High Performance Computing with R

Drew Schmidt

June 10, 2015

Support

Division of Mathematical Sciences, National Science Foundation, Award No. 1418195, 2014-2017.

Tutorial Structure

- (30 Minutes) XSEDE, Introduction, etc.

- (45 Minutes) Interfacing to Compiled Code . . .
- (30 Minutes) Exercises + Break

- (45 Minutes) Parallelism
- (30 Minutes) Exercises, questions, discussions, demos, etc.

Tutorial Goals

We hope to introduce you to:

1. Why and how to interface R to C++.
2. The state of parallelism in R.
3. Basic distributed computing with R.

Exercises

Each section has a complement of exercises to give hands-on reinforcement of ideas introduced in the lecture.

1. More exercises are given than you have time to complete.
2. Later exercises are more difficult than earlier ones.
3. Some exercises require use of things not explicitly shown in lecture; look through the documentation mentioned in the slides to find the information you need.

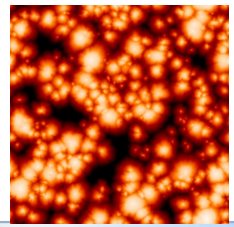
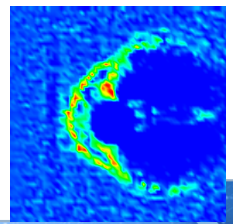
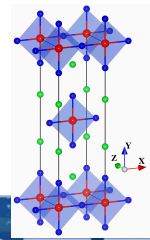
Part I

Intro

XSEDE

Extreme Science and Engineering Discovery Environment

- Extreme Science and Engineering Discovery Environment
- Follow on NSF project to TeraGrid in 2012
- Centers operate machines, and XSEDE provides seamless infrastructure for allocations, access, and training
- Researchers propose resource use through XRAS
- Supports thousands of scientists in fields such as:
  - Chemistry
  - Bioinformatics
  - Materials Science
  - Data Sciences

XSEDE

XSEDE Allocations

- Want to use XSEDE resources to teach a class?
  - https://portal.xsede.org/allocations-overview#types-education
- Just looking to try out a larger resource or a special resource your campus doesn't have?
  - https://portal.xsede.org/allocations-overview#types-startup

XSEDE Allocations

- See a Campus Champion
  - https://www.xsede.org/current-champions
- Ready to scale up your research?
  - https://portal.xsede.org/allocations-overview#types-research

More "helpful" resources

xsede.org→User Services

- Resources available at each Service Provider
  - User Guides describing memory, number of CPUs, file systems, etc.
  - Storage facilities
  - Software (Comprehensive Search)
- Training: portal.xsede.org → Training
  - Course Calendar
  - On-line training
  - Certifications
- Get face-to-face help from XSEDE experts at your institution; contact your local Campus Champions.
- Extended Collaborative Support (formerly known as Advanced User Support (AUSS))

Why XSEDE?

- Free!!!!!!!!
- Excellent support.
- ECSS
- Many machines pre-configured with R (and RStudio).
- Good compilers.
- Fast math libraries.

GPU's

Using with R:

- R Packages: gputools, HiPLARM, Rth, ...
- Rolling your own: CUDA, thrust, OpenACC, OpenMP*

Intel Xeon Phi

Using with R:

- Packages: HiPLAR, anything using BLAS
- Rolling your own: LAPACK, OpenMP, OpenACC

Resources:

- Fast parallel computing with Intel Phi coprocessors
- *Performance Evaluation of R with Intel Xeon Phi Coprocessor*, El-Khamra, Gaffney, et al.

2 Advanced Computing Resources

- Accelerator Cards
- Hadoop

Hadoop/Spark

- RHIPE
- RHadoop
- SparkR

Resources for Learning R

- *The Art of R Programming* by Norm Matloff: http://nostarch.com/artofr.htm
- *An Introduction to R* by Venables, Smith, and the R Core Team: http://cran.r-project.org/doc/manuals/R-intro.pdf
- *The R Inferno* by Patrick Burns: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Mathesaurus: http://mathesaurus.sourceforge.net/
- R programming for those coming from other languages: http://www.johndcook.com/R_language_for_programmers.html
- *aRrgh: a newcomer's (angry) guide to R*, by Tim Smith and Kevin Ushey: http://tim-smith.us/arrgh/

Other Invaluable Resources

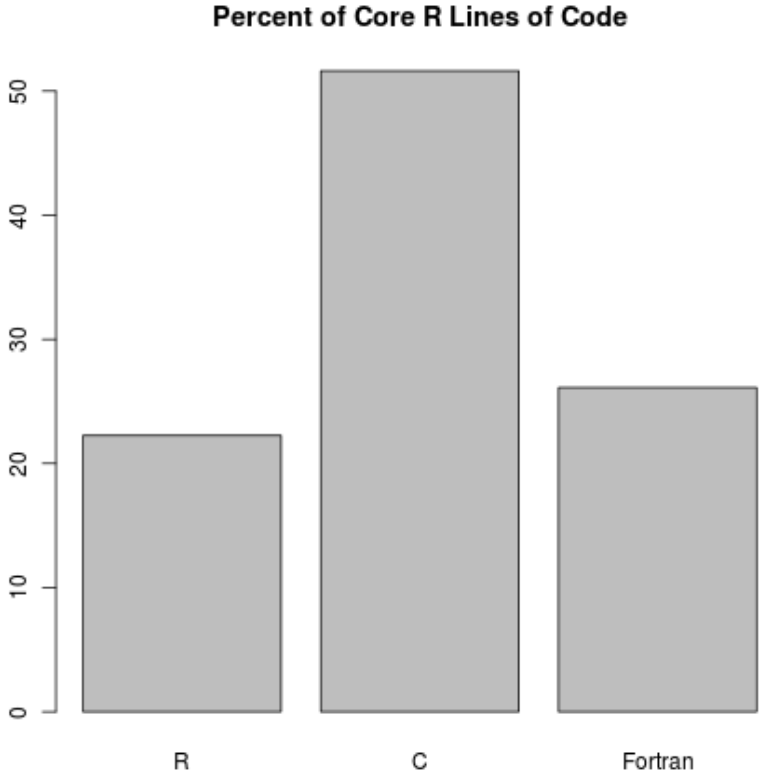
- *R Installation and Administration*: http://cran.r-project.org/doc/manuals/R-admin.html
- *Task Views*: http://cran.at.r-project.org/web/views
- *Writing R Extensions*: http://cran.r-project.org/doc/manuals/R-exts.html
- Mailing list archives: http://tolstoy.newcastle.edu.au/R/
- The `[R]` stackoverflow tag.
- The `#rstats` hastag on Twitter.

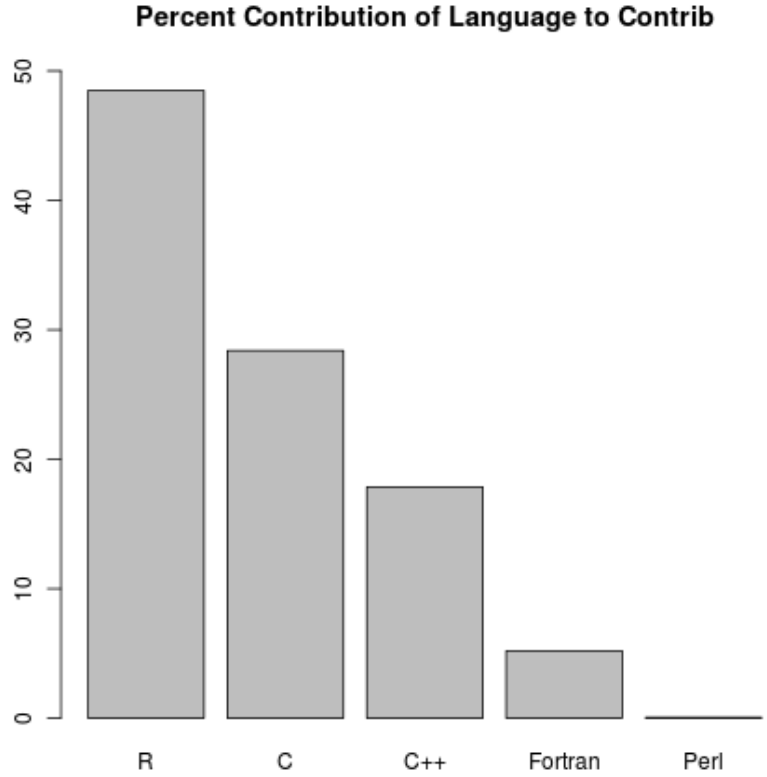
Part II

Interfacing to Compiled Code

What Language is R Written In?

- R is mostly a C program
- R extensions are mostly R programs





Foreign Language Interfaces

- C/C++: `.Call()`, `.C()` (deprecated)
- Fortran: `.Call()`, `.Fortran()` (deprecated)
- Java: rJava package
- Python: rPython package
- ...

For the remainder, we will focus on C++ via Rcpp.

4 Introduction to Rcpp

- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help

What Rcpp **is**

- R interface to compiled code.
- Package ecosystem (Rcpp, RcppArmadillo, RcppEigen, . . .).
- Utilities to make writing C++ more convenient for R users.
- **A tool which requires C++ knowledge to effectively utilize.**
- GPL licensed (like R).

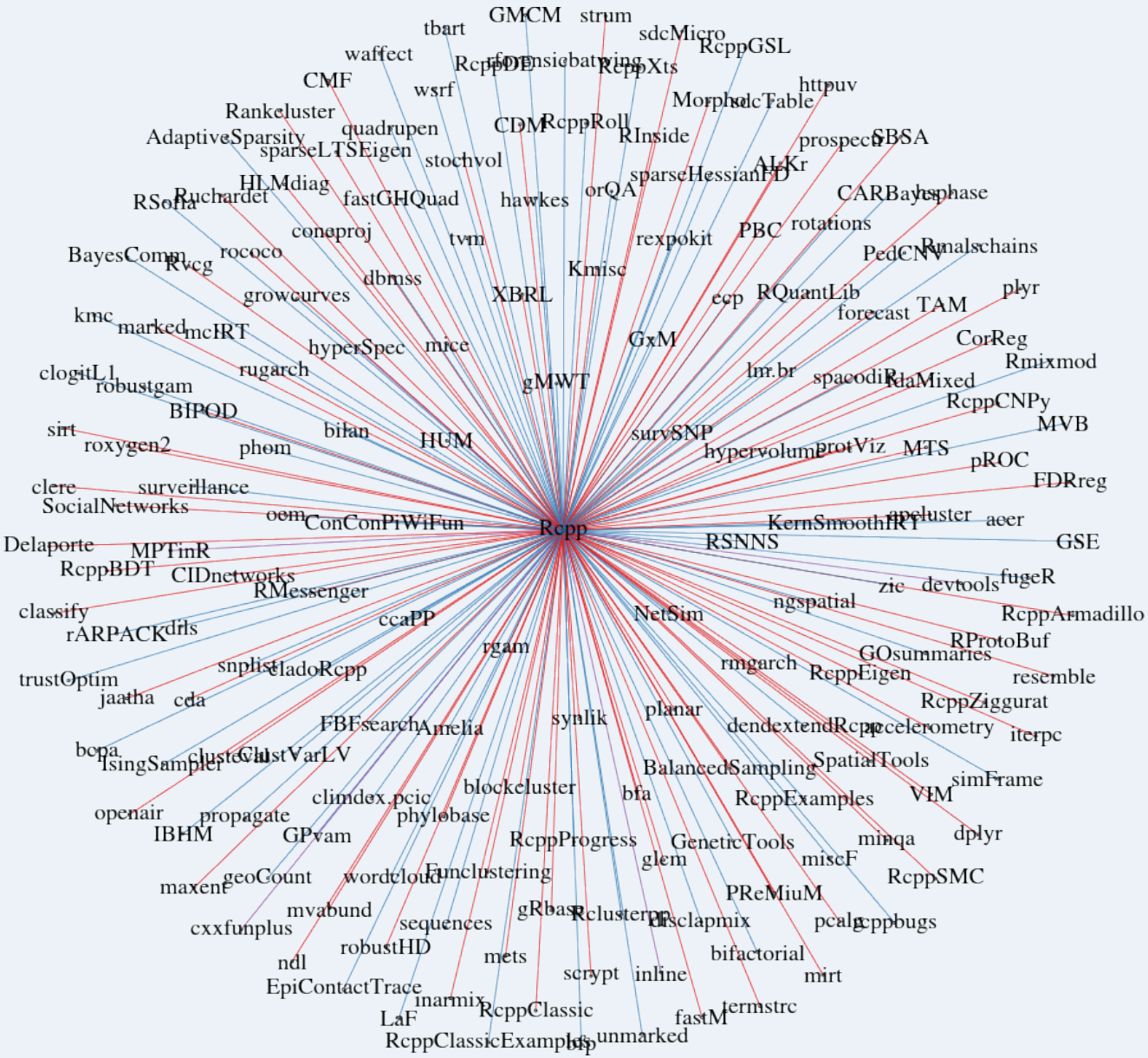
What Rcpp **is not**

- Magic.
- Automatic R-to-C++ converter.
- A way around having to learn C++.
- A tool to make existing R functionality faster (unless you rewrite it!).
- As easy to use as R.

Advantages of Rcpp

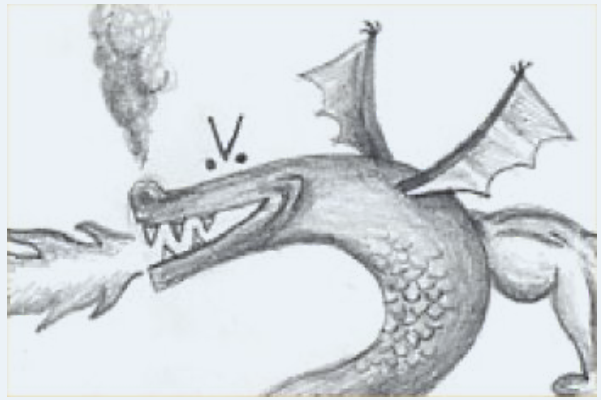
- Compiled code is *fast*.
- Easy to install.
- Easy to use (comparatively).
- Better documented than alternatives.
- Large, friendly, helpful community.

Rcpp Package Dependencies



Disadvantages

- It's C++ (there be dragons).
- Difficult to debug/profile.
- Rcpp designed to only work with R.



4 Introduction to Rcpp

- Foreign Language Interfaces
- What is Rcpp?
- Documentation and Help

Documentation

- The numerous Rcpp vignettes
  http://cran.r-project.org/web/packages/Rcpp/index.html
  (start with Introduction, quickref, and FAQ).
- *High Performance Functions with Rcpp*, Hadley Wickham:
  http://adv-r.had.co.nz/Rcpp.html
- *Seamless R and C++ Integration with Rcpp* (book), http://www.amazon.com/Seamless-Integration-Rcpp-Dirk-Eddelbuettel/dp/1461468671/ref=sr_1_1?ie=UTF8

Where to Get Help

- The documentation.
- The `[rcpp]` tag on stackoverflow.
- Rcpp-devel list: http://lists.r-forge.r-project.org/mailman/listinfo/rcpp-devel

Advice

New to C++?

- Get a good book on just C++.
- Be patient. C++ is really hard.
- Learn the art of reading template explosions.

Know R?

- Never use . in object names.
- Lines end with ;.
- Returns of functions must be explicitly named.

Know C++?

- No voids.
- If data is modified, do it in a copy.
- R functions are not thread safe!!!

C/C++ API's and Extensions for R

- The native C interface.
- Rcpp
  - RcppArmadillo
  - RcppCNPy
  - RcppEigen
  - RcppGSL
  - RcppRedis
  - ...
- Rcpp11, Rcpp14, ...

C vs Rcpp

To see the difference, let's construct:

```
1 list(a=1L, b=2.0)
```

using the native C interface and with Rcpp.

The C Interface

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

SEXP examplefun(){
  SEXP ret, retnames, a, b;
  PROTECT(a = allocVector(INTSXP, 1));
  PROTECT(b = allocVector(REALSXP, 1));

  INTEGER(a)[0] = 1;
  REAL(b)[0] = 2.0;

  PROTECT(ret = allocVector(VECSXP, 2));
  SET_VECTOR_ELT(ret, 0, a);
  SET_VECTOR_ELT(ret, 1, b);

  PROTECT(retnames = allocVector(STRSXP, 2));
  SET_STRING_ELT(retnames, 0, mkChar("a"));
  SET_STRING_ELT(retnames, 1, mkChar("b"));
  setAttrib(ret, R_NamesSymbol, retnames);

  UNPROTECT(4);
  return ret;
}
```

Rcpp

```
1  #include <Rcpp.h>
2
3  // [[Rcpp::export]]
4  Rcpp::List examplefun()
5  {
6    Rcpp::IntegerVector a(1);
7    Rcpp::NumericVector b(1);
8
9    a[0] = 1;
10   b[0] = 2.0;
11
12   Rcpp::List ret =
13     Rcpp::List::create(Rcpp::Named("a") = a,
14                        Rcpp::Named("b") = b);
15
16   return ret;
17 }
```

C vs Rcpp

- I can't in good conscience describe C++ as *good for beginners*.
- Rcpp is cleaner.
- Like C++? You'll *love* Rcpp.

5 Using Rcpp

- C vs Rcpp
- Using Rcpp with R

Rcpp

What about compiling, linking, loading, wrapping, etc?

Building with Rcpp

We will be using `sourceCpp()` to build our examples:

1. Create C++ function as string in R.
2. Use sourceCpp to generate wrapper.
3. Call your function in R.

sourceCpp(): Create C++ Function

```
code <- '
#include <Rcpp.h>

// [[Rcpp::export]]
int plustwo(int n)
{
  return n+2;
}
'
```

sourceCpp(): Use sourceCpp

```
1 library(Rcpp)
2 sourceCpp(code=code)
```

sourceCpp(): Call Your Function in R

```
1 plustwo(1)
2 #  [1]  3
```

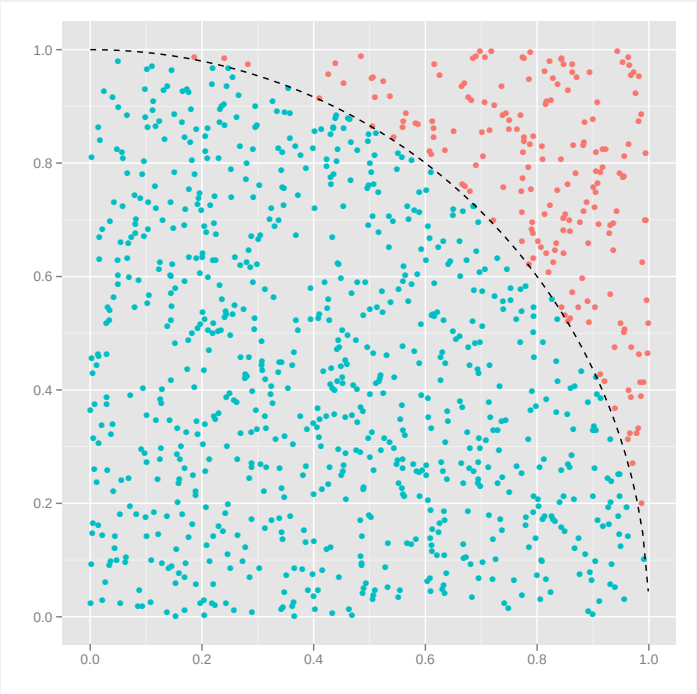
6 The Typical Monte Carlo Simulation for Estimating π

- Background and Outline
- Implementation
- Summary

Example 1 : Monte Carlo Simulation to Extimate π

Sample N uniform observations (x_i, y_i) in the unit square $[0, 1] \times [0, 1]$. Then

$$\pi \approx 4 \left(\frac{\#\ Inside\ Circle}{\#\ Total} \right) = 4 \left(\frac{\#\ \text{Blue}}{\#\ \text{Blue} + \#\ \text{Red}} \right)$$



Outline

1. Implement in R using loops.
2. Implement in R using vectorization.
3. Implement in C++ with Rcpp.
4. Benchmark.
5. Examine other performance considerations.

6 The Typical Monte Carlo Simulation for Estimating π

- Background and Outline
- Implementation
- Summary

Example 1: Monte Carlo Simulation Code

R Code (loops)

```
1  mcsim_r <- function(n)
2  {
3    r <- 0L
4
5    for (i in 1:n){
6      u <- runif(1)
7      v <- runif(1)
8
9      if (u^2 + v^2 <= 1)
10       r <- r + 1
11   }
12
13   return( 4*r/n )
14 }
```

Example 1: Monte Carlo Simulation Code

R Code (vectorized)

```
1 mcsim_r_vec <- function(n)
2 {
3   x <- matrix(runif(n * 2), ncol=2)
4   r <- sum(rowSums(x^2) <= 1)
5
6   return( 4*r/n )
7 }
```

Example 1: Monte Carlo Simulation Code

Rcpp Code

```
code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
double mcsim_rcpp(const int n)
{
  int i, r = 0;
  double u, v;

  for (i=0; i<n; i++){
    u = R::runif(0, 1);
    v = R::runif(0, 1);

    if (u*u + v*v <= 1)
      r++;
  }

  return (double) 4.*r/n;
}
"

library(Rcpp)
sourceCpp(code=code)
```

Example 1: Monte Carlo Simulation Code

Benchmarking the Methods

```
1 library(rbenchmark)
2
3 n <- 100000L
4
5 benchmark(R.loop = mcsim_r(n),
6           R.vec = mcsim_r_vec(n),
7           C = mcsim_c(n),
8           Rcpp = mcsim_rcpp(n),
9           columns=c("test", "replications", "elapsed",
              "relative"))
```

```
    test replications elapsed relative
3   Rcpp          100   0.309    1.000
1 R.loop          100  65.543  212.113
2  R.vec          100   1.989    6.437
```

Example 1: Monte Carlo Simulation Code

Benchmarking the Methods

```
1 library(rbenchmark)
2
3 n <- 10000000L
4
5 benchmark(R.vec = mcsim_r_vec(n),
6           Rcpp = mcsim_rcpp(n),
7           columns=c("test", "replications", "elapsed",
              "relative"))
```

```
   test replications elapsed relative
2  Rcpp          100  30.825    1.000
1 R.vec          100 135.075    4.382
```

What About the Compiler?

Benchmarking the Methods

```
1  library(rbenchmark)
2  library(compiler)
3
4  mcsim_r <- cmpfun(mcsim_r)
5  mcsim_r_vec <- cmpfun(mcsim_r_vec)
6  mcsim_rcpp <- cmpfun(mcsim_rcpp)
7
8  n <- 100000L
9
10 benchmark(R.loop = mcsim_r(n),
11           R.vec = mcsim_r_vec(n),
12           Rcpp = mcsim_rcpp(n),
13           columns=c("test", "replications", "elapsed",
                 "relative"))
```

```
     test replications elapsed relative
3    Rcpp          100   0.311    1.000
1 R.loop          100  55.125  177.251
2  R.vec          100   1.107    3.559
```

Memory Usage in Bytes (roughly)

Loops:

$$\underbrace{4(n+3)}_{\text{Integers}} + \underbrace{8 \cdot 3}_{\text{Doubles}}$$

Vectorized:

$$\underbrace{4n}_{\text{Integers}} + \underbrace{8(2+2n)}_{\text{Doubles}}$$

Rcpp

$$\underbrace{4 \cdot 3}_{\text{Integers}} + \underbrace{8 \cdot 3}_{\text{Doubles}}$$

Summary

For $n = 100,000$ iterations and 100 replicates:

	Loops	Vectorized	Rcpp
Avg Runtime (seconds)	0.65543	0.01999	0.00309
Avg Compiled Runtime (seconds)	0.55125	0.1107	0.00311
Memory Usage	1.526 MiB	13.733 MiB	36 bytes

Processor:	Core i5 Sandy Bridge
R Version:	3.1.2
C++ Compiler:	clang++ 3.5.0
CXX Flags:	`-O3 -fpic`

Some Thoughts

- Bad R often looks like good C/C++.
- The bytecode compiler helps, but not much.
- R's memory footprint is terrible.

7 Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary

Cosine Similarity

Recall from vector calculus that for vectors x and y

$$cos(x, y) = \|x\| \, \|y\| \, \cos(\theta(x, y))$$

We define

$$\text{cosim}(x, y) := \cos(\theta(x, y)) = \frac{x \cdot y}{\|x\| \, \|y\|}$$

Cosine Similarity Matrix

The cosine similarity matrix of a given (possibly non-square) matrix is the matrix of all pairwise similarities of the columns, i.e., given

$$X_{n,p} = [x_1, \ldots, x_p]$$

We take

$$\text{cosim}(X)_{ij} = \text{cosim}(x_i, x_j)$$

7 Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary

Original implementation

From CRAN's lsa package version 0.73 (in R/lsa.R)

```
1  cosine <- function (x, y = NULL){
2      if (is.matrix(x) && is.null(y)) {
3          co = array(0, c(ncol(x), ncol(x)))
4          f = colnames(x)
5          dimnames(co) = list(f, f)
6          for (i in 2:ncol(x)) {
7              for (j in 1:(i - 1)) {
8                  co[i, j] = cosine(x[, i], x[, j])
9              }
10         }
11         co = co + t(co)
12         diag(co) = 1
13         return(as.matrix(co))
14     }
15     else if (is.vector(x) && is.vector(y))
16         return(crossprod(x, y)/sqrt(crossprod(x) * crossprod(y)))
17     else
18         stop("argument mismatch.")
19 }
```

R Improvements 1

```
1  cosine_loop <- function(x){
2    cp <- crossprod(x)
3    dg <- diag(cp)
4
5    co <- matrix(0.0, length(dg), length(dg))
6
7    for (j in 2L:length(dg)){
8      for (i in 1L:(j-1L)){
9        co[i, j] <- cp[i, j] / sqrt(dg[i] * dg[j])
10     }
11   }
12
13   co <- co + t(co)
14   diag(co) <- 1.0
15
16   return( co )
17 }
```

Rcpp 1

```
1  library(Rcpp)
2
3  code <- "
4  #include <Rcpp.h>
5
6  // [[Rcpp::export]]
7  Rcpp::NumericMatrix fill_loop(Rcpp::NumericMatrix cp,
       Rcpp::NumericVector dg){
8    const unsigned int n = cp.nrow();
9    Rcpp::NumericMatrix co(n, n);
10
11   // Fill lower triangle and diagonal
12   for (int j=0; j<n; j++){
13     for (int i=0; i<=j; i++){
14       if (i == j)
15         co(j, j) = 1.0;
16       else
17         co(i, j) = cp(i, j) / std::sqrt(dg[i] * dg[j]);
18     }
19   }
20
```

Rcpp 2

```
21    // Copy lower triangle to upper
22    for (int j=0; j<n; j++){
23      for (int i=j+1; i<n; i++)
24        co(i, j) = co(j, i);
25    }
26
27    return co;
28  }
29  "
30  sourceCpp(code=code)
31
32
33  cosine_Rcpp <- function(x){
34    cp <- crossprod(x)
35    dg <- diag(cp)
36
37    co <- fill_loop(cp, dg)
38
39    return( co )
40  }
```

Rcpp 1

```
1  library(rbenchmark)
2
3  reps <- 10
4
5  for (i in 1:10){
6    n <- i*100
7    x <- matrix(rnorm(n*n), n, n)
8
9    benchmark(cosine(x), cosine_loop(x), cosine_Rcpp(x),
         replications=reps, columns=c("test",
10 "relative"))
11 }
```

Relative Performance

Matrix Dimension	cosine()	cosine_loop()	cosine_Rcpp()
100x100	340	44.5	1
200x200	535.167	57	1
300x300	441.632	42.895	1
400x400	495.176	42.412	1
500x500	519.877	41.456	1
600x600	512.264	36.758	1
700x700	392.114	25.486	1
800x800	474.341	28.498	1
900x900	523.841	29.367	1
1000x1000	459.322	23.995	1

Relative Performance with Bytecode Compilation

Matrix Dimension	cosine()	cosine_loop()	`cosine_Rcpp()`
100x100	300	25.5	1
200x200	360.25	25.125	1
300x300	454.059	29.941	1
400x400	252.885	14.705	1
500x500	315.518	17.671	1
600x600	323.662	15.398	1
700x700	430.507	18.169	1
800x800	385.504	15.043	1
900x900	469.728	16.709	1
1000x1000	505.706	16.625	1

7 Computing the Cosine Similarity Matrix

- Background and Outline
- Implementation
- Benchmarks
- Summary

Summary

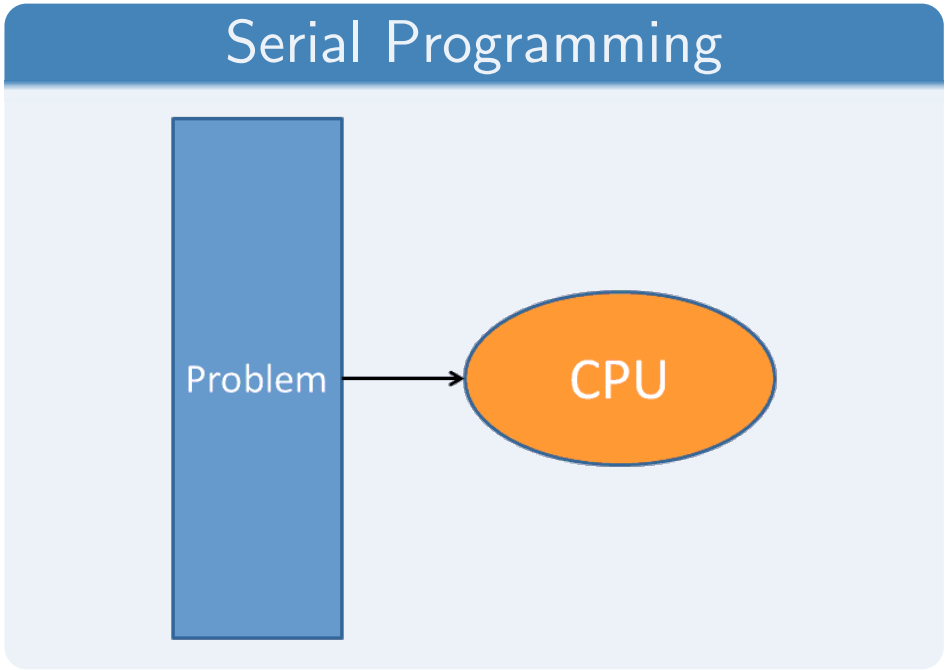
- Bad R often looks like good C/C++.
- Compiled code can be much faster than R code.
- Vectorized code better than loops, but worse than more tailored compiled code.

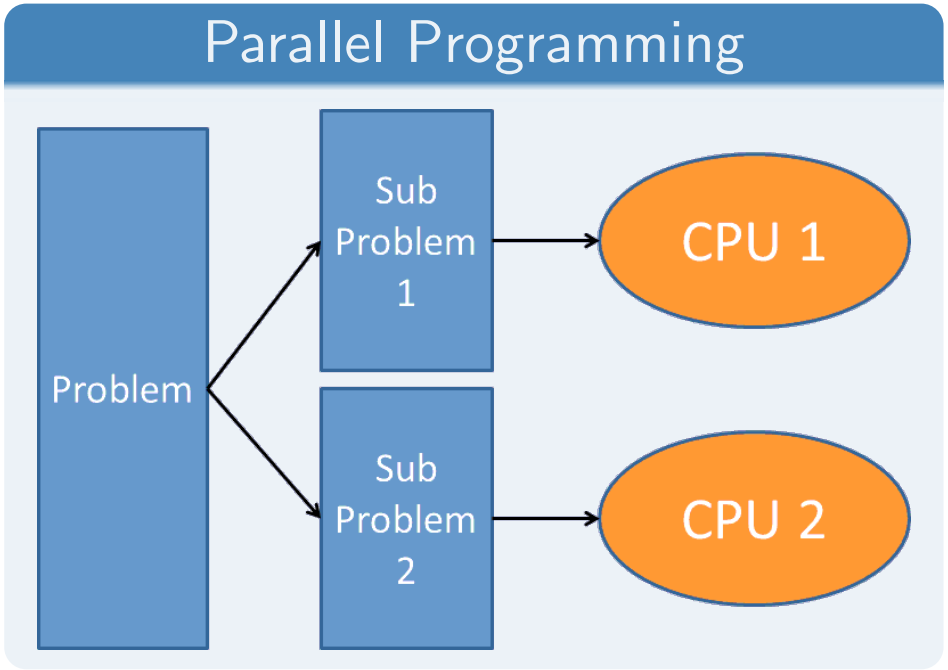
Exercises

Part III

Parallelism

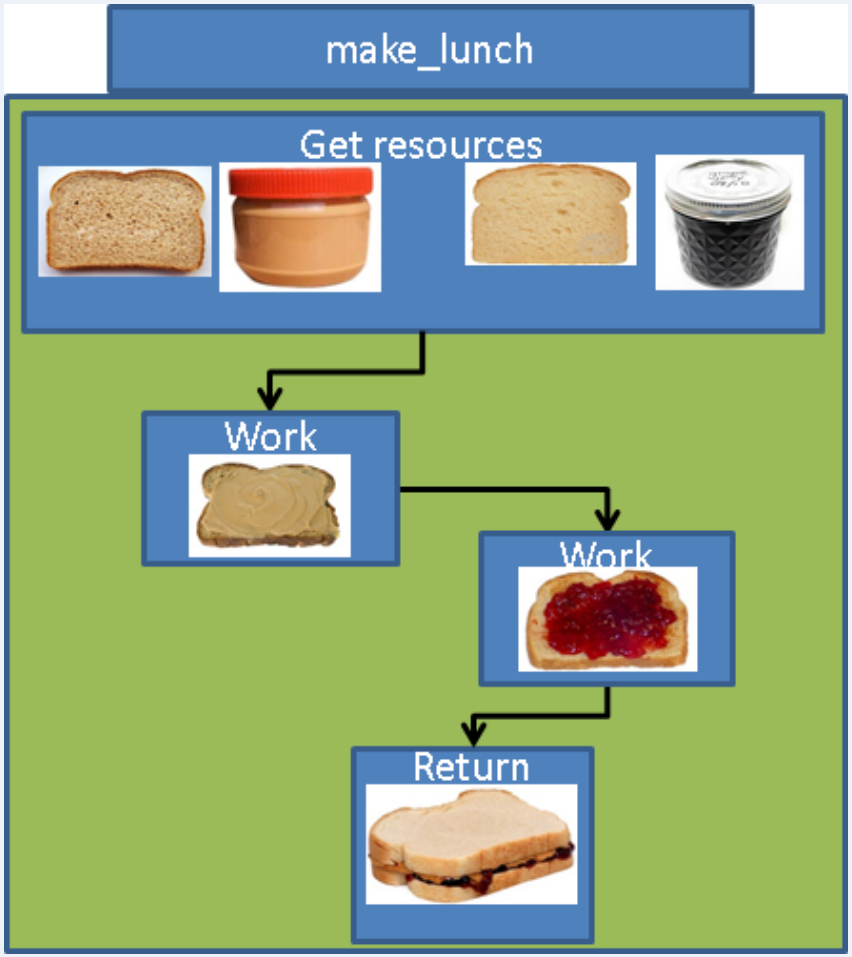
Parallelism



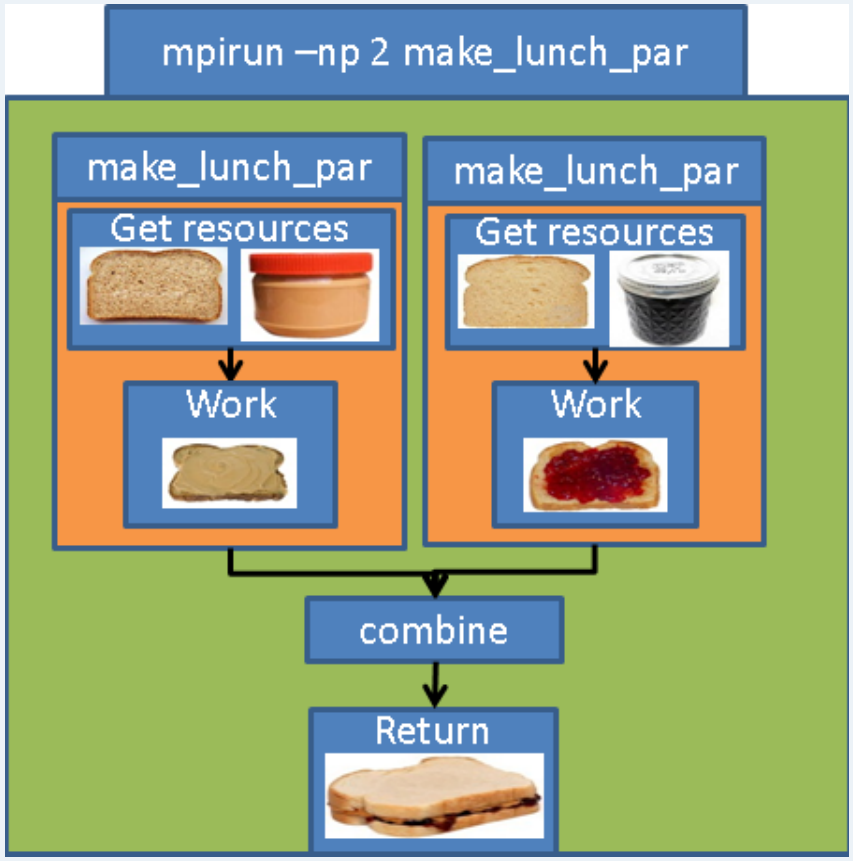


Parallelism

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

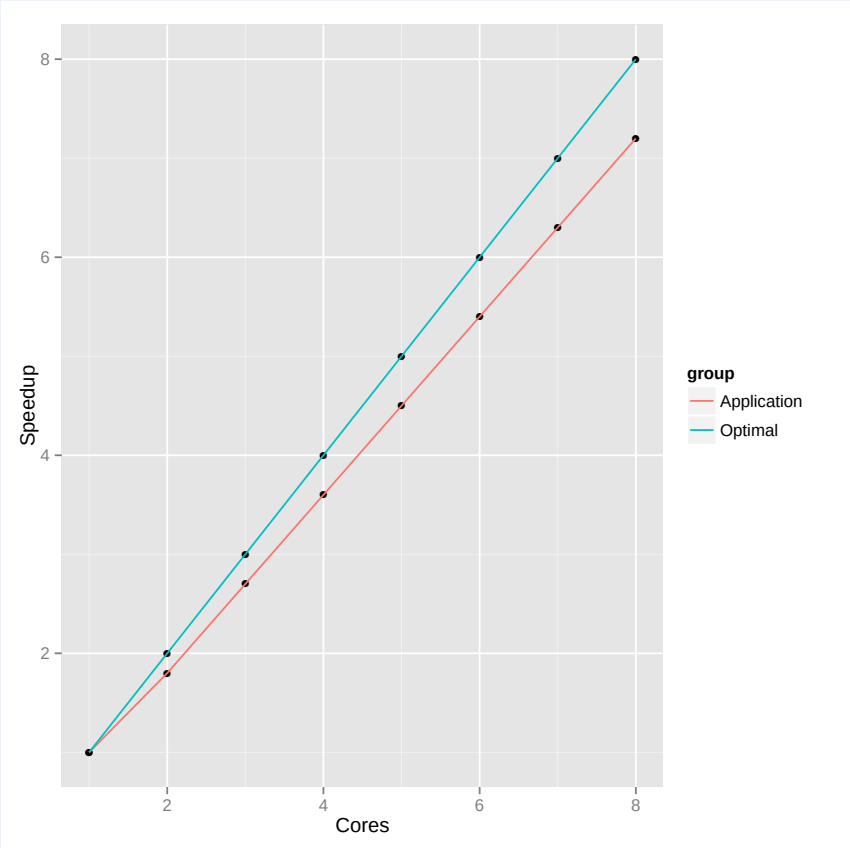
- *Wallclock Time*: Time of the clock on the wall from start to finish
- *Speedup*: unitless measure of improvement; more is better.

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

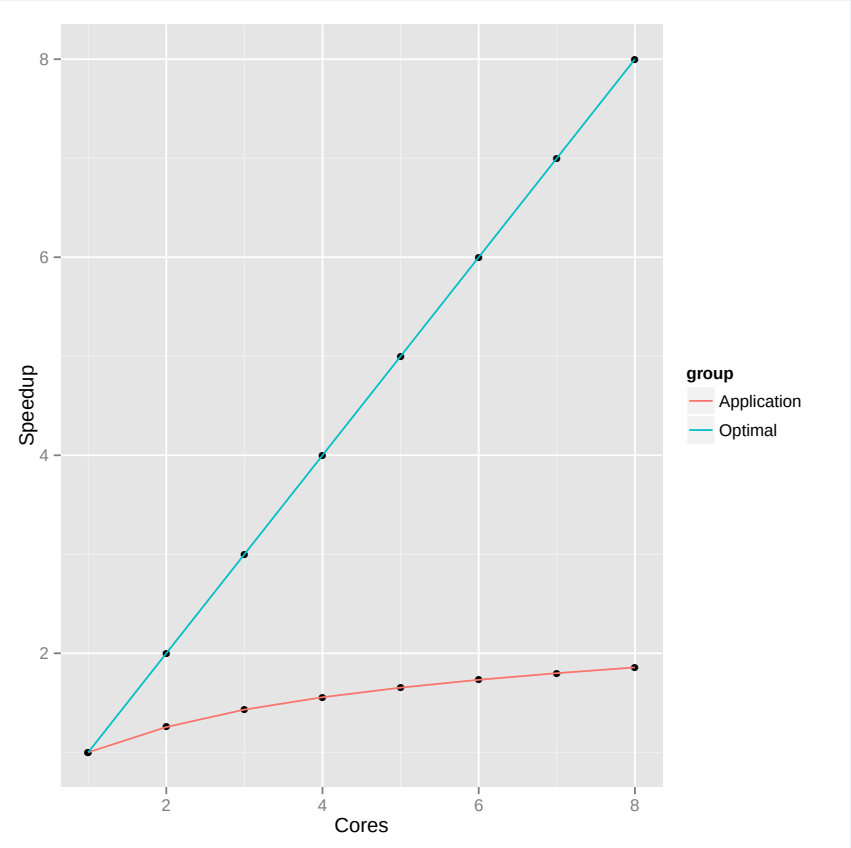
  - n_1 is often taken to be 1
  - In this case, comparing parallel algorithm to serial algorithm

Speedup

Good Speedup



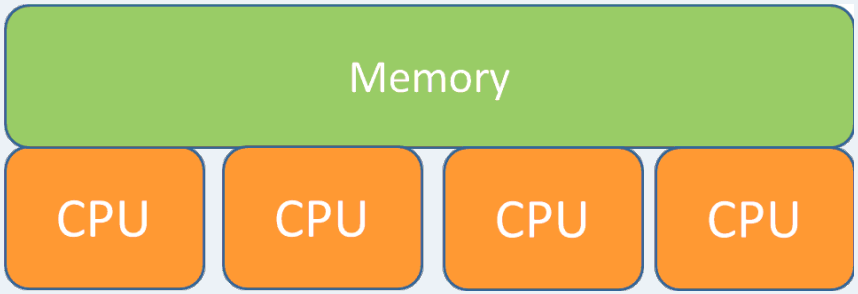
Bad Speedup



Shared and Distributed Memory Machines

Shared Memory

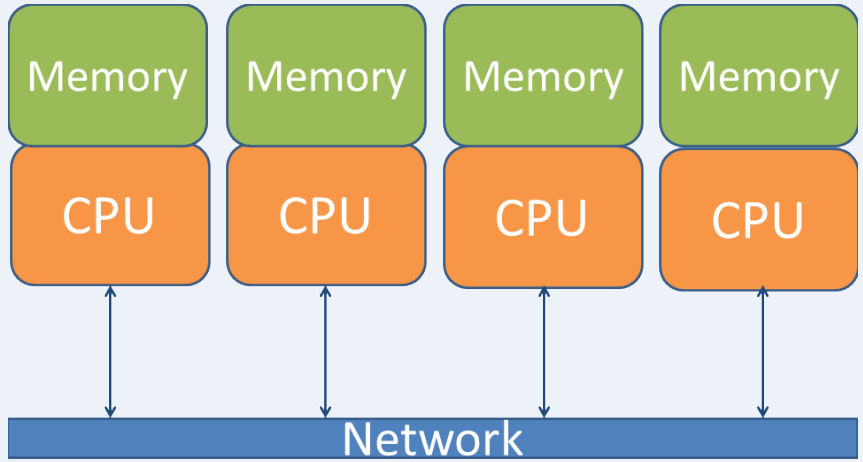
Direct access to read/change memory (one node)



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

Titan, Oak Ridge National Lab
299,008 cores
584 TB RAM

Parallel Programming Packages for R

Shared Memory

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

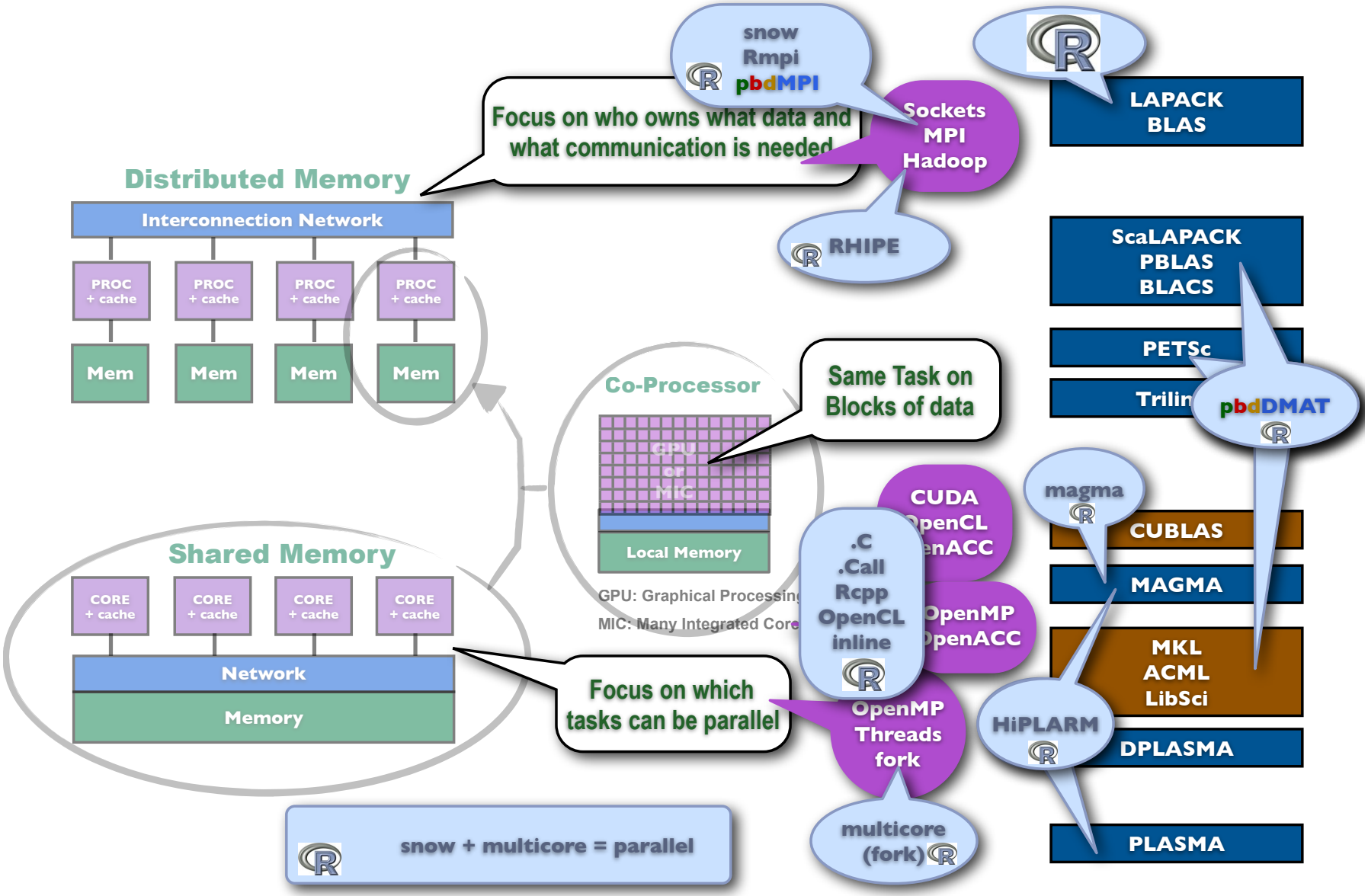
Distributed

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

CRAN HPC Task View

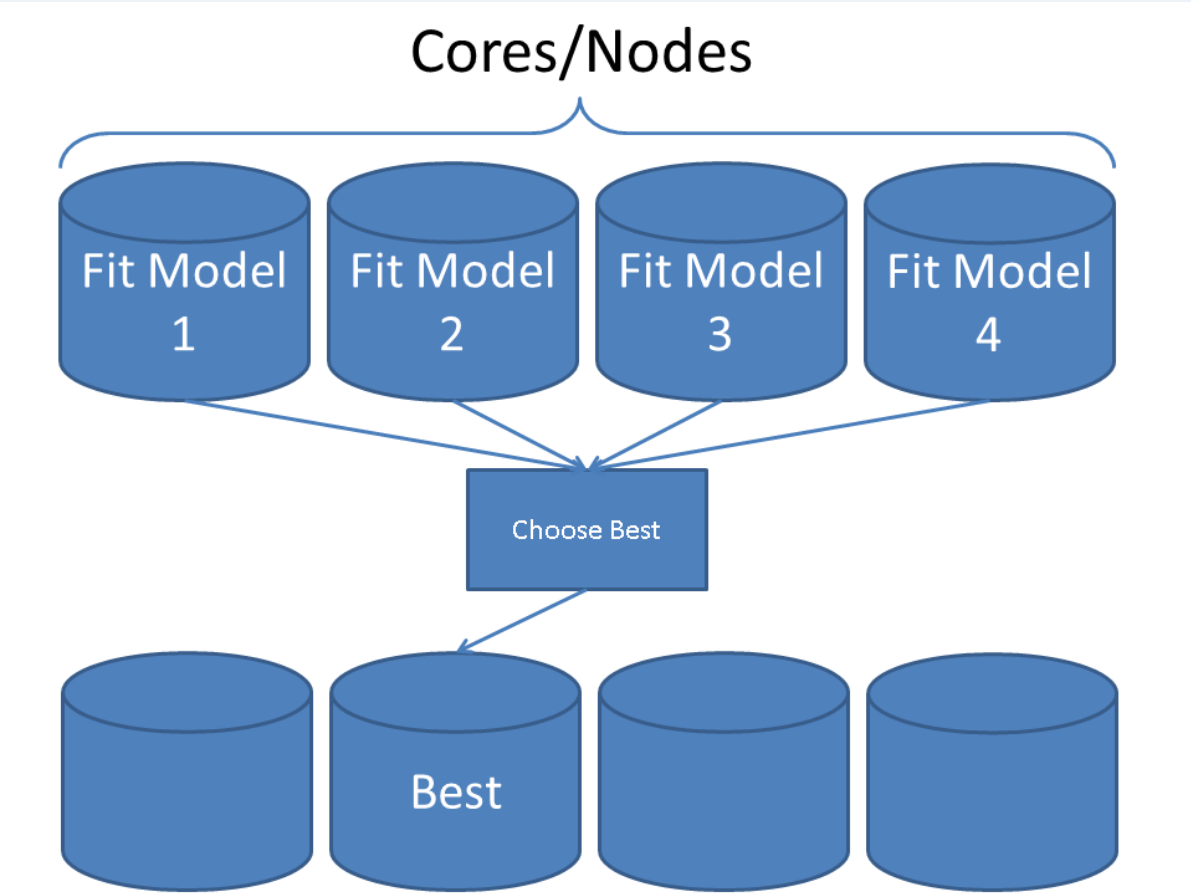
For more examples, see: http://cran.r-project.org/web/views/HighPerformanceComputing.html

Parallel Programming Packages for R



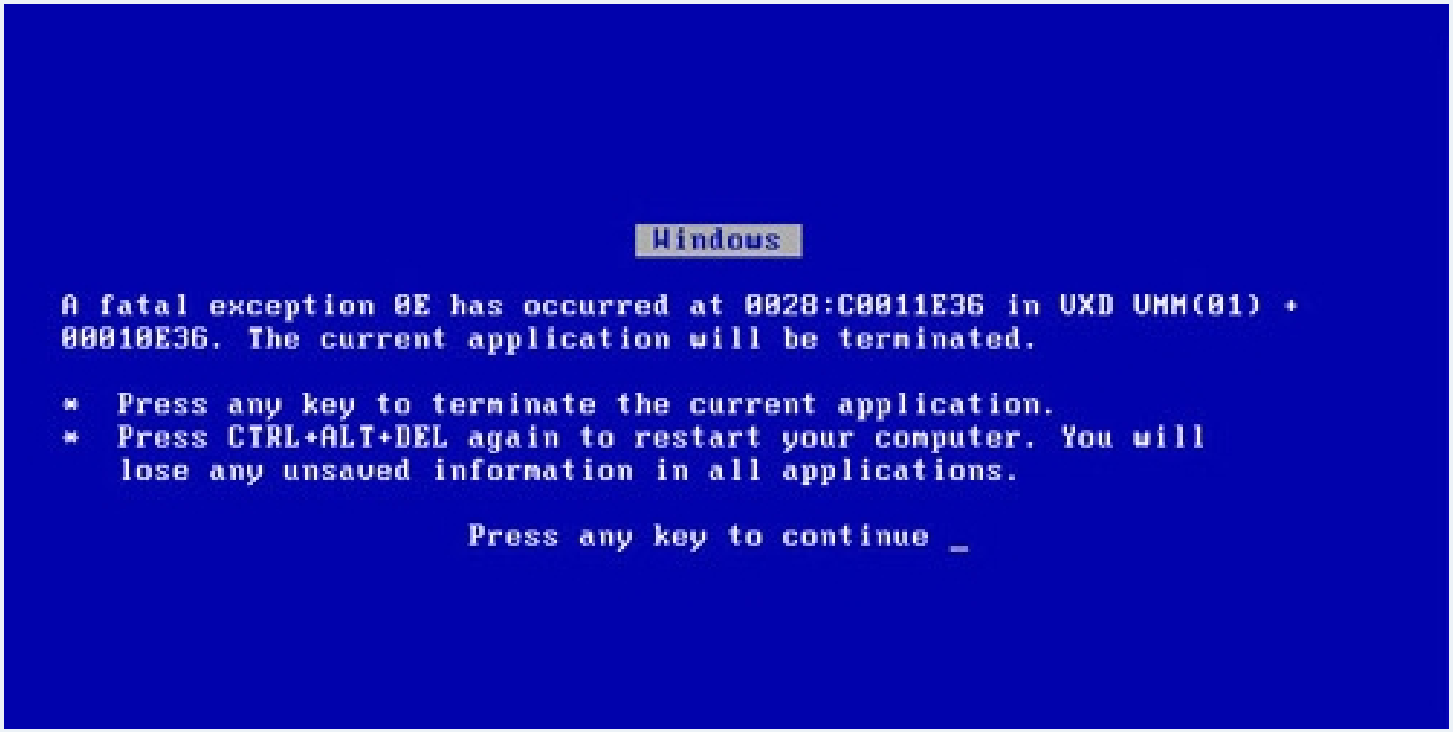
Independence

- Parallelism requires *independence*.
- Separate evaluations of R functions is embarrassingly parallel.



Portability

Many parallel R packages break on Windows



RNG's in Parallel

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

Parallel Programming: In Theory

Parallel Programming: In Practice

Summary

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

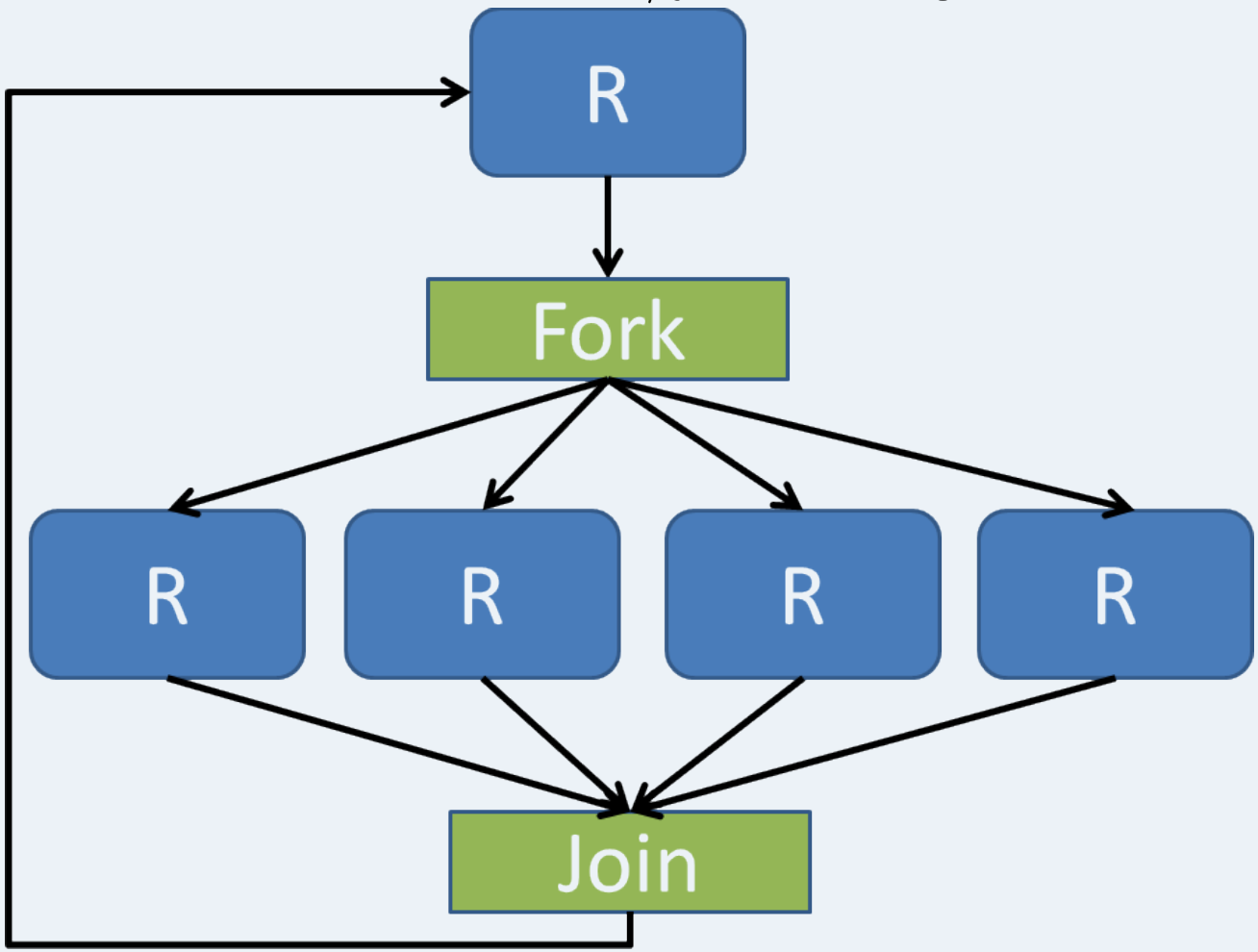
The parallel Package

- Comes with R $\geq$ 2.14.0
- Has 2 disjoint interfaces.

$$\textbf{parallel} = \textbf{snow} + \textbf{multicore}$$

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: snow

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

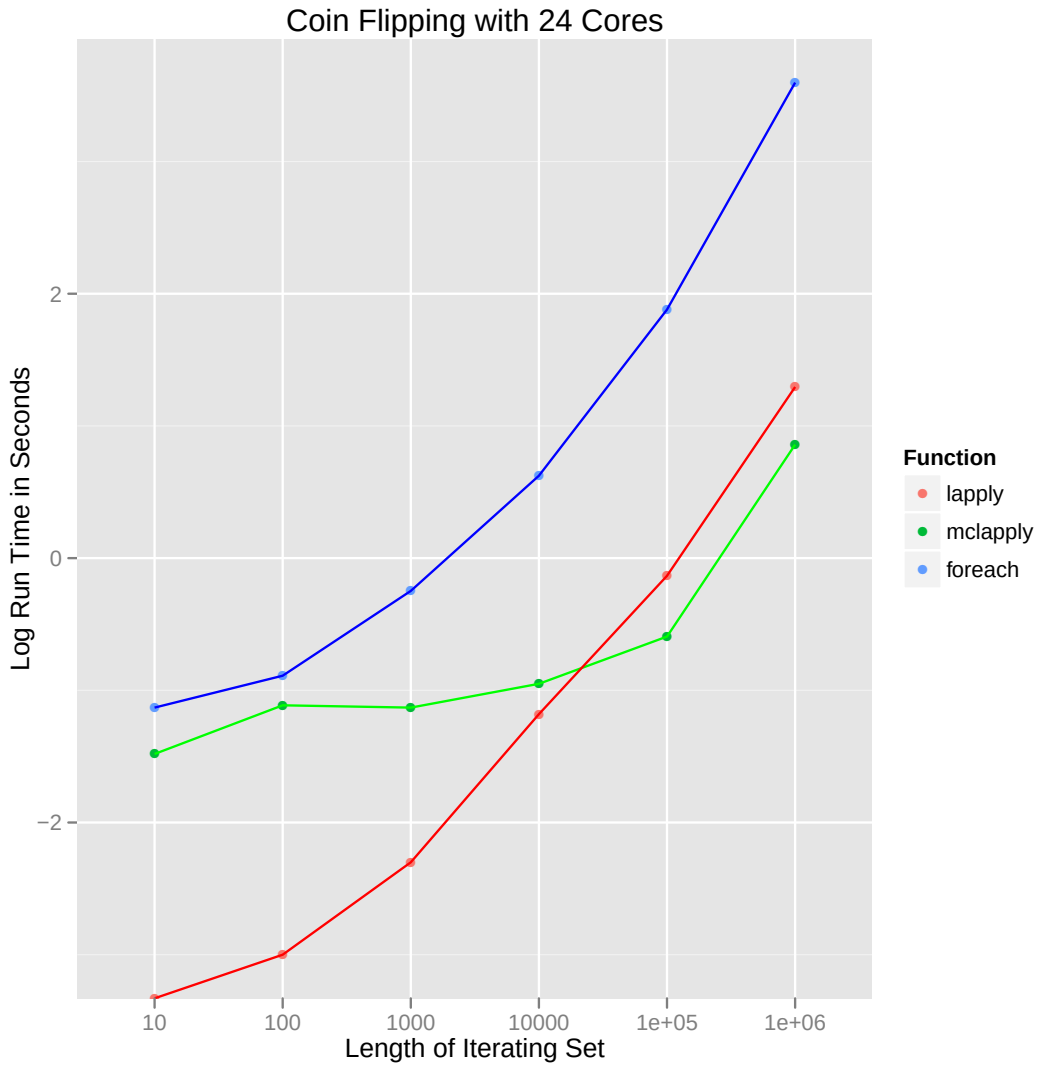
The foreach Package: The Idea

Unify the disparate interfaces.

The foreach Package

+ Works on all platforms (if backend does).
+ 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 if you aren't careful!

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.

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.

Hasty Explanation of MPI

- 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

```
1  mpi.spawn.Rslaves(nslaves=2)
2  #          2 slaves are spawned successfully. 0 failed.
3  # master (rank 0, comm 1) of size 3 is running on: wootabega
4  # slave1 (rank 1, comm 1) of size 3 is running on: wootabega
5  # slave2 (rank 2, comm 1) of size 3 is running on: wootabega
6
7  mpi.remote.exec(paste("I
       am",mpi.comm.rank(),"of",mpi.comm.size()))
8  # $slave1
9  # [1] "I am 1 of 3"
10 #
11 # $slave2
12 # [1] "I am 2 of 3"
13
14 mpi.exit()
```

Using Rmpi from snow

```
1 library(snow)
2 library(Rmpi)
3
4 cl <- makeCluster(2, type = "MPI")
5 clusterCall(cl, function() Sys.getpid())
6 clusterCall(cl, runif, 2)
7 stopCluster(cl)
8 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 > typeof(1)
2 [1] "double"
3 > typeof(2)
4 [1] "double"
5 > typeof(1:2)
6 [1] "integer"
```

Summary

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

Recall: Parallel R Packages

Shared Memory

1. **foreach**
2. **parallel**
3. **snow**
4. **multicore**

Distributed

1. **Rmpi**
2. **RHIPE, RHadoop**
3. **pbdR**

(and others. . .)

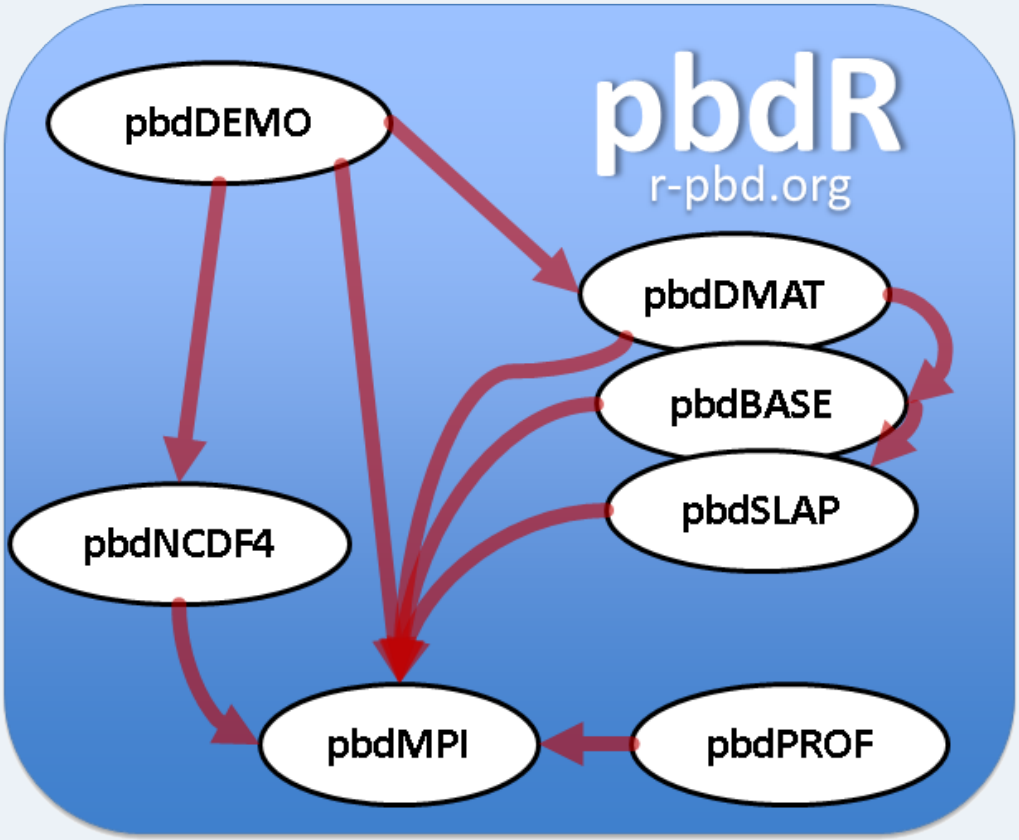
Programming with Big Data in R (pbdR)

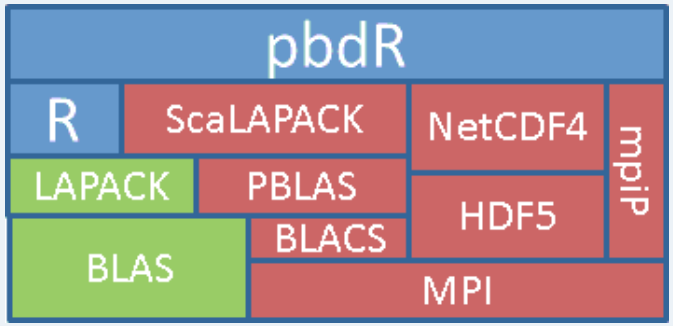
Striving for *Productivity, Portability, Performance*

- *Free*[a] R packages.
- Bridging high-performance compiled code with high-productivity of R
- Scalable, big data analytics.
- Offers implicit and explicit parallelism.
- Methods have syntax *identical* to R.

[a] MPL, BSD, and GPL licensed

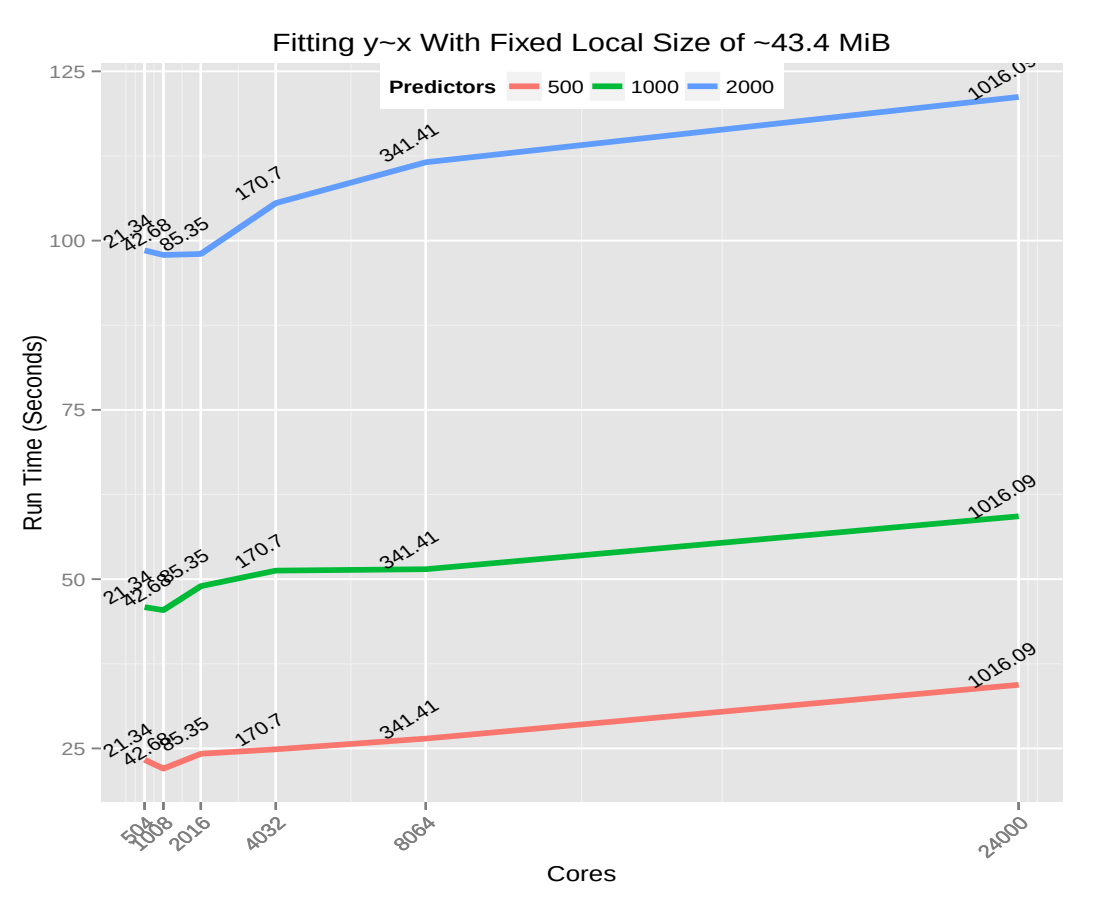
pbdR Packages





Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark




```
x <- ddmatrix("rnorm", nrow=m, ncol=n)
y <- ddmatrix("rnorm", nrow=m, ncol=1)
mdl <- lm.fit(x=x, y=y)
```

pbdR Scripts

- They're just R scripts.
- Can't run interactively (with more than 1 rank).
- We can use **pbdinline** to get "pretend interactivity".

ddmatrix: 2-dimensional Block-Cyclic with 6 Processors

$$x = \left[\begin{array}{cc|cc|cc|cc|c} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} & x_{17} & x_{18} & x_{19} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} & x_{48} & x_{49} \\ \hline x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} & x_{57} & x_{58} & x_{59} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} & x_{67} & x_{68} & x_{69} \\ \hline x_{71} & x_{72} & x_{73} & x_{74} & x_{75} & x_{76} & x_{77} & x_{78} & x_{79} \\ x_{81} & x_{82} & x_{83} & x_{84} & x_{85} & x_{86} & x_{87} & x_{88} & x_{89} \\ \hline x_{91} & x_{92} & x_{93} & x_{94} & x_{95} & x_{96} & x_{97} & x_{98} & x_{99} \end{array}\right]_{9\times 9}$$

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

Understanding `ddmatrix`: Local View

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{17} & x_{18} \\ x_{21} & x_{22} & x_{27} & x_{28} \\ \hline x_{51} & x_{52} & x_{57} & x_{58} \\ x_{61} & x_{62} & x_{67} & x_{68} \\ \hline x_{91} & x_{92} & x_{97} & x_{98} \end{array}\right]_{5\times4} \quad \left[\begin{array}{cc|c} x_{13} & x_{14} & x_{19} \\ x_{23} & x_{24} & x_{29} \\ \hline x_{53} & x_{54} & x_{59} \\ x_{63} & x_{64} & x_{69} \\ \hline x_{93} & x_{94} & x_{99} \end{array}\right]_{5\times3} \quad \left[\begin{array}{cc} x_{15} & x_{16} \\ x_{25} & x_{26} \\ \hline x_{55} & x_{56} \\ x_{65} & x_{66} \\ \hline x_{95} & x_{96} \end{array}\right]_{5\times2}$$

$$\left[\begin{array}{cc|cc} x_{31} & x_{32} & x_{37} & x_{38} \\ x_{41} & x_{42} & x_{47} & x_{48} \\ \hline x_{71} & x_{72} & x_{77} & x_{78} \\ x_{81} & x_{82} & x_{87} & x_{88} \end{array}\right]_{4\times4} \quad \left[\begin{array}{cc|c} x_{33} & x_{34} & x_{39} \\ x_{43} & x_{44} & x_{49} \\ \hline x_{73} & x_{74} & x_{79} \\ x_{83} & x_{84} & x_{89} \end{array}\right]_{4\times3} \quad \left[\begin{array}{cc} x_{35} & x_{36} \\ x_{45} & x_{46} \\ \hline x_{75} & x_{76} \\ x_{85} & x_{86} \end{array}\right]_{4\times2}$$

$$\text{Processor grid} = \left|\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \end{array}\right| = \left|\begin{array}{ccc} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{array}\right|$$

Methods for class ddmatrix

pbdDMAT has over 100 methods with *identical* syntax to R:

- `` `[` ``, `rbind()`, `cbind()`, ...
- `lm.fit()`, `prcomp()`, `cov()`, ...
- `` `%*%` ``, `solve()`, `svd()`, `norm()`, ...
- `median()`, `mean()`, `rowSums()`, ...

Serial Code

```
1 cov(x)
```

Parallel Code

```
1 cov(x)
```

ddmatrix Syntax

```
1 cov.x <- cov(x)
2 pca <- prcomp(x)
3 x <- x[, -1]
4 col.sd <- apply(x, MARGIN=2, FUN=sd)
```

Part IV

Wrapup

13 Wrapup

Thanks so much for attending!

Questions?

schmidt@math.utk.edu