Speeding up Your R Code

Serial solutions before parallel solutions

- User R code often inefficient (high-level code = deep complexity)
  - Profile and improve code first
  - Vectorize loops if possible
  - Compute once if not changing
  - Know when copies are made
- Improve matrix algebra speed with a fast multithreaded library such as OpenBLAS
- Move kernels into compiled language, such as C/C++
- Then consider parallel computation (multicore and distriuted)
- If memory bound, consider distributed parallel solutions

Profiling Why Profile?

- Because performance matters.
- Bad practices scale up!
- Your bottlenecks may surprise you.
- One line of R code can touch a lot of data.
- High-level language has deep complexity
- There is no compiler to optimize code

Performance Profiling Tools: `system.time()`

`system.time()` is a basic R utility for timing expressions

```
x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)

system.time(t(x) %*% x)

system.time(crossprod(x))

system.time(cov(x))
```

Performance Profiling Tools: `Rprof()`

Samples call stack (default every 0.02 seconds)

fastR/profile.R

```
> x <- matrix( rnorm( 10000*250 ), nrow = 10000, ncol = 250)
> Rprof()
> invisible( prcomp( x ) )
> Rprof( NULL )
> summaryRprof()
$by.self
                self.time self.pct total.time total.pct
"La.svd"             0.64    78.05       0.70     85.37
"%*%"                0.06     7.32       0.06      7.32
"aperm.default"      0.04     4.88       0.04      4.88
"is.finite"          0.04     4.88       0.04      4.88
"matrix"             0.04     4.88       0.04      4.88

$by.total
                 total.time total.pct self.time self.pct
"prcomp.default"       0.82    100.00      0.00     0.00
"prcomp"               0.82    100.00      0.00     0.00
"svd"                  0.72     87.80      0.00     0.00
"La.svd"               0.70     85.37      0.64    78.05
"%*%"                  0.06      7.32      0.06     7.32
### output truncated by presenter

$sample.interval
[1] 0.02

$sampling.time
[1] 0.98
```

Performance Profiling Tools: `Rprof()`

fastR/profile.R

```
> Rprof( interval = .99 )
> invisible( prcomp( x ) )
> Rprof( NULL )
> summaryRprof()
$by.self
[1] self.time  self.pct   total.time total.pct
<0 rows> (or 0-length row.names)

$by.total
[1] total.time total.pct  self.time  self.pct
<0 rows> (or 0-length row.names)

$sample.interval
[1] 0.99

$sampling.time
[1] 0
```

Performance Profiling Tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

fastR/benchmark.R

```
x <- matrix( rnorm( 10000*500 ), nrow = 10000, ncol = 500 )

f <- function( x ) t( x ) %*% x
g <- function( x ) crossprod( x )

library( rbenchmark )
benchmark( f( x ), g( x ) )

#   test replications elapsed relative
# 1 f(x)          100  64.153    2.063
# 2 g(x)          100  31.098    1.000
```

Performance Profiling Tools: pbdPAPI

pbdPAPI enables PAPI library for accessing hardware counters (only unix):

PAPI_cache_access.R

```
library(pbdPAPI)
library(inline)
bad_cache_access <- "
␣␣int␣i,␣j;
␣␣const␣int␣n␣=␣INTEGER(n_)[0];
␣␣Rcpp::NumericMatrix␣x(n,␣n);
␣␣for␣(i=0;␣i<n;␣i++)
␣␣␣␣for␣(j=0;␣j<n;␣j++)
␣␣␣␣␣␣x(i,␣j)␣=␣1.;
␣␣return␣x;
"
```

```
good_cache_access <- "
␣␣int␣i,␣j;
␣␣const␣int␣n␣=␣INTEGER(n_)[0];
␣␣Rcpp::NumericMatrix␣x(n,␣n);
␣␣for␣(j=0;␣j<n;␣j++)
␣␣␣␣for␣(i=0;␣i<n;␣i++)
␣␣␣␣␣␣x(i,␣j)␣=␣1.;
␣␣return␣x;
"
```

```
bad <- cxxfunction(signature(n_ = "integer"), body = bad_cache_access,
    plugin = "Rcpp")
good <- cxxfunction(signature(n_ = "integer"), body = good_cache_access,
     plugin = "Rcpp")
n <- 10000L

### Summary of cache misses
system.cache(bad(n))
system.cache(good(n))

### Ratio of total cache misses to total cache accesses
system.cache(bad(n), events = "l2.ratio")
system.cache(good(n), events = "l2.ratio")
```

Performance Profiling Tools: pbdPAPI

```r
library(pbdPAPI)

x <- runif(1e6)

### Sorting is not a floating point operation.
system.flops(sort(x))

### It does require lots of memory access, though.
system.cache(sort(x))

system.utilization(sort(x))
```

Performance Profiling Tools: pbdPROF

```r
library(pbdPROF)
prof <- read.prof("output.mpiP")
plot(prof, plot.type="messages2")
```

Profiling Summary

- *Profile, profile, profile.*
- Use `system.time()` to get a general sense of a method.
- Use **rbenchmark's `benchmark()` to compare 2 methods.**
- **Use `Rprof()` for more detailed profiling.**
- **More advanced profiling: pbdPAPI**
- **Parallel code profiling pbdPROF.**

Vectorizing

vectorizing.R

```
n <- 1e5
x <- seq( 0, 1, length.out = n )
f <- function( x ) exp( x^3 + 2.5*x^2 + 12*x + 0.12 )
y1 <- numeric( n )

set.seed( 12345 )
system.time(
  for( i in 1:n )
    y1[ i ] <- f( x[ i ] ) + rnorm( 1 )
)

set.seed( 12345 )
system.time(
  y2 <- f( x ) + rnorm( n )
)

all.equal( y1, y2 )
```

Compute Once if not Changing

Bad Loop

```
for (i in 1:n){
  Y <-  t(A) %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}

Q
```

Good Loop (from pbdML)

```
tA <- t(A)

for (i in 1:n){
  Y <- tA %*% Q
  Q <- qr.Q(qr(Y))
  Y <- A %*% Q
  Q <- qr.Q(qr(Y))
}

Q
```

Example from a Real R Package

Exerpt from Original function

```
while(i<=N){
  for(j in 1:i){
     d.k <- as.matrix(x)[l==j,l==j]
       ...
```

Exerpt from Modified function

```
x.mat <- as.matrix(x)

while(i<=N){
  for(j in 1:i){
     d.k <- x.mat[l==j,l==j]
       ...
```

By changing just 1 line of code, performance of the main method improved by **over 3.5**$\times$!

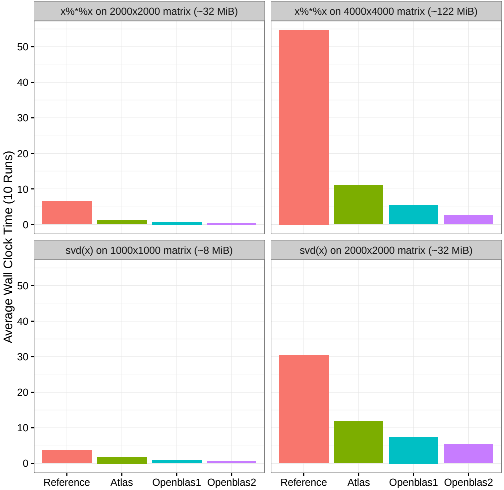
OpenBLAS (multithreaded Basic Linar Algebra Subroutines)

- pbdr-workshop container includes OpenBLAS and **openblasctl**
- **Native Install: http://www.openblas.net/**
- **Batch thread control: OPENBLAS_NUM_THREADS=n**
- **Dynamic thread control: "wrathematics/openblasctl" micropackage on GitHub**

```
x <- matrix( rnorm( 1e6*2e2 ), nrow = 1e6 )

system.time( y <- crossprod( x ) )

library( openblasctl )

openblas_set_num_threads( 1 )
system.time( y <- t( x ) %*% x )
system.time( z <- crossprod( x ) )

openblas_set_num_threads( 2 )
system.time( y <- t( x ) %*% x )
system.time( z <- crossprod( x ) )

openblas_set_num_threads( 4 )
system.time( y <- t( x ) %*% x )
system.time( z <- crossprod( x ) )
```

- Same is available with Intel's MKL

Benefit of OpenBLAS



Schmidt, Chen, Matheson, and Ostrouchov (2017). Programming with BIG Data in R: Scaling Analytics from One to Thousands of Nodes, Big Data Research, 8, p.1-11.