Introduction to High-Performance R

UseR! 2008 Tutorial

Dirk Eddelbuettel

TU Dortmund
August 11, 2008

Motivation

What describes our current situation?

- Moore's Law: Computers keep getting *faster and faster*.
- But at the same time out datasets get *bigger and bigger*.
- And our research ambitions get *bigger and bigger too*.
- So we're still *waiting and waiting . . .*

Hence: A need for higher / faster / further / ... computing with R.

Motivation cont.

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

We will look at *vectorisation*, a key method for speed improvements, as well as various ways to *compile code*.

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

Next, we look at ways to compute with R *beyond the memory limits* imposed by the R engine.

Last but not least we look at ways to *automate* running R code.

Outline

Profiling

We need to know where our code spends the time it takes to compute our tasks. Measuring 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.
- The `profr` package can visualize `Rprof` data.

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

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

Lastly, we can also profile compiled code.

RProf example

In this example (taken from the manual), the two calls to `Rprof` turn profiling on and off, respectively.

```
library(MASS); library(boot)
storm.fm <- nls(Time ~ b*Viscosity/(Wt - c), stormer, \
                start = c(b=29.401, c=2.2183))
st <- cbind(stormer, fit=fitted(storm.fm))
storm.bf <- function(rs, i) {
    st$Time <-  st$fit + rs[i]
    tmp <- nls(Time ~ (b * Viscosity)/(Wt - c), st, \
               start = coef(storm.fm))
    tmp$m$getAllPars()
}
rs <- scale(resid(storm.fm), scale = FALSE) # remove mean
Rprof("boot.out")
storm.boot <- boot(rs, storm.bf, R = 4999) # pretty slow
Rprof(NULL)
```

RProf example cont.

We can run the example via either one of

```
cat profilingExample.R | R --no-save     ## N = 4999
cat profilingSmall.R | R --no-save       ## N = 99
```

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

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

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

```
R CMD Prof boot.out | less
```

Third, `profr` can directly profile, evaluate, and optionally plot, an expression. Note that we reduce *N* here:

```
plot(pr <- profr(storm.boot <- boot(rs, storm.bf, R = 99) ))
```

In this example, the code is already very efficient and no 'smoking gun' reveals itself for further improvement.

profr example

The profr function can be very useful for its 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
Rprof("/tmp/sillysum.out", interval=ival)
a <- sillysum(1e6); Rprof(NULL)
plot(parse_rprof("/tmp/sillysum.out", interval=ival))
```

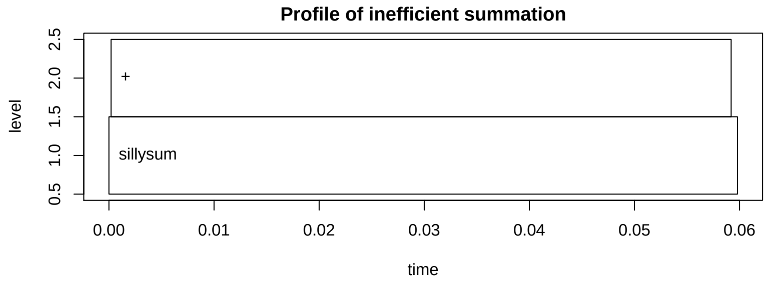
and a more efficient solution where we use a larger *N*:

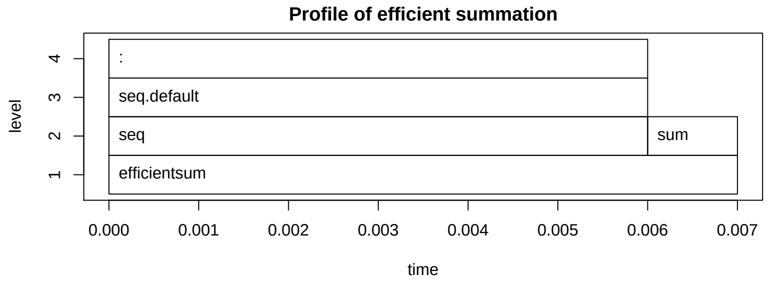
```
efficientsum <- function(N) { s <- sum(seq(1,N)); s }
ival <- 1/5000
Rprof("/tmp/effsum.out", interval=ival)
a <- efficientsum(1e7); Rprof(NULL)
plot(parse_rprof("/tmp/effsum.out", interval=ival))
```

We can run the complete example via

```
cat rprofChartExample.R | R --no-save
```

profr example cont.





RProfmem example

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`:

```
Rprofmem("/tmp/boot.memprof", threshold=1000)
storm.boot <- boot(rs, storm.bf, R = 4999)
Rprofmem(NULL)
```

Looking at the results files shows, and we quote, that *apart from some initial and final work in 'boot' there are no vector allocations over 1000 bytes.*

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

Profiling compiled code

Profiling compiled code typically entails rebuilding the binary and libraries with the `-gp` compiler option. In the case of R, a complete rebuild is required.

Add-on tools like `valgrind` and `kcachegrind` can be helpful.

Two other options are mentioned in the *R Extensions* manual section of profiling for Linux.

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 package which we will illustrate.

Profiling with Google Perftools

The Google Perftools package provides 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 profile.

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

Profiling with Google Perftools

```
# 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.

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
 11.513   0.048  11.560
>

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

[1] 5e+13
   user  system elapsed
  0.104   0.084   0.187
>
```

Replacing the loop yielded a gain of a factor of more than 100. Hence it pays to know the corpus of available functions.

Vectorisation cont.

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

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

Vectorisation cont.

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

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

The naive time is

```
> mean(replicate(10, system.time(dd.for.c())["elapsed"]))
[1] 0.3003
```

Vectorisation cont.

The case study discusses two important points that bear repeating:

- pre-allocating space helps with performance:
  `val <- double(10000)`
- switching to faster functions can help too as `tabulate` outperforms `table`.

Vectorisation cont.

However, by far the largest improvement comes from eliminating the four loop 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
```

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

Accelerated R with just-in-time compilation

Stephen Milborrow recently released a set of patches to R that allow 'just-in-time compilation' of loops and arithmetic expression. 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.

`Ra` and `jit` are still pretty young and not widely deployed yet. They are available in Debian and should be in the next Ubuntu release.

Optimised Blas

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

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

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

An example from the old README.Atlas, running with R 2.7.0, follows:

Optimised Blas cont.

```
# with Atlas
> mm <- matrix(rnorm(4*10^6), ncol = 2*10^3)
> mean(replicate(10, system.time(crossprod(mm))["elapsed"]))
[1] 3.8465

# with basic. non-optmised Blas,
# ie after dpkg --purge atlas3-base libatlas3gf-base
> mm <- matrix(rnorm(4*10^6), ncol = 2*10^3)
> mean(replicate(10, system.time(crossprod(mm))["elapsed"]))
[1] 8.9776
```

So for pure linear algebra problems, we may get an improvement by a factor of two or larger by using binary code that is optimised for the cpu class. This is likely to be more pronounced on multi-cpu machines.

Higher increases are possibly by 'tuning' the library, see the Atlas documentation.

From Blas to GPUs.

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

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

Development kits are available as e.g Nvidia CUDA, and some initial work on integration with R has been undertaken but there appear to no easy-to-install and easy-to-use kits for R – yet.

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

Compiled Code

Beyond smarter code (using e.g. vectorised expression and/or just-in-time compilation), compiled subroutines or accelerated libraries, the most direct speed gain is to switch 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++

Another 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 packages like `RApache` or `littler` offer concrete examples. This does however require a greater familiarity with R internals.

Compiled Code: The Basics

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

The canonical example in the documentation is the convolution function:

```
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

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

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 simply becomes easier using the function name and the vector arguments as all 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 Fortan, C or C++ code. Taking the first example:

```
## 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.

We can run the example via `cat inline.Fortan.R | R -no-save`.

Compiled Code: inline cont.

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

```
## Use of .Call convention with C code
## Multyplying 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])
```

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 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 analysis the result in R it still more than twice as fast than the previous best solution.

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

Compiled Code: RCpp

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

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

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

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

RCpp example

The simplest version can be set up as follows:

```
#include <Rcpp.hpp>

RcppExport SEXP dd_rcpp(SEXP v) {
  SEXP  rl = R_NilValue;           // Use this when there is nothing to be 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 to be returned as a list to R.
  rs.add("vec", vec);              // vec as named element with name 'vec'
  rl = rs.getReturnList();         // 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.

RCpp example cont.

```
#include <Rcpp.hpp>

RcppExport SEXP dd_rcpp(SEXP v) {
  SEXP  rl = R_NilValue;         // Use this when there is nothing to be returned.
  char* exceptionMesg = NULL;    // msg var in case of error

  try {
    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 to be returned as a list to R.
    rs.add("vec", vec);          // vec as named element with name 'vec'
    rl = rs.getReturnList();     // Get the list to be returned to R.
  } catch(std::exception& ex) {
    exceptionMesg = copyMessageToR(ex.what());
  } catch(...) {
    exceptionMesg = copyMessageToR("unknown reason");
  }

  if (exceptionMesg != NULL)
    error(exceptionMesg);

  return rl;
}
```

RCpp example cont.

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

```
$ R CMD SHLIB dd.rcpp.cpp -lrcpp; strip dd.rcpp.so
```

```
g++ -I/usr/share/R/include -fpic  -g -O2 -c dd.rcpp.cpp -o dd.rcpp.o
g++ -shared -o dd.rcpp.so dd.rcpp.o -lrcpp -L/usr/lib/R/lib -lR
```

Note the extra link instruction `-lrcpp` as well as the `strip` command to remove extraneous debugging information.

RCpp example cont.

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

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

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

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

```
[1] 0.00047
```

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

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

Embarassingly parallel

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

- NWS
- Rmpi
- snow
- papply
- taskPR

NWS Intro

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

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

On Debian and Ubuntu, installing the `python-nwsserver` package on at least the server node, and installing `r-cran-nws` on each client is all that is needed.

Other system may need to install the `twisted` framework for `Python` first.

NWS data store example

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

```
ws <- netWorkSpace('r place')    # create a 'value store'

nwsStore(ws, 'x', 1)             # place a value (as a 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 more
```

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

NWS sleigh example

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

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

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

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

# run a lapply-style eachWorker over each element of list
eachElem(s, function(x) {x+1}, list(1:10))

stopSleigh(s)
```

NWS sleigh cont.

The NWS framework and `sleigh` object will be described in more more detail in the presentation by David Henderson et al on Wednesday morning at UseR! 2008.

Also of note is the extended `caretNWS` version of `caret`. It uses `nws` and `sleigh` for embarassingly parallel task in classification and regression model training and evaluation: bagging, boosting, cross-validation, bootstrapping, ...

Rmpi

`Rmpi` is a CRAN package that provides and 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 standard 'Hello, World!' program:

```
#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 -n 8 -H ron,joe,wayne,tony /tmp/mpiHelloWorld
```

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

Notice how the order of execution is indeterminate.

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

```
#!/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 -n 8 -H ron,joe,wayne,tony /tmp/mpiHelloWorld.r
Hello, rank 0 size 8 on ron
Hello, rank 4 size 8 on ron
Hello, rank 3 size 8 on tony
Hello, rank 7 size 8 on tony
Hello, rank 6 size 8 on wayne
Hello, rank 2 size 8 on wayne
Hello, rank 5 size 8 on joe
Hello, rank 1 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,wayne,tony\
     r -lRmpi -e'cat("Hello", \
     mpi.comm.rank(0), "of", \
     mpi.comm.size(0), "on", \
     mpi.get.processor.name(), "\n")'
```

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

snow

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

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

A simple example:

```
nbNodes <- 8
cl <- makeCluster(nbNodes, "MPI")
clusterCall(cl, function() Sys.info()[c("nodename","machine")]
```

snow: Example

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

```
       4 slaves are spawned successfully. 0 failed.
     nodename
[1,] "joe"
[2,] "ron"
[3,] "joe"
[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 two 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.

papply, biopara and taskPR

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

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

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

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

slurm resource management and queue system

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

Paraphrasing from the slurm website:

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

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

slurm example

Slurm wraps around Open MPI. That is an advantage inasmuch as it permits use of `Rmpi` and other recent MPI-using applications built against Open MPI.

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

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

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

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

slurm example

Additional coomand-line tools of interest are `salloc`, `sbatch`, `scontrol` 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 in Wednesday’s presentation.

Using all those cores

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

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

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

They can be found at
`http://www.stat.uiowa.edu/~luke/R/experimental/`

pnmath and pnmath0

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

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

Profiling is probably the best way to assess the possible usefulness. As a quick illustrations, 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,

   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.

Extending physical RAM limits

Two recently released CRAN packages (both of which will have UseR! 2008 presentations too) 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 objects not managed by R

All of these 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 provides a way to operate 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)
```

ff

ff is the winner of the UseR! 2007 'large datasets' competition.

An illustration of ff use, taken from an example in the package:

```
data("trees")
# create ffm object, convert 'trees' data, creates two files
m <- ffm("foom.ff", c(31, 3))
m[1:31, 1:3] <- trees[1:31, 1:3]
# create a ffm.data.frame wrapper around the ffm object
ffmdf <- ffm.data.frame(m, c("Girth", "Height", "Volume"))
# define formula and fit the model
fg        <- log(Volume) ~ log(Girth) + log(Height)
ffmdf.out <- bigglm(fg,data=ffmdf,chunksize=10,sandwich=TRUE)
```

The ffm function creates a flat-file based matrix which is then filled with data from 'trees' dataset, and converted into an 'ffm.data.frame' which biglm can operate on.

Running object.size() on the ff object shows that it occupies less memory than the (puny) trees dataset.

bigmemory

The bigmemory package allows us to allocate and access memory managed by the operating system but 'outside' of the view of R.

```
> object.size( big.matrix(1000,1000, "double") )
[1] 372
> object.size( matrix(double(1000*1000), ncol=1000) )
[1] 8000112
```

Here we see that 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.

bigmemory cont.

We can illustrate `bigmemory` by adapting the previous example:

```
bm <- as.big.matrix(as.matrix(trees), type="double")
colnames(bm) <- colnames(trees)
fg <- log(Volume) ~ log(Girth) + log(Height)
bm.out <- biglm.big.matrix(fg, data=bm, chunksize=10, \
                           sandwich=TRUE)
```

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.

`bigmemory` can also provide shared memory allocation: one (large) object can accessed by several R process as proper locking mechanisms are provided.

littler

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

```
#!/usr/bin/env r
# a simple example to install one or more packages

if (is.null(argv) | length(argv)<1) {
  cat("Usage: 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, dependencies=TRUE)
```

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

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

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

Rscript

`Rscript` can be used in a similar fashion.

Previously we had to use

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

or some shell-script varitions around this theme.

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

RPy

RPy packages provides access to R from Python:

```
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 conversion back on

fit = r.predict(NNModel,XG)
print fit
```

Wrapping up

In this tutorial session, we have

- seen several ways to profile execution times;
- looked a different vectorisation examples, as well speed increases from using compiled code;
- provided a brief introduction to parallel execution frameworks such as `NWS`, `MPI` and `snow`;
- looked at packages such as `ff` and `bigmemory` that can help with larger data sets;
- briefly looked at ways to script R tasks using `littler` and `Rscript`

Appendix: Software Support

The tutorial is supported by a 'live cdrom'. The (updated) iso file `Quantian_UseR2008_tutorial_v2.iso` can be downloaded from `http://quantian.alioth.debian.org/`. Version one is also at `http://http://quantian.fhcrc.org`.

The iso image contains a complete Debian operating sytem including the graphical KDE user interface. All the software demonstrated during the tutorial is available and fully functional. This includes

- R and all packages used,
- the accelerated Ra variant,
- Open MPI, NWS, Slurm and more,
- Emacs, ESS and a few other tools.

The versions correspond to the to the late-July 2008 snapshot of the upcoming Debian release.

Appendix: Software Support cont.

The iso file can be burned to a cdrom that can be used to boot up a workstation.

Alternatively, virtualisation software such as

- *VMware Player* (Windows, Linux),
- *VMware Fusion* (Mac OS X),
- *VirtualBox* (Windows, Linux) or
- *QEMU* (Linux)

can be used to run a ’virtual’ guest computer alongside the host computer.

The software can also be installed to disk and updated using standard Debian tools; see the documentation for the ’Debian Live’ tools used.

Appendix: Software Support cont.

Known issues with the provided iso file are:

- The cdrom appears to fail on some Dell models, there may be a BIOS incompatibility with the syslinux bootloader. Failures with the Parallels virtualisation for OS X were also reported.
- No wireless extensions: if a laptop is booted off a cdrom, chances are that wireless will not be supported due to lack of binary firmware.
- The first release lacked Ra and the Open MPI compilers.
- With the most recent VirtualBox releases 1.6.*, the screen resolution defaults to a 1280x1024 even if the host is running at a smaller resolution.