

Essentials of High Performance and Parallel Statistical Computing with R

1. Introduction, Resources, and Resource Managers

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017



Objectives

Our main goal is to engage statistical researchers to become acquainted with essential concepts of parallel computing in a familiar environment of R. Our philosophy is for users to be aware of the underlying hardware and software reality without having to worry about syntax details of low-level libraries and languages. By the end of this course, we expect that participants are able to develop their own applications, utilizing platforms ranging from multicore laptops through high performance computing resources.



Outline

- ① Introduction, Resources, and Resource Managers (this section)
- ② Speeding up Your R Code
- ③ Parallel Hardware and Programming Paradigms
- ④ Multicore Parallel Computing
- ⑤ Distributed Parallel Computing with pbdMPI
- ⑥ Distributed Parallel Computing with pbdDMAT
- ⑦ Going Interactive with pbdR Client-Server



Compute Resources



- Your laptop has 2-4 cores, a few GiB ram.
- Say you need more...
- What are our options?



Campus Cluster



- Pros:

- Cheap to free.
- Probably available right now.

- Cons:

- Need campus affiliation
- Not all schools have one.
- Usually very full.
- Often not the best equipment.



XSEDE (NSF)



- Pros:

- Free to academia (National Science Foundation funded)
- Many unique resources available.
- Extensive support and collaboration available (ECSS).

- Cons:

- Proposal process (ranges from rubber stamp to very competitive, depending on what you want).
- Not just anyone can get access (focused on academia).

See: <https://www.xsede.org/>



OLCF, ALCF, NERSC (DOE)



- Pros:

- Free! (Department of Energy funded).
- Resources are “leadership class”: largest systems in the United States.

- Cons:

- Proposal process (highly competitive).
- Resources are “leadership class”: hard to use.

If interested, start with a [DD](#).

Consider an INCITE proposal in a few years.



The Cloud



- Pros:

- Can get access RIGHT THIS VERY SECOND (if you can pay)
- Lots of guides/documentation.
- Good enough for most.

- Cons:

- \$\$\$
- Low quality hardware.
- Support virtually non-existent.

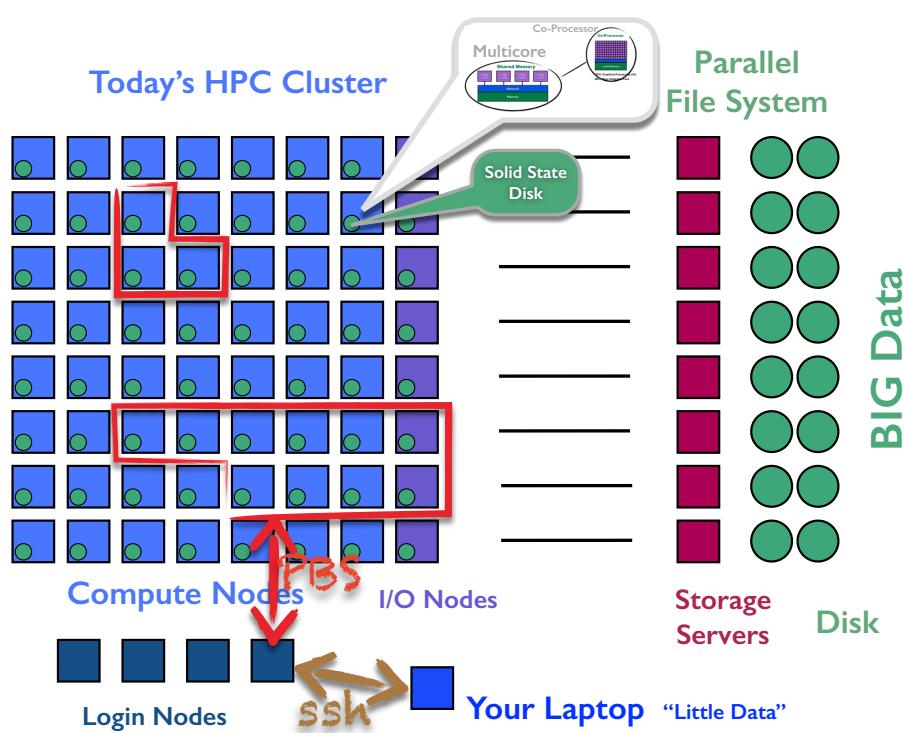


Some Other Concerns

- Cloud: easy to install software, hard to use distributed.
- Clusters: exact opposite!
- Clusters getting better with singularity. The Cloud...???



Anatomy of a Cluster



PBS qsub for Batch and Interactive Use

After login on a “Login Node”, a job scheduler like PBS is used to allocate node resources and schedule jobs for a user. An example batch submission script, which allows you to logout after submission and collect your results when you log back in:

..../scripts/intro/script.pbs

```

1 #!/bin/sh
2 #PBS -A your_account
3 #PBS -l nodes=4:ppn=16,walltime=00:30:00
4 #PBS -q batch
5 #PBS -o test.log
6 #PBS -e test.err
7
8 module load PE-gnu
9 module load r
10
11 cd your_working_directory
12
13 mpirun -np 64 Rscript your_script_code.r

```

Batch use

At the Login Node shell prompt:

```
1 $ qsub script.pbs
```

PBS qsub for Batch and Interactive Use

Interactive batch use

Interactive use example on an ORNL cluster Rhea. While the allocation is interactive (you get a shell prompt from one of your allocated nodes), the `mpirun` submission is still batch. Here, you wait for your code to complete.

```

1 $ qsub -I -X -A your_account -l walltime=0:30:00 -l nodes=4
2 qsub: waiting for job 233324.rhea-mgmt2g.ccs.ornl.gov to start
3 qsub: job 233324.rhea-mgmt2g.ccs.ornl.gov ready
4
5 $ module load PE-gnu
6 $ module load r
7 $ cd your_working_directory
8 $ mpirun -np 64 Rscript your_script_code.r

```

Starting Docker with RStudio Server on a Mac

Start the pbdr-workshop container in a terminal window (after installing Docker and the container):

```
1 $ docker run -i -t -v your_scripts_directory/scripts:/home/pbdR/scripts
   -p 8787:8787 -p 55555:55555 rbigdata/pbdr-workshop
2 pbdR@117792a0e5ba:~$
```

Makes the container contents available, gives you a shell prompt, mounts your scripts directory inside the container, enables RStudio Server port 8787, and enables **remoter** package port 55555.

To quit:

```
1 pbdR@117792a0e5ba:~$ exit
```

To connect a browser RStudio client

```
1 http://localhost:8787
```

Username/password is pbdR/pbdR



Starting Docker with RStudio Server on Windows

Start the pbdr-workshop container by clicking the icon (Docker Quickstart Terminal), then:

```
1 $ docker run -i -t -v your_scripts_directory/scripts:/home/pbdR/scripts
   -p 8787:8787 -p 55555:55555 rbigdata/pbdr-workshop
2 pbdR@117792a0e5ba:~$
```

Makes the container contents available, gives you a shell prompt, mounts your scripts directory inside the container, enables RStudio Server port 8787, and enables **remoter** package port 55555.

To quit:

```
1 pbdR@117792a0e5ba:~$ exit
```

To connect a browser RStudio client

```
1 http://192.168.99.100:8787
```

Username/password is pbdR/pbdR



Essentials of High Performance and Parallel Statistical Computing with R

2. Speeding up Your R Code

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017



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 distributed)
- 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

..../scripts/fastR/system.time.R

```

1 x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)
2
3 system.time(t(x) %*% x)
4
5 system.time(crossprod(x))
6
7 system.time(a <- cov(x))
8
9 system.time(b <- crossprod(scale(x, scale=FALSE))/(nrow(x) - 1))
10
11 all.equal(a, b)

```



Performance Profiling Tools: Rprof()

Samples call stack (default every 0.02 seconds)

scripts/fastR/profile.R

```

1 > x <- matrix( rnorm( 10000*250 ), nrow = 10000, ncol = 250)
2 > Rprof()
3 > invisible( prcomp( x ) )
4 > Rprof( NULL )
5 > summaryRprof()
6 $by.self
7           self.time self.pct total.time total.pct
8 "La.svd"        0.64    78.05     0.70    85.37
9 "%*%"         0.06     7.32     0.06     7.32
10 "aperm.default" 0.04     4.88     0.04     4.88
11 "is.finite"    0.04     4.88     0.04     4.88
12 "matrix"        0.04     4.88     0.04     4.88
13
14 $by.total
15          total.time total.pct self.time self.pct
16 "prcomp.default" 0.82   100.00     0.00     0.00
17 "prcomp"         0.82   100.00     0.00     0.00
18 "svd"            0.72    87.80     0.00     0.00
19 "La.svd"         0.70    85.37     0.64    78.05
20 "%*%"           0.06     7.32     0.06     7.32
21 ### output truncated by presenter
22
23 $sample.interval
24 [1] 0.02
25
26 $sampling.time
27 [1] 0.98

```



Performance Profiling Tools: Rprof()

scripts/fastR/profile.R

```

1 > Rprof( interval = .99 )
2 > invisible( prcomp( x ) )
3 > Rprof( NULL )
4 > summaryRprof()
5 $by.self
6 [1] self.time self.pct total.time total.pct
7 <0 rows> (or 0-length row.names)
8
9 $by.total
10 [1] total.time total.pct self.time self.pct
11 <0 rows> (or 0-length row.names)
12
13 $sample.interval
14 [1] 0.99
15
16 $sampling.time
17 [1] 0

```



Performance Profiling Tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

scripts/fastR/benchmark.R

```

1 x <- matrix( rnorm( 10000*500 ), nrow = 10000, ncol = 500 )
2
3 f <- function( x ) t( x ) %*% x
4 g <- function( x ) crossprod( x )
5
6 library( rbenchmark )
7 benchmark( f( x ), g( x ) )
8
9 #      test replications elapsed relative
10 # 1 f(x)          100  64.153    2.063
11 # 2 g(x)          100  31.098    1.000

```



Performance Profiling Tools: pbdPAPI

PAPI_cache_access.R

```

1 library( pbdPAPI )
2 library( inline )

1 bad_cache_access <- "
2   int i, j;
3   const int n = INTEGER( n_ )[0];
4   Rcpp::NumericMatrix x( n, n );
5   for ( i=0; i<n; i++ )
6     for ( j=0; j<n; j++ )
7       x( i, j ) = 1.;
8   return x;
9 "
1 good_cache_access <- "
2   int i, j;
3   const int n = INTEGER( n_ )[0];
4   Rcpp::NumericMatrix x( n, n );
5   for ( j=0; j<n; j++ )
6     for ( i=0; i<n; i++ )
7       x( i, j ) = 1.;
8   return x;
9 "
1 bad <- cxxfunction( signature( n_ = "integer" ), body =
2   bad_cache_access, plugin = "Rcpp" )
2 good <- cxxfunction( signature( n_ = "integer" ), body =
3   good_cache_access, plugin = "Rcpp" )
3 n <- 10000L
4
5 ### Summary of cache misses
6 system.cache( bad( n ) )
7 system.cache( good( n ) )
8
9 ### Ratio of total cache misses to total cache accesses
10 system.cache( bad( n ), events = "l2.ratio" )
11 system.cache( good( n ), events = "l2.ratio" )

```



Performance Profiling Tools: pbdPAPI

..../scripts/fastR/PAPI_sort.R

```

1 library(pbdPAPI)
2
3 x <- runif(1e6)
4
5 ##### Sorting is not a floating point operation.
6 system.flops(sort(x))
7
8 ##### It does require lots of memory access, though.
9 system.cache(sort(x))
10
11 system.utilization(sort(x))

```



Performance Profiling Tools: pbdPROF

..../scripts/fastR/pbdPROF.R

```

1 library(pbdPROF)
2
3 prof <- read.prof("output.mpiP")
4
5 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

```

1 n <- 1e5
2 x <- seq( 0, 1, length.out = n )
3 f <- function( x ) exp( x^3 + 2.5*x^2 + 12*x + 0.12 )
4 y1 <- numeric( n )
5
6 set.seed( 12345 )
7 system.time(
8   for( i in 1:n )
9     y1[ i ] <- f( x[ i ] ) + rnorm( 1 )
10 )
11
12 set.seed( 12345 )
13 system.time(
14   y2 <- f( x ) + rnorm( n )
15 )
16
17 all.equal( y1, y2 )

```



Compute Once if not Changing

Bad Loop

```

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

```

Good Loop (from pbdML)

```

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 a Real R Package

Excerpt from Original function

```

1 while(i<=N){
2   for(j in 1:i){
3     d.k <- as.matrix(x)[l==j,l==j]
4     ...

```

Excerpt from Modified function

```

1 x.mat <- as.matrix(x)
2
3 while(i<=N){
4   for(j in 1:i){
5     d.k <- x.mat[l==j,l==j]
6     ...

```

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



OpenBLAS (multithreaded Basic Linear Algebra Subroutines)

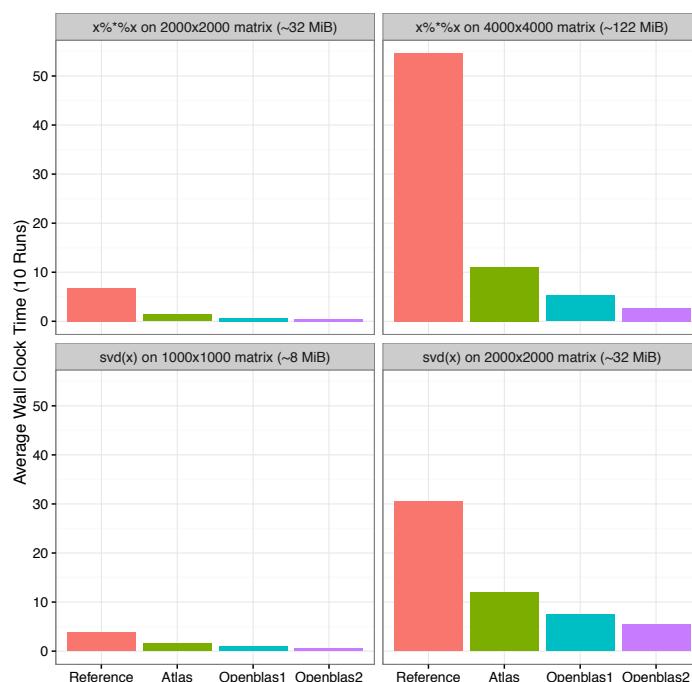
- pbdr-workshop container includes OpenBLAS and **openblasctl**
- Native Install: <http://www.openblas.net/>
- Batch thread control: OPENBLAS_NUM_THREADS=n
- Dynamic thread control: “wrathematics/openblasctl” micropackage on GitHub
..../scripts/fastR/openblas.R

```

1 x <- matrix( rnorm( 1e6*1e2 ), nrow = 1e6 )
2
3 library(memuse)
4 howbig(1e6, 1e2, unit="GB")
5 sessionInfo()
6
7 library( openblasctl )
8
9 openblas_set_num_threads( 1 )
10 system.time( y <- t( x ) %*% x )
11 openblas_set_num_threads( 2 )
12 system.time( y <- t( x ) %*% x )
13 openblas_set_num_threads( 4 )
14 system.time( y <- t( x ) %*% x )
15
16 openblas_set_num_threads( 1 )
17 system.time( z <- crossprod( x ) )
18 openblas_set_num_threads( 2 )
19 system.time( z <- crossprod( x ) )
20 openblas_set_num_threads( 4 )
21 system.time( z <- crossprod( x ) )

```

Benefit of OpenBLAS



Inserting C/C++ Code Into R

.Call

- Standard R interface to C code
- Lightweight but clunky

Rcpp: Incorporating C++ code into R

Authors: Dirk Eddelbuettel and Romain Francois

- Simplifies integrating C++ code with R
- Maps R objects (vectors, matrices, functions, environments, . . .) to dedicated C++ classes
- Broad support for C++ Standard Template Library idioms.
- C++ code can be compiled, linked and loaded on the fly, or added via packages.
- Error and exception code handling



Rcpp Example: A simple row max calculation

`cat ex_max.cpp`

```

1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 //[[Rcpp::export]]
5
6 NumericVector row_max( NumericMatrix m )
7 {
8     int nrows = m.nrow( );
9     NumericVector maxPerRow( nrows );
10
11    for ( int i = 0; i < nrows; i++ )
12    {
13        maxPerRow[ i ] = Rcpp::max( m( i, _ ) );
14    }
15
16    return ( maxPerRow );
17 }
```

One can get configuration values by

```

1 setenv PKG_CXXFLAGS 'Rscript -e "Rcpp:::CxxFlags( )"'
2 setenv PKG_LIBS 'Rscript -e "Rcpp:::LdFlags( )"'
```



Rcpp Example (con'd)

A simple row max calculation

cat ex_max.R

```

1 library( Rcpp )
2 Sys.setenv( "PKG_CXXFLAGS" =
3   "-I /sw/redhat6/r/3.3.0/rhel6_gnu4.8.2/lib64/R/library/Rcpp/include" )
4 Sys.setenv( "PKG_LIBS"="-lm" )
5
6 sourceCpp( "ex_max.cpp" )
7
8 set.seed( 27 )
9 X <- matrix( rnorm( 4 * 4 ), 4, 4 )
10 X
11
12 print( "Rcpp" )
13 row_max( X )

```

Rscript ex_max.R

```

1 Rscript ex_max.R
2      [,1]      [,2]      [,3]      [,4]
3 [1,]  1.9071626 -1.093468881  2.13463789  1.5702953
4 [2,]  1.1448769  0.295241218  0.23784461  0.1580101
5 [3,] -0.7645307  0.006885942 -1.28512736 -0.7457995
6 [4,] -1.4574325  1.157410886  0.03482725 -1.0688030
7 [1] "Rcpp"
8 [1]  2.134637891  1.144876890  0.006885942  1.157410886

```



RcppArmadillo Example: Eigenvalue calculation

- The RcppArmadillo package is a set of bindings to the Armadillo C++ library.
- Armadillo is a templated C++ linear algebra library that uses supplied BLAS and LAPACK.
- Includes some machine learning libraries
- BLAS and LAPACK are also directly engaged from R.
- Probably not much faster than R direct but not having to come back out to R if C++ code needs to use linear algebra can produce gains.

cat eigen.cpp

```

1 #include <RcppArmadillo.h>
2 // [[Rcpp::depends(RcppArmadillo)]]
3 // [[Rcpp::export]]
4
5 arma::vec getEigenValues( arma::mat M )
6 {
7   return ( arma::eig_sym( M ) );
8 }

```



RcppArmadillo Example (con'd)

cat eigen.R

```

1 library( Rcpp )
2 library( RcppArmadillo )
3 Sys.setenv( "PKG_CXXFLAGS" = "-I
              /sw/redhat6/r/3.3.0/rhel6_gnu4.8.2/lib64/R/library/RcppArmadillo/include"
            )
4 Sys.setenv( "PKG_LIBS" = "-lm"
      )
6 sourceCpp( "eigen.cpp" )
7
8 set.seed( 27 )
9 X <- matrix( rnorm( 4 * 4 ), 4, 4 )
10 Z <- X %*% t( X )
11 print( "RcppArmadillo" )
12 getEigenValues( Z )
13
14 print( "R" )
15 eigen( Z )$values

```

Rscript eigen.R

```

1 [1] "RcppArmadillo"
2           [,1]
3 [1,]  0.03779289
4 [2,]  0.85043786
5 [3,]  2.03877658
6 [4,] 17.80747601
7 [1] "R"
8 [1] 17.80747601  2.03877658  0.85043786  0.03779289

```


Essentials of High Performance and Parallel Statistical Computing with R

3. Parallel Hardware and Programming Paradigms

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017

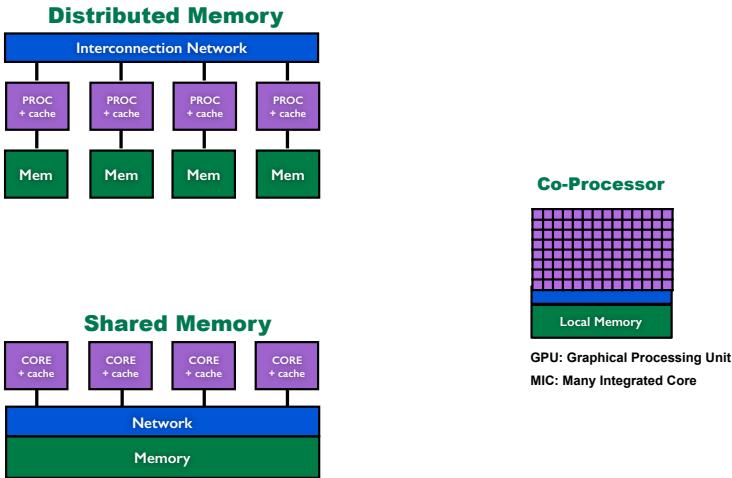


Outline

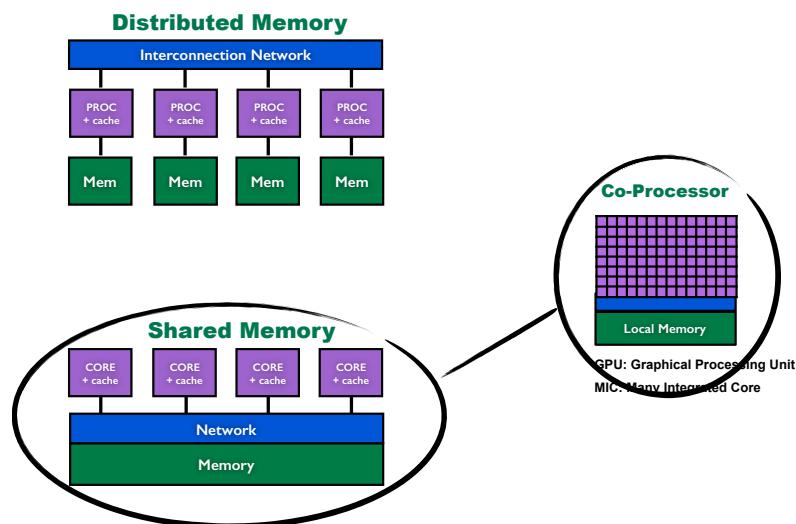
- Brief introduction to parallel hardware and software
 - Hardware flavors
 - Low-level native software
 - R interfaces to native software
 - Scalable libraries
 - R and scalable libraries
- Parallel programming paradigms
 - Shared memory vs. distributed memory
 - Manager-workers and fork-join
 - MapReduce
 - SPMD - single program, multiple data
 - Data-flow



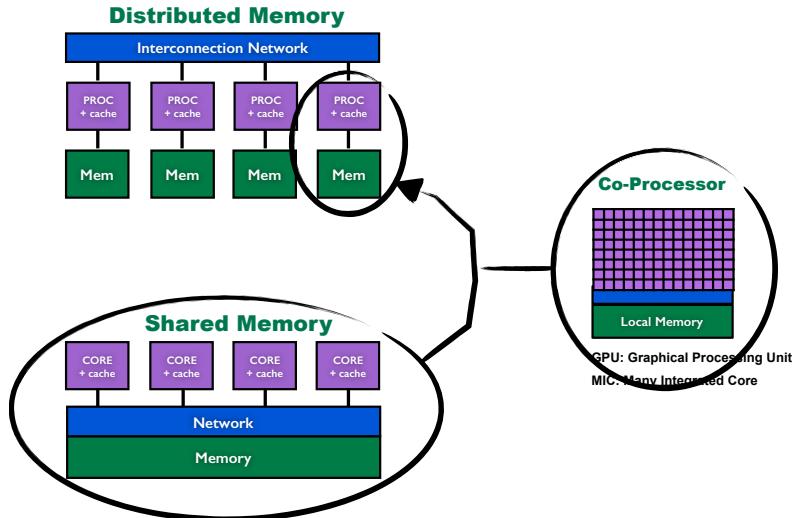
Three Basic Flavors of Hardware



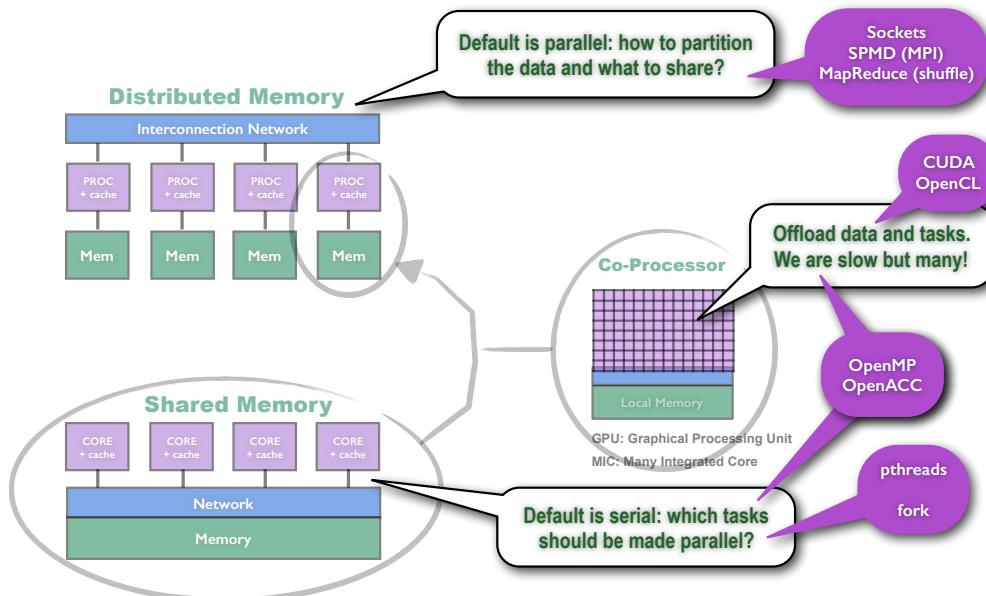
Your Laptop or Desktop



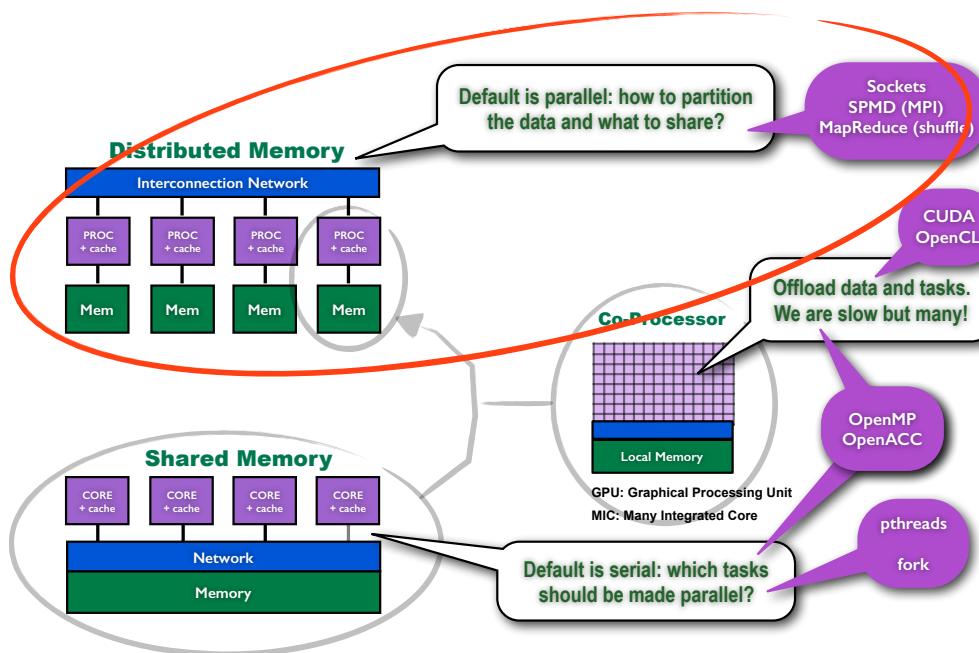
Server to Cluster to Supercomputer



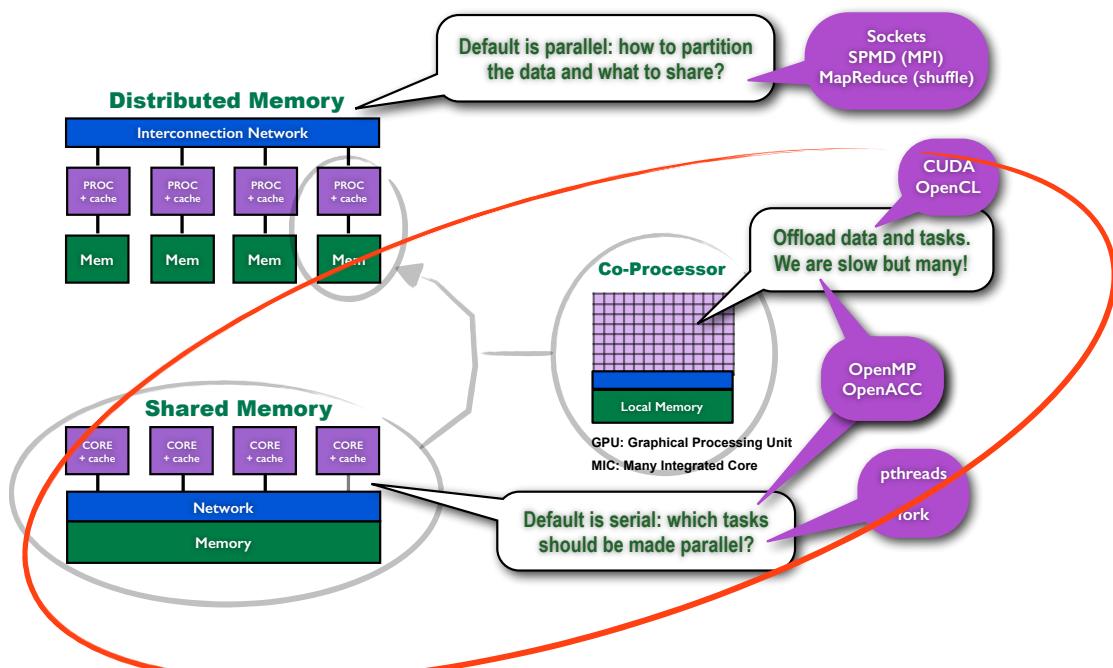
“Native” Programming Models and Tools



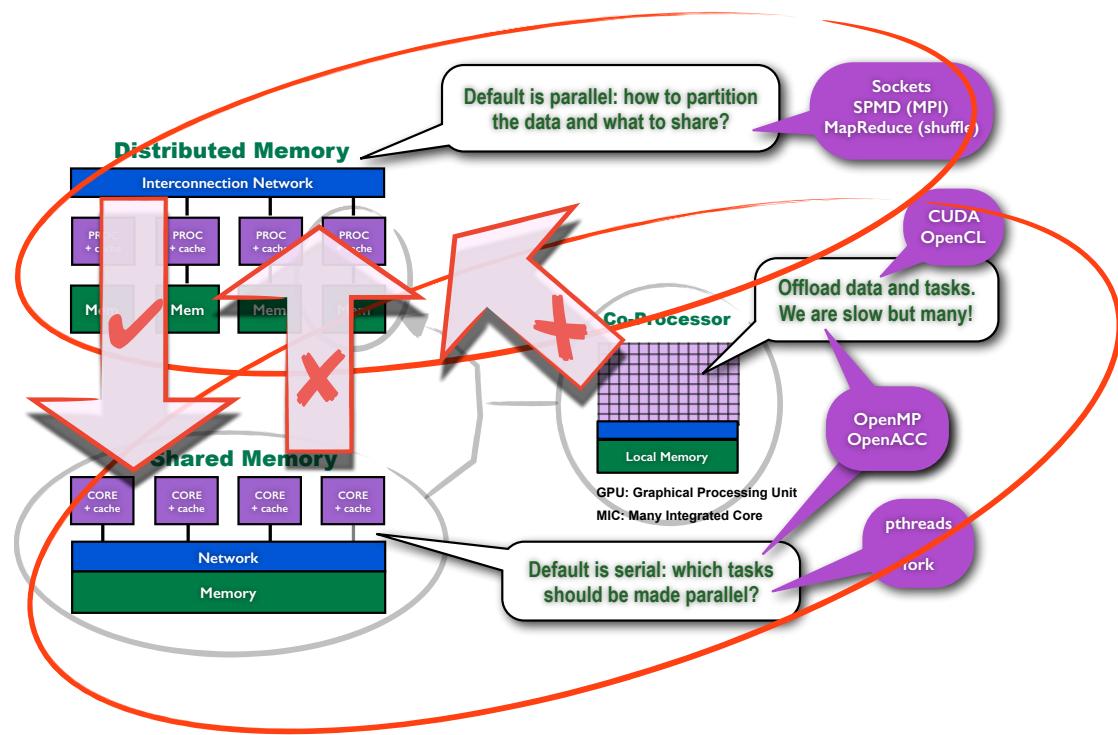
30+ Years of Parallel Computing Research



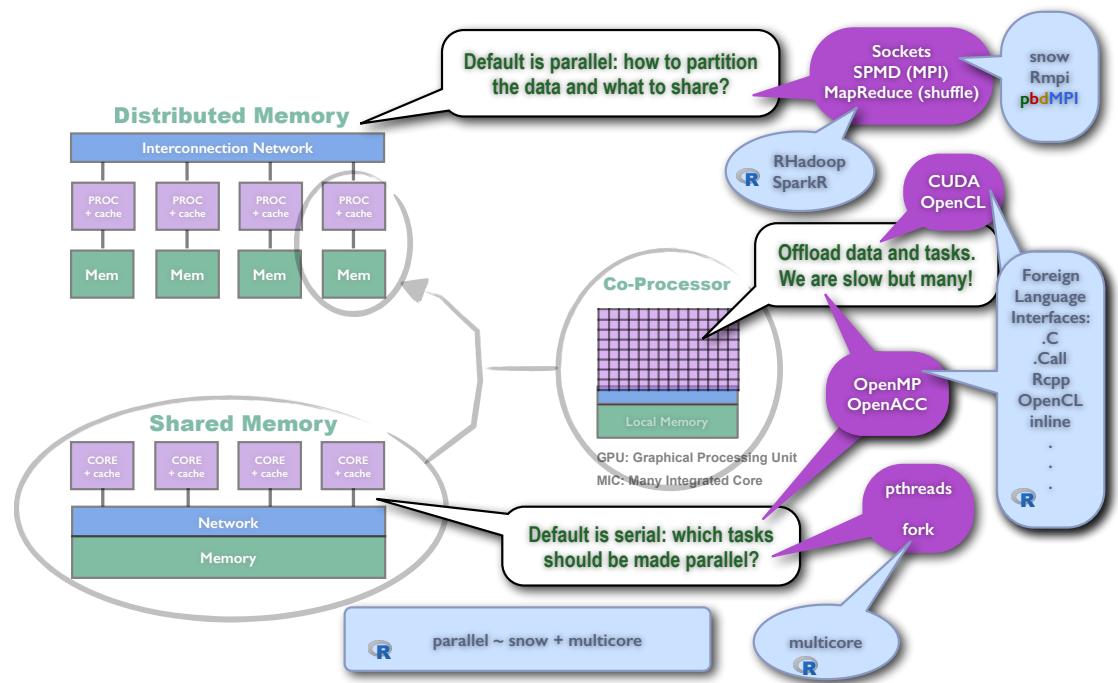
Last 10+ years of Advances



Distributed Programming Works in Shared Memory



R Interfaces to Low-Level Native Tools



Some packages in R for parallel computing

- parallel: multicore + snow
 - multicore: an interface to unix fork (no Windows)
 - snow: simple network of workstations
- pbdMPI, pbdDMAT and other pbd: use HPC concepts, simplify, and use scalable libraries
- foreach, doParallel: interface to hide hardware reality, can be difficult to debug
- Rmpi: simplified with pbdMPI for SPMD
- RHadoop, RHipe: limited to MapReduce, slow because file-backed
- datadr: divide-recombine, currently MapReduce/HADOOP back end
- SparkR: in-memory MapReduce, limited to Shuffle, MPI is faster and more flexible



Manager-Workers

- ① A serial program (Manager) divides up work and/or data
 - ② Manager sends work (and data) to workers
 - ③ Workers run in parallel without interaction
 - ④ Manager collects/combines results from workers
- Divide-Recombine fits this model
 - Concept appears similar to interactive and to client-server



MapReduce

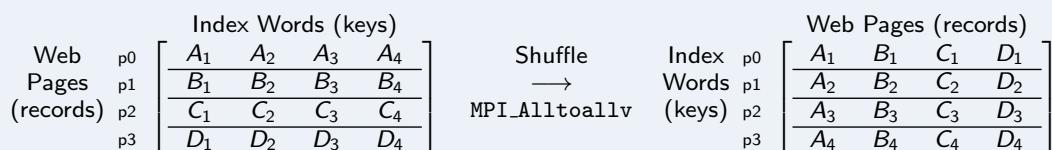
- A concept born of a search engine
- Decouples some coupled problems with an all-to-all communication: shuffle
- User needs to decompose computation into Map and Reduce steps
- User writes two serial codes: Map and Reduce



MapReduce: a Parallel Search Engine Concept

Search MANY documents

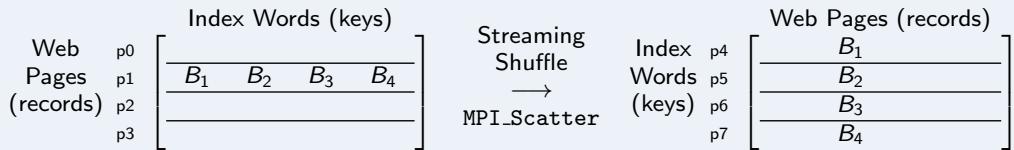
Serve MANY users



Matrix transpose in another language?



Can use different sets of processors



SPMD: Single Program Multiple Data

Write one general program so many copies of it can run asynchronously and cooperate (usually via MPI) to solve the problem.

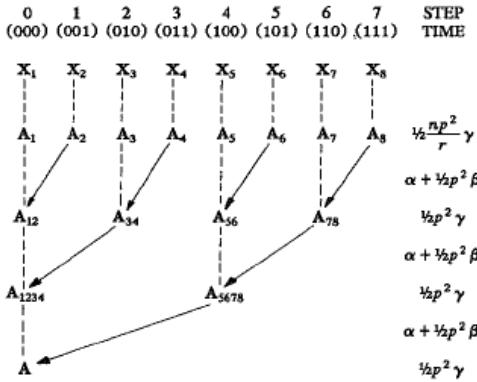
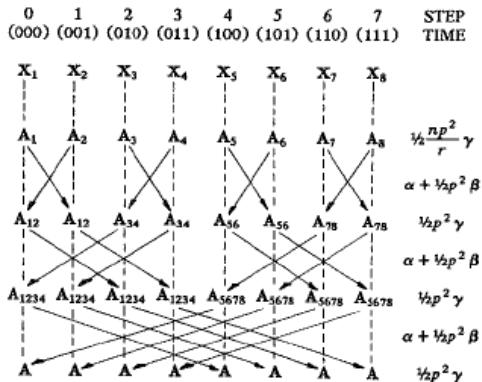
- The prevalent way of distributed programming in HPC for 30+ years
- Can handle tightly coupled parallel computations
- It is designed for batch computing
- There is usually no manager - rather, all cooperate
- Prime driver behind MPI specification
- Way to program server side in client-server



SPMD

$$A = X^T X = \sum_i X_i^T X_i, \text{ where } X = \begin{bmatrix} X_1 \\ \vdots \\ X_8 \end{bmatrix} \quad (\text{Row-Block partition})$$

Reduction on a 1986 hypercube: Managing bitflips and dimensions

Fig. 4. Computation of $A = XX^T$ on an 8-processor hypercube, with final result on processor 0.Fig. 6. Computation of $A = XX^T$ on an 8-processor hypercube, with final result on all processors.

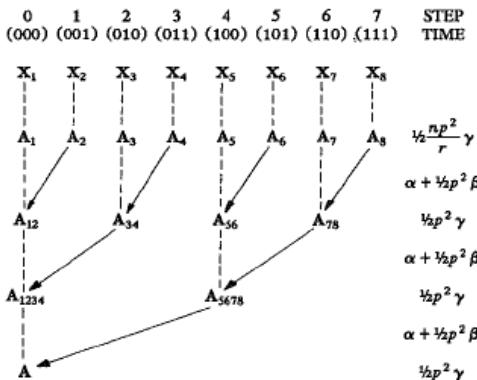
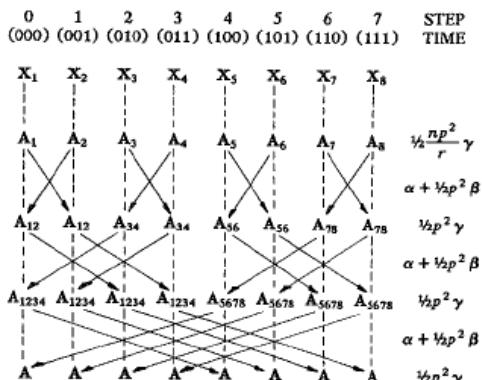
^aOstrouchov (1987). Parallel Computing on a Hypercube: An overview of the architecture and some applications. *Proceedings of the 19th Symposium on the Interface of Computer Science and Statistics*, p.27-32.

SPMD

$$A = X^T X = \sum_i X_i^T X_i, \text{ where } X = \begin{bmatrix} X_1 \\ \vdots \\ X_8 \end{bmatrix} \quad (\text{Row-Block partition})$$

Today: $A = \text{reduce}(\text{crossprod}(X))$ $A = \text{allreduce}(\text{crossprod}(X))$

pbdr

Fig. 4. Computation of $A = XX^T$ on an 8-processor hypercube, with final result on processor 0.Fig. 6. Computation of $A = XX^T$ on an 8-processor hypercube, with final result on all processors.

SPMD and MapReduce

(MPI and Shuffle)

One parallel code or pairs of serial codes

- SPMD: write one code that generalizes the serial code
- MapReduce: decompose serial code into pairs of map and reduce serial functions and a control code

Concepts differ in approach to communication

- SPMD makes communication explicit, gives choices (MPI)
- MapReduce hides communication, uses one choice (shuffle)
- MPI reduce operations take advantage of recursive doubling:
 - Most reductions are associative and commutative
 - Reducing n items takes $\log_2(n)$ steps
 - Fewer additions and fewer communication steps
 - Vendor MPI implementations are architecture-aware

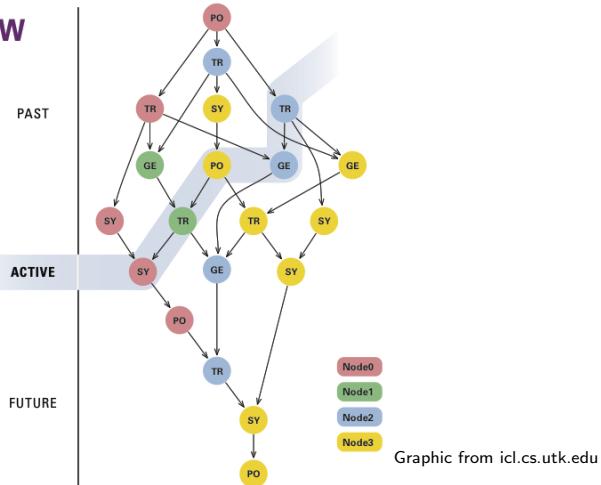


Data-flow: Parallel Runtime Scheduling and Execution Controller (PaRSEC)

EFFICIENT DATA FLOW REPRESENTATION

FEATURES

- Supports Distributed Heterogeneous Platforms
- Sustained Performance
- NUMA & Cache Aware Scheduling
- State-of-the-art Algorithms
- Capacity Level Scalability
- Performance Portability
- Implicit Communication
- Communication Overlapping



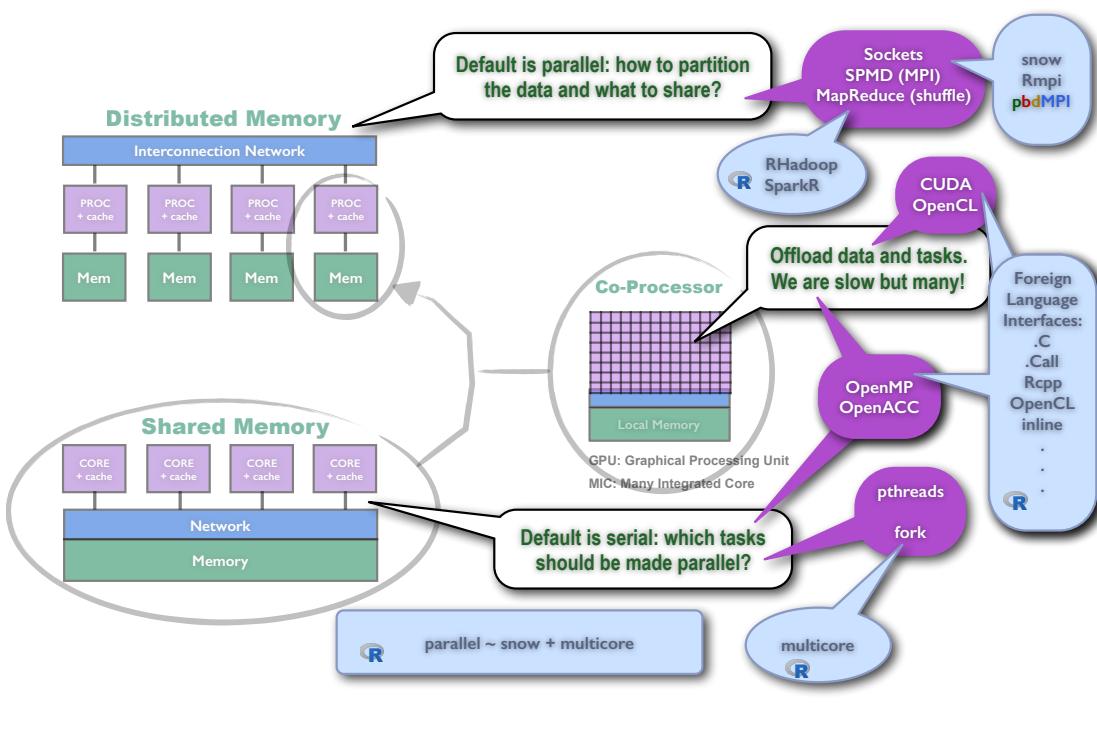
Graphic from icl.cs.utk.edu

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," IEEE Computing in Science and Engineering, Vol. 15, No. 6, 36-45, November, 2013.

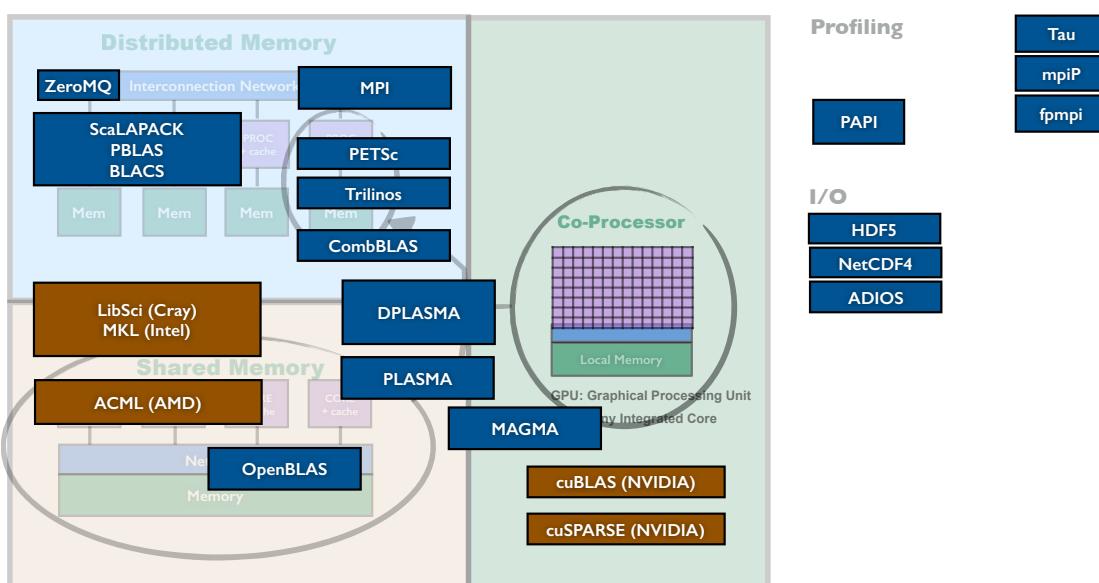
- Master data-flow controller runs distributed on all nodes
- Dynamic generation of current level in flow graph
- Effectively removes collective synchronizations for additional speedup



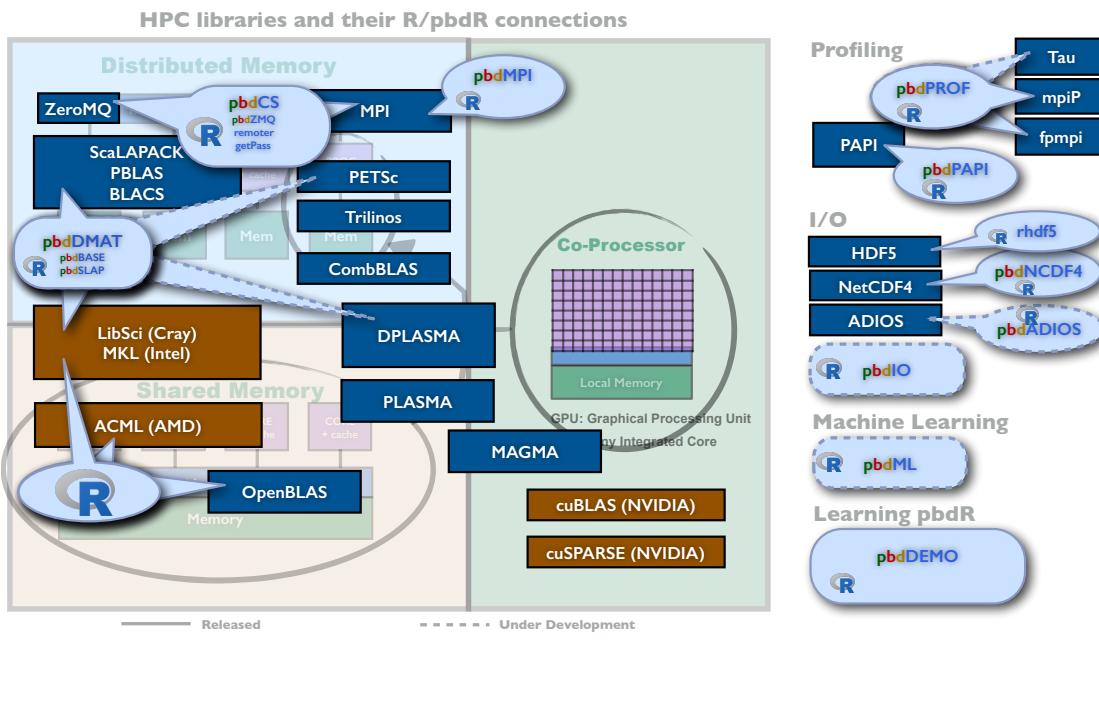
Recall: Hardware flavors and Low-Level Native Tools



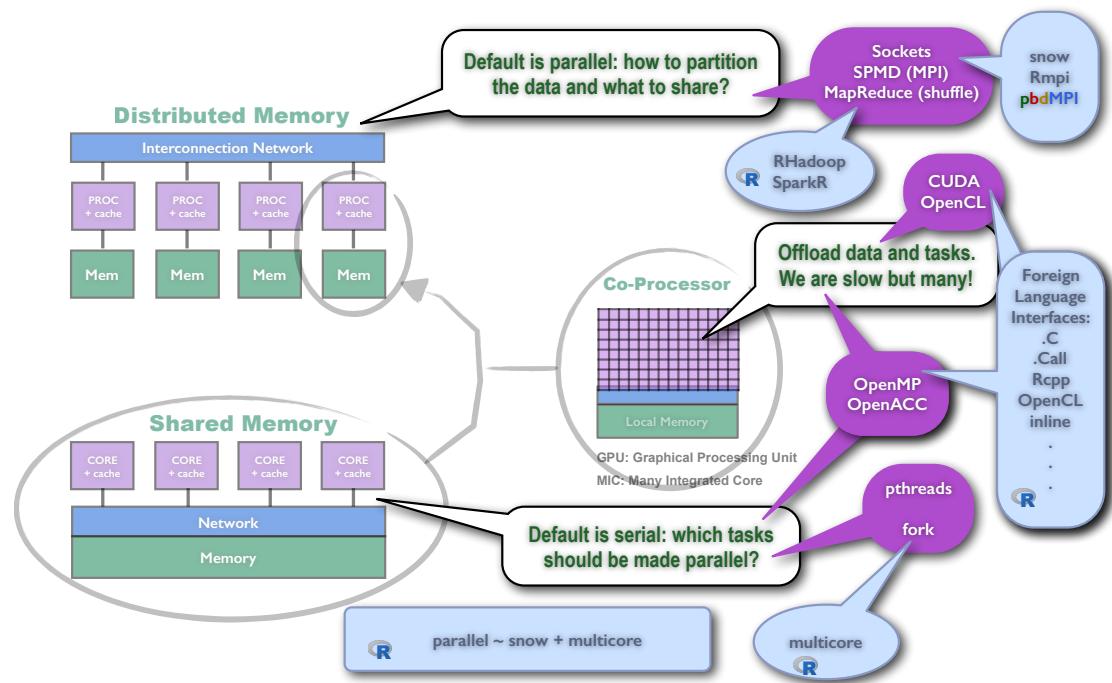
Scalable Libraries Mapped to Hardware



R and pbdR Interfaces to HPC Libraries



Recall: Hardware flavors and Low-Level Native Tools



Essentials of High Performance and Parallel Statistical Computing with R

4. Multicore Parallel Computing

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017

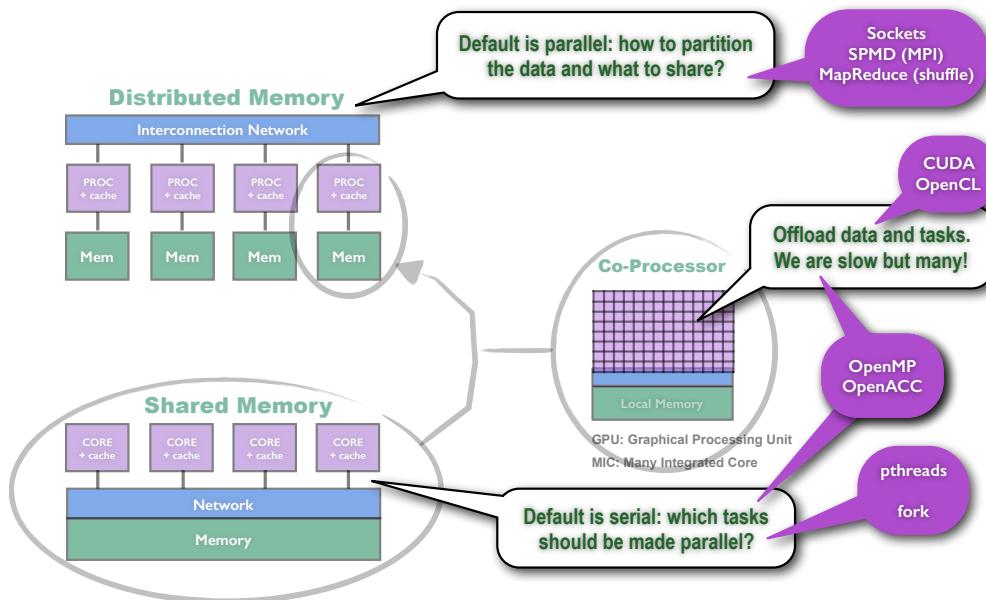


Outline

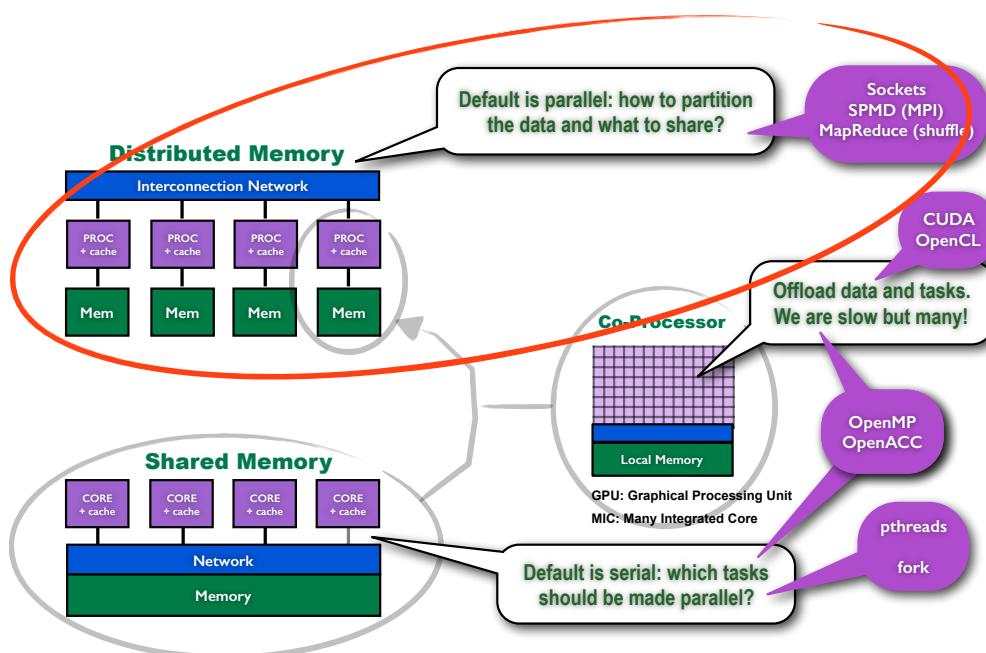
- Brief introduction to parallel hardware and software
- Using the parallel package in R



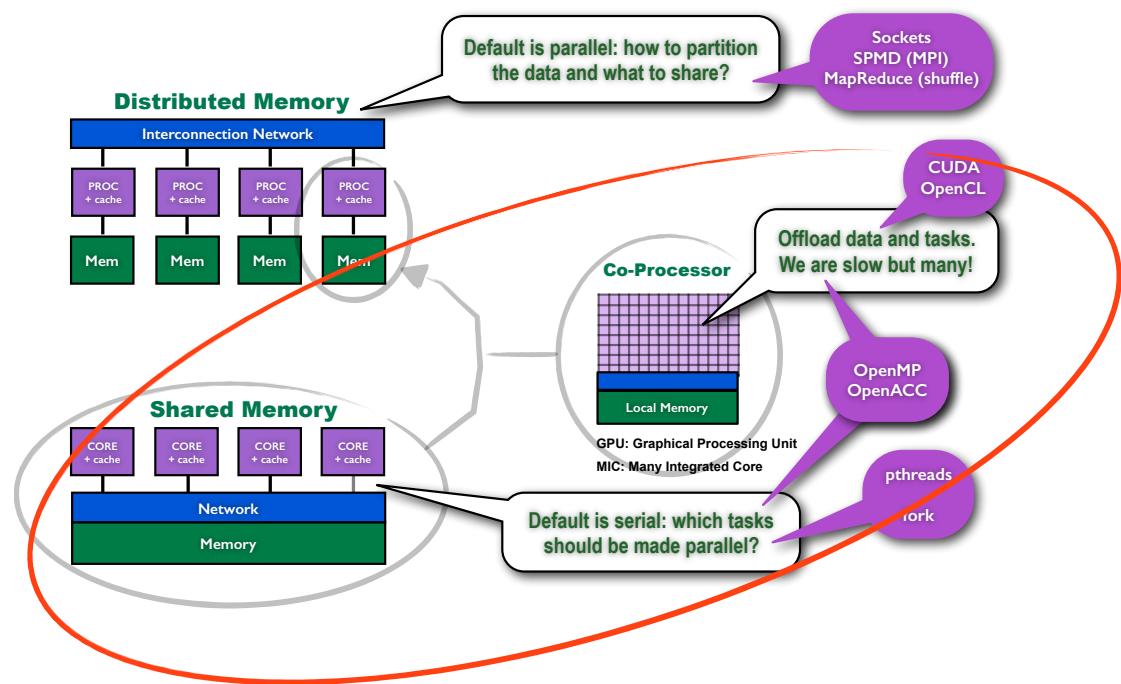
“Native” Programming Models and Tools



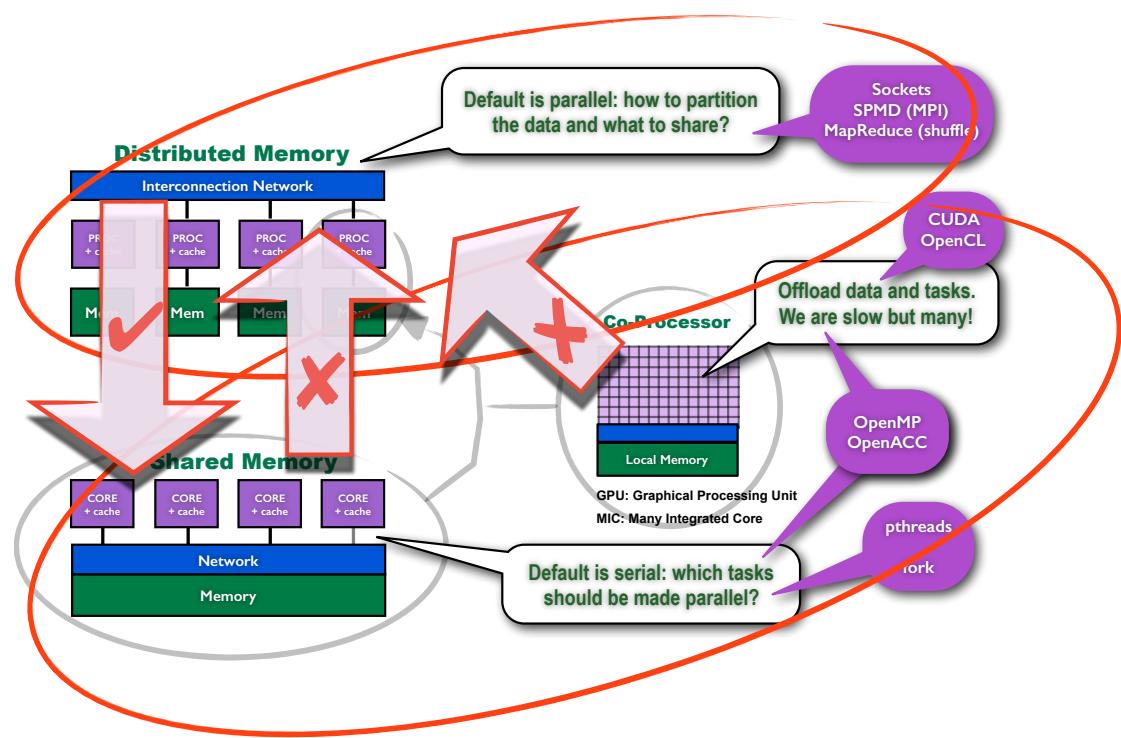
30+ Years of Parallel Computing Research



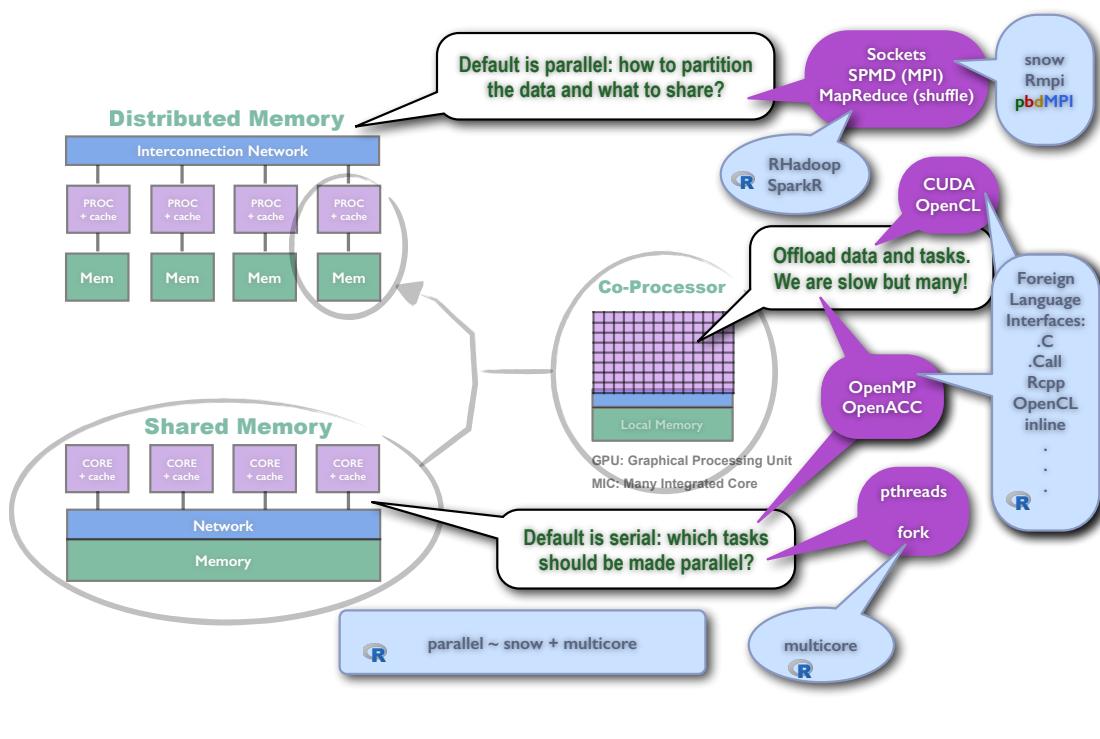
Last 10 years of Advances



Distributed Programming Works in Shared Memory



R Interfaces to Low-Level Native Tools



Using the **parallel** package in R

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

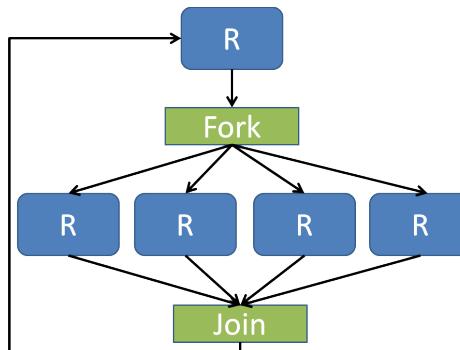
parallel \approx **snow** + **multicore**

- Prefer to use only multicore components
 - Clever and efficient use of the Unix `fork`
- Aside: Distributed computing more transparent and scalable with SPMD/MPI via **pbdr** packages

The parallel Package: mclapply

A simple Fork-Join parallel programming paradigm

- + Data shared if not modified (“copy-on-write” by OS)
- + Very efficient.
- No Windows support.
- Not as efficient as threads (C, C++, FORTRAN).



The parallel Package: mclapply

```

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

..../scripts/multicore/mclapply.R

```

1 x <- lapply(1:1e3, function(x) median(rnorm(1e5)))
2
3 library(parallel)
4 detectCores()
5 x.mc2 <- mclapply(1:1e3, function(x) median(rnorm(1e5)), mc.cores = 2)
6 x.mc4 <- mclapply(1:1e3, function(x) median(rnorm(1e5)), mc.cores = 4)
  
```



Bootstrap k-means centers from iris data

..../scripts/multicore/3kmp-mclapply.R

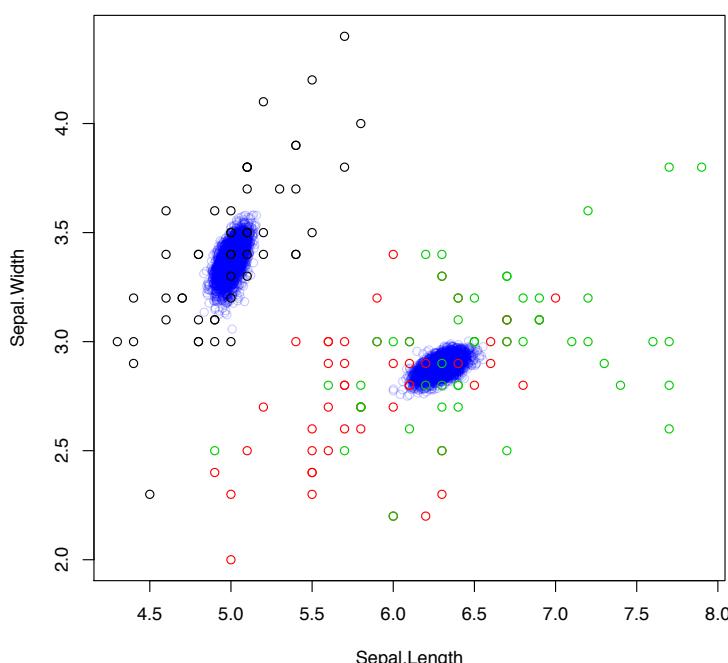
```

1 library(parallel, quietly = TRUE)
2 X <- as.matrix(iris[, 1:4])
3 k <- 3
4 RNGkind("L'Ecuyer-CMRG") # mc
5 set.seed(seed = 123)
6 FUN <- function(n) {
7   isample <- sample(nrow(X), nrow(X), replace = TRUE)
8   kmeans(X[isample, ], k, iter.max = 100, nstart = 100)$centers
9 }
10
11 means <- do.call(rbind, mclapply(1:5000, FUN, mc.cores = 2))
12
13 pdf("means-k3.pdf")
14 plot(X[, c(1, 2)], type = "n")
15 points(means[, c(1, 2)], col = rgb(0, 0, 1, 0.2))
16 points(X[, c(1, 2)], col = iris[, 5])
17 dev.off()

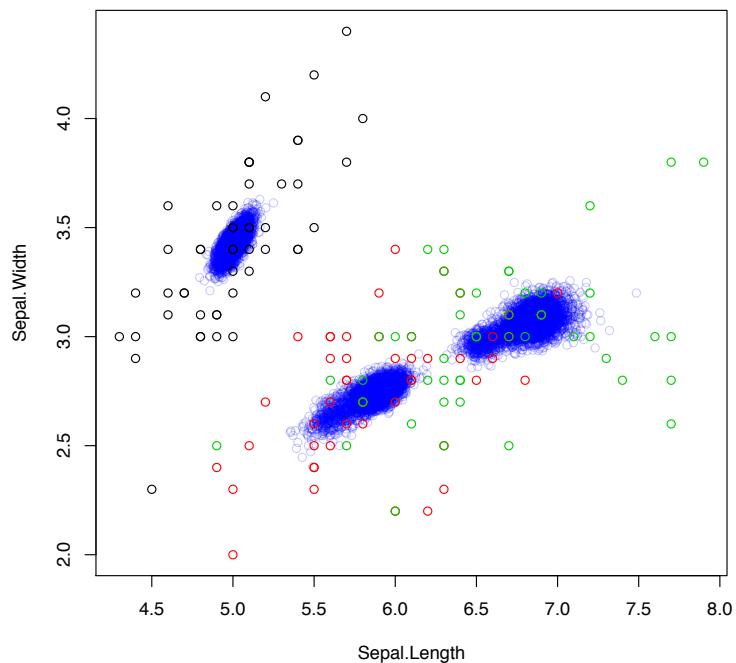
```



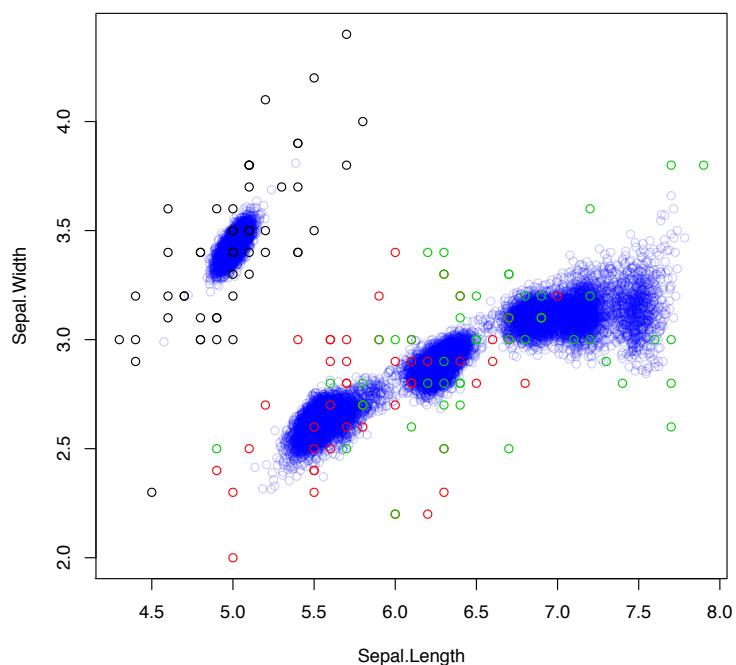
Bootstrap k-means centers from iris data: k=2



Bootstrap k-means centers from iris data: k=3



Bootstrap k-means centers from iris data: k=4



Machine Learning Example: Random Forest

Letter Recognition data from package **mlbench** (20,000 × 17)

1	[,1]	lettr	capital letter
2	[,2]	x.box	horizontal position of box
3	[,3]	y.box	vertical position of box
4	[,4]	width	width of box
5	[,5]	high	height of box
6	[,6]	onpix	total number of on pixels
7	[,7]	x.bar	mean x of on pixels in box
8	[,8]	y.bar	mean y of on pixels in box
9	[,9]	x2bar	mean x variance
10	[,10]	y2bar	mean y variance
11	[,11]	xybar	mean x y correlation
12	[,12]	x2ybr	mean of x^2 y
13	[,13]	xy2br	mean of x y^2
14	[,14]	x.ege	mean edge count left to right
15	[,15]	xegvy	correlation of x.ege with y
16	[,16]	y.ege	mean edge count bottom to top
17	[,17]	yegvx	correlation of y.ege with x

P. W. Frey and D. J. Slate (Machine Learning Vol 6/2 March 91): "Letter Recognition Using Holland-style Adaptive Classifiers".



Example: Random Forest Code

Build many simple models from subsets, use model averaging to predict

..../scripts/multicore/rf-serial.R

```

1 library(randomForest)
2 library(mlbench)
3 data(LetterRecognition)
4 set.seed(seed = 123)

5
6 n <- nrow(LetterRecognition)
7 n_test <- floor(0.2 * n)
8 i_test <- sample.int(n, n_test)
9 train <- LetterRecognition[-i_test,]
10 test <- LetterRecognition[i_test,]

11
12 rf.all <- randomForest(lettr ~ ., train, ntree = 500, norm.votes = FALSE)
13 pred <- predict(rf.all, test)
14 cat("Proportion Correct:", sum(pred == test$lettr)/n_test, "\n")

```



Example: Random Forest Code

Split learning by blocks of trees. Split prediction by blocks of rows.

.. /scripts/multicore/rf-mc.R

```

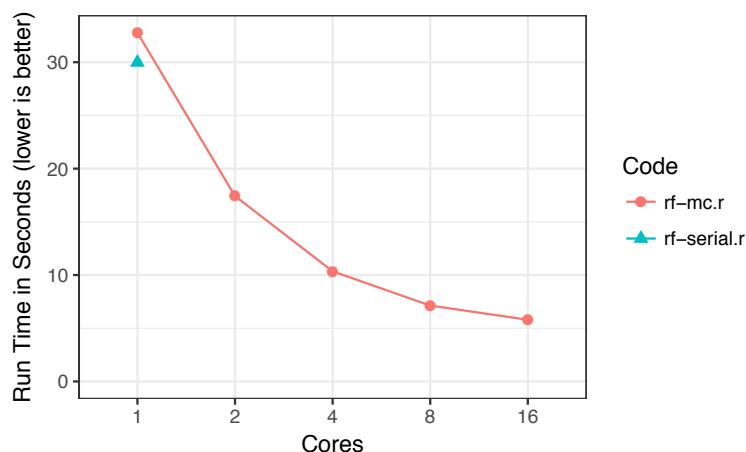
1 library(randomForest)
2 library(mlbench)
3 data(LetterRecognition)
4 library(parallel)
5 RNGkind("L'Ecuyer-CMRG") # mc
6 set.seed(seed = 123)
7
8 n <- nrow(LetterRecognition)
9 n_test <- floor(0.2 * n)
10 i_test <- sample.int(n, n_test)
11 train <- LetterRecognition[-i_test, ]
12 test <- LetterRecognition[i_test, ]
13
14 nc <- detectCores() # mc
15 ntree <- lapply(splitIndices(500, nc), length) # mc
16 rf <- function(x) randomForest(lettr~., train, ntree=x,
   norm.votes=FALSE) # mc
17 rf.out <- mclapply(ntree, rf, mc.cores = nc) # mc
18 rf.all <- do.call(combine, rf.out) # mc
19
20 crows <- splitIndices(nrow(test), nc) # mc
21 rfp <- function(x) as.vector(predict(rf.all, test[x, ])) # mc
22 cpred <- mclapply(crows, rfp, mc.cores = nc) # mc
23 pred <- do.call(c, cpred) # mc
24 cat("Proportion Correct:", sum(pred == test$lettr)/n_test, "\n")

```



Example: Random Forest Code

Scaling results on ORNL's Rhea[†]: 16 cores per node



[†] a 512-node commodity-type Linux cluster

Essentials of High Performance and Parallel Statistical Computing with R

5. Distributed Parallel Computing with pbdMPI

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

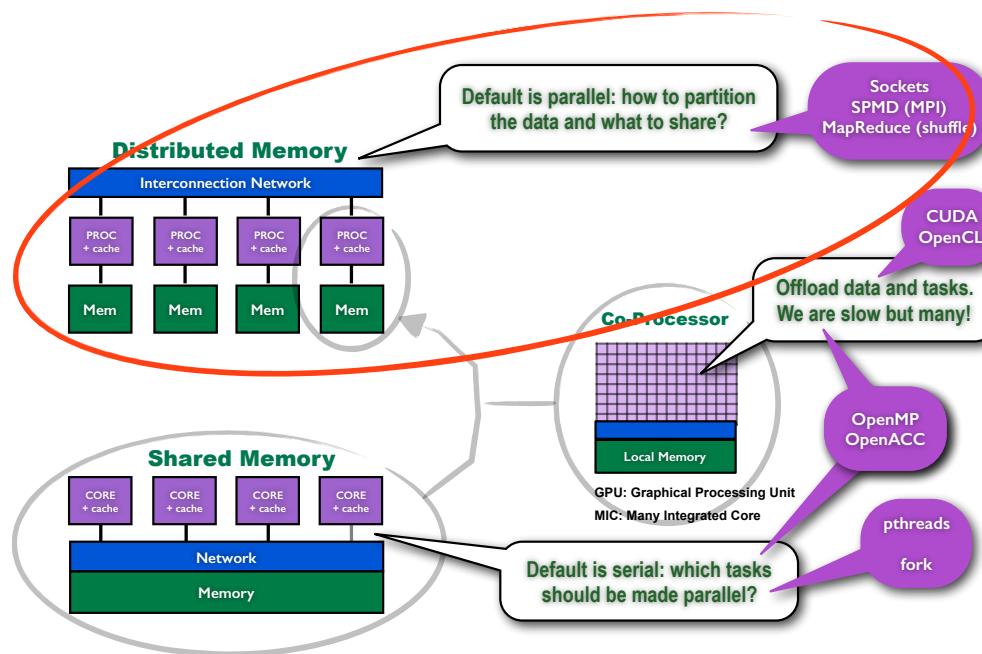
¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

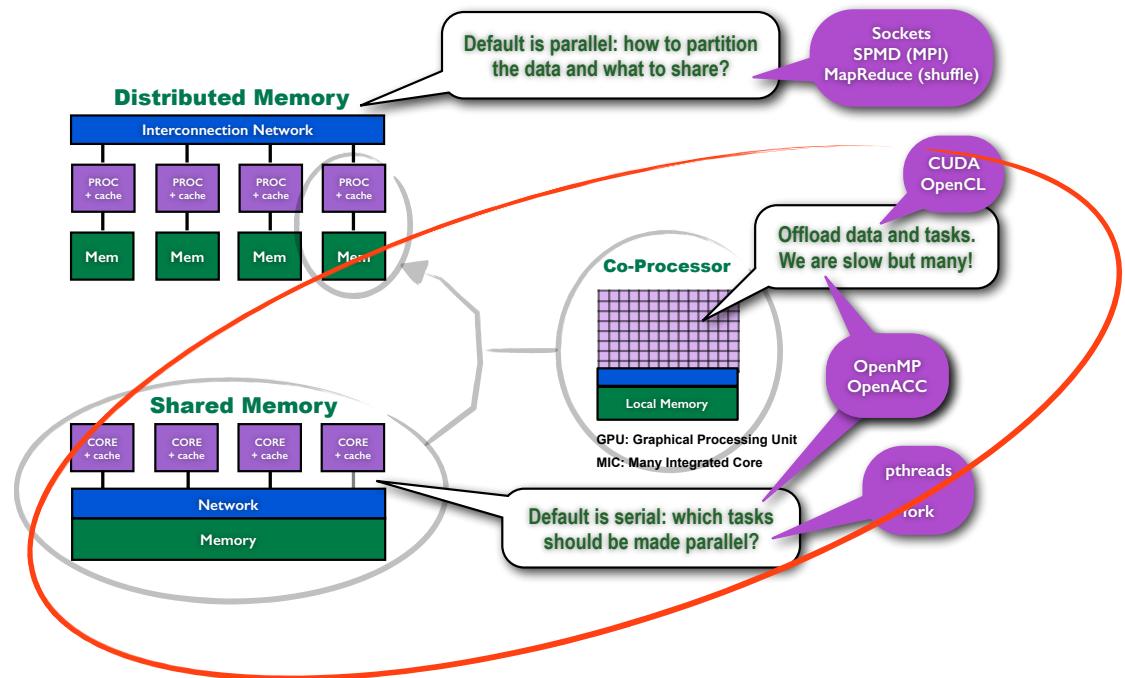
Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017



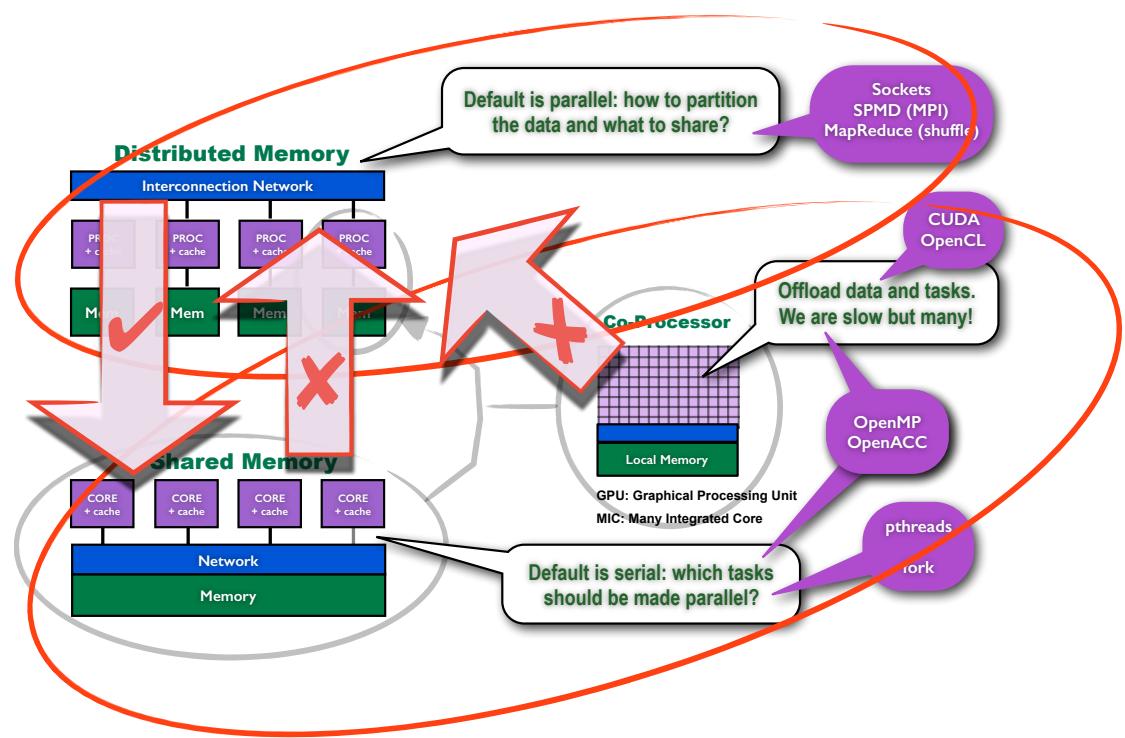
30+ Years of Parallel Computing Research



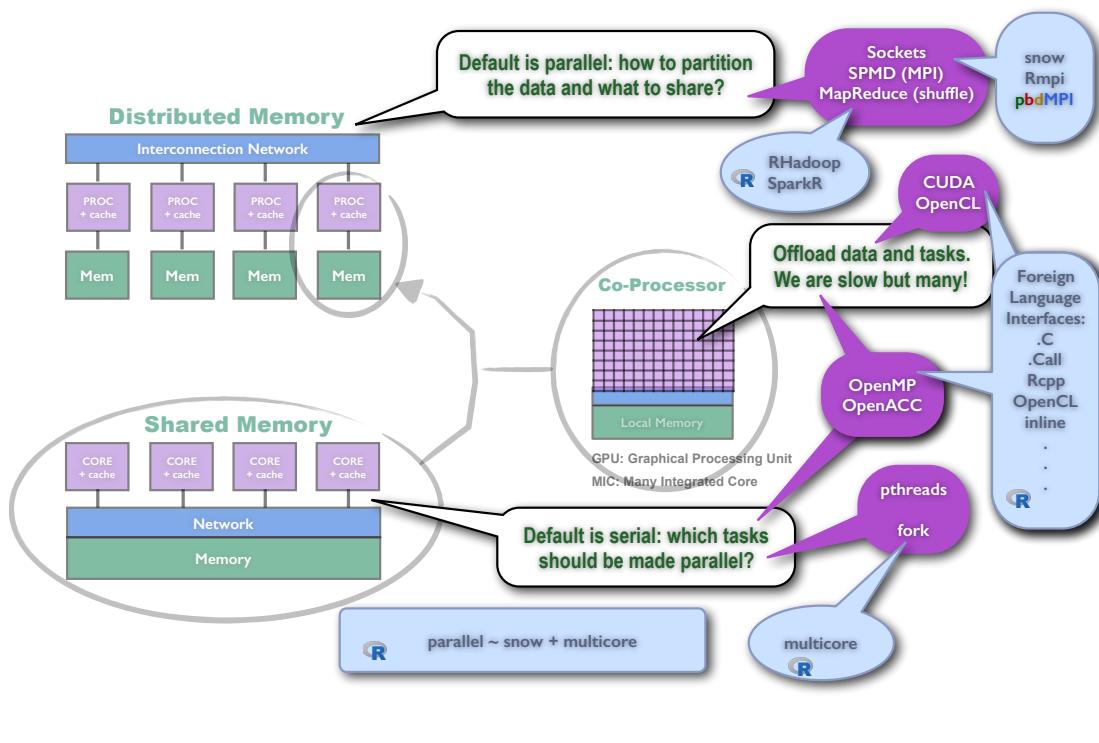
Last 10 years of Advances



Distributed Programming Works in Shared Memory



R Interfaces to Low-Level Native Tools



Distributed Parallel Computing with pbdR

- Why manager-workers is not scalable (ask Amdahl)
- pbdMPI - SPMD and managing your own communication
- Simple examples
- Parallel bootstrap
- Parallel crossvalidation
- Parallel random forest
- Tall skinny matrices



Manager-Workers to Cooperating Workers

Manager-Workers

- A serial program (Manager) divides up work and/or data
- Manager gets data and sends chunks to workers - **serial**
- Workers run without interaction - **parallel**
- Manager collects/combines results from workers - **serial**
- Divide-Recombine fits this model

Communicating Workers: SPMD

- Workers get their own data - **parallel**
- Workers can run complex SPMD codes - **parallel**
- Workers combine results - **parallel**
 - Associative and commutative reductions with recursive doubling



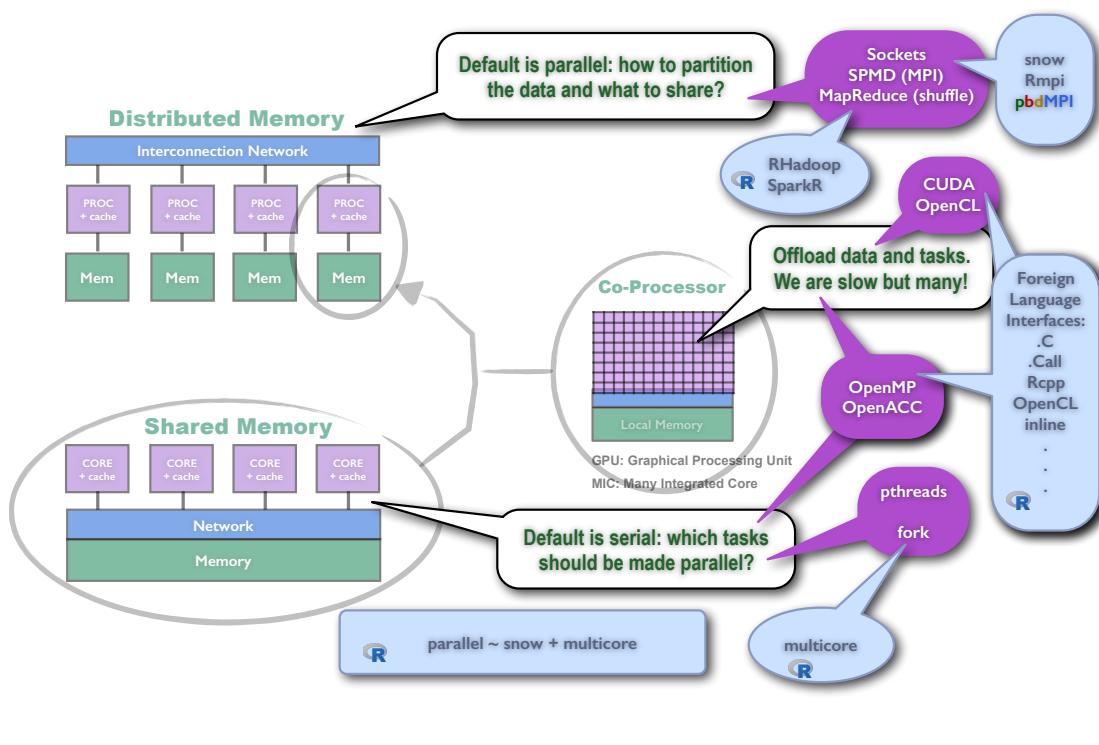
SPMD: Single Program Multiple Data

Write one general program so many copies of it can run asynchronously and cooperate (usually via MPI) to solve the problem.

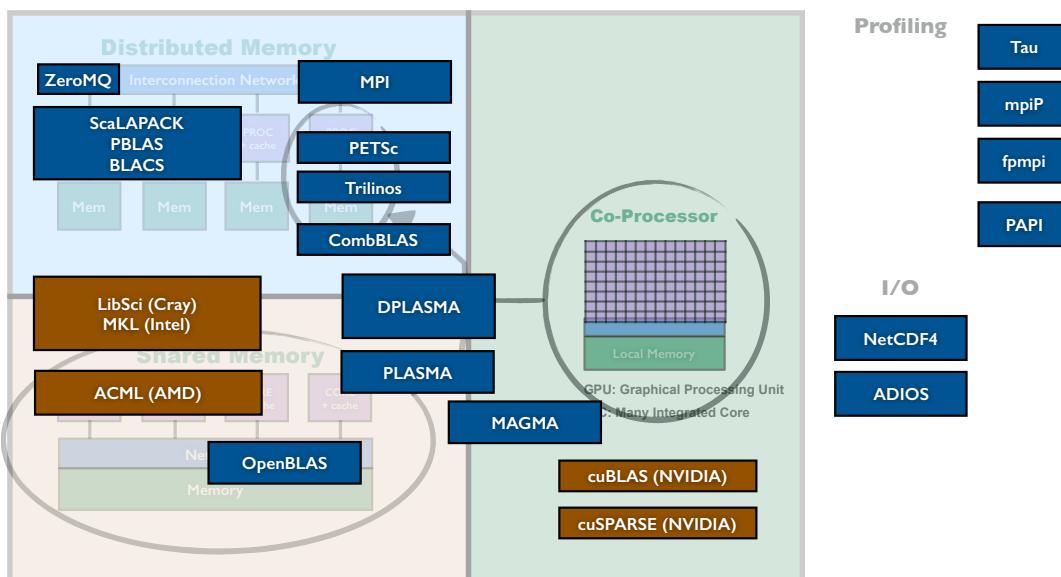
- The prevalent way of distributed programming in HPC
- Can handle tightly coupled parallel computations
- It is designed for batch computing
- There is usually no manager - rather, all cooperate
- Prime driver behind MPI specification
- Way to program server side in client-server



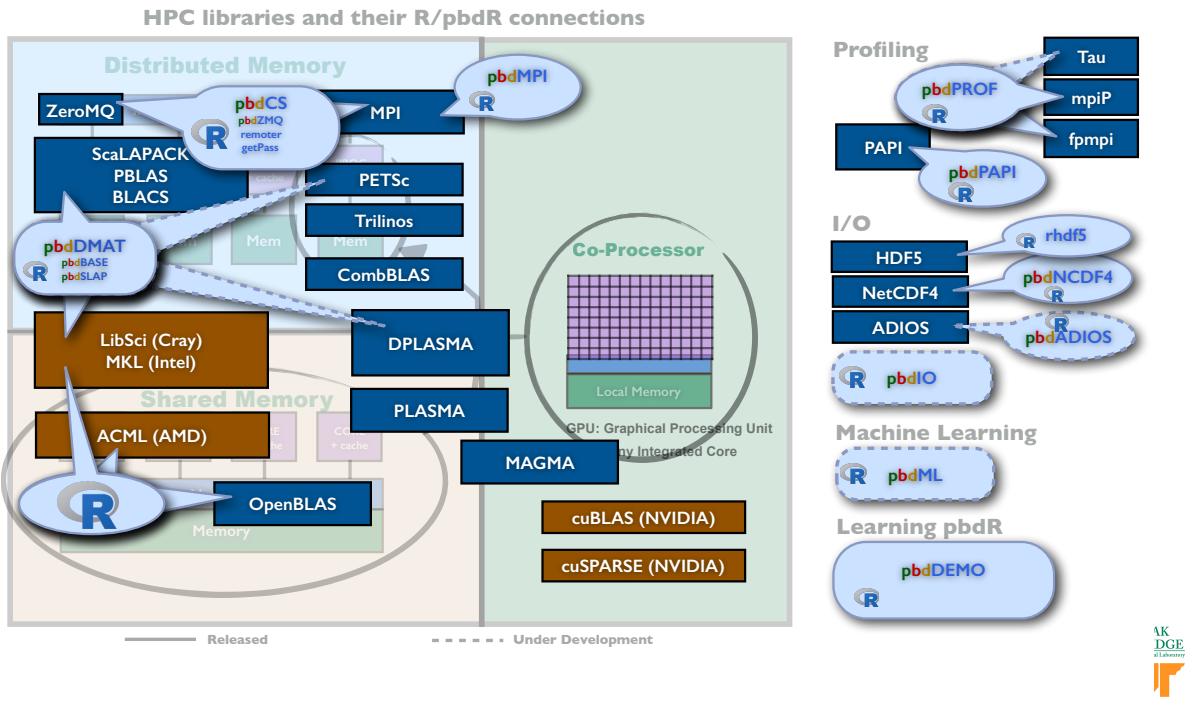
R Interfaces to Low-Level Native Tools



Scalable Libraries



R and **pbdR** Interfaces to HPC Libraries



pbdMPI: a High Level Interface to MPI

- Requires MPI installed (<https://www.open-mpi.org>)
- API is simplified: defaults in control objects
- Fast array and matrix methods without serialization
- S4 methods: extensible to complex R objects
- Additional error checking

pbdMPI (S4)	Rmpi
<code>allreduce</code>	<code>mpi.allreduce</code>
<code>allgather</code>	<code>mpi.allgather, mpi.allgatherv, mpi.allgather.Robj</code>
<code>bcast</code>	<code>mpi.bcast, mpi.bcast.Robj</code>
<code>gather</code>	<code>mpi.gather, mpi.gatherv, mpi.gather.Robj</code>
<code>recv</code>	<code>mpi.recv, mpi.recv.Robj</code>
<code>reduce</code>	<code>mpi.reduce</code>
<code>scatter</code>	<code>mpi.scatter, mpi.scatterv, mpi.scatter.Robj</code>
<code>send</code>	<code>mpi.send, mpi.send.Robj</code>



Hello World

..../scripts/pbdMPI/hello.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 text <- paste( "Hello, world from", comm.rank() )
4 print( text )
5
6 finalize()

```



Hello Print World

..../scripts/pbdMPI/hello-p.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 print( "Hello, world print" )
4
5 comm.print( "Hello, world comm.print" )
6
7 comm.print( "Hello from all", all.rank = TRUE, quiet = TRUE )
8
9 finalize()

```



Print My Rank

..../scripts/pbdMPI/rank.r

```

1 library( pbdMPI , quiet = TRUE )
2
3 my.rank <- comm.rank()
4 comm.print( my.rank , all.rank = TRUE )
5
6 finalize()

```



Random Gather and Reduce

..../scripts/pbdMPI/gt.r

```

1 library( pbdMPI , quiet = TRUE )
2
3 comm.set.seed( seed = 1234567 , diff = TRUE )
4
5 n <- sample( 1:10 , size = 1 )
6
7 gt <- gather( n )
8 comm.print( unlist( gt ) )
9
10 sm <- allreduce( n , op = 'sum' )
11 comm.print( sm , all.rank = TRUE )
12
13 finalize()

```



Map Reduce?

..../scripts/pbdMPI/map-reduce.r

```

1 library( pbdMPI , quiet = TRUE )
2
3 ## Your "Map" code
4 n <- comm.rank() + 1
5
6 ## Now "Reduce" but give the result to all
7 all_sum <- allreduce( n ) # Sum is default
8
9 text <- paste( "Hello: n is", n, "sum is", all_sum )
10 comm.print( text , all.rank = TRUE )
11
12 finalize ()

```



Broadcast

..../scripts/pbdMPI/bcast.r

```

1 library( pbdMPI , quiet = TRUE )
2
3 if ( comm.rank() == 0 ){
4   x <- matrix( 1:4 , nrow = 2 )
5 } else {
6   x <- NULL
7 }
8
9 y <- bcast( x )
10
11 comm.print( y , all.rank = TRUE )
12 comm.print( x , all.rank = TRUE )
13
14 finalize()

```



Rethinking system.time()

..../scripts/pbdMPI/timer.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 comm.set.seed( seed = 1234567, diff = T )
4
5 test <- function( timed )
6 {
7   ltime <- system.time( timed )[ 3 ]
8
9   mintime <- allreduce( ltime, op='min' )
10  maxtime <- allreduce( ltime, op='max' )
11  meantime <- allreduce( ltime, op='sum' ) / comm.size()
12
13  return( data.frame( min = mintime, mean = meantime, max = maxtime ) )
14 }
15
16 # generate 10,000,000 random normal values (total)
17 times <- test( rnorm( 1e7/comm.size() ) ) # ~76 MiB of data
18 comm.print( times )
19
20 finalize()

```



What's My Job?

..../scripts/pbdMPI/jid.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 k <- get.jid( 10 ) # note: pbdIO has comm.chunk()
4 my.rank <- comm.rank()
5 comm.cat( my.rank, ":", k, "\n", all.rank = TRUE, quiet = TRUE )
6
7 finalize()

```



Reduce a Matrix

..../scripts/pbdMPI/reduce-mat.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 x <- matrix( 10*comm.rank() + (1:6), nrow = 2 )
4
5 comm.print( x, all.rank = TRUE )
6
7 z <- allreduce( x ) # knows it's a matrix
8
9 comm.print( z, all.rank = TRUE )
10
11 finalize()

```



Computing Pi by Simulation

..../scripts/pbdMPI/mcsim.r

```

1 ##### Compute pi by simulation
2 library( pbdMPI, quiet = TRUE )
3
4 comm.set.seed( seed = 1234567, diff = TRUE )
5
6 my.N <- 1e7 %/% comm.size()
7 my.X <- matrix( runif( my.N * 2 ), ncol = 2 )
8 my.r <- sum( rowSums( my.X^2 ) <= 1 )
9 r <- allreduce( my.r )
10 PI <- 4*r / ( my.N * comm.size() )
11 comm.print( PI )
12
13 finalize()

```



Covariance the Hard Way

..../scripts/pbdMPI/cov.r

```

1 library( pbdMPI, quiet = TRUE )
2
3 comm.set.seed( seed = 1234567, diff = TRUE )
4
5 ## Generate 10 rows and 3 columns of data per process
6 my.X <- matrix( rnorm(10*3), ncol = 3 )
7
8 ## Compute mean
9 N <- allreduce( nrow( my.X ), op = "sum" )
10 mu <- allreduce( colSums( my.X ) / N, op = "sum" )
11
12 ## Sweep out mean and compute crossproducts sum
13 my.X <- sweep( my.X, STATS = mu, MARGIN = 2 )
14 Cov.X <- allreduce( crossprod( my.X ), op = "sum" ) / ( N - 1 )
15
16 comm.print( Cov.X )
17
18 finalize()

```



Least Squares Fit via Normal Equations

..../scripts/pbdMPI/ols.r

```

1 ##### Least Squares Fit via Normal Equations (see lm.fit for a better way)
2 library( pbdMPI, quiet = TRUE )
3
4 comm.set.seed( seed = 12345, diff = TRUE )
5
6 ## 10 rows and 3 columns of data per process
7 my.X <- matrix( rnorm(10*3), ncol = 3 )
8 my.y <- matrix( rnorm(10*1), ncol = 1 )
9
10 ## Form the Normal Equations components
11 my.Xt <- t( my.X )
12 Xtx <- allreduce( my.Xt %*% my.X, op = "sum" )
13 Xty <- allreduce( my.Xt %*% my.y, op = "sum" )
14
15 ## Everyone solve the Normal Equations
16 ols <- solve( Xtx, Xty )
17
18 comm.print( ols )
19
20 finalize()

```



Bootstrapping k-means on Iris data

..../scripts/pbdMPI/bootstrap-kmp.r

```

1 library( pbdMPI, quietly = TRUE )
2 X <- as.matrix( iris[, 1:4] )
3 k <- 3
4 comm.set.seed( seed = 123, diff = TRUE )
5 FUN <- function( n ) {
6   isample <- sample( nrow(X), nrow(X), replace = TRUE )
7   kmeans( X[ isample, ], k, iter.max = 100, nstart = 100 )$centers
8 }
9
10 my.means <- do.call( rbind, lapply( get.jid( 5000 ), FUN ) )
11 means <- gather( my.means )
12
13 if( comm.rank() == 0 ){
14   means <- do.call( rbind, means )
15   pdf( "means.pdf" )
16   plot( X[, c(1, 2)], type = "n" )
17   points( means[, c(1, 2)], col = rgb( 0, 0, 1, 0.2 ) )
18   points( X[, c(1, 2)], col = iris[, 5] )
19   dev.off()
20 }
21 finalize()

```



Bootstrapping k-means on Iris data + parallel plots

..../scripts/pbdMPI/bootstrap-kmpp.r

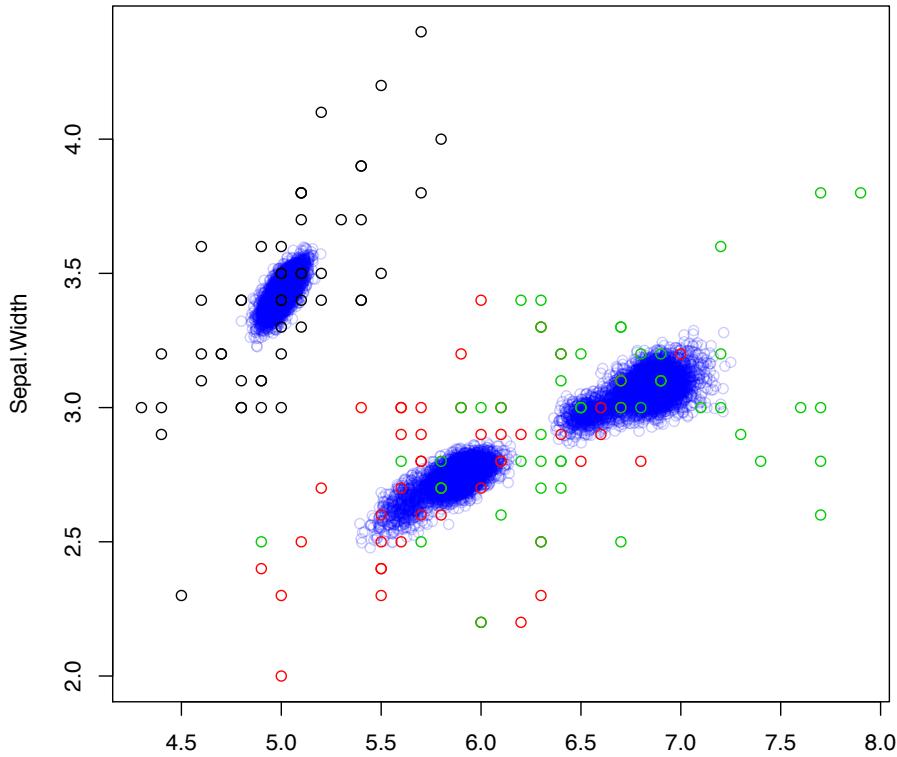
```

1 ##### file "bootstrap_kmpp.r"
2 library(pbdMPI, quietly = TRUE)
3 X <- as.matrix(iris[, 1:4])
4 k <- 3
5 comm.set.seed(seed = 123, diff = TRUE)
6 FUN <- function(n) {
7   isample <- sample(nrow(X), nrow(X), replace = TRUE)
8   kmeans(X[isample, ], k, iter.max = 100, nstart = 100)$centers
9 }
10 my.samples <- get.jid(5000)
11 my.means <- do.call(rbind, lapply(my.samples, FUN))
12 means <- do.call(rbind, allgather(my.means))
13
14 plots <- combn(ncol(X), 2)
15 my.plots <- plots[, get.jid(ncol(plots)), drop = FALSE]
16 plotFUN <- function(cols) {
17   pdf(paste(k, "means", paste(cols, collapse = "-"), ".pdf", sep = ""))
18   plot(X[, cols], type = "n")
19   points(means[, cols], col = rgb(0, 0, 1, 0.2))
20   points(X[, cols], col = iris[, 5])
21   dev.off()
22 }
23 bits <- apply(my.plots, 2, plotFUN)
24 finalize()

```



Bootstrapping k-means on Iris data



Crossvalidation: Sombrero Function Data Setup

`..../scripts/pbdMPI/crossvalidate-setup.r`

```

1 suppressMessages(library( locfit , quietly = TRUE))
2 # set.seed( 12345 )
3
4 f <- function( x ) {                                     # Sombrero function
5   rho <- sqrt( x[ 1 ]^2 + x[ 2 ]^2 )
6   2 * besselJ( pi * rho, 1 ) / ( pi * rho )
7 }
8 x.seq <- y.seq <- seq( -pi, pi, length.out = 100 )
9 z <- matrix( NA, 100, 100 )                           # Evaluate f on 100x100
10 grid
11 for ( i in 1:length( x.seq ) ) {
12   for ( j in 1:length( y.seq ) ) {
13     z[ i, j ] <- f( c( x.seq[ i ], y.seq[ j ] ) )
14   }
15 }
16 # image(z, col = rainbow(128))
17
18 sigma.true <- 0.1                                     # Generate observations with error
19 n <- 1000
20 x1 <- runif( n, -pi, pi )
21 x2 <- runif( n, -pi, pi )
22 y <- numeric( n )
23 f.true <- numeric( n )
24 for ( i in 1:n ) {
25   y[ i ] <- f( c( x1[ i ], x2[ i ] ) ) + rnorm( 1, 0, sigma.true )
26   f.true[ i ] <- f( c( x1[ i ], x2[ i ] ) )
27 }
28 dat <- data.frame( x = x1 - x2, y = y - f.true )

```



Crossvalidation: Nearest (alpha) Neighbor Fitting

..../scripts/pbdMPI/crossvalidate.r

```

1 set.seed(12345)
2 source( "crossvalidate-setup.r" )
3
4 K <- 5
5 fold.grp <- ( 1:n - 1 ) %% K + 1
6 folds <- split( 1:n, f = fold.grp )
7 y.hat.cv <- numeric( n )
8 alpha <- seq( 0.02, 0.3, 0.001 )
9
10 SSPE.cv <- numeric( length( alpha ) )
11
12 for ( j in 1:length( alpha ) ) {
13   for ( k in 1:K ) {
14     idx <- folds[[ k ]]
15     fit.fold.out <- locfit( y ~ lp( x1, x2, nn = alpha[ j ] ), data
16                             = dat[ -idx, ] )
17     y.hat.cv[ idx ] <- predict( fit.fold.out, newdata = dat[ idx, ] )
18   }
19   SSPE.cv[ j ] <- sum( ( y - y.hat.cv )^2 )
20 }
21 plot( alpha, SSPE.cv )

```



Crossvalidation: Parallel Nearest (alpha) Neighbor Fitting

..../scripts/pbdMPI/crossvalidate.pbdR.r

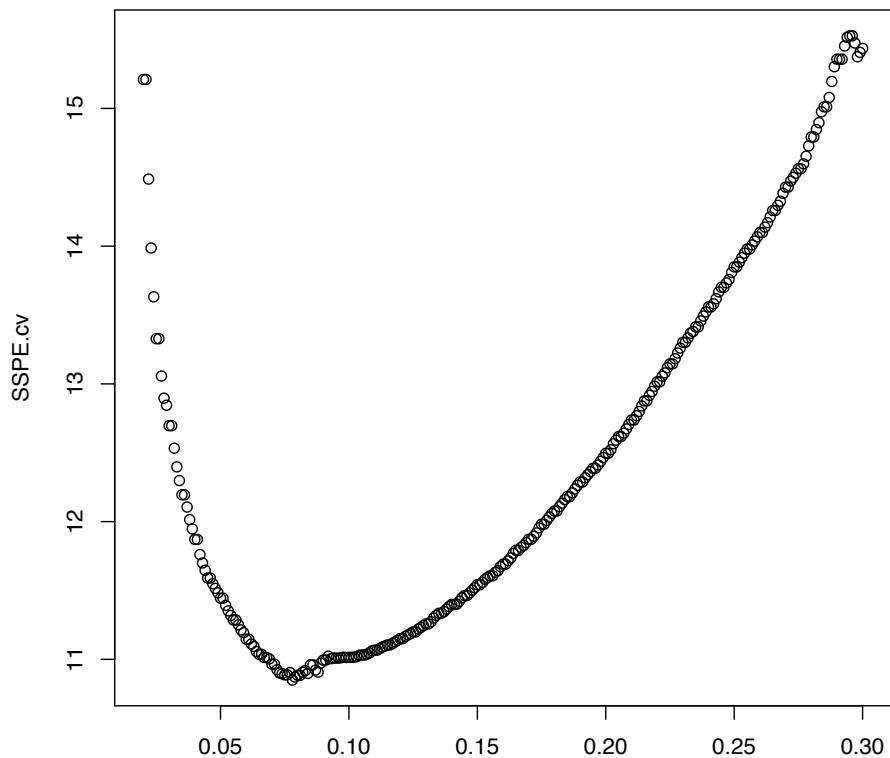
```

1 library(pbdMPI, quietly = TRUE)
2 comm.set.seed(12345, diff = FALSE)
3 source( "crossvalidate-setup.r" )
4
5 K <- 5
6 fold.grp <- ( 1:n - 1 ) %% K + 1
7 folds <- split( 1:n, f = fold.grp )
8 y.hat.cv <- numeric( n )
9 alpha <- seq( 0.02, 0.3, 0.001 )
10
11 my.alpha.i <- get.jid( length( alpha ) ) # pbdR
12 SSPE.cv <- numeric( length( my.alpha.i ) ) # pbdR
13 for ( j in my.alpha.i ) { # pbdR
14   for ( k in 1:K ) {
15     idx <- folds[[ k ]]
16     fit.fold.out <- locfit( y ~ lp( x1, x2, nn = alpha[ j ] ), data
17                             = dat[ -idx, ] )
18     y.hat.cv[ idx ] <- predict( fit.fold.out, newdata = dat[ idx, ] )
19   }
20   SSPE.cv[ j - my.alpha.i[1] + 1 ] <- sum( ( y - y.hat.cv )^2 ) # pbdR
21 }
22 SSPE.cv <- do.call( c, allgather( SSPE.cv ) ) # pbdR
23
24 if( comm.rank() == 0 ) { # pbdR
25   plot( alpha, SSPE.cv )
26 } # pbdR

```



Crossvalidation: alpha SSPE curve



Machine Learning Example: Random Forest

Letter Recognition data from package **mlbench** ($20,000 \times 17$)

A A A A A A A A	lettr	capital letter
B B B B B B B B	x.box	horizontal position of box
C C C c C c c C	y.box	vertical position of box
G F G F F F F F	width	width of box
K K K K K K K K	high	height of box
S S S S S S S S	onpix	total number of on pixels
X x X X X X X X	x.bar	mean x of on pixels in box
	y.bar	mean y of on pixels in box
	x2bar	mean x variance
	y2bar	mean y variance
	xybar	mean x y correlation
	x2ybr	mean of x^2 y
	xy2br	mean of x y^2
	x.ege	mean edge count left to right
	xegvy	correlation of x.ege with y
	y.ege	mean edge count bottom to top
	yegvx	correlation of y.ege with x

P. W. Frey and D. J. Slate (Machine Learning Vol 6/2 March 91): "Letter Recognition Using Holland-style Adaptive Classifiers".



Example: Random Forest Code

Build many simple models from subsets, use model averaging to predict

Serial Code rf-serial.r

```

1 library(randomForest)
2 library(mlbench)
3 data(LetterRecognition) # 26 Capital Letters Data 20,000 x 17
4 set.seed(seed=123)
5 n <- nrow(LetterRecognition)
6 n_test <- floor(0.2*n)
7 i_test <- sample.int(n, n_test) # Use 1/5 of the data to test
8 train <- LetterRecognition[-i_test, ]
9 test <- LetterRecognition[i_test, ]
10
11 ## train random forest
12 rf.all <- randomForest(lettr ~ ., train, ntree=500, norm.votes=FALSE)
13
14 ## predict test data
15 pred <- predict(rf.all, test)
16 correct <- sum(pred == test$lettr)
17 cat("Proportion Correct:", correct/n_test, "\n")

```



Example: Random Forest Code

Split learning by blocks of trees. Split prediction by blocks of rows.

Parallel Code rf-mpi.r

```

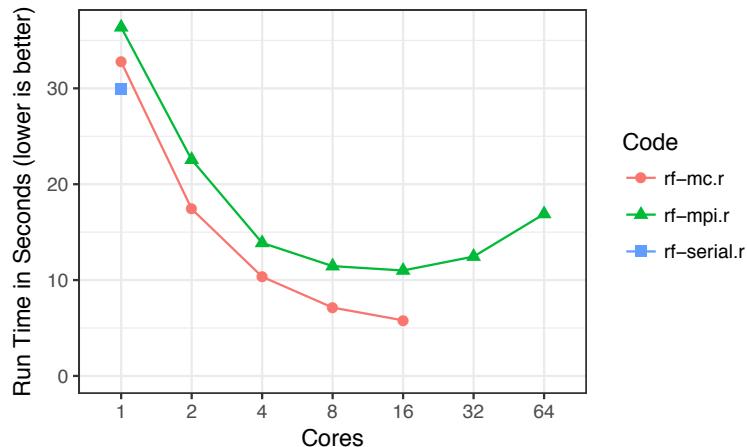
1 library(randomForest)
2 library(mlbench)
3 data(LetterRecognition)
4 comm.set.seed(seed=123, diff=FALSE) # same training data
5 n <- nrow(LetterRecognition)
6 n_test <- floor(0.2*n)
7 i_test <- sample.int(n, n_test) # Use 1/5 of the data to test
8 train <- LetterRecognition[-i_test, ]
9 test <- LetterRecognition[i_test, ][get.jid(n_test), ]
10
11 comm.set.seed(seed=1e6*runif(1), diff=TRUE)
12 my.rf <- randomForest(lettr ~ ., train, ntree=500//comm.size(),
13                         norm.votes=FALSE)
13 rf.all <- do.call(combine, allgather(my.rf))
14
15 pred <- predict(rf.all, test)
16 correct <- allreduce(sum(pred == test$lettr))
17 comm.cat("Proportion Correct:", correct/n_test, "\n")

```



Example: Random Forest Code

Scaling results on ORNL's Rhea[†]: 16 cores per node



- LetterRecognition data set too small for more than 16 cores
- Ideally, a combination of MPI and fork would be used
 - One MPI process per node
 - One data set copy per node
 - All cores busy with fork

[†] a 512-node commodity-type Linux cluster



Recall profiling min.basis() in fda.usc package

```

1 > summaryRprof()
2 $by.total
3                               total.time total.pct self.time self.pct
4 "min.basis"                12.32    100.00     0.00     0.00
5 "type.CV"                  6.54     53.08     0.02     0.16
6 "S.basis"                  5.76     46.75     0.00     0.00
7 "drop"                     4.20     34.09     0.00     0.00
8 "norm.fdata"                4.20     34.09     0.00     0.00
9 "metric"                   4.18     33.93     1.04     8.44
10 "%*%"                      3.98     32.31     3.98    32.31
11 "getbasispenalty"          2.72     22.08     0.02     0.16
12 "bsplinepen"                2.68     21.75     0.36     2.92
13 "int.simpson2"              2.54     20.62     1.96    15.91
14 "t"                           2.10     17.05     0.10     0.81
15 "ppBspline"                 1.60     12.99     0.82     6.66
16 . . .

```



min.basis() 110 lines

SPMD: Add 5, change 3

```

1 min.basis <- function(fdataobj, type.CV = GCV.S, . . ., . . .)
2 {
3   . . . 13 lines
4   library(pbddMPI)
5   init()
6   my.k <- get.jid(lenlambda)
7   my.gcv <- array(Inf, dim = c(lenbasis, length(my.k)))
8   . . . 36 lines
9   for (i in 1:lenbasis) {
10     . . . 3 lines
11     for (k in my.k) {
12       S2 <- S.basis(tt, base, lambda[k])
13       my.gcv[i, k - my.k[1] + 1] <-
14         type.CV(fdataobj, S = S2, W = W, trim = par.CV$trim,
15           draw = par.CV$draw, . . .)
16     }
17   }
18   gcv <- do.call(cbind, allgather(my.gcv))
19   finalize()
. . . 48 lines

```



Running parallel min.basis()

Beacon cluster <http://nics.utk.edu/beacon> (16 cores/node)

```

1 -bash-4.1$ time Rscript fda-s.r
2 real 0m28.906s    user 0m28.587s    sys 0m0.138s
3
4 -bash-4.1$ time Rscript fda-pbdR.r
5 real 0m31.090s    user 0m30.654s    sys 0m0.163s
6
7 -bash-4.1$ time mpirun -np 1 Rscript fda-pbdR.r
8 real 0m31.435s    user 0m30.611s    sys 0m0.205s
9
10 -bash-4.1$ time mpirun -np 2 Rscript fda-pbdR.r
11 real 0m18.398s    user 0m35.346s    sys 0m0.444s
12
13 -bash-4.1$ time mpirun -np 4 Rscript fda-pbdR.r
14 real 0m10.590s    user 0m39.252s    sys 0m1.023s
15
16 -bash-4.1$ time mpirun -np 8 Rscript fda-pbdR.r
17 real 0m7.203s     user 0m52.165s    sys 0m1.843s
18
19 -bash-4.1$ time mpirun -np 16 Rscript fda-pbdR.r
20 real 0m5.800s     user 1m18.676s    sys 0m5.876s
21
22 -bash-4.1$ time mpirun -np 32 Rscript fda-pbdR.r
23 real 0m4.965s     user 1m3.870s    sys 0m6.571s

```



kazaam package: “shaq” class for tall and skinny matrices

..../scripts/pbdMPI/shaq.r

```

1 suppressMessages( library( kazaam ) )
2 source("shaq_rhdf5_section.r")
3
4 ## create shaq class matrix from local pieces
5 B <- shaq( A, rowsA, ncol(A), checks = FALSE )
6
7 ## svd ( shaq class: tall-skinny matrix )
8 Res <- svd( B, nu = 0, nv = 0 )
9
10 barrier( )
11
12 comm.cat( "results top( 16 ):", Res$d[ 1:16 ], "\n", quiet = TRUE )
13
14 ## what's in shaq? (repeat "eigen only" without shaq class)
15 Ac <- allreduce( crossprod( A ) )
16 Res2 <- eigen( Ac, only.values = TRUE )
17 comm.cat( "results2 top( 16 ):", sqrt( Res2$values[ 1:16 ] ), "\n",
18 quiet = TRUE )
19
20 finalize( )

```



kazaam for tall and skinny matrices

..../scripts/pbdMPI/shaq_rhdf5_section.r

```

1 suppressMessages( library( rhdf5 ) )
2 suppressMessages( library( pbdIO ) )
3
4 filename <- "big_hdf5_data.h5"
5 dataset <- "/dataset"
6
7 ## get and broadcast the number of rows to all processors
8 if ( comm.rank() == 0 ) {
9   h5f <- H5Fopen( filename )
10  h5d <- H5Dopen( h5f, dataset )
11  h5s <- H5Dget_space( h5d )
12  rowsA <- H5Sget_simple_extent_dims( h5s )$size[1]
13
14  H5Dclose( h5d )
15  H5Fclose( h5f )
16 } else {
17   rowsA <- 0
18 }
19 rowsA <- bcast( rowsA )
20
21 ## get my local row indices
22 my_rows <- comm.chunk( rowsA, form = "vector", type = "balance" )
23
24 ## parallel read of local data
25 A <- h5read(filename, dataset, index=list( my_rows, NULL ) )
26 H5close( )

```



Essentials of High Performance and Parallel Statistical Computing with R

6. Distributed Parallel Computing with pbdDMAT

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017

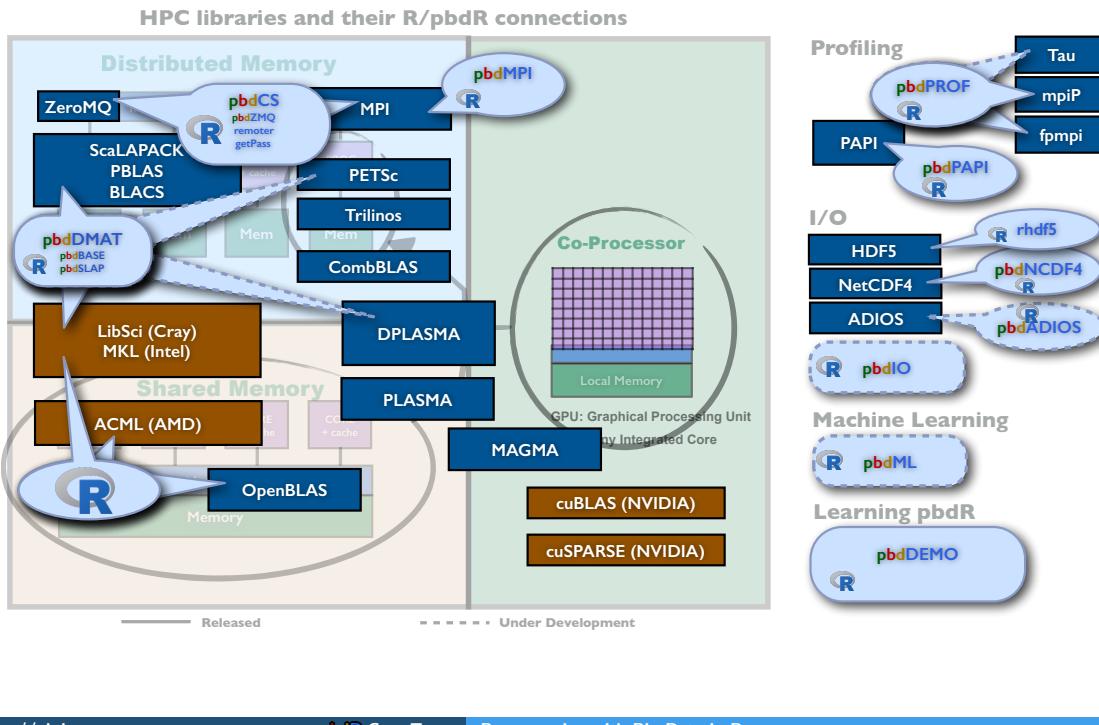


Outline

- pbdDMAT: Distributed matrix computation
- pbdML: Distributed machine learning



R and **pbdR** Interfaces to HPC Libraries



Distributed Matrix Computations

A matrix is mapped to a processor grid shape

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

(a) 1×6
(ICTXT=1)

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

(b) 2×3
(User defined)

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

(c) 3×2
(ICTXT=0)

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

(d) 6×1
(ICTXT=2)

Table: Processor Grid Shapes with 6 Processors

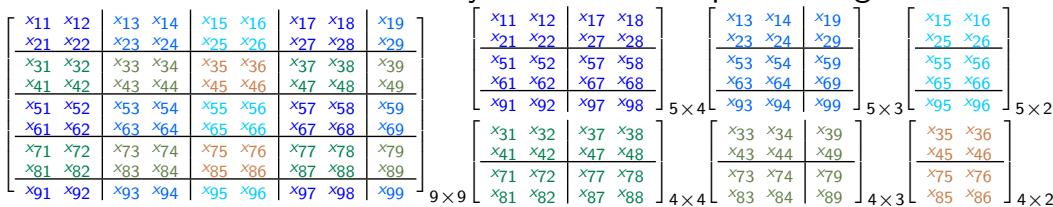
Distributed Matrix Operations

Powered by ScaLAPACK, PBLAS, and BLACS (MKL, SciLIB)

pbdDMAT

- Block-cyclic data layout for scalability and efficiency
- No change in R syntax
- High-level convenience for data layout redistributions
 - Row-major data: read row-block then convert to block-cyclic
 - Column-major data: read column-block then convert to block-cyclic

Global and local views of block-cyclic on a 2×3 processor grid



pbdR No change in syntax

New data redistribution functions

```

1 x <- x[ -1, 2:5 ]
2 x <- log( abs( x ) + 1 )
3 x.pca <- prcomp( x )
4 xtx <- t( x ) %*% x
5 ans <- svd( solve( xtx ) )

```

The above (and over 100 other functions) runs on 1 core with R
or 10,000 cores with **pbdR** ddmatrix class

```

1 > showClass( "ddmatrix" )
2 Class "ddmatrix" [package "pbdDMAT"]
3 Slots:
4   Name:      Data      dim      ldim     bldim    ICTXT
5   Class:    matrix numeric numeric numeric numeric

```

```

1 > x <- as.rowblock( x )
2 > x <- as.blockcyclic( x )
3 > x <- redistribute( x, bldim=c(8, 8), ICTXT = 0 )

```



Generate a Matrix and Distribute

..../scripts/pbdDMAT/gen-dist.r

```

1 suppressPackageStartupMessages( library( pbdDMAT ) )
2 init.grid()
3
4 ## Construct on 0 and convert to distributed (communicates)
5 if ( comm.rank() == 0 ){
6   x <- matrix( 1:100, nrow = 10, ncol = 10 )
7 } else {
8   x <- NULL
9 }
10 x.dmat <- as.ddmatrix( x, bldim = c( 2, 2 ) )
11
12 x.dmat
13 comm.print( submatrix( x.dmat ), all.rank = TRUE )
14
15 ## More conveniently (no communication)
16 y.dmat <- ddmatrix( 1:100, nrow = 10, ncol = 10, bldim = c( 2, 2 ) )
17 comm.print( submatrix( y.dmat ), all.rank = TRUE )
18
19 all.equal( x.dmat, y.dmat )
20
21 ## Some ddmatrix defaults
22 comm.cat(".pbd_env$BLDIM =", .pbd_env$BLDIM, "\n")
23 comm.cat(".pbd_env$ICTXT =", .pbd_env$ICTXT, "\n")
24
25 finalize()

```



Generate Identity or Zero ddmatrix

..../scripts/pbdDMAT/gen-01.r

```

1 library( pbdDMAT, quiet = TRUE )
2 init.grid()
3
4 zero.dmat <- ddmatrix( 0, nrow = 10, ncol = 10 )
5 zero.dmat
6
7 id.dmat <- diag( 1, nrow = 10, ncol = 10, type = "ddmatrix" )
8 id.dmat
9
10 comm.print( submatrix( id.dmat ), all.rank = TRUE )
11
12 diag.dmat <- diag( 1:5, nrow = 10, ncol = 10, type = "ddmatrix" )
13 diag.dmat
14
15 comm.print( submatrix( diag.dmat ), all.rank = TRUE )
16
17 comm.set.seed( seed = 1234567, diff = TRUE )
18 rand.dmat <- ddmatrix( "rnorm", nrow = 100, ncol = 100, mean = 10, sd =
19   100 )
20 rand.dmat
21
22 finalize()

```



Distributed Construction of a ddmatrix

..../scripts/pbdDMAT/ddmatrix.r

```

1 suppressPackageStartupMessages( library( pbdDMAT ) )
2 init.grid()
3
4 ## construct local piece of a row-block distributed matrix
5 x <- matrix( 100*comm.rank() + 1:(3*5), ncol = 5 )
6 comm.print( x, all.rank = TRUE )
7
8 ## get correct row-block dimensions and glue into ddmatrix
9 gdim <- c( allreduce( nrow( x ) ), ncol( x ) )
10 bd <- c( allreduce( nrow( x ), op = "max" ), ncol( x ) )
11 xnew <- new( "ddmatrix", Data = x, dim = gdim, bldim = bd, ldim = dim( x )
12   ), ICTXT = 2 )
12 print( xnew )
13 comm.print( submatrix( xnew ), all.rank = TRUE )
14
15 xt.cb <- as.colblock( xnew )
16 print( xt.cb )
17 comm.print( submatrix( xt.cb ), all.rank = TRUE )
18
19 xt.bc <- as.blockcyclic( xnew, bldim = c( 2, 2 ) )
20 print( xt.bc )
21 comm.print( submatrix( xt.bc ), all.rank = TRUE )
22
23 xt <- as.rowblock( t( xnew ) )
24 print( xt )
25 comm.print( submatrix( xt ), all.rank = TRUE )
26 finalize()

```



Extract Rows and Columns of a ddmatrix

..../scripts/pbdDMAT/extract.r

```

1 library( pbdDMAT, quiet = TRUE )
2 init.grid()
3
4 .BLDIM <- c( 1, 1 )
5
6 x.dmat <- ddmatrix( 1:30, nrow = 10 )
7 y.dmat <- x.dmat[ c( 1, 3, 5, 7, 9 ), -3 ]
8
9 comm.print( submatrix( y.dmat ), all.rank = TRUE )
10
11 comm.print( y.dmat )
12 y <- as.matrix( y.dmat )
13 comm.print( y )
14
15 finalize()

```



apply() functions on a ddmatrix

..../scripts/pbdDMAT/apply.r

```

1 suppressPackageStartupMessages( library( pbdDMAT ) )
2 init.grid()
3
4 x.dmat <- ddmatrix( "rnorm", nrow = 10, ncol = 3 )
5 comm.print( x.dmat )
6
7 colsd <- apply( X = x.dmat, FUN = sd, MARGIN = 2 )
8 colsd.loc <- as.matrix( colsd )
9
10 comm.print( colsd )
11 comm.print( colsd.loc )
12
13 finalize()

```

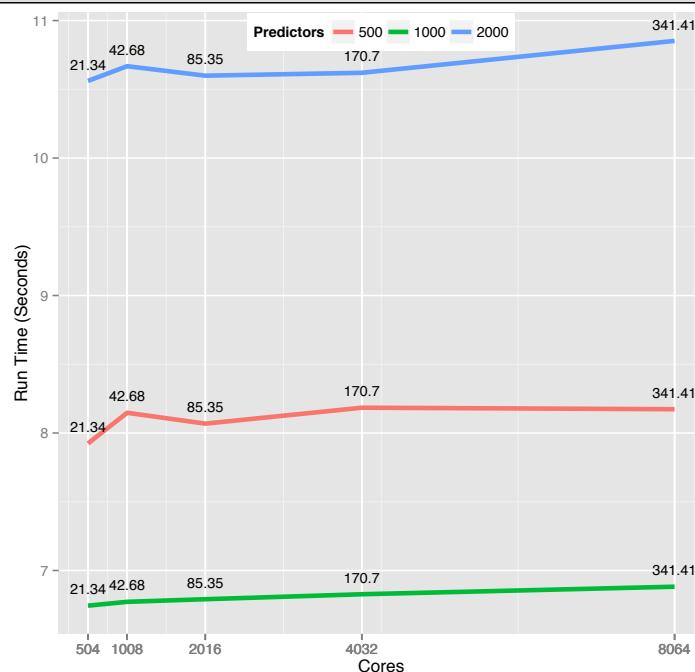


Covariance

```

1 x <- ddmatrix("rnorm", nrow=n, ncol=p, mean=mean, sd=sd)
2 cov.x <- cov(x)

```

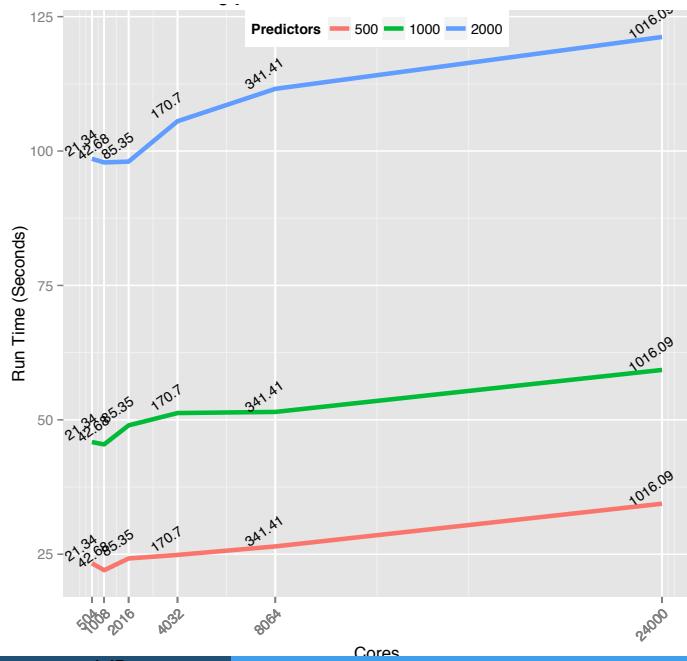


Regression

```

1 beta.true <- ddmatrix("runif", nrow=p, ncol=1)
2 y <- x %*% beta.true
3 beta.est <- lm.fit(x, y)$coefficients

```



Singular Value Decomposition

Principal Components Analysis

Given an $n \times p$ matrix X , there exists a decomposition

$$X = UDV^T,$$

where U and V are orthonormal and D is diagonal.

Principal components analysis (PCA) usually centers and possibly scales columns of X before applying SVD.

Truncated SVD uses first k columns from U and V , U_k and V_k , respectively, and the first k values from D_k . This gives an approximation

$$X \approx U_k D_k V_k^T.$$

The proportion of variability in X explained by the k principal components is

$$\frac{\sum_{i=1}^k d_i^2}{\sum_{i=1}^m d_i^2}, \text{ where } m = \min(n, p).$$



prcomp() function

..../scripts/pbdDMAT/pca.r

```

1 suppressPackageStartupMessages( library( pbdDMAT ) )
2 init.grid()
3
4 n <- 1e4
5 p <- 250
6
7 comm.set.seed( seed = 1234567, diff = TRUE )
8 x.dmat <- ddmatrix( "rnorm", nrow = n, ncol = p, mean = 100, sd = 25 )
9
10 x.dmat
11
12 pca <- prcomp( x = x.dmat, retx = TRUE, scale = TRUE )
13
14 ## Truncate at 90%
15 prop_var <- cumsum( pca$sdev ) / sum( pca$sdev )
16 i <- max( min( which( prop_var > 0.9 ) ) - 1, 1 )
17
18 y.dmat <- pca$x[ , 1:i ]
19
20 comm.cat( "\nCols: ", i, "\n", quiet = TRUE )
21 comm.cat( "%Cols:", i / dim( x.dmat )[ 2 ], "\n\n", quiet = TRUE )
22
23 finalize()

```



Truncated SVD from random projections¹

PROTOTYPE FOR RANDOMIZED SVD
Given an $m \times n$ matrix A , a target number k of singular vectors, and an exponent q (say, $q = 1$ or $q = 2$), this procedure computes an approximate rank- $2k$ factorization $U\Sigma V^$, where U and V are orthonormal, and Σ is nonnegative and diagonal.*

Stage A:

- 1 Generate an $n \times 2k$ Gaussian test matrix Ω .
- 2 Form $Y = (AA^*)^q\Omega$ by multiplying alternately with A and A^* .
- 3 Construct a matrix Q whose columns form an orthonormal basis for the range of Y .

Stage B:

- 4 Form $B = Q^*A$.
- 5 Compute an SVD of the small matrix: $B = \tilde{U}\Sigma V^*$.
- 6 Set $U = Q\tilde{U}$.

Note: The computation of Y in step 2 is vulnerable to round-off errors. When high accuracy is required, we must incorporate an orthonormalization step between each application of A and A^* ; see Algorithm 4.4.

ALGORITHM 4.4: RANDOMIZED SUBSPACE ITERATION
Given an $m \times n$ matrix A and integers ℓ and q , this algorithm computes an $m \times \ell$ orthonormal matrix Q whose range approximates the range of A .

- 1 Draw an $n \times \ell$ standard Gaussian matrix Ω .
- 2 Form $Y_0 = A\Omega$ and compute its QR factorization $Y_0 = Q_0R_0$.
- 3 for $j = 1, 2, \dots, q$
 - 4 Form $\tilde{Y}_j = A^*Q_{j-1}$ and compute its QR factorization $\tilde{Y}_j = \tilde{Q}_j\tilde{R}_j$.
 - 5 Form $Y_j = A\tilde{Q}_j$ and compute its QR factorization $Y_j = Q_jR_j$.
 - 6 end
 - 7 $Q = Q_q$.

Serial R

```

1 randSVD <- function(A, k, q=3)
{
  ## Stage A
  Omega <- matrix(rnorm(n*2*k),
                   nrow=n, ncol=2*k)
  Y <- A %*% Omega
  Q <- qr.Q(qr(Y))
  At <- t(A)
  for(i in 1:q)
  {
    Y <- At %*% Q
    Q <- qr.Q(qr(Y))
    Y <- A %*% Q
    Q <- qr.Q(qr(Y))
  }
  ## Stage B
  B <- t(Q) %*% A
  U <- La.svd(B)$u
  U <- Q %*% U
  U[, 1:k]
}

```

¹Halko, Martinsson, and Tropp. 2011. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions *SIAM Review* **53** 217–288

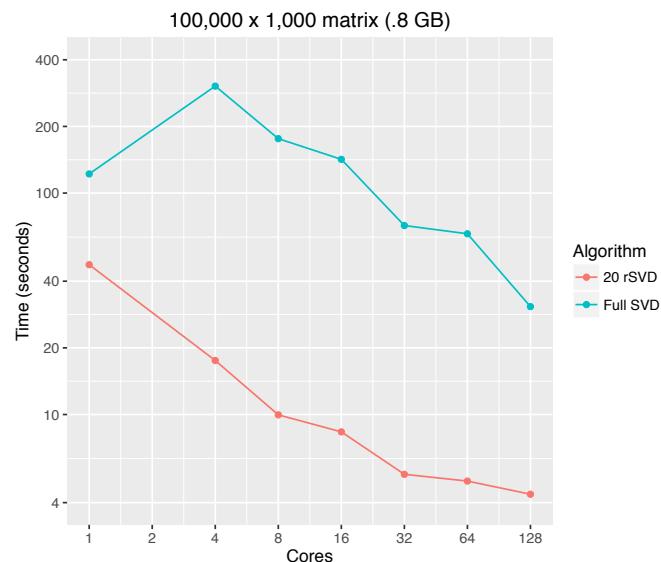


From serial pseudocode to parallel scalable code in one day!

Parallel pbdR

```

1 randSVD <- function(A, k, q=3)
2 {
3   ## Stage A
4   Omega <- ddmatrix("rnorm", nrow=n, ncol=2*k)
5   Y <- A %*% Omega
6   Q <- qr.Q(qr(Y))
7   At <- t(A)
8   for(i in 1:q)
9   {
10     Y <- At %*% Q
11     Q <- qr.Q(qr(Y))
12     Y <- A %*% Q
13     Q <- qr.Q(qr(Y))
14   }
15
16   ## Stage B
17   B <- t(Q) %*% A
18   U <- La.svd(B)$u
19   U <- Q %*% U
20   U[, 1:k]
21 }
```



Big Benchmark (134 GB) on Big Clusters

rPCA centers, optionally scales, and uses rsvd

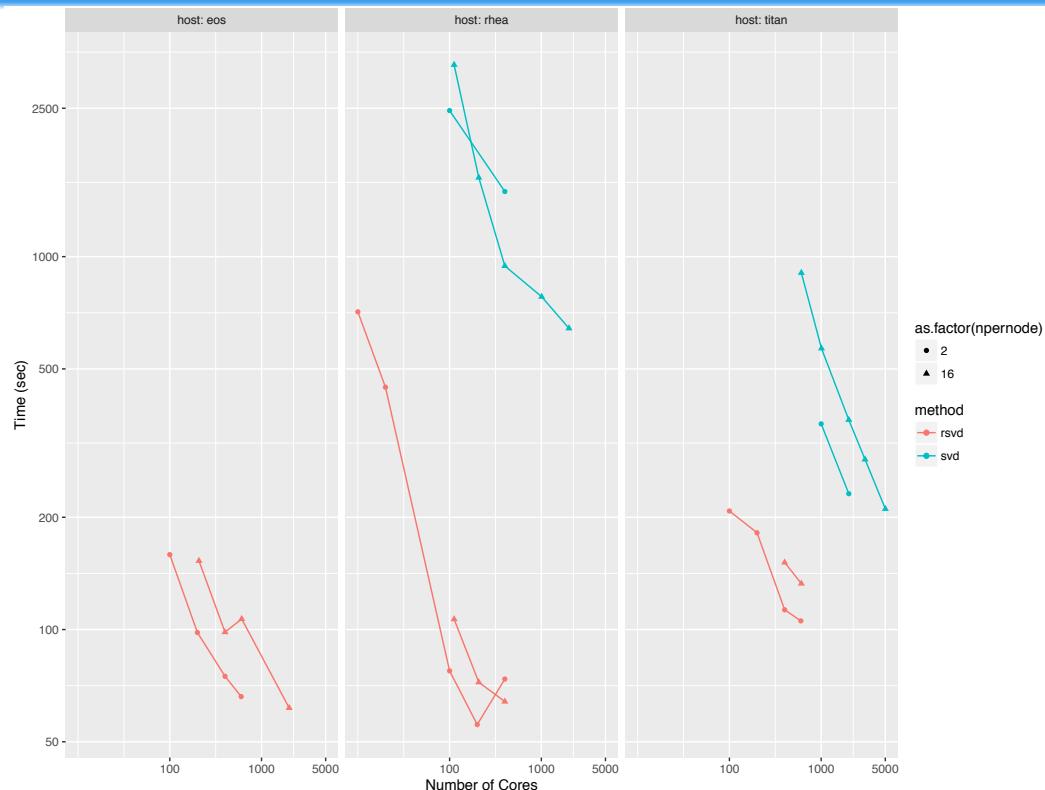
fastpca.R

```

1 suppressPackageStartupMessages(library(rhdf5))
2 suppressPackageStartupMessages(library(pbdIO))
3 suppressPackageStartupMessages(library(pbdML))
4 init.grid()
5
6 var <- "your_hdf5_full_variable_name"
7 h5f <- H5Fopen( "your_hdf5_file_name" )
8 h5d <- H5Dopen( h5f, var )
9 h5s <- H5Dget_space( h5d )
10 dims <- H5Sget_simple_extent_dims( h5s )$size
11 rows <- dims[2] # row-major to column-major
12 cols <- dims[1] # because data written by C/Py
13
14 ## read C/Py-written blocks of rows into R blocks of columns
15 my_rows <- comm.chunk(rows, form="vector", type="equal")
16 A <- t(h5read(h5f, var, index=list(NULL, my_rows)))
17
18 ## add glue to make a global column-block ddmatrix
19 A <- new("ddmatrix", Data=A, dim=c(rows, cols), ldim=dim(A), bldim=dim(A), ICTXT=2)
20
21 ## rearrange into block-cyclic
22 A <- as.blockcyclic( A )
23
24 ## get 32 top singular values and vectors
25 Res <- rPCA( A, k = 32 )
26
27 ## print the singular values
28 comm.print( Res$d )
29
30 finalize()
```



Big Benchmark (134 GB) on Big Clusters



Big Benchmark (134 GB) on HPC Wire



Since 1986 - Covering the Fastest Computers in the World and the People Who Run Them

Click the Logos below for our

[Subscribe to receive news](#)

[Home](#)
[News](#)
[Technologies](#)
[Sectors](#)
[Exascale](#)
[Resources](#)
[Specials](#)
[Events](#)

July 6, 2016

OLCF Researchers Scale R to Tackle Big Science Data Sets

John Russell



Sometimes lost in the discussion around big data is the fact that big science has long generated huge data sets. In fact, large-scale simulations that run on leadership-class supercomputers work at such high speeds and resolution that they generate unprecedented amounts of data. The size of these datasets—ranging from a few gigabytes to hundreds of terabytes—makes managing and analyzing the resulting

pbdR computes in under a minute what takes hours on Spark MLlib.



Essentials of High Performance and Parallel Statistical Computing with R

7. Going Interactive with pbdR Client Server

Instructors: George Ostrouchov¹ and Wei-Chen Chen²

¹ Oak Ridge National Laboratory and University of Tennessee

² U.S. Food and Drug Administration

Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017



Outline

- Interactive parallel computing
- One local client and one remote serial server
- One local client and remote parallel servers
- Other applications

See demo at <https://github.com/snoweye/user2016.demo>



Interactive parallel computing

What should be considered?

- Front end interface
- Security
- Back end framework
- Graphics and visualization
- Performance trade off
- Remote procedure calls



One local client and one remote serial server

Why do we need this?

- Ans: Just for fun

pbdrZMQ + remoter

- Communication other than MPI
- Where data and code should stay? Local or remote?
- What should be in local?
- Data transfer

Reference: <https://snoweye.github.io/pbdr/tutorial5.html>



One local client and remote parallel servers

Why do we need this?

- Ans: Distributed Read, Compute, Statistics, Output, and Inference, ...

pbdrZMQ + remoter + pbdrCS

- ZMQ over MPI or MPI over ZMQ?
- How about ZMQ with MPI?
- Where computation should be executed?

Reference: <https://snoweye.github.io/pbdr/tutorial6.html>



Other applications

Do we really need others?

- Go supercomputing
- Go web
- Go mobile



Course materials prepared by

pbdR Core Team:

Wei-Chen Chen, US FDA
George Ostrouchov, ORNL & UTK
Drew Schmidt, ORNL

with assistance from:

Michael Matheson, ORNL
Andrew Raim, US Census Bureau



Support

This work used resources of the [Oak Ridge Leadership Computing Facility](#) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work also used resources of the [National Institute for Computational Sciences](#) at the University of Tennessee, Knoxville, which is supported by the Office of Cyberinfrastructure of the U.S. National Science Foundation. This material is based upon work supported by the National Science Foundation Division of Mathematical Sciences under Grant No. 1418195.

