Introduction to
High-Performance Computing with R

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

The Institute of Statistical Mathematics
Tachikawa, Tokyo, Japan
27 November 2009

Outline

1 Motivation
2 Tools for automation and scripting
3 Measuring and profiling
4 Speeding up
5 Compiled Code
6 Explicitly and Implicitly Parallel
7 Out-of-memory processing
8 Summary
Motivation: What describes our current situation?

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**: everything from faster DNA sequencing to larger experiments to per-item RFID scanning in retail — 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

Not all large data sets are suitable for R, and data is frequently pruned, filtered or condensed down to manageable size (where the exact meaning of manageable will vary by user).
We look at ways to ‘script’ running R code which is helpful for both automation and debugging.

We will then measure using profiling tools to analyse and visualize performance; we will also look 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 session 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:

```r
> cat("Hello
")
Hello
```

Source code listings are boxed and with lines numbers

```r
1 cubed <- function(n) {
2 m <- n^3
3 return(m)
4 }
```
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
  ```

  However, this feels somewhat cumbersome. Variable expansion by the shell may interfere as well.
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:

```r
#!/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)
```
If saved as install.r, we can call it via
$ install.r ff bigmemory

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

For debugging, the following proves useful:
r --package pkgA,pkgB --eval "code(1,2)"

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

A simple example about using pipes:
$ du -csk /usr/local/lib/R/site-library/* | \ awk ‘!/total$/ {print $1}’ | \ ~/svn/littler/examples/fsizes.r

<table>
<thead>
<tr>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>218</td>
<td>540</td>
<td>864</td>
<td>972</td>
<td>3620</td>
</tr>
</tbody>
</table>

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

Due to implementation details, \texttt{r} starts up faster than \texttt{Rscript}.

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

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

The \texttt{RPy} and \texttt{RPy2} packages provides access from \texttt{Python}:

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

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()")
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.

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.

```r
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:

```r
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 $N$:

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

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

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

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.
# 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.

This can generate complete (yet complex) graphs.
Another output format is used by the \textit{callgrind} analyser that is part of \textit{valgrind}—a frontend to a variety of analysis tools such as \textit{cachegrind} (cache simulator), \textit{callgrind} (call graph tracer), \textit{helpgrind} (race condition analyser), \textit{massif} (heap profiler), and \textit{memcheck} (fine-grained memory checker).

For example, the KDE frontend \textit{kcachegrind} can be used to visualize the profiler output as follows:

\begin{verbatim}
google-pprof --callgrind \ /usr/bin/r /tmp/gpProfile.log \ > googlePerftools.callgrind
kcachegrind googlePerftools.callgrind
\end{verbatim}
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

```c
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()`.
Revisiting our trivial trivial example from the preceding section:

```r
> 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.

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

```r
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

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

The case study discusses two important points that bear repeating:

- pre-allocating space helps with performance:
  ```r
  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`:

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

The time for the most efficient solution is:

```r
> 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 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.

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

```r
> 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.
Accelarated R with just-in-time compilation

The last looping example can be improved with jit:

```r
    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)
    }
```

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

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

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

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

[1] 2.6465

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

[1] 16.42813
```

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

```c
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);
```

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

```c
__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);
```
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.

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>user</th>
<th>system</th>
<th>elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
<td>148.65</td>
<td>0.070</td>
<td>148.716</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>3925.73</td>
<td>0.010</td>
<td>3925.735</td>
</tr>
</tbody>
</table>

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

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, ...

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

This requires some familiarity with R internals though the `Rcpp` and `RInside` packages aim to hide much of this complexity.
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];
}
```

Dirk Eddelbuettel
High-Perf. Computing with R @ ISM, Japan, Nov 2009

The convolution function is called from R by

```r
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.
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:

```r
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.

```r
## 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)
```

**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`.

---

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

```r
## Use of .Call convention with C code
## Multiply each image in a stack with a 2D Gaussian at a given position
code <- "
SEXP res:
int nprotect = 0, nx, ny, nz, x, y;
PROTECT(res = Rf_duplicate(a)); nprotect++;

nx = INTEGER(GET_DIM(a))[0];
ny = INTEGER(GET_DIM(a))[1];
nz = INTEGER(GET_DIM(a))[2];
double sigma2 = REAL(s)[0] * REAL(s)[0], d2 ;
double cx = REAL(centre)[0], cy = REAL(centre)[1], *data, *rdata;
for (int im = 0; im < nz; im++) {
  data = &(REAL(a)[im*nx*ny]); rdata = &(REAL(res)[im*nx*ny]);
  for (x = 0; x < nx; x++)
    for (y = 0; y < ny; y++) {
      d2 = (x-cx)*(x-cx) + (y-cy)*(y-cy);
      rdata[x + y*nx] = data[x + y*nx] * exp(-d2/sigma2);}
}
UNPROTECT(nprotect);
return res;"
funx <- cfunction(signature(a="array", s="numeric", centre="numeric"), code)
x <- array(runif(50*50), c(50,50,1))
res <- funx(a=x, s=10, centre=c(25,15)) ## actual call of compiled function
if (interactive()) image(res[1,])
```
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

```r
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)
```

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

```r
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.
**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.

```
#include <Rcpp.h>

RcppExport SEXP dd_rcpp(SEXP v) {

  SEXP rl = R_NilValue;  // Use this when nothing is returned

  RcppVector<int> vec(v);  // vec parameter viewed as vector of doubles
  int n = vec.size(), 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;

  RcppResultSet rs;  // Build result set returned as list to R
  rs.add("vec", vec);  // vec as named element with name 'vec'
  rl = rs.getList();  // Get the list to be returned to R.

  return rl;
}
```

but it is actually preferable to use the exception-handling feature of C++ as in the slightly longer next version.
We can create a shared library from the source file as follows:

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

g++ -I/usr/share/R/include \n    -I/usr/lib/R/site-library/Rcpp/lib \n    -fpic -g -O2 \n    -c dd.rcpp.cpp -o dd.rcpp.o

g++ -shared -o dd.rcpp.so dd.rcpp.o \n    -L/usr/lib/R/site-library/Rcpp/lib \n    -lRcpp -Wl,-rpath,/usr/lib/R/site-library/Rcpp/lib \n    -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.

```r
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:

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

by initialising a `RcppParams` object and extracting as in

```r
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");
```
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.

---

```
#include <cstdio>
extern "C" {
#include <gsl/gsl_multifit.h>
}
#include <Rcpp.h>

RcppExport SEXP gsl_multifit(SEXP Xsexp, SEXP Ysexp) {
  SEXP r = R_NilValue;
  char *exceptionMesg=NULL;

  try {
    RcppMatrixView<double> Xr(Xsexp);
    RcppVectorView<double> Yr(Ysexp);

    int i, j, n = Xr.dim1(), k = Xr.dim2();
    double chisq;

    gsl_matrix *X = gsl_matrix Alloc(n, k);
    gsl_vector *y = gsl_vector Alloc(n);
    gsl_vector *c = gsl_vector Alloc(k);
    gsl_matrix *cov = gsl_matrix Alloc(k, k);

    for (i = 0; i < n; i++)
      for (j = 0; j < k; j++)
        gsl_matrix_set (X, i, j, Xr(i,j));
    gsl_vector_set (y, i, Yr(i));

    // your code here

  } catch (exceptionMesg)
  {
    Rcpp::stop(exceptionMesg);
  }
  return r;
}
```

---

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

```r
27
gsl_multifit_linear_workspace *work = gsl_multifit_linear_alloc (n, k);
gsl_multifit_linear (X, y, c, cov, &chisq, work);
gsl_multifit_linear_free (work);

RcppMatrix<double> CovMat(k, k);
RcppVector<double> Coef(k);
for (i = 0; i < k; i++) {
    for (j = 0; j < k; j++)
        CovMat(i, j) = gsl_matrix_get(cov, i, j);
    Coef(i) = gsl_vector_get(c, i);
}
gsl_matrix_free(X);
gsl_vector_free(y);
gsl_vector_free(c);
gsl_matrix_free(cov);
RcppResultSet rs;
rs.add("coef", Coef);
rs.add("covmat", CovMat);
rl = rs.getReturnList();
}
catch (std::exception & ex) {
    exceptionMesg = copyMessageToR(ex.what());
    catch (...) {
        exceptionMesg = copyMessageToR("unknown reason");
    }
    if (exceptionMesg != NULL) Rf_error(exceptionMesg);
    return rl;
}
```

We can build a shared library for R via

```bash
RCPP_CXXFLAGS='Rscript -e 'Rcpp:::CxxFlags()''
RCPP_LIBS='Rscript -e 'Rcpp:::LdFlags()''

PKG_CPPFLAGS="-W ${RCPP_CXXFLAGS}" \
    PKG_LIBS="-lgsl -lblas ${RCPP_LIBS}" \
    R CMD SHLIB gsl_multifit_in_R.cpp
```

and run the example code via

```r
dyn.load("gsl_multifit_in_R.so")
## generate X and y
N <- 100
mean(replicate(N, system.time(val <- .Call("gsl_multifit", X, y))['elapsed']), trim=0.05)
```
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.

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

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

```c
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

```r
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`

![Profiling Diagram](image_url)
Two tips for easing builds with \texttt{Rcpp}:

For command-line use, a shortcut is to copy \texttt{Rcpp.h} to \\
\texttt{/usr/local/include}, and \texttt{libRcpp.so} to \\
\texttt{/usr/local/lib}. The earlier example reduces to \\
\texttt{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 \texttt{src/Mak
evars} with \\
\# compile flag providing header directory \\
\texttt{PKG_CXXFLAGS=’Rscript -e ’Rcpp:::CxxFlags()’} \\
\# link flag providing library and path \\
\texttt{PKG_LIBS=’Rscript -e ’Rcpp:::LdFlags()’} \\
See \texttt{help(Rcpp-package)} for more details.

\begin{verbatim}
#include "RInside.h" // for the embedded R via RInside
#include "Rcpp.h" // for the R / Cpp interface

int main(int argc, char *argv[]) {
    RInside R(argc, argv); // create an embedded R instance
    std::string txt = "Hello, world!\n"; // assign a standard C++ string to 'txt'
    R.assign( txt, "txt" ); // assign string var to R variable 'txt'
    std::string evalstr = "cat(txt)";
    R.parseEvalQ( evalstr ); // eval the init string, ignoring any returns
    exit(0);
}
\end{verbatim}

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

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

int main(int argc, char* argv[]) {
  const int mdim = 4;
  std::string evalstr = "cat('Running ls()\n'); print(ls());\n  cat('Showing M\n'); print(M);  
  cat('Showing colSums()\n');\n  Z <- colSums(M); print(Z); Z";  ## returns Z
  RInside R(argc, argv);
  SEXP ans;
  std::vector<std::vector<double>> myMatrix = createMatrix(mdim);
  R.assign("myMatrix", "M");  // assign STL matrix to R's 'M' var
  R.parseEval(evalstr, ans);   // eval the init string — Z is now in ans
  RcppVector<double> vec(ans); // convert RcppVector to STL vector
  for (unsigned int i = 0; i < v.size(); i++)
    std::cout << "In C++ element " << i << " is " << v[i] << std::endl;
  exit(0);
}

Debugging example: valgrind

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

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

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

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 0x40F16E6: {within /usr/lib/R/lib/libR.so}
#==21642== by 0x40EDA74: 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== 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
```
The state after the fix:

```plaintext
(...)
#==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 0x445F4B: ???
#==3820== by 0x44FCEC: ???
#==3820== by 0x43AB0F1: getpwuid_r@GLIBC_2.1.2 (getXXbyYY_r.c:226)
#==3820== by 0x43AA76: getpwuid (getXXbyYY.c:116)
#==3820== by 0x419F412: (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== 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.
Several CRAN (or R-Forge) packages provide the ability to execute R code in parallel:

- **NWS**
- **Rmpi**
- **snow** (using MPI, PVM, NWS or sockets)
- **multicore**
- **foreach** with **doMC**, **doSNOW**, **doMPI**
- plus several others (**rpvm**, **papply**, **taskPR**...)

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

NWS ("NetWorkSpaces") is an alternative to MPI (see below). It is based on Python and cross-platform. 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 and Ubuntu, installing the **python-nwsserver** package on at least the server node, and installing **r-cran-nws** on each client is all that is needed. Other systems may need to install the **twisted** framework for Python first.
NWS data store example

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

```r
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

# 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.

```r
# 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)
```

Dirk Eddelbuettel  High Perf. Computing with R @ ISM, Japan, Nov 2009
Also of note is the extended caretNWS version of caret by Max Kuhn, and described in a recent JSS 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 embarassingly parallel task: 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 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:

```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.
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, 
```
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

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
Hello 2 of 8 on tony
Hello 1 of 8 on joe
Hello 5 of 8 on joe
6 of 8 on tony
**Rmpi** offers a large number of 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.

---

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:

```r
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

```r
$ 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()'
	nodename
[1,] "joe"
[2,] "tony"
[3,] "mccoy"
[4,] "ron"

4 slaves are spawned successfully. 0 failed.

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).
```

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

```r
cl <- makeCluster(4, "MPI")
res <- parSapply(cl, params, \n   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.
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.

The multicore package by Simon Urbanek is a fairly recent addition to CRAN. It 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.
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`.

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

```R
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`. 
REvolution Computing released several packages to CRAN to elegantly work with serial or parallel loops:

- iterators
- foreach
- backends for `%dopar%` doMC (for multicore) and doSNOW (for snow)

Another backend package, doMPI for Rmpi, is currently under active development and should be on CRAN in due course.

**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.
The **foreach** package provides a new looping construct which can 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:  

```r
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:

```r
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:

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

```r
library(foreach)

# function that creates an iterator that returns subvectors
ivect <- 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=ivect(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")
```

The key in the `foreach` demo was the line

```
z <- foreach(y=ivect(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 `ivect` 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%`. 

Dirk Eddelbuettel  High Perf. Computing with R @ ISM, Japan, Nov 2009
In order to run this code in parallel using **multicore**, we simply use

```r
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)`.

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

```r
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)`. 
Using all those cores

Multi-core hardware is now a default, and the number of cores per cpus is expected to increase dramatically over the next few years. 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/](http://www.stat.uiowa.edu/~luke/R/experimental/)

Other (related) approaches are of course [multicore](http://www.stat.uiowa.edu/~luke/R/experimental/) discussed above 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.

Dirk Eddelbuettel
High-Perf. Computing with R @ ISM, Japan, Nov 2009

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.

Dirk Eddelbuettel
High-Perf. Computing with R @ ISM, Japan, Nov 2009
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.

A simple example:

```bash
$ 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
```

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

---

Use scripting with `r` and `slurm`

As discussed at the beginning, the `r` command of the `littler` package (as well as `R`’s `Rscript`) provide more robust alternatives to ‘batch’ of `R`.

We saw that `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 with slurm.
Having introduced snow, slurm and r, we would like to combine them.

However, there 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

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:

```r
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()
}
```
For example

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

suppressMessages(library(Rmpi))
suppressMessages(library(snow))

mpirank <- mpi.comm.rank(0)
if (mpirank == 0) {
  cat("Launching master, mpi rank=", mpirank, "\n")
  makeMPIcluster()
} else {
  cat("Launching slave with, mpi rank=", mpirank, "\n")
  sink(file="/dev/null")
  slaveLoop(makeMPImaster())
  mpi.finalize()
  q()
}

stopCluster(cl)
```

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.
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.
```r
$ 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"
[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
```

```r
A complete example

```r
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()
```
This uses \texttt{RDieHarder} to test four Mersenne-Twister implementations at once.

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

\begin{verbatim}
pdf("/tmp/snowRDH.pdf")
lapply(res, function(x) plot(x))
dev.off()

print( do.call(rbind,
             lapply(res, function(x) { x[[1]] } )))
\end{verbatim}
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 `rprng` 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,
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.
The `biglm` package operates on 'larger-than-memory' datasets by operating on 'chunks' of data at a time.

```r
make.data <- function ... # see 'help(bigglm)'
dataurl <-
    "http://faculty.washington.edu/tlumley/NO2.dat"
airpoll <- make.data(dataurl1, 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), \n    start=c(2,0,0,0),maxit=10)
summary(b)
```

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

---

`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.
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()
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>ff_matrix</td>
<td>2088</td>
<td>1e+08</td>
</tr>
<tr>
<td>b</td>
<td>numeric</td>
<td>32</td>
<td>1e+00</td>
</tr>
<tr>
<td>k</td>
<td>numeric</td>
<td>32</td>
<td>1e+00</td>
</tr>
<tr>
<td>n</td>
<td>numeric</td>
<td>32</td>
<td>1e+00</td>
</tr>
</tbody>
</table>

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).

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

```r
> 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 \( \times \) 1000 elements occupies only 372 bytes of memory. The actual size of 800 mb is allocated by the operating system, and R interfaces it via an 'external pointer' object.

We can illustrate *bigmemory* use of *biglm*:

```r
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.
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) {
  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:

```r
# 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
}
date() ## time ?
```

---

Example: Sequential data access

### Take three: `foreach()` and multicore

```r
# 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
}
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
```

Outline

1. Motivation
2. Tools for automation and scripting
3. Measuring and profiling
4. Speeding up
5. Compiled Code
6. Explicitly and Implicitly Parallel
7. Out-of-memory processing
8. Summary
In this tutorial session, we covered

- *scripting* and automation using littler, Rscript and RPy
- *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
- working with *large datasets* that exceed the available memory size

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
  

- Updated versions of the tutorial may appear at [http://dirk.eddelbuettel.com/presentations.html](http://dirk.eddelbuettel.com/presentations.html)

Do not hesitate to email me at [edd@debian.org](mailto:edd@debian.org)
Thank You!