these,"Month"], na.rm=TRUE)
  12*minYear + minMonth
}
planeStart
stopCluster(cl)
date()  ## about 3.5 hours
```

Section Summary

We looked at

- `biglm` which enables (generalized) linear models to be fit in ’chunks’,
- `ff` which permits efficient storage of large data sets in file-based storage,
- `bigmemory`, part of a suite of packages, for shared-memory (or file-backed) analysis.

We illustrate their use with examples from the ASA dataviz competition.

Outline

1. Motivation
2. Automation and scripting
3. Measuring and profiling
4. Speeding up
5. Compiled Code
6. Implicitly Parallel
7. Explicitly Parallel
8. Out-of-memory processing
9. Summary

Wrapping up

In this tutorial session, we covered

- *scripting* and automation using littler and Rscript
- *profiling* and tools for *visualising profiling* output
- gaining speed using *vectorisation*, *Ra* and *just-in-time* compilation
- even more speed via *compiled code* using tools like *inline* and *Rcpp*, and how to embed R in C++ programs
- running R code in *parallel*, explicitly and implicitly

Wrapping up

Further questions ?

Two good resources are

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

Further resources:

- (Some) scripts are at http://dirk.eddelbuettel.com/code/hpcR/
- Updated versions of the tutorial may appear at http://dirk.eddelbuettel.com/presentations.html

Do not hesitate to email me at edd@debian.org

Thank You!