High Performance Computing with R

Drew Schmidt

April 6, 2014

Contents

1. Introduction
   - Compute Resources
   - HPC Myths

Compute Resources

- Your laptop.
- Your own server.
- The cloud.
- NSF resources.

XSEDE

- (+) Free*!!!
- (+) Access to *massive* compute resources.
- (+) OS image and software managed by others.
- (-) 1-3 month turnaround for new applications.
- (-) Application consists of more than "here's my credit card."
- (-) Some restrictions apply.

https://www.xsede.org/allocations

What is High Performance Computing (HPC)?

High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

HPC Myths

1. We don't need to worry about it.
2. HPC requires a supercomputer.
3. HPC is only for academics.
4. HPC is too expensive.
5. You can't do HPC with R.
6. There's no need for HPC in biology.
7. Every question requires an HPC solution.

Companies Using HPC

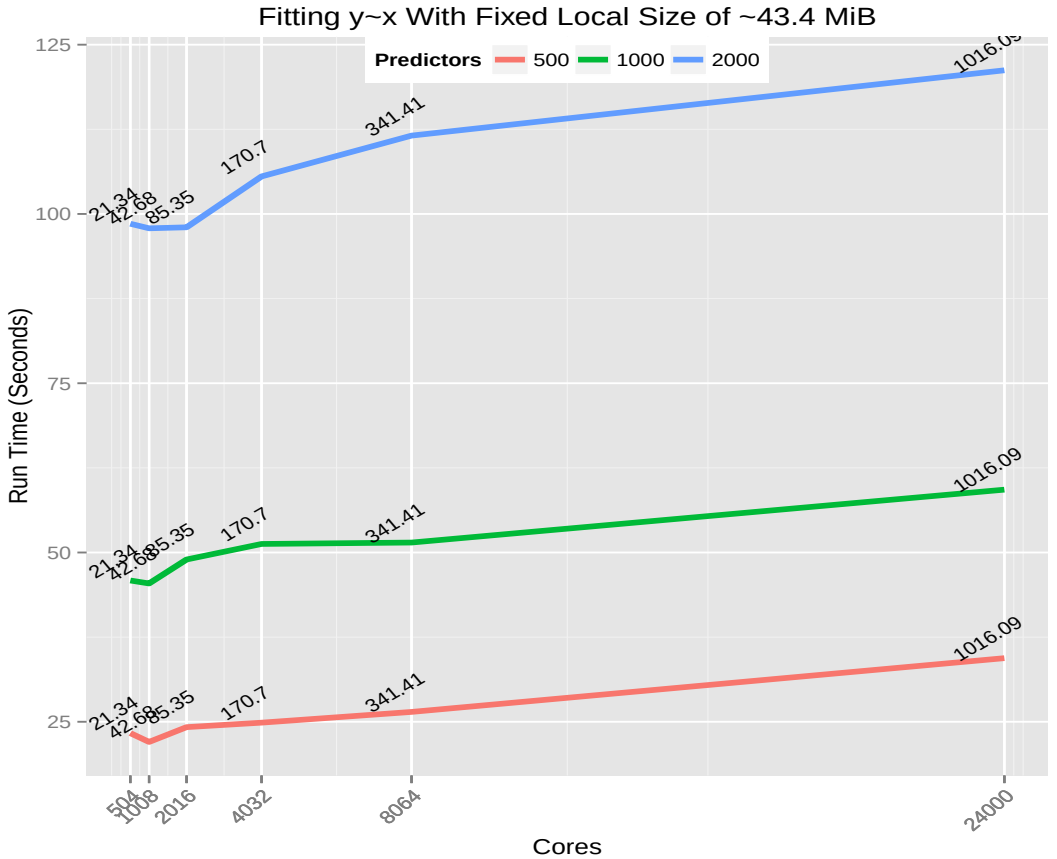
bing™

Johnson&Johnson

Google

Programming with Big Data in R (pbdR)

pbdR
Programming with Big Data in R



HPC in the Biological Sciences

According to Manuel Peitsch, co-founder of the Swiss Institute of Bioinformatics, HPC is "essential" and plays a critical role in the life sciences in 4 ways:

1. Massive amounts of data generated by modern 'omics' and genome sequencing technologies.
2. Modeling increasingly large biomolecular systems using quantum mechanics/molecular mechanics and molecular dynamics.
3. Modeling biological networks and simulating how network perturbations lead to adverse outcomes and disease.
4. Simulation of organ function.

2 Profiling and Benchmarking

- Why Profile?
- Profiling R Code
- Other Ways to Profile
- Summary

Why Profile?

- Because performance matters.
- Your bottlenecks may surprise you.
- Because R is dumb.
- R users claim to be data people. . . so act like it!

Compilers often correct bad behavior...

A Really Dumb Loop

```
int main(){
    int x, i;
    for (i=0; i<10;
        i++)
        x = 1;
    return 0;
}
```

clang -O3 example.c

```
main:
        .cfi_startproc
# BB#0:
        xorl
            %eax,
            %eax
        ret
```

clang example.c

```
main:
        .cfi_startproc
# BB#0:
        movl    $0, -4(%rsp)
        movl    $0, -12(%rsp)
.LBB0_1:
        cmpl    $10, -12(%rsp)
        jge     .LBB0_4
# BB#2:
        movl    $1, -8(%rsp)
# BB#3:
        movl    -12(%rsp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rsp)
        jmp     .LBB0_1
.LBB0_4:
        movl    $0, %eax
        ret
```

R will not!

Dumb Loop

```
1 for (i in 1:n){
2   tA <- t(A)
3   Y <- tA %*% Q
4   Q <- qr.Q(qr(Y))
5   Y <- A %*% Q
6   Q <- qr.Q(qr(Y))
7 }
8
9 Q
```

Better Loop

```
1  tA <- t(A)
2
3 for (i in 1:n){
4   Y <- tA %*% Q
5   Q <- qr.Q(qr(Y))
6   Y <- A %*% Q
7   Q <- qr.Q(qr(Y))
8 }
9
10 Q
```

Example from the clusterGenomics Package

Exerpt from Original findW function

```
1  n <- nrow(as.matrix(dX))
2
3  while(k<=K){
4      for(i in 1:k){
5        #Sum of within-cluster dispersion:
6        d.k <- as.matrix(dX)[labX==i,labX==i]
7        D.k <- sum(d.k)
8        ...
```

Exerpt from Modified findW function

```
1  dX.mat <- as.matrix(dX)
2  n <- nrow(dX.mat)
3
4  while(k<=K){
5      for(i in 1:k){
6        #Sum of within-cluster dispersion:
7        d.k <- dX.mat[labX==i,labX==i]
8        D.k <- sum(d.k)
9        ...
```

By changing just 2 lines of code, I was able to improve the speed of his method by **over 350%**!

Runtime Tools

Getting simple timings as a basic measure of performance is easy, and valuable.

- `system.time()`
- **rbenchmark**
- `Rprof()`

Performance Profiling Tools: `system.time()`

`system.time()` is a basic R utility for giving run times of expressions

```
> x <- matrix(rnorm(10000*500), nrow=10000, ncol=500)
> system.time(t(x) %*% x)
   user  system elapsed
  0.459   0.028   0.488
> system.time(crossprod(x))
   user  system elapsed
  0.234   0.000   0.234
> system.time(cov(x))[3]
elapsed
  1.428
```

Performance Profiling Tools: `system.time()`

Improving the **rexpokit** Package

```
library(rexpokitold)
system.time(expokit_dgpadm_Qmat(x))[3]
# 5.496

library(rexpokit)
system.time(expokit_dgpadm_Qmat(x))[3]
# 4.164

5.496/4.164
# 1.319885
```

Performance Profiling Tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

```
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
```

Rprof()

A very useful tool for profiling R code

```
Rprof(filename="Rprof.out", append=FALSE, interval=0.02,
  memory.profiling=FALSE, gc.profiling=FALSE,
  line.profiling=FALSE, numfiles=100L, bufsize=10000L)
```

Rprof()

```
data(iris)
myris <- iris[1:100, ]
fam <- binomial(logit)

Rprof()
mymdl <- replicate(1, myglm <- glm(Species ~ .,
    family=fam, data=myris))
Rprof(NULL)
summaryRprof()

Rprof()
mymdl <- replicate(10, myglm <- glm(Species ~ .,
    family=fam, data=myris))
Rprof(NULL)
summaryRprof()

Rprof()
mymdl <- replicate(100, myglm <- glm(Species ~ .,
    family=fam, data=myris))
Rprof(NULL)
summaryRprof()
```

Other Profiling Tools

- `Rprofmem()`
- `tracemem()`
- perf (Linux)
- PAPI, TAU, ...

Profiling with **pbdPROF**

1. Rebuild pbdR packages

```
R CMD INSTALL
    pbdMPI_0.2-1.tar.gz \
    --configure-args= \
    "--enable-pbdPROF"
```

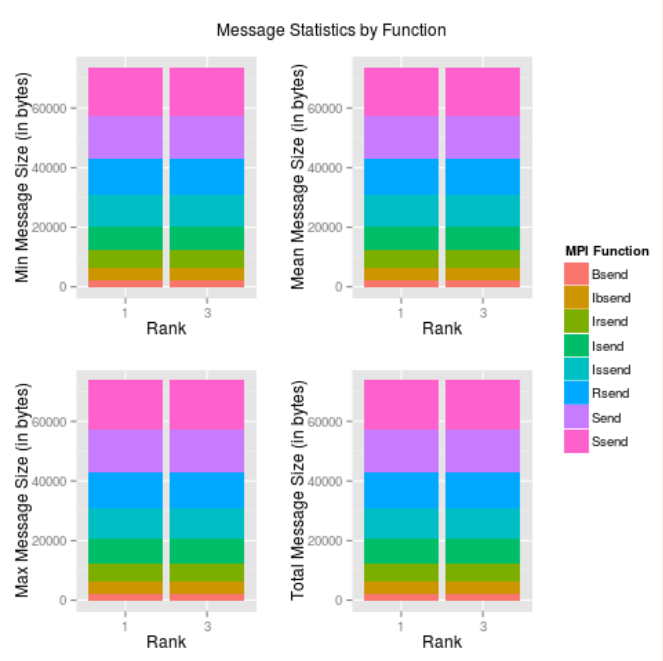
2. Run code

```
mpirun -np 64 Rscript
    my_script.R
```

3. Analyze results

```
1 library(pbdPROF)
2 prof <- read.prof(
      "profiler_output.mpiP")
3 plot(prof)
```

Publication-quality graphs



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.
- Other tools exist for more hardcore applications.

3 Writing Better R Code

- Functions
- Loops, Ply Functions, and Vectorization
- Summary

Serial R Improvements

Slow serial code $\Longrightarrow$ slow parallel code

Function Evaluation

- Function calls are *comparatively* expensive (≈10x slower than C)
- In absolute terms, the abstraction is worth the price.
- Recursion *sucks*. Avoid at all costs.

Recursion 1

```
1  fib1 <- function(n)
2  {
3    if (n == 0 || n == 1)
4      return( 1L )
5    else
6      return( fib1(n-1) + fib1(n-2) )
7  }
8
9
10 fib2 <- function(n)
11 {
12   if (n == 0 || n == 1)
13     return( 1L )
14
15   f0 <- 1L
16   f1 <- 1L
17
18   i <- 1L
19   fib <- 0L
```

Recursion 2

```
20    while (i < n)
21    {
22      fib <- f0 + f1
23      f0 <- f1
24      f1 <- fib
25      i <- i+1
26    }
27
28    return( fib )
29  }
```

Recursion 3

```
library(rbenchmark)
n <- 20

benchmark(fib1(n), fib2(n))
#       test replications elapsed relative
# 1 fib1(n)          100   2.047   1023.5
# 2 fib2(n)          100   0.002      1.0

system.time(fib1(45))[3]
2934.146
system.time(fib2(45))[3]
3.0616e-5

# Relative performance
2934.146/3.0616e-5
# 95837013
```

Loops, Plys, and Vectorization

- Loops are slow.
- `apply()`, `Reduce()` are just `for` loops.
- `Map()`, `lapply()`, `sapply()`, `mapply()` (and most other core ones) are *not* `for` loops.
- Vectorization is the fastest of these options, but tends to be much more memory wasteful.
- *Ply functions are not vectorized*.

Loops: Best Practices

- *Profile, profile, profile.*
- Evaluate how practical it is to rewrite as an `lapply()`, vectorize, or push to compiled code.
- *Preallocate, preallocate, preallocate.*

Loops 1

```
1  f1 <- function(n){
2    x <- c()
3    for (i in 1:n){
4      x <- c(x, i^2)
5    }
6
7    x
8  }
9
10
11 f2 <- function(n){
12   x <- integer(n)
13   for (i in 1:n){
14     x[i] <- i^2
15   }
16
17   x
18 }
```

Loops 2

```
library(rbenchmark)
n <- 1000

benchmark(f1(n), f2(n))
   test replications elapsed relative
1 f1(n)          100   0.234    2.412
2 f2(n)          100   0.097    1.000
```

Ply's: Best Practices

- Most ply's are just shorthand/higher expressions of loops.
- Generally not much faster (if at all), especially with the compiler.
- Thinking in terms of `lapply()` can be useful however...

Vectorization

- `x+y`
- `x[, 1] <- 0`
- `rnorm(1000)`

Plys and Vectorization

```
1 f3 <- function(n){
2   sapply(1:n, function(i) i^2)
3 }
4
5 f4 <- function(n){
6   (1:n)*(1:n)
7 }
```

```
library(rbenchmark)
n <- 1000

benchmark(f1(n), f2(n), f3(n), f4(n))
    test replications elapsed relative
1 f1(n)          100   0.210      210
2 f2(n)          100   0.096       96
3 f3(n)          100   0.084       84
4 f4(n)          100   0.001        1
```

Loops, Plys, and Vectorization 1

```
1  f1 <- function(ind){
2    sum <- 0
3    for (i in ind){
4      if (i%%2 == 0)
5        sum <- sum - log(i)
6      else
7        sum <- sum + log(i)
8    }
9    sum
10 }
11
12 f2 <- function(ind){
13   sum(sapply(X=ind, FUN=function(i) if (i%%2==0) -log(i)
         else log(i)))
14 }
15
16 f3 <- function(ind){
17   sign <- replicate(length(ind), 1L)
18   sign[seq.int(from=2, to=length(ind), by=2)] <- -1L
```

Loops, Plys, and Vectorization 2

```
19
20    sum(sign*log(ind))
21  }
22
23
24  library(rbenchmark)
25  ind <- 1:50000
26  benchmark(f1(ind), f2(ind), f3(ind))
```

```
library(rbenchmark)
ind <- 1:50000

benchmark(f1(ind), f2(ind), f3(ind), replications=10)
      test replications elapsed relative
1 f1(ind)           10   0.428    1.309
2 f2(ind)           10   1.018    3.113
3 f3(ind)           10   0.327    1.000
```

Loops, Plys, and Vectorization 3

Summary

- Avoid recursion at all costs.
- Vectorize when you can.
- Pre-allocate your data in loops.

4 All About Compilers and R

- Building R with a Different Compiler
- The Bytecode Compiler
- Bringing Compiled C/C++/Fortran to R
- Summary

Better Compiler

- GNU (gcc/gfortran) and clang/gfortran are free and will compile anything, but don't produce the fastest binaries.
- Don't even bother with anything from Microsoft.
- Intel icc is very fast on intel hardware. ($\approx$ 20% over GNU)

$$\text{Better compiler} \implies \text{Faster R}$$

Compiling R with icc and ifort

- Faster, but not painless.
- Requires Intel Composer suite license ($$$)
- Improvements are most visible on Intel hardware.
- See Intel's help pages for details.

The Compiler Package

- Released in 2011 (Tierney)
- Bytecode: sort of like machine code for interpreters...
- Improves R code speed 2-5% generally.
- Does best on loops.

Bytecode Compilation

- By default, packages are not (bytecode) compiled.
- Exceptions: base (**base**, **stats**, ...) and recommended (**MASS**, **Matrix**, ...) packages.
- Downsides to package compilation: (1) bigger install size, (2) longer install process.
- Upsides: faster.

Compiling a Function

```
1  test <- function(x) x+1
2  test
3  # function(x) x+1
4
5  library(compiler)
6
7  test <- cmpfun(test)
8  test
9  # function(x) x+1
10 # <bytecode: 0x38c86c8>
11
12 disassemble(test)
13 # list(.Code, list(7L, GETFUN.OP, 1L, MAKEPROM.OP, 2L,
       PUSHCONSTARG.OP,
14 #      3L, CALL.OP, 0L, RETURN.OP), list(x + 1, `+`,
       list(.Code,
15 #      list(7L, GETVAR.OP, 0L, RETURN.OP), list(x)), 1))
```

Compiling Packages

From R

```
1 install.packages("my_package", type="source",
      INSTALL_opts="--byte-compile")
```

From The Shell

```
1 export R_COMPILE_PKGS=1
2 R CMD INSTALL my_package.tar.gz
```

Compiling YOUR Package

- In the DESCRIPTION file, you can set `ByteCompile: yes` to require bytecode compilation (overridden by `--no-byte-compile`).
- Not recommended during development.
- CRAN may yell at you.

(Machine Code) Compiled Code

- Moving to compiled code can be difficult.
- But performance is *very compelling*.
- We will talk about one popular way on Tuesday: **Rcpp**.

Extra Credit

Compare the bytecode of these two functions:

Wasteful

```
1  f <- function(A, Q){
2    n <- ncol(A)
3    for (i in 1:n){
4     tA <- t(A)
5      Y <- tA %*% Q
6      Q <- qr.Q(qr(Y))
7      Y <- A %*% Q
8      Q <- qr.Q(qr(Y))
9    }
10
11   Q
12 }
```

Less Wasteful

```
1  g <- function(A, Q){
2    n <- ncol(A)
3    tA <- t(A)
4    for (i in 1:n){
5      Y <- tA %*% Q
6      Q <- qr.Q(qr(Y))
7      Y <- A %*% Q
8      Q <- qr.Q(qr(Y))
9    }
10
11   Q
12 }
```

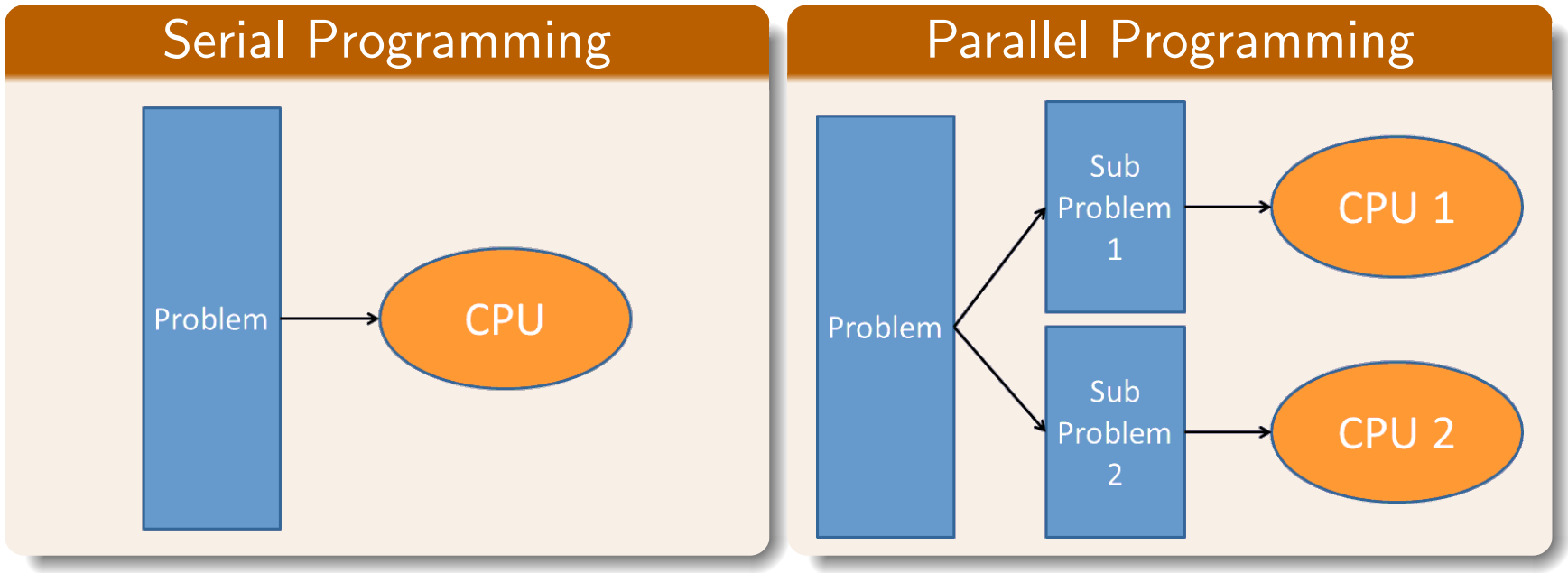
Summary

- Compiling R itself with a different compiler can improve performance, but is non-trivial.
- The compiler package offers small, but free speedup.
- The (bytecode) compiler works best on loops.

5 An Overview of Parallelism

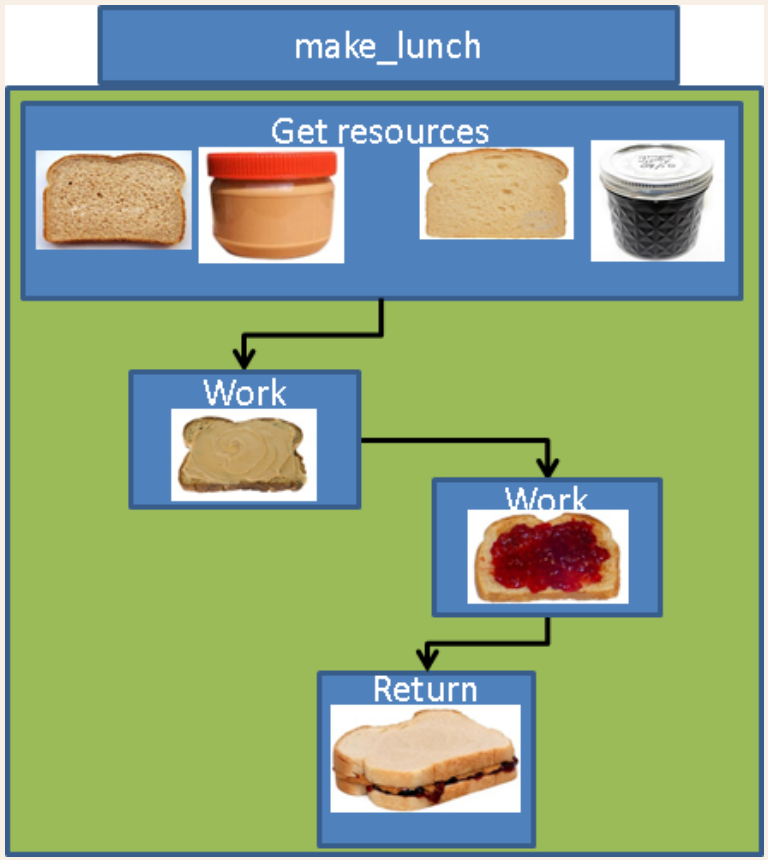
- Terminology: Parallelism
- Choice of BLAS Library
- Guidelines
- Summary

Parallelism

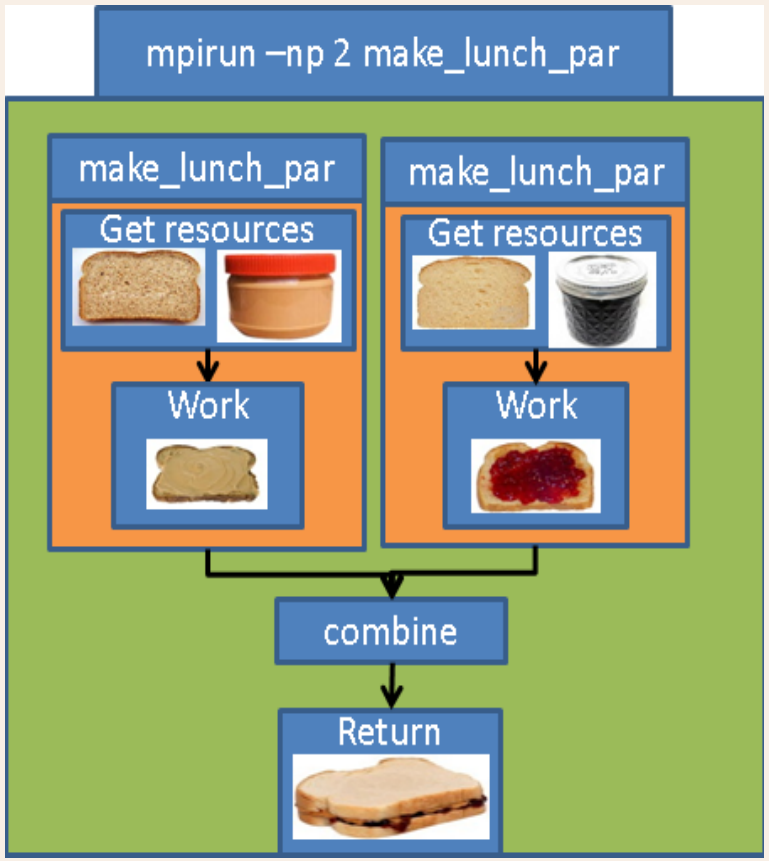


Parallelism

Serial Programming



Parallel Programming



Parallel Programming Vocabulary: Difficulty in Parallelism

1. *Implicit parallelism*: Parallel details hidden from user
   Example: Using multi-threaded BLAS

2. *Explicit parallelism*: Some assembly required...
   Example: Using the `mclapply()` from the **parallel** package

3. *Embarrassingly Parallel* or *loosely coupled*: Obvious how to make parallel; lots of independence in computations.
   Example: Fit two independent models in parallel.

4. *Tightly Coupled*: Opposite of embarrassingly parallel; lots of dependence in computations.
   Example: Speed up model fitting for one model.

Speedup

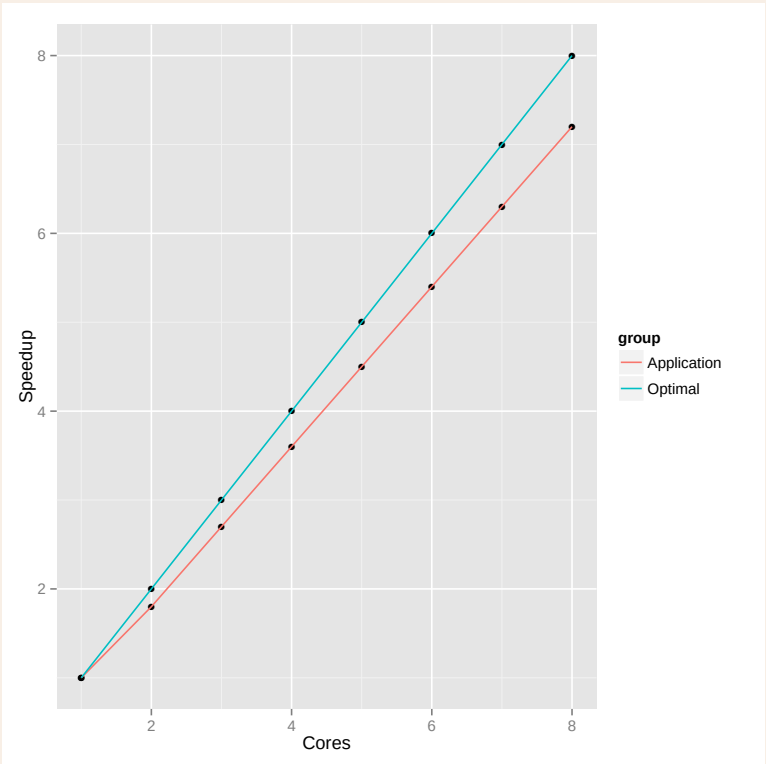
- *Wallclock Time*: Time of the clock on the wall from start to finish
- *Speedup*: unitless measure of improvement; more is better.

$$S_{n_1,n_2} = \frac{\text{Time for } n_1 \text{ cores}}{\text{Time for } n_2 \text{ cores}}$$

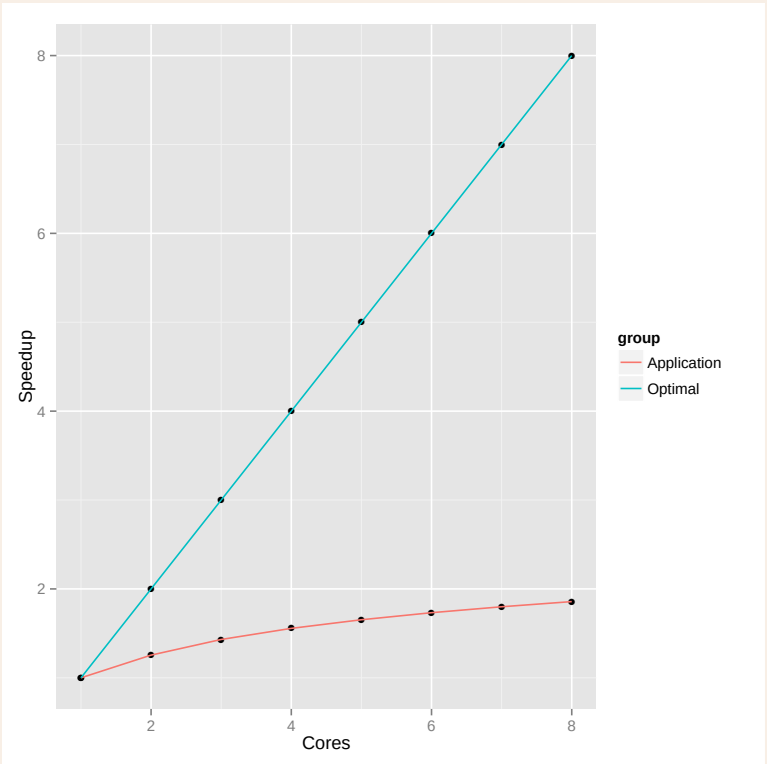
  - n_1 is often taken to be 1
  - In this case, comparing parallel algorithm to serial algorithm

Speedup

Good Speedup



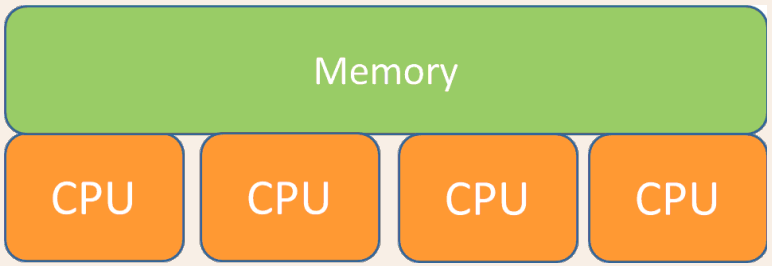
Bad Speedup



Shared and Distributed Memory Machines

Shared Memory

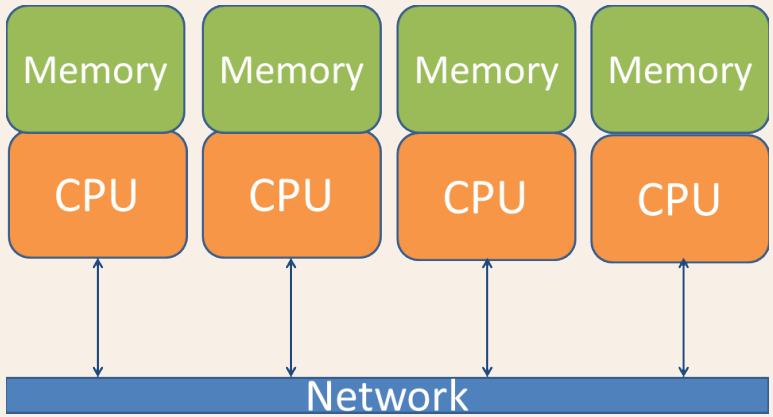
Direct access to read/change memory (one node)



Examples: laptop, GPU, MIC

Distributed

No direct access to read/change memory (many nodes); requires communication



Examples: cluster, server, supercomputer

Shared and Distributed Memory Machines

Shared Memory Machines

Thousands of cores

Nautilus, University of Tennessee
1024 cores
4 TB RAM

Distributed Memory Machines

Hundreds of thousands of cores

Kraken, University of Tennessee
112,896 cores
147 TB RAM

Shared and Distributed Programming from R

Shared Memory

Examples: **parallel**, **snow**, **foreach**, **gputools**, **HiPLARM**

Distributed

Examples: **pbdR**, **Rmpi**, **RHadoop**, **RHIPE**

CRAN HPC Task View

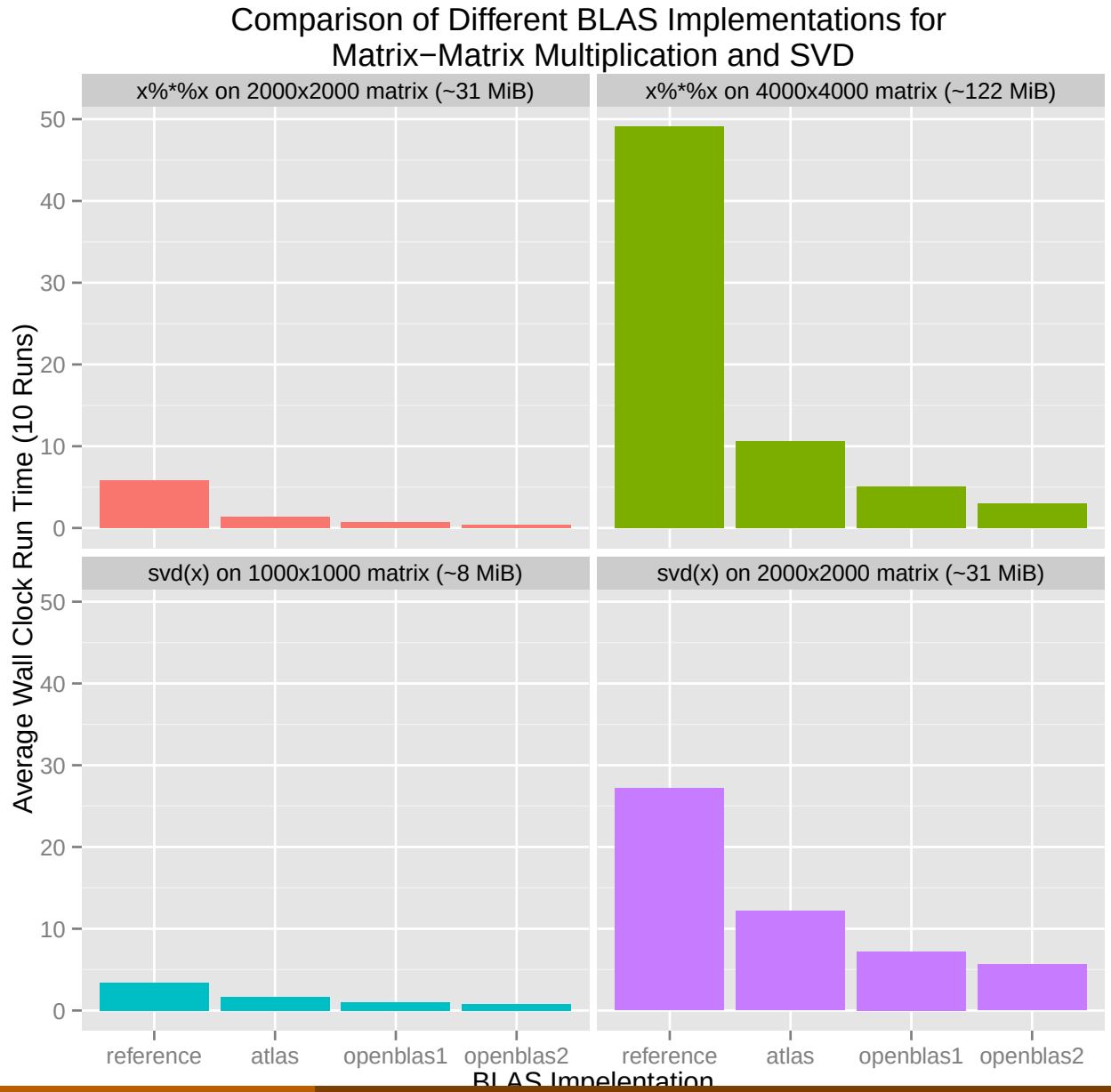
For more examples, see: http://cran.r-project.org/web/views/HighPerformanceComputing.html

The BLAS

- Basic Linear Algebra Subprograms.
- Simple vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3).
- R uses BLAS (and LAPACK) for most linear algebra operations.
- There are different implementations available, with massively different performance.
- Several multithreaded BLAS libraries exist.

Benchmark

```
1  set.seed(1234)
2  m <- 2000
3  n <- 2000
4  x <- matrix(
5    rnorm(m*n),
6    m, n)
7
8  object.size(x)
9
10 library(rbenchmark)
11
12 benchmark(x%*%x)
13 benchmark(svd(x))
```



Using openblas

On Debian and derivatives:

```
sudo apt-get install libopenblas-dev
sudo update-alternatives --config libblas.so.3
```

Warning: doesn't play nice with the **parallel** package!

Independence

- Parallelism requires *independence*.
- Separate evaluations of R functions is embarrassingly parallel.
- For bio applications, this may mean splitting calculations by gene.

Portability

- Not all packages (or methods within a package) support all OS's.
- In the HPC world, that usually means "doesn't work on Windows".

RNG's in Parallel

- Be careful!
- Aided by **rlecuyer**, **rsprng**, and **doRNG** packages.

Summary

- Many kinds of parallelism available to R.
- Better/parallel BLAS is free speedup for linear algebra, but takes some work.

6 Shared Memory Parallelism in R

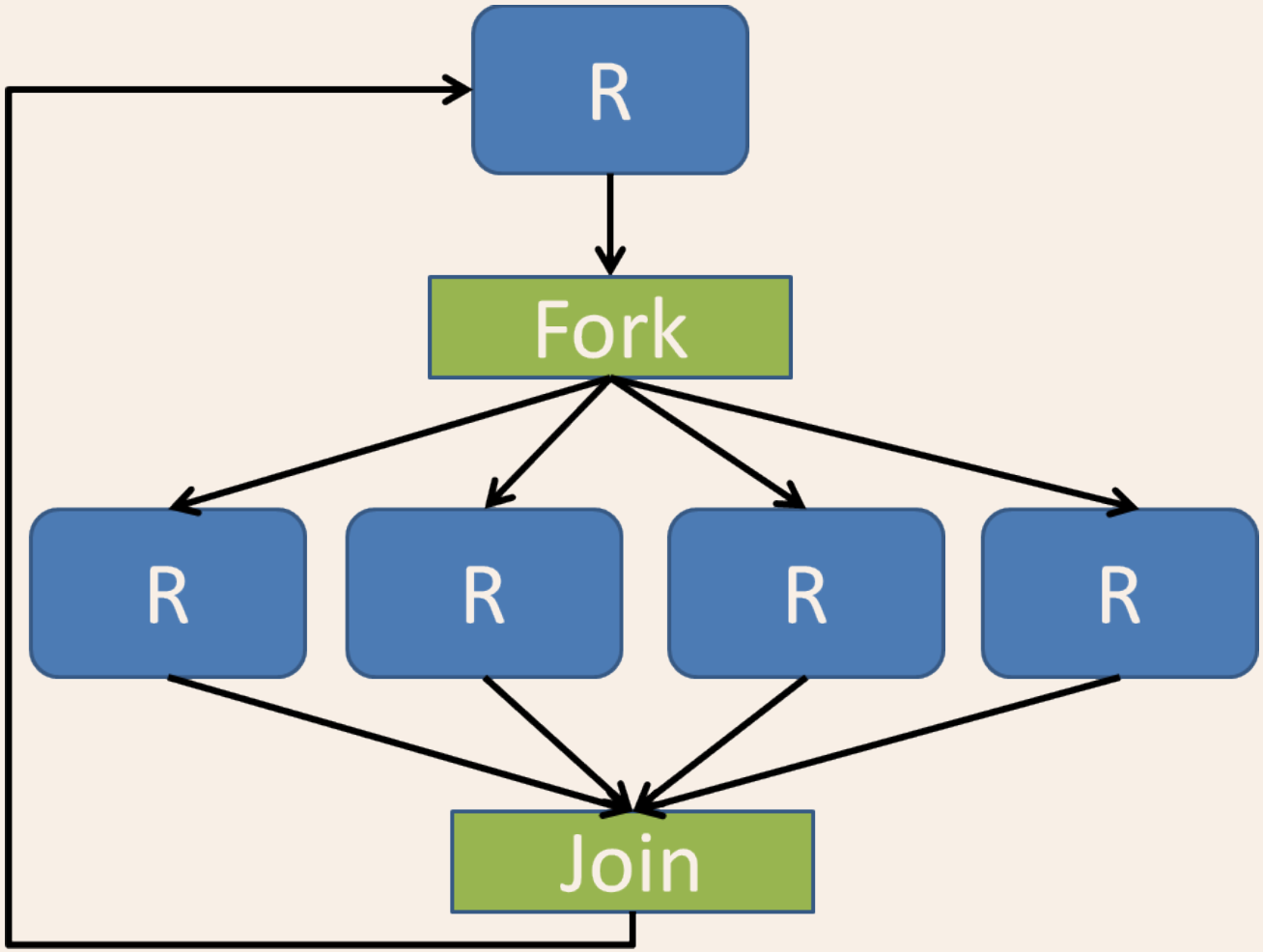
- The parallel Package
- The foreach Package

The parallel Package

- Comes with R $\geq$ 2.14.0
- Includes **multicore** + most of **snow**.
- As such, has 2 disjoint interfaces.

The parallel Package: multicore

Operates on fork/join paradigm.



The parallel Package: multicore

- (+) Data copied to child on write (handled by OS)
- (+) Very efficient.
- (-) No Windows support.
- (-) Not as efficient as threads.

The parallel Package: multicore

```
mclapply(X, FUN, ...,
  mc.preschedule=TRUE, mc.set.seed=TRUE,
  mc.silent=FALSE, mc.cores=getOption("mc.cores", 2L),
  mc.cleanup=TRUE, mc.allow.recursive=TRUE)
```

```
x <- lapply(1:10, sqrt)

library(parallel)
x.mc <- mclapply(1:10, sqrt)

all.equal(x.mc, x)
# [1] TRUE
```

The parallel Package: multicore

```
1 simplify2array(mclapply(1:10, function(i) Sys.getpid(),
      mc.cores=4))
2 # [1] 27452 27453 27454 27455 27452 27453 27454 27455
      27452 27453
3
4 simplify2array(mclapply(1:2, function(i) Sys.getpid(),
      mc.cores=4))
5 # [1] 27457 2745
```

The parallel Package: snow

- Uses sockets.
- (+) Works on all platforms.
- (-) More fiddley than `mclapply()`.
- (-) Not as efficient as forks.

The parallel Package: multicore

```
1 ### Set up the worker processes
2 my.cl <- makeCluster(detectCores())
3 my.cl
4 # socket cluster with 4 nodes on host   localhost
5
6 parSapply(cl, 1:5, sqrt)
7
8 stopCluster(my.cl)
```

The parallel Package: Summary

All

- `detectCores()`
- `splitIndices()`

multicore

- `mclapply()`
- `mcmapply()`
- `mcparallel()`
- `mccollect()`
- and others...

snow

- `makeCluster()`
- `stopCluster()`
- `parLapply()`
- `parSapply()`
- and others...

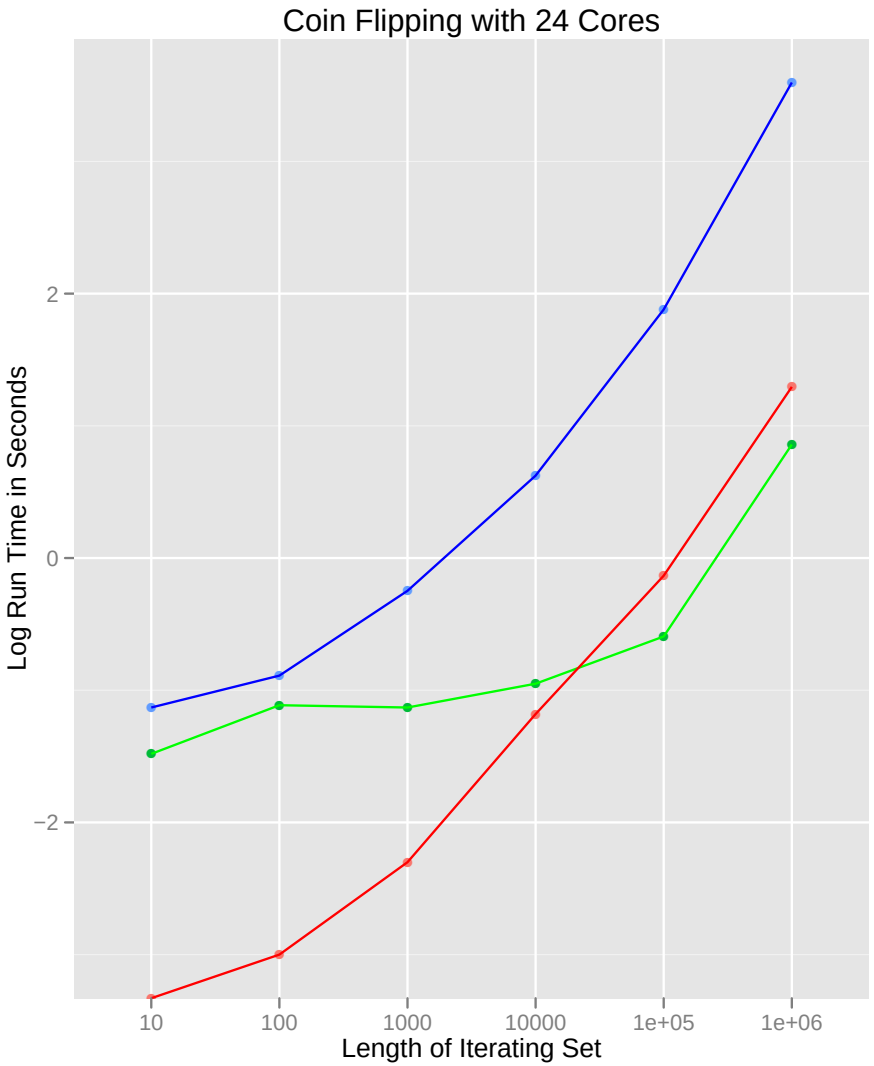
The foreach Package

- On Cran (Revolution Analytics).
- Main package is **foreach**, which is a single interface for a number of "backend" packages.
- Backends: **doMC**, **doMPI**, **doParallel**, **doRedis**, **doRNG**, **doSNOW**.

The foreach Package

- (+) Works on all platforms (with correct backend).
- (+) Can even work serial with minor notational change.
- (+) Write the code once, use whichever backend you prefer.
- (-) Really bizarre, non-R-ish synatx.
- (-) Efficiency issues ???

Efficiency Issues ???




```
1  ### Bad performance
2  foreach(i=1:len)
       %dopar% tinyfun(i)
3
4  ### Expected
       performance
5  foreach(i=1:ncores)
       %dopar% {
6    out <-
         numeric(len/ncores)
7    for (j in
         1:(len/ncores))
8      out[i] <-
           tinyfun(j)
9    out
10 }
```

The foreach Package: General Procedure

- Load **foreach** and your backend package.
- Register your backend.
- Call `foreach`

Using foreach: serial

```
1  library(foreach)
2
3  ### Example 1
4  foreach(i=1:3) %do% sqrt(i)
5
6  ### Example 2
7  n <- 50
8  reps <- 100
9
10 x <- foreach(i=1:reps) %do% {
11   sum(rnorm(n, mean=i)) / (n*reps)
12 }
```

Using foreach: Parallel

```
1  library(foreach)
2  library(<mybackend>)
3
4  register<MyBackend>()
5
6  ### Example 1
7  foreach(i=1:3) %dopar% sqrt(i)
8
9  ### Example 2
10 n <- 50
11 reps <- 100
12
13 x <- foreach(i=1:reps) %dopar% {
14   sum(rnorm(n, mean=i)) / (n*reps)
15 }
```

foreach backends

multicore

```
1 library(doParallel)
2 registerDoParallel(cores=ncores)
3 foreach(i=1:2) %dopar% Sys.getpid()
```

snow

```
1 library(doParallel)
2 cl <- makeCluster(ncores)
3 registerDoParallel(cl=cl)
4
5 foreach(i=1:2) %dopar% Sys.getpid()
6 stopCluster(cl)
```

foreach Summary

- Make sure to register your backend.
- Different backends may have different performance.
- Use `%dopar%` for parallel foreach.
- `%do%` and `%dopar%` *must* appear on the same line as the `foreach()` call.

7 Distributed Memory Parallelism with R

- Distributed Memory Parallelism
- Rmpi
- pbdMPI vs Rmpi
- Summary

Why Distribute?

- Nodes only hold so much ram.
- Commodity hardware: $\approx 32 - 64$ gib.
- With a few exceptions (**ff**, **bigmemory**), R does computations in memory.
- If your problem doesn't fit in the memory of one node...

Packages for Distributed Memory Parallelism in R

- **Rmpi**, and **snow** via **Rmpi**
- **RHIPE** and **RHadoop** ecosystem
- **pbdR** ecosystem (Monday)

Hasty Explanation of MPI

- We will return to this on Monday.
- MPI = Message Passing Interface
- Recall: Distributed machines can't directly manipulate memory of other nodes.
- Can *indirectly* manipulate them, however...
- Distinct nodes collaborate by passing messages over network.

Rmpi Hello World

```
mpi.spawn.Rslaves(nslaves=2)
#         2 slaves are spawned successfully. 0 failed.
# master (rank 0, comm 1) of size 3 is running on:
    wootabega
# slave1 (rank 1, comm 1) of size 3 is running on:
    wootabega
# slave2 (rank 2, comm 1) of size 3 is running on:
    wootabega

mpi.remote.exec(paste("I
    am",mpi.comm.rank(),"of",mpi.comm.size()))
# $slave1
# [1] "I am 1 of 3"
#
# $slave2
# [1] "I am 2 of 3"

mpi.exit()
```

Using Rmpi from snow

```
library(snow)
library(Rmpi)

cl <- makeCluster(2, type = "MPI")
clusterCall(cl, function() Sys.getpid())
clusterCall(cl, runif, 2)
stopCluster(cl)
mpi.quit()
```

Rmpi Resources

- **Rmpi** tutorial: http://math.acadiau.ca/ACMMaC/Rmpi/
- **Rmpi** manual: http://cran.r-project.org/web/packages/Rmpi/Rmpi.pdf

pbdMPI vs Rmpi

- **Rmpi** is interactive; **pbdMPI** is exclusively batch.
- **pbdMPI** is easier to install.
- **pbdMPI** has a simpler interface.
- **pbdMPI** integrates with other pbdR packages.

Example Syntax

Rmpi

```
1 # int
2 mpi.allreduce(x, type=1)
3 # double
4 mpi.allreduce(x, type=2)
```

pbdMPI

```
1 allreduce(x)
```

Types in R

```
1 > is.integer(1)
2 [1] FALSE
3 > is.integer(2)
4 [1] FALSE
5 > is.integer(1:2)
6 [1] TRUE
```

Summary

- Distributed parallelism is necessary when computations no longer fit in ram.
- Several options available; most go beyond the scope of this talk.
- More on pbdR Monday!

Exercises 1

1. Suppose we wish to store the square root of all integers from 1 to 10000 in a vector. Do this in each of the following ways, and compare them with **rbenchmark**:
   - for loop without initialization
   - for loop with initialization
   - Ply function
   - vectorization
2. Revisit the previous example, evaluating the different implementations with `Rprof()`.
3. Count the number of integer multiples of 5 or 17 which are less than 10,000,000.
   - Solve this with `lapply()`.
   - Solve this with vectorization.
   - Solve this with `mclapply()` and 2 cores.

Exercises 2

4. The Monte Hall game is a well known "paradox" from elementary probability. From Wikipedia:

   > *Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?*

   Simulate one million trials of the Monte Hall game on 2 cores, switching doors every time, to computationally verify the elementary probability result. Compare the run time against the 1 core run time.

Exercises 3

5. The following is a more substantive example that utilizes multiple cores to perform a real analysis task. Run and evaluate this example. The example is modified from an example of Wei-Chen's from the **pbdDEMO** package. There are 148 EIAV sequences in the data set. They are sequencing from multiple blood serum collected longitudinally from one EIA horse over periodical fever cycles after EIAV infection. The virus population evolved within the sick horse over time. Some subtype can break the horse's immune system. It was to identify how many subtypes were evolving within the horse, which type is associated with disease early onset, where/which sample/time point to isolate that subtype virus. Moreover, which mutated region of sequence was critical for that subtype in order to break the immune system.

Exercises 4

```
1  library(phyclust, quietly = TRUE)
2  library(parallel)
3
4  ### Load data
5  data.path <- paste(.libPaths()[1],
       "/phyclust/data/pony524.phy", sep = "")
6  pony.524 <- read.phylip(data.path)
7  X <- pony.524$org
8  K0 <- 1
9  Ka <- 2
10
11 ### Find MLEs
12 ret.K0 <- find.best(X, K0)
13 ret.Ka <- find.best(X, Ka)
14 LRT <- -2 * (ret.Ka$logL - ret.K0$logL)
15
16 ### The user defined function
17 FUN <- function(jid){
```

Exercises 5

```
18    X.b <- bootstrap.seq.data(ret.K0)$org
19
20    ret.K0 <- phyclust(X.b, K0)
21    repeat{
22      ret.Ka <- phyclust(X.b, Ka)
23      if(ret.Ka$logL > ret.K0$logL){
24        break
25      }
26    }
27
28    LRT.b <- -2 * (ret.Ka$logL - ret.K0$logL)
29    LRT.b
30  }
31
32  ### Task pull and summary
33  ret <- mclapply(1:100, FUN)
34  LRT.B <- unlist(ret)
35  cat("K0: ", K0, "\n",
36      "Ka: ", Ka, "\n",
```

Exercises 6

```
37      "logL K0: ", ret.K0$logL, "\n",
38      "logL Ka: ", ret.Ka$logL, "\n",
39      "LRT: ", LRT, "\n",
40      "p-value: ", mean(LRT > LRT.B), "\n", sep = "")
```

9 Wrapup

Important Topics Not Discussed Here

- Distributed computing (for real) — pbdR on Monday.
- Utilizing compiled code — Rcpp on Tuesday.
- Multithreading.
- GPU's and MIC's.
- R+Hadoop.

Thanks for coming!

Questions?