Essentials of High Performance and Parallel Statistical Computing with R

1. Introduction, Resources, and Resource Managers

Instructors: George Ostrouchov[1] and Wei-Chen Chen[2]

[1] Oak Ridge National Laboratory and University of Tennessee
[2] U.S. Food and Drug Administration

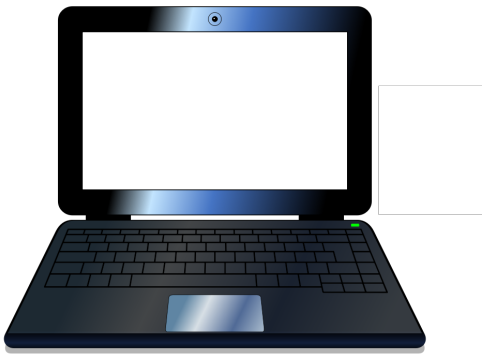
Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017

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

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 +
  - 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! (taxpayer funded).
  - Resources are "leadership class": nothing faster available to you on earth.
- 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

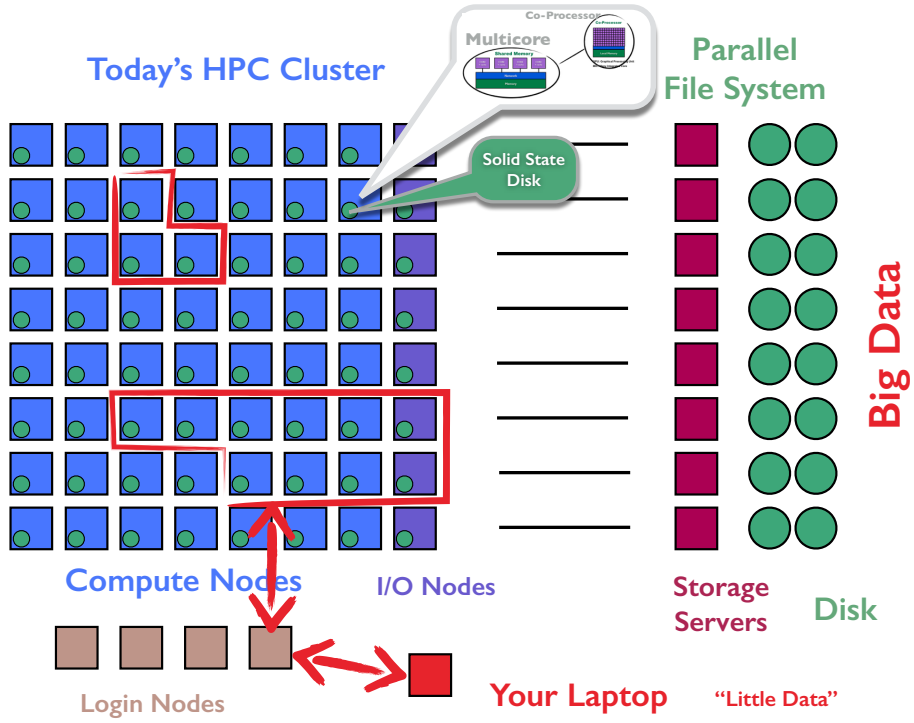
amazon web services™ | EC2

- 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

code/script.pbs

```
#!/bin/sh
#PBS -A your_account
#PBS -l nodes=4:ppn=16,walltime=00:30:00
#PBS -q batch
#PBS -o test.log
#PBS -e test.err

module load PE-gnu
module load r

cd your_working_directory

mpirun -np 64 Rscript test.R
```

Batch use

```
$ qsub script.pbs
```

PBS qsub for Batch and Interactive Use

Interactive batch use

```
$ qsub -I -X -A your_account -l walltime=0:30:00 -l nodes=4
qsub: waiting for job 233324.rhea-mgmt2g.ccs.ornl.gov to start
qsub: job 233324.rhea-mgmt2g.ccs.ornl.gov ready

$ module load PE-gnu
$ module load r
$ cd your_working_directory
$ mpirun -np 64 Rscript test.R
```

Essentials of High Performance and Parallel Statistical Computing with R

2. Speeding up Your R Code

Instructors: George Ostrouchov[1] and Wei-Chen Chen[2]

[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 distriuted)
- If memory bound, consider distributed parallel solutions

Profiling — Why Profile?

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

Performance Profiling Tools: `system.time()`

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

code/system.time.R

```
1  x <- matrix(rnorm(20000*750), nrow=20000, ncol=750)
2
3  system.time(t(x) %*% x)
4  #    user  system elapsed
5  #   2.187   0.032   2.324
6
7  system.time(crossprod(x))
8  #    user  system elapsed
9  #   1.009   0.003   1.019
10
11 system.time(cov(x))
12 #    user  system elapsed
13 #   6.264   0.026   6.338
```

Performance Profiling Tools: Rprof()

Samples call stack (default every 0.02 seconds)

code/profile.R

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

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

$sample.interval
[1] 0.02

$sampling.time
[1] 0.98
```

Performance Profiling Tools: Rprof()

code/profile.R

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

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

$sample.interval
[1] 0.99

$sampling.time
[1] 0
```

Performance Profiling Tools: rbenchmark

rbenchmark is a simple package that easily benchmarks different functions:

code/benchmark.R

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

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

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

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

Performance Profiling Tools: pbdPAPI

PAPI_cache_access.R

```
library( pbdPAPI )
library( inline )
```

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

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

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

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

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

Performance Profiling Tools: pbdPAPI

code/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

code/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

Exerpt from Original function

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

Exerpt 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**$\times$!

OpenBLAS (multithreaded Basic Linar Algebra Subroutines)

- Install: http://www.openblas.net/
- Batch thread control: OPENBLAS_NUM_THREADS=n
- Dynamic thread control: "wrathematics/openblasctl" micropackage on GitHub

code/openblas.R

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

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

library( openblasctl )

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

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

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

- Same is available with Intel's MKL

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

```
#include <Rcpp.h>
using namespace Rcpp;

//[[Rcpp::export]]

NumericVector row_max( NumericMatrix m )
{
   int nrows = m.nrow( );
   NumericVector maxPerRow( nrows );

   for ( int i = 0; i < nrows; i++ )
   {
      maxPerRow[ i ] = Rcpp::max( m( i, _ ) );
   }

   return ( maxPerRow );
}
```

One can get configuration values by

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

Rcpp Example (con'd)

A simple row max calculation

cat ex_max.R

```
library( Rcpp )
Sys.setenv( "PKG_CXXFLAGS" =
 "-I /sw/redhat6/r/3.3.0/rhel6_gnu4.8.2/lib64/R/library/Rcpp/include" )
Sys.setenv( "PKG_LIBS"="-lm" )

sourceCpp( "ex_max.cpp" )

set.seed( 27 )
X <- matrix( rnorm( 4 * 4 ), 4, 4 )
X

print( "Rcpp" )
row_max( X )
```

Rscript ex_max.R

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

```
#include <RcppArmadillo.h>
//[[Rcpp::depends(RcppArmadillo)]]
//[[Rcpp::export]]

arma::vec getEigenValues( arma::mat M )
{
    return ( arma::eig_sym( M ) );
}
```

RcppArmadillo Example (con'd)

cat eigen.R

```
library( Rcpp )
library( RcppArmadillo )
Sys.setenv( "PKG_CXXFLAGS" = "-I
    /sw/redhat6/r/3.3.0/rhel6_gnu4.8.2/lib64/R/library/RcppArmadillo/include"
    )
Sys.setenv( "PKG_LIBS"="-lm"
)
sourceCpp( "eigen.cpp" )

set.seed( 27 )
X <- matrix( rnorm( 4 * 4 ), 4, 4 )
Z <- X %*% t( X )
print( "RcppArmadillo" )
getEigenValues( Z )

print( "R" )
eigen( Z )$values
```

Rscript eigen.R

```
[1] "RcppArmadillo"
            [,1]
[1,]  0.03779289
[2,]  0.85043786
[3,]  2.03877658
[4,] 17.80747601
[1] "R"
[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 Ostrouchov[1] and Wei-Chen Chen[2]

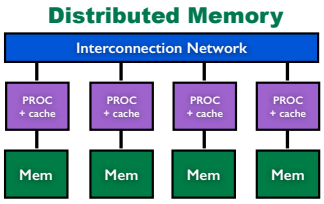
[1] Oak Ridge National Laboratory and University of Tennessee
[2] U.S. Food and Drug Administration

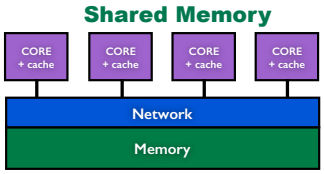
Professional Development Continuing Education Course
Joint Statistical Meetings, Baltimore, August 1, 2017

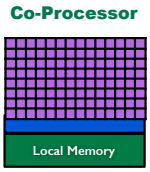
Outline

- 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

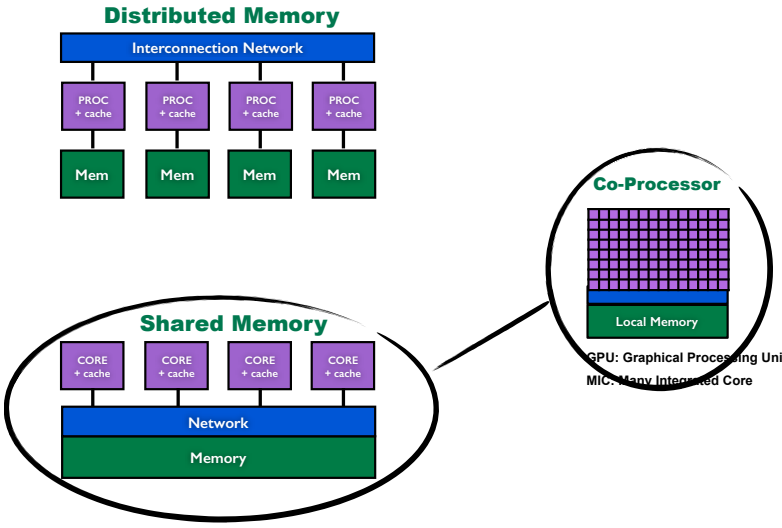




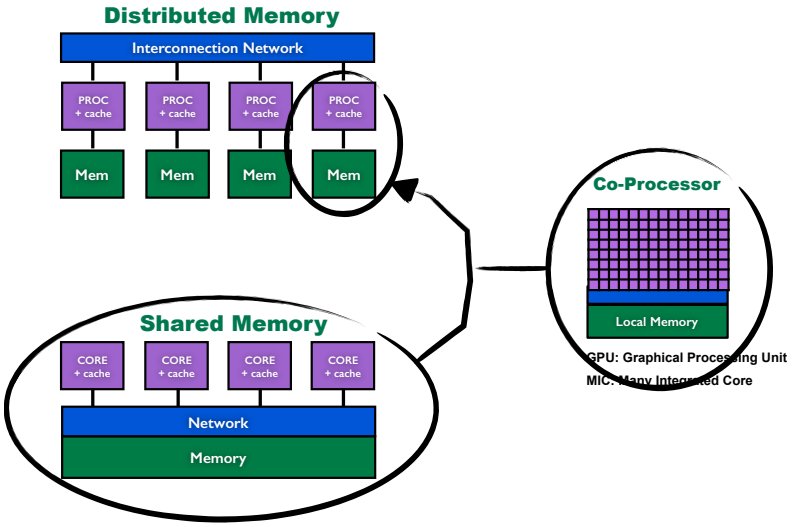


GPU: Graphical Processing Unit
MIC: Many Integrated Core

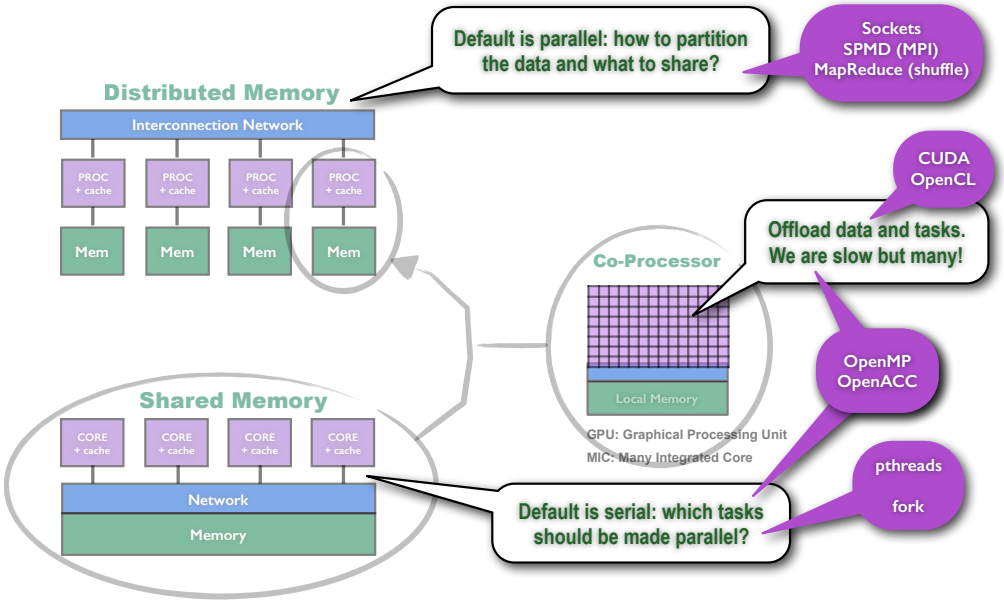
Your Laptop or Desktop



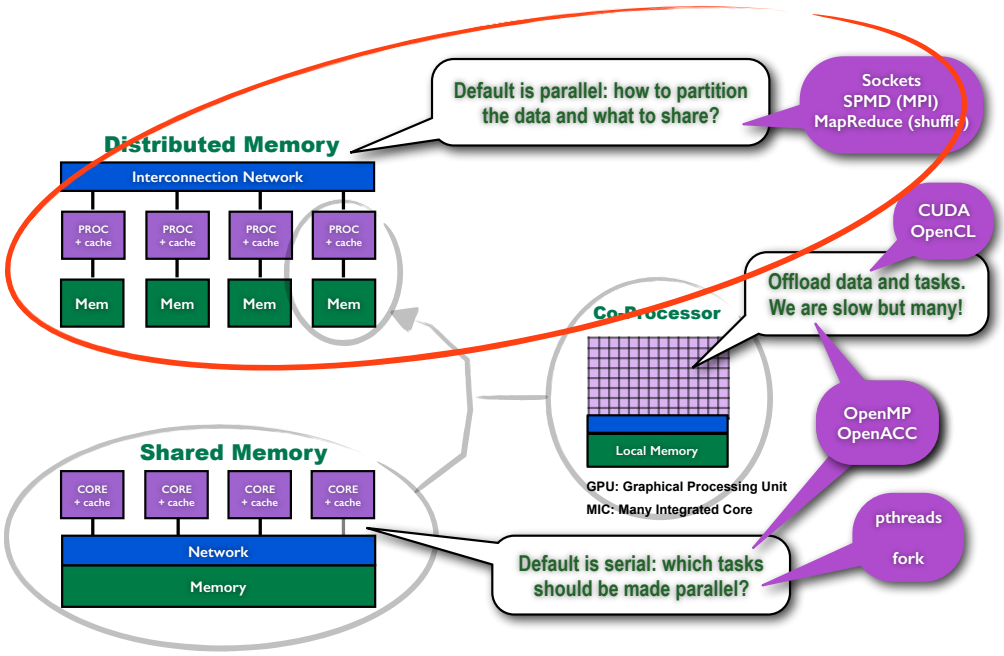
Server to Cluster to Supercomputer



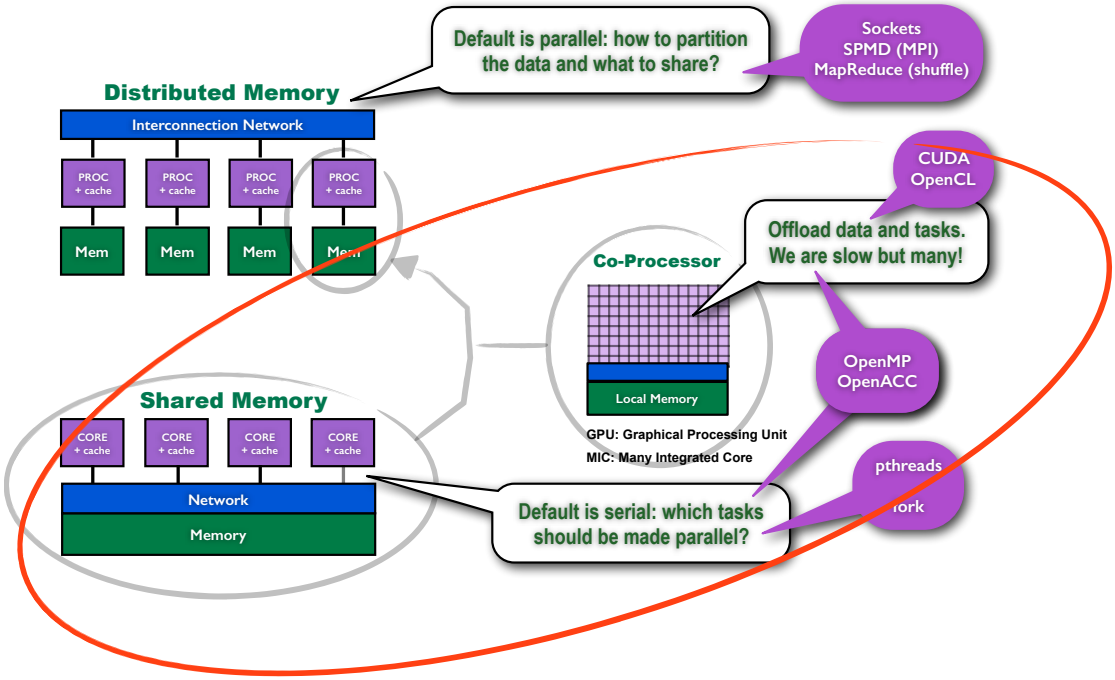
"Native" Programming Models and Tools



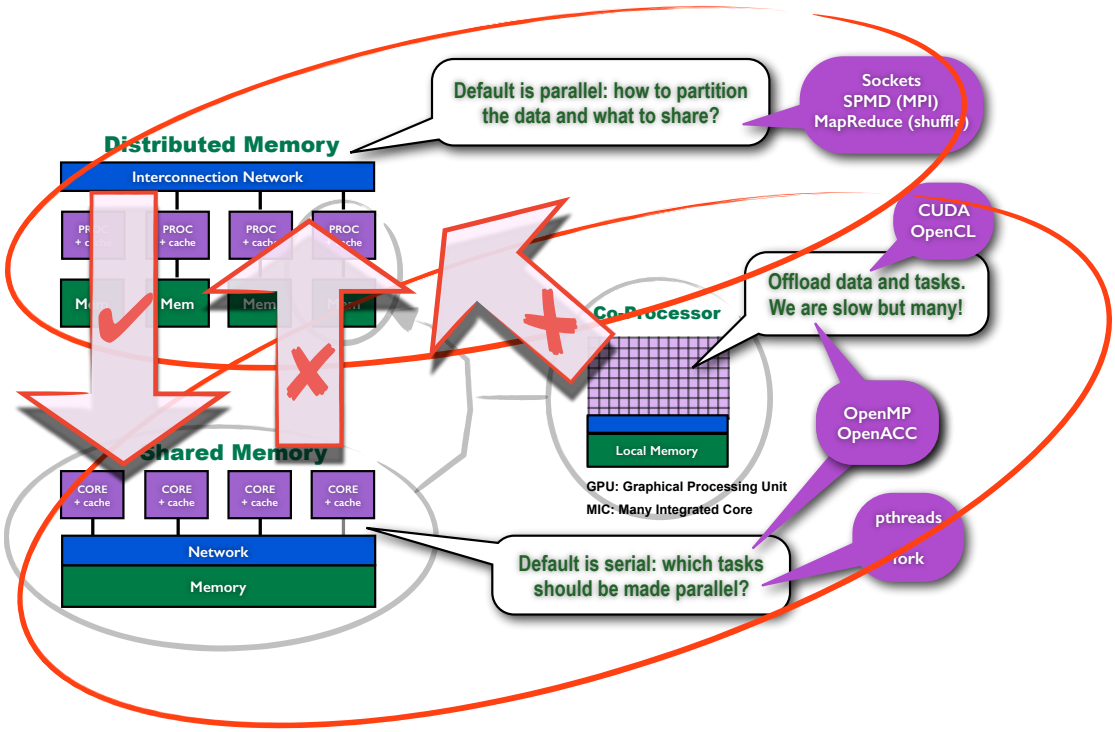
30+ Years of Parallel Computing Research



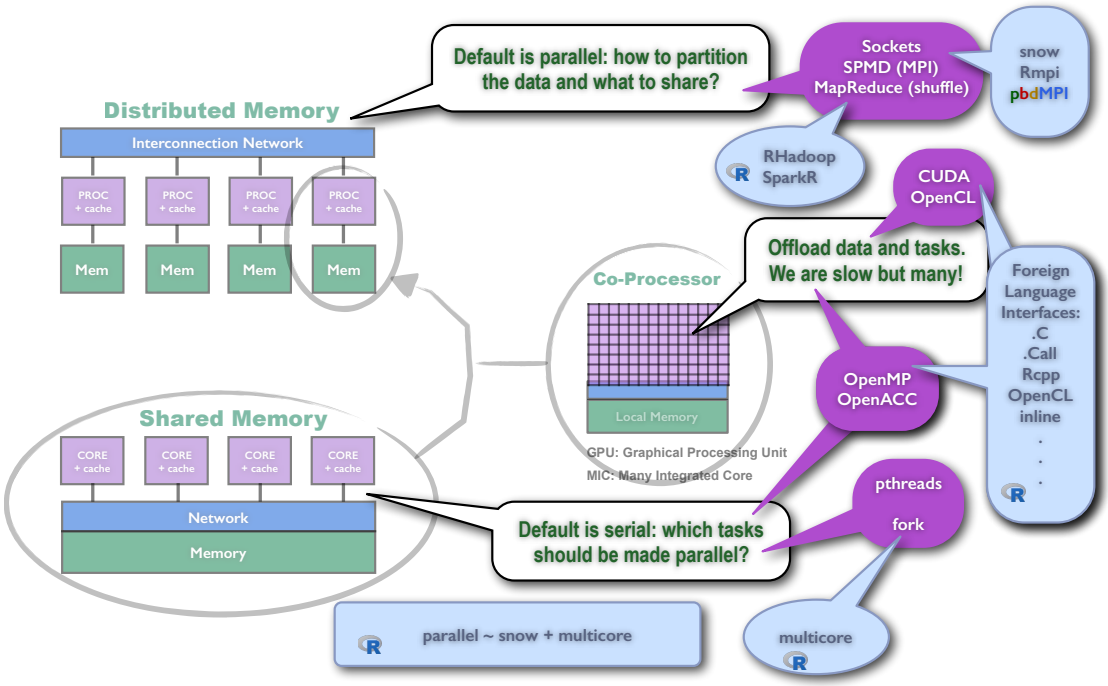
Last 10+ years of Advances



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: iterface 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

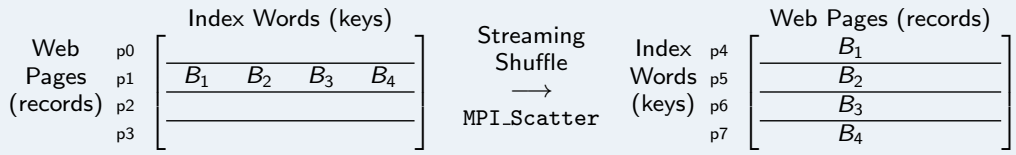
		Index Words (keys)			
Web Pages (records)	p0	A_1	A_2	A_3	A_4
	p1	B_1	B_2	B_3	B_4
	p2	C_1	C_2	C_3	C_4
	p3	D_1	D_2	D_3	D_4

Shuffle $\longrightarrow$ `MPI_Alltoallv`

		Web Pages (records)			
Index Words (keys)	p0	A_1	B_1	C_1	D_1
	p1	A_2	B_2	C_2	D_2
	p2	A_3	B_3	C_3	D_3
	p3	A_4	B_4	C_4	D_4

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

Reduction on a 1986 hypercube: Managing bitflips and dimensions

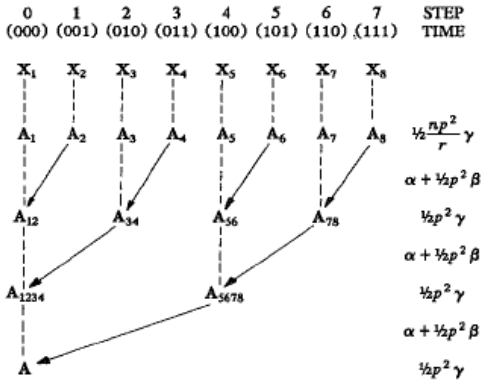


FIG. 4. *Computation of* $\mathbf{A} = \mathbf{X}'\mathbf{X}$ *on an 8-processor hypercube, with final result on processor 0.*

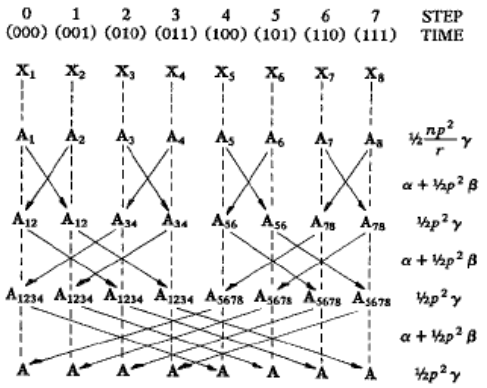


FIG. 6. *Computation of* $\mathbf{A} = \mathbf{X}'\mathbf{X}$ *on an 8-processor hypercube, with final result on all processors.*

[a]

[a] Ostrouchov (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.

3. Parallel Hardware and Programming Paradigms | Distributed Parallel Computing Paradigms

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

Today: A = reduce(crossprod(X)) A = allreduce(crossprod(X)) pbdR

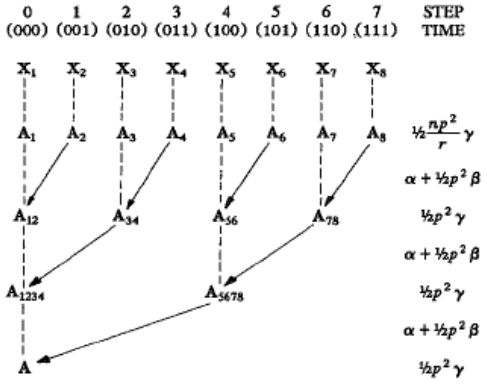


FIG. 4. *Computation of* $\mathbf{A} = \mathbf{X}'\mathbf{X}$ *on an 8-processor hypercube, with final result on processor 0.*

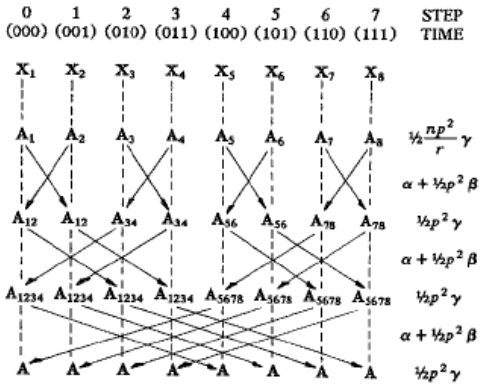


FIG. 6. *Computation of* $\mathbf{A} = \mathbf{X}'\mathbf{X}$ *on an 8-processor hypercube, with final result on all processors.*

[a]

[a] Ostrouchov (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 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

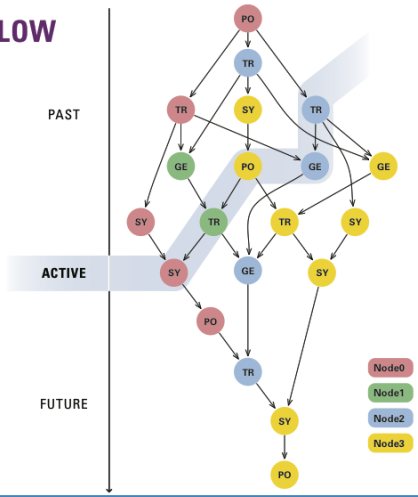
Concepts differ in approach to communication

- 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

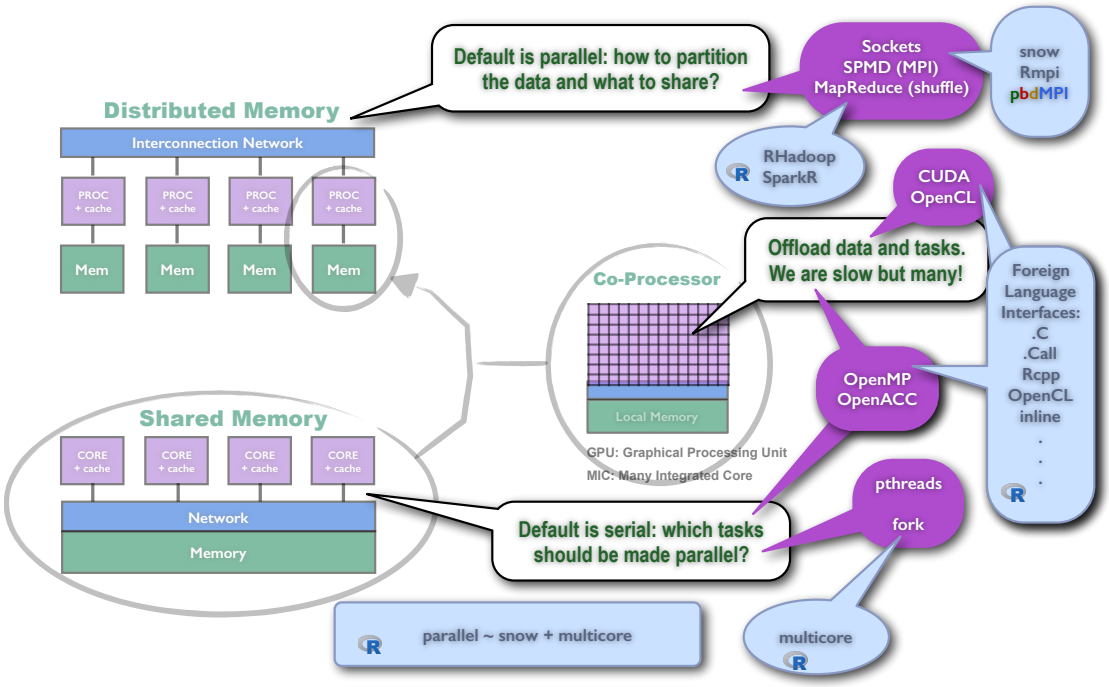


Graphic from icl.cs.utk.edu

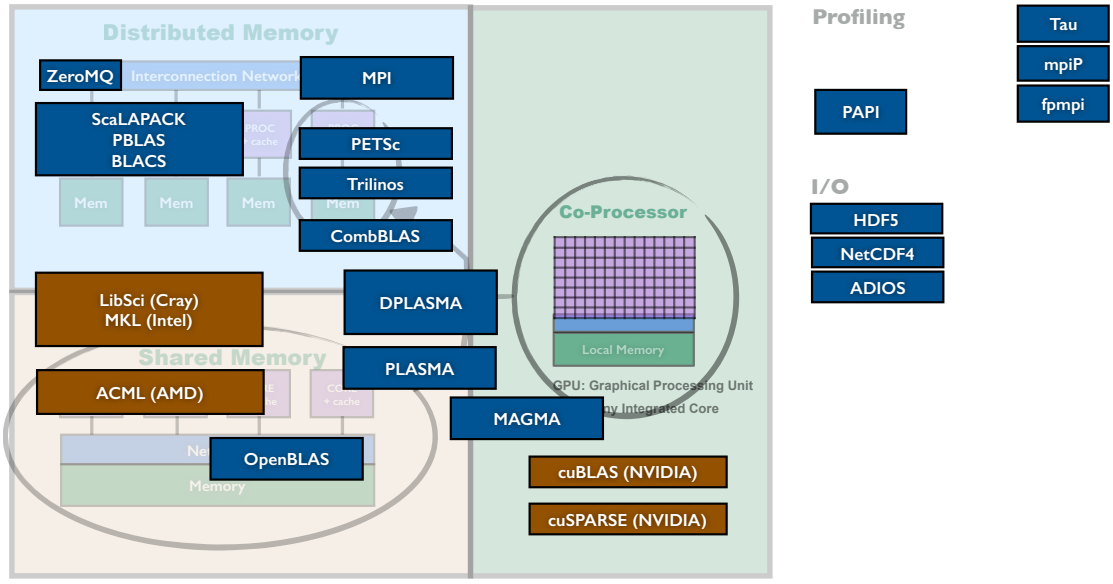
Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," IEEE Computing in Science and Engineering, Vol. 15, No. 6, 36-45, November, 2013.

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

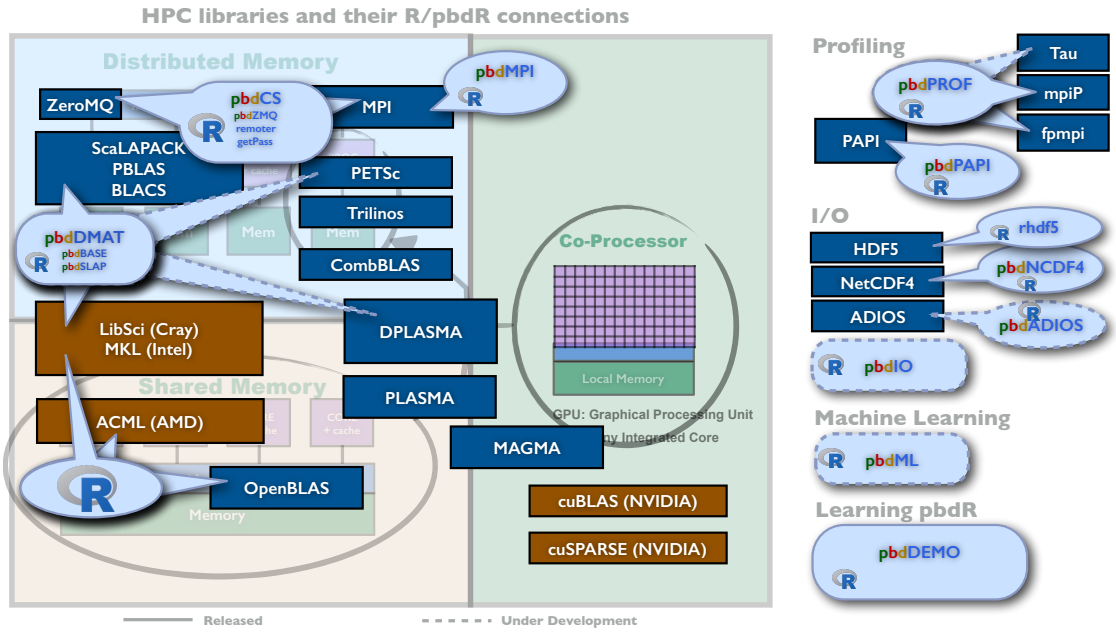
Recall: Hardware flavors and Low-Level Native Tools



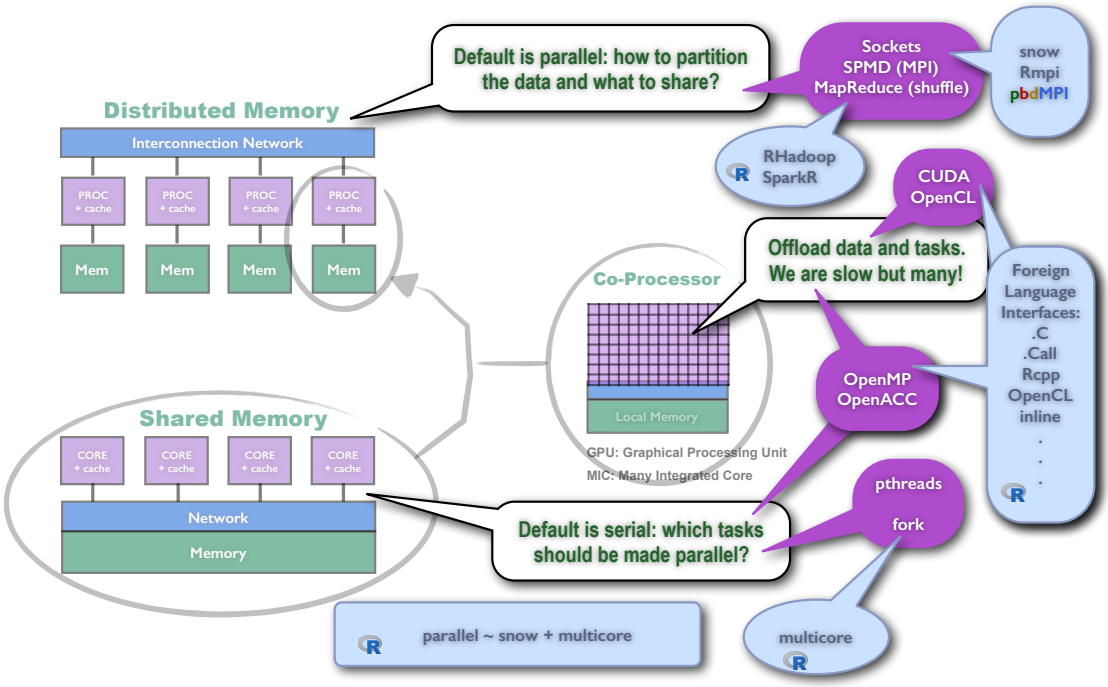
Scalable Libraries Mapped to Hardware



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 Ostrouchov[1] and Wei-Chen Chen[2]

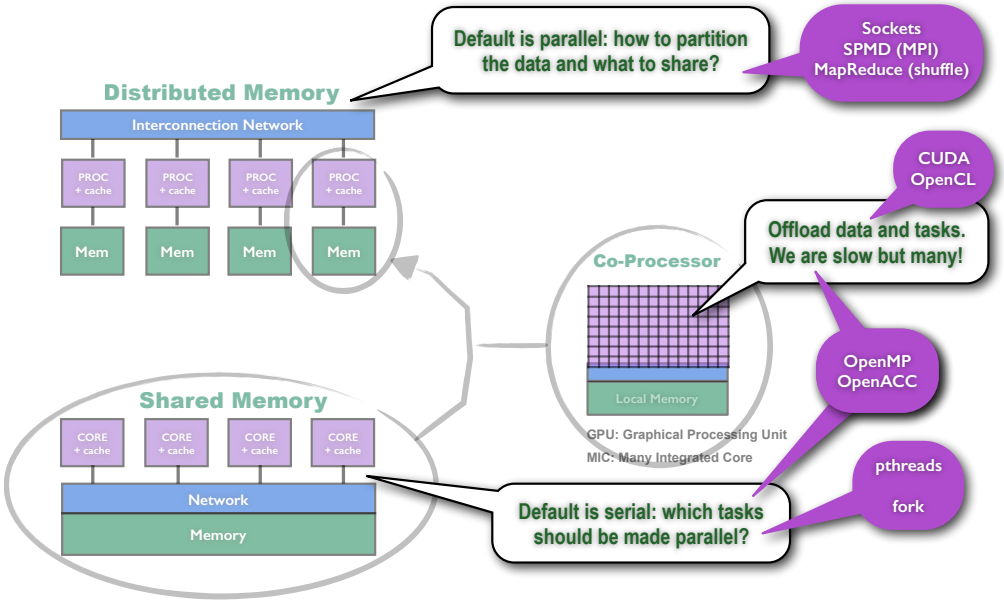
[1] Oak Ridge National Laboratory and University of Tennessee
[2] U.S. Food and Drug Administration

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

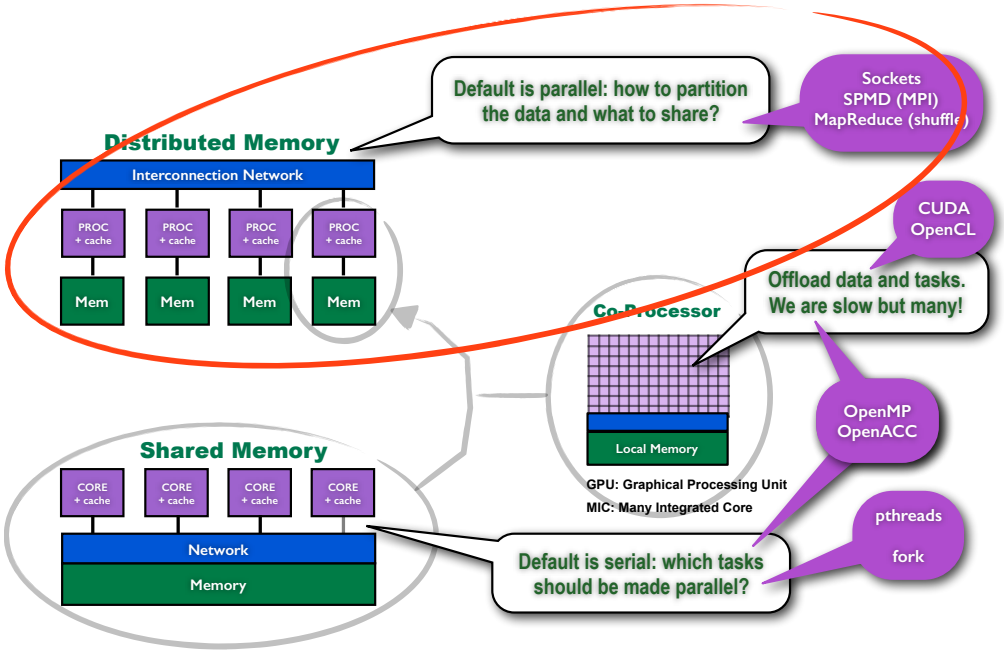
Outline

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

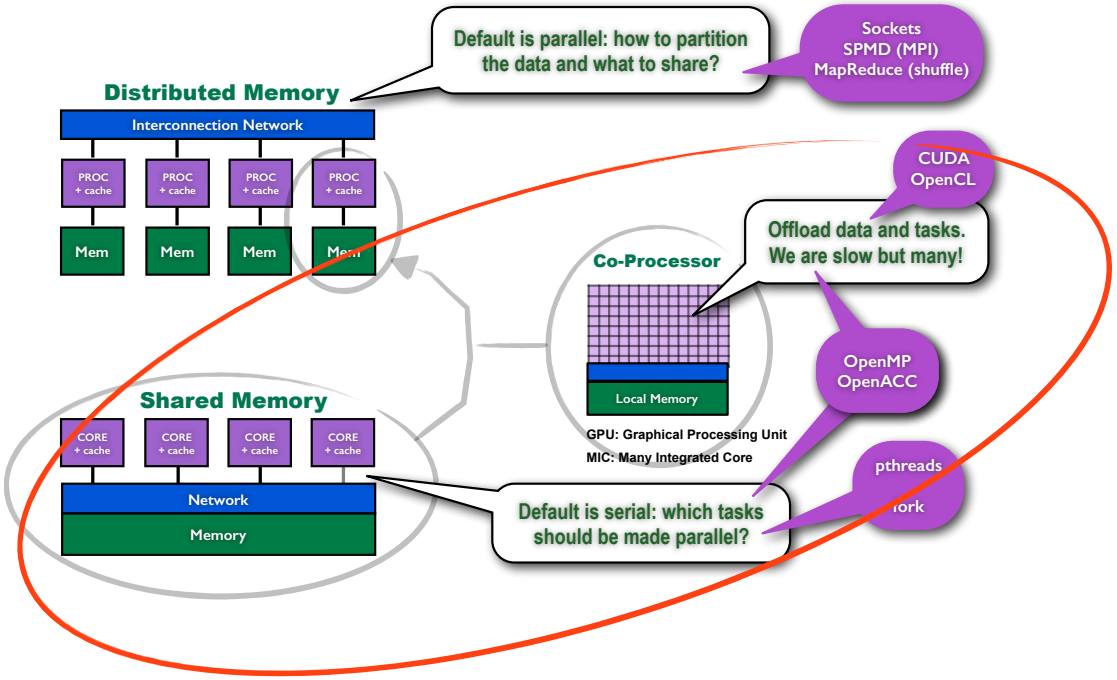
"Native" Programming Models and Tools



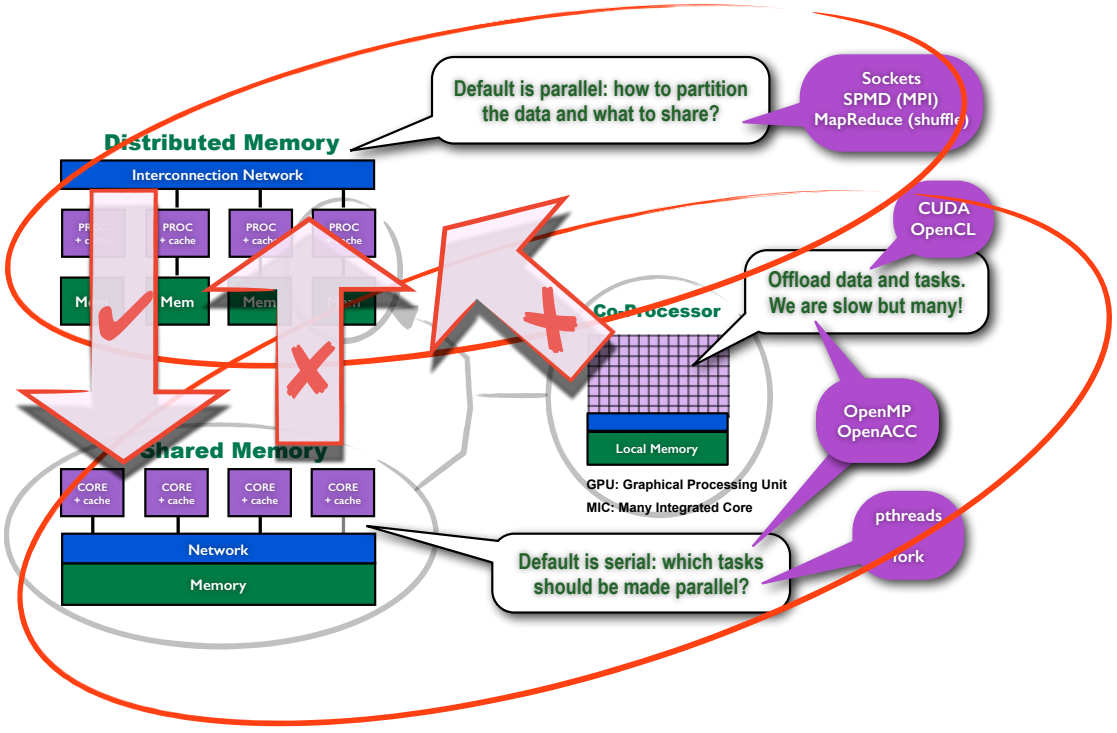
30+ Years of Parallel Computing Research



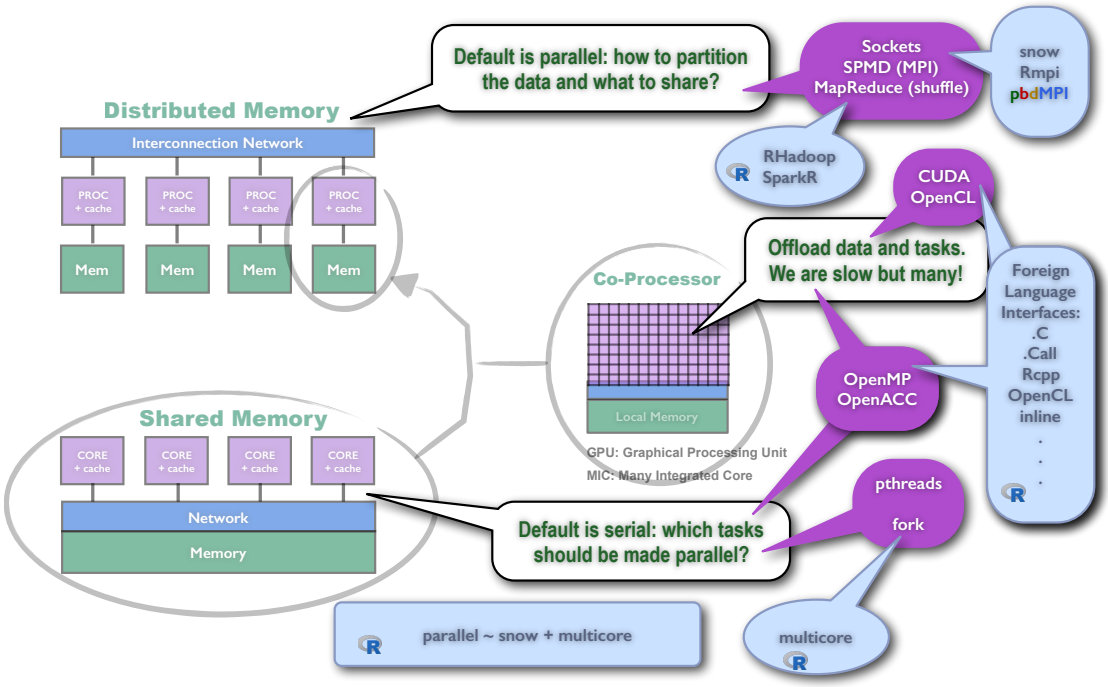
Last 10 years of Advances



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.

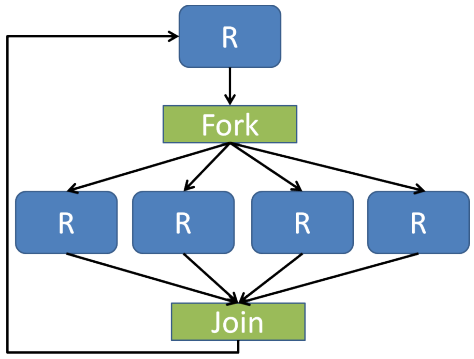
$$\textbf{parallel} \approx \textbf{snow} + \textbf{multicore}$$

- Prefer to use only multicore components
  - Clever and efficient use of the Unix `fork`
- 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

```
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)
```

code/mclapply.R

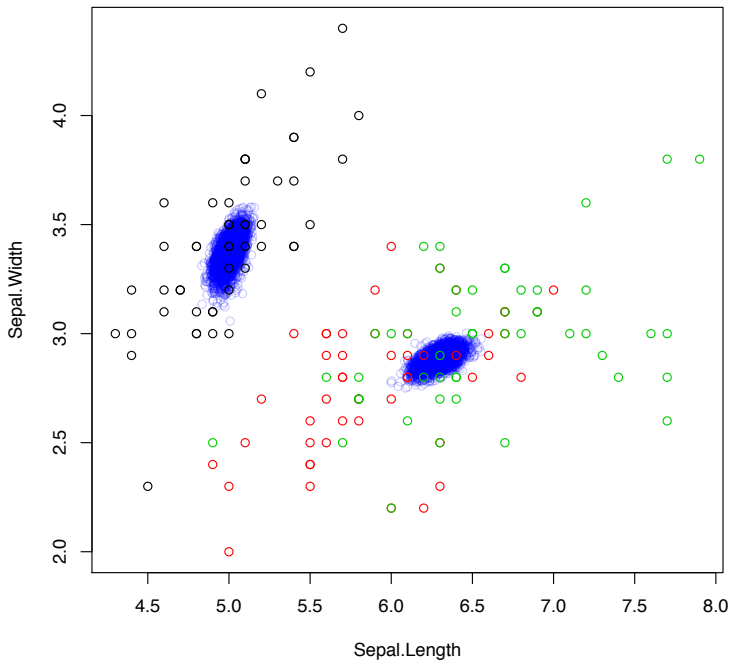
```
x <- lapply(1:1e3, function(x) median(rnorm(1e5)))

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

Bootstrap k-means centers from iris data

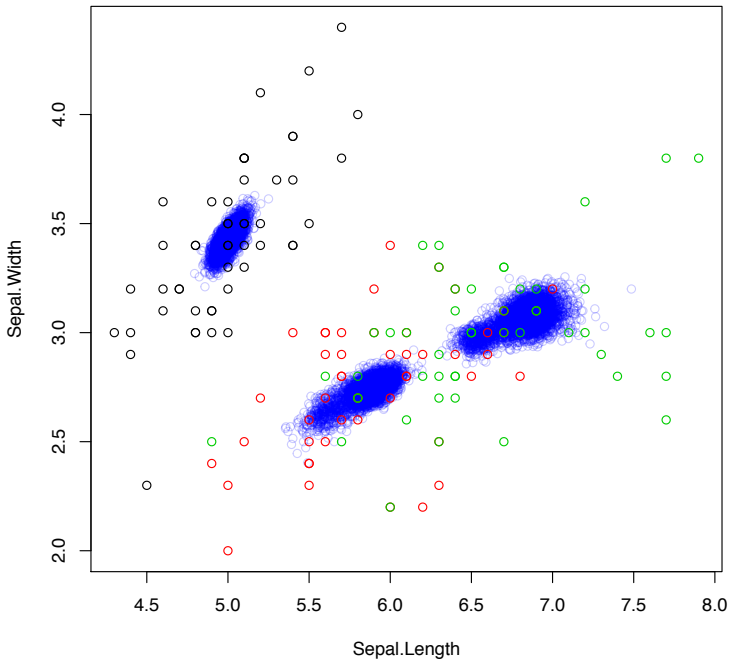
code/3kmp–mclapply.R

```
library(parallel, quietly = TRUE)
X <- as.matrix(iris[, 1:4])
k <- 3
set.seed(seed = 123)
FUN <- function(n) {
  isample <- sample(nrow(X), nrow(X), replace = TRUE)
  kmeans(X[isample, ], k, iter.max = 100, nstart = 100)$centers
}

means <- do.call(rbind, mclapply(1:5000, FUN, mc.cores = 2))

pdf("means.pdf")
  plot(X[, c(1, 2)], type = "n")
  points(means[, c(1, 2)], col = rgb(0, 0, 1, 0.2))
  points(X[, c(1, 2)], col = iris[, 5])
dev.off()
```

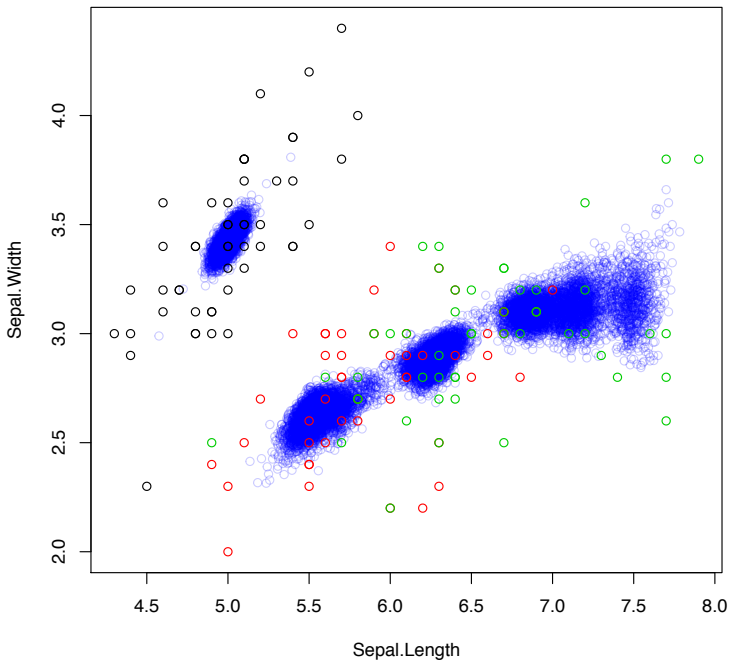

Bootstrap k-means centers from iris data: k=2



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

code/rf–serial.R

```
library(randomForest)
library(mlbench)
data(LetterRecognition)
set.seed(seed = 123)

n <- nrow(LetterRecognition)
n_test <- floor(0.2 * n)
i_test <- sample.int(n, n_test)
train <- LetterRecognition[-i_test,]
test <- LetterRecognition[i_test,]

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

Example: Random Forest Code

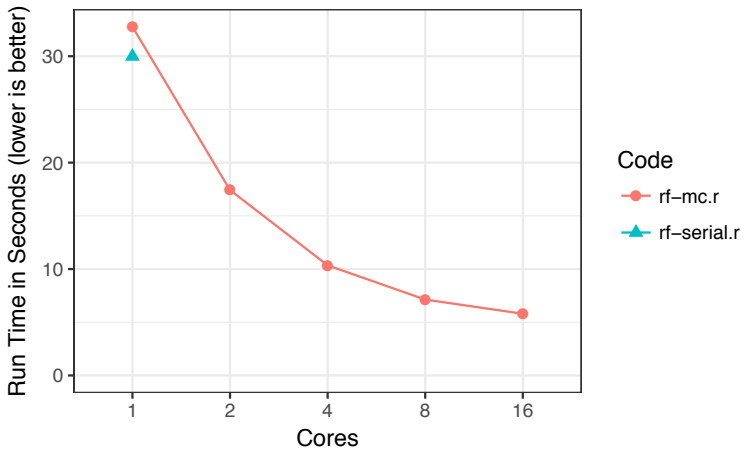
Split learning by blocks of trees. Split prediction by blocks of rows.

code/rf–mc.R

```
library(randomForest)
library(mlbench)
data(LetterRecognition)
library(parallel)
RNGkind("L'Ecuyer-CMRG")  # mc
set.seed(seed = 123)

n <- nrow(LetterRecognition)
n_test <- floor(0.2 * n)
i_test <- sample.int(n, n_test)
train <- LetterRecognition[-i_test, ]
test <- LetterRecognition[i_test, ]

nc <- detectCores()  # mc
ntree <- lapply(splitIndices(500, nc), length)  # mc
rf <- function(x) randomForest(lettr~., train, ntree=x,
    norm.votes=FALSE)# mc
rf.out <- mclapply(ntree, rf, mc.cores = nc)  # mc
rf.all <- do.call(combine, rf.out)  # mc

crows <- splitIndices(nrow(test), nc)  # mc
rfp <- function(x) as.vector(predict(rf.all, test[x, ]))  # mc
cpred <- mclapply(crows, rfp, mc.cores = nc)  # mc
pred <- do.call(c, cpred)  # mc
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

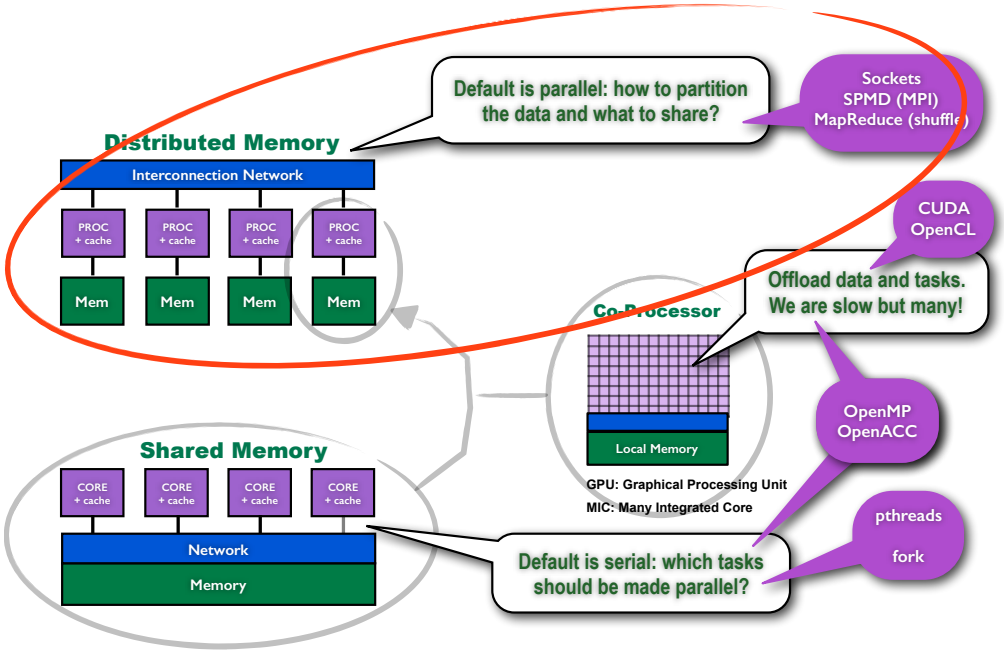
5. Distributed Parallel Computing with pbdMPI

Instructors: George Ostrouchov[1] and Wei-Chen Chen[2]

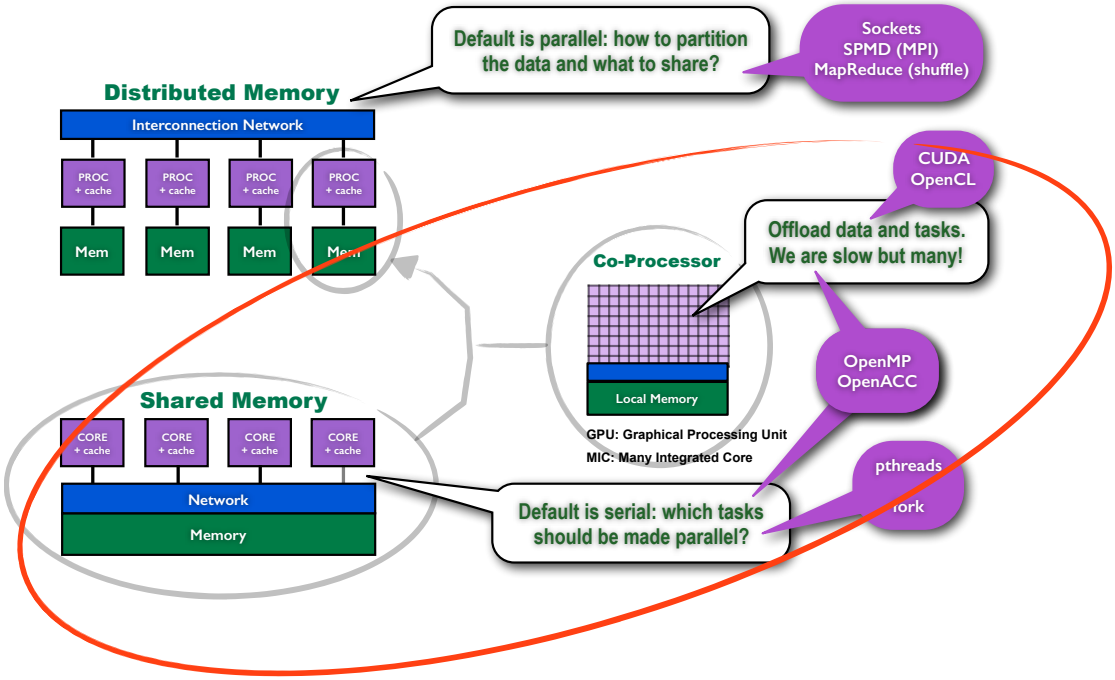
[1] Oak Ridge National Laboratory and University of Tennessee
[2] U.S. Food and Drug Administration

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

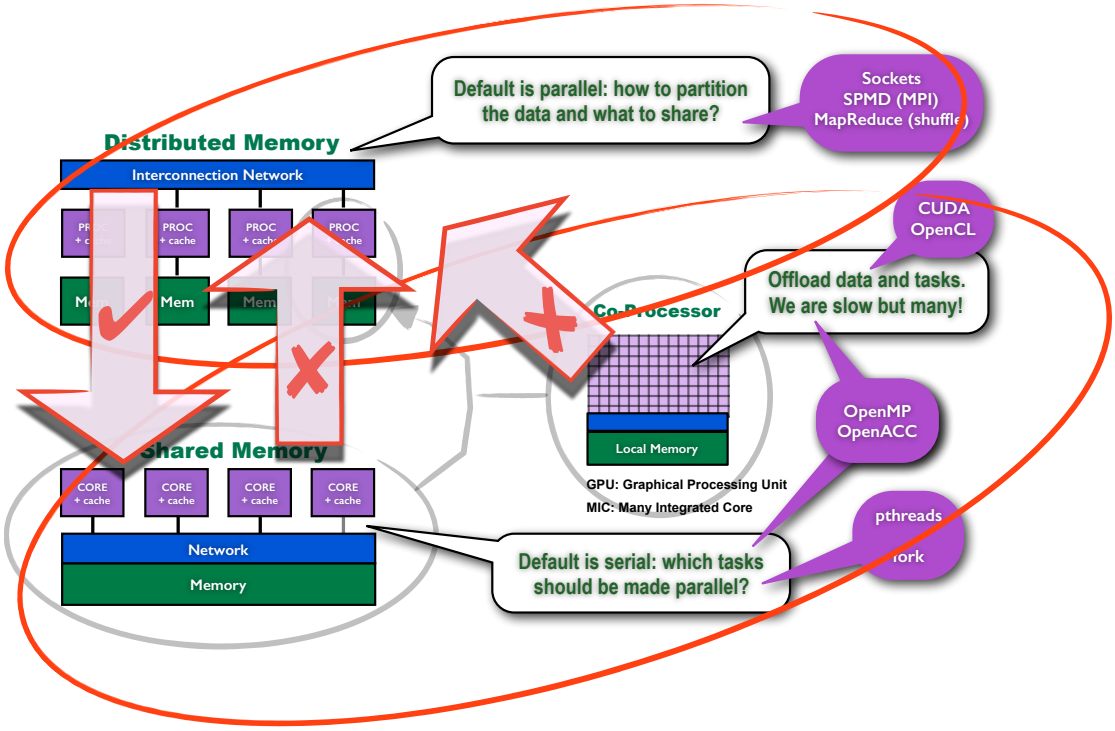
30+ Years of Parallel Computing Research



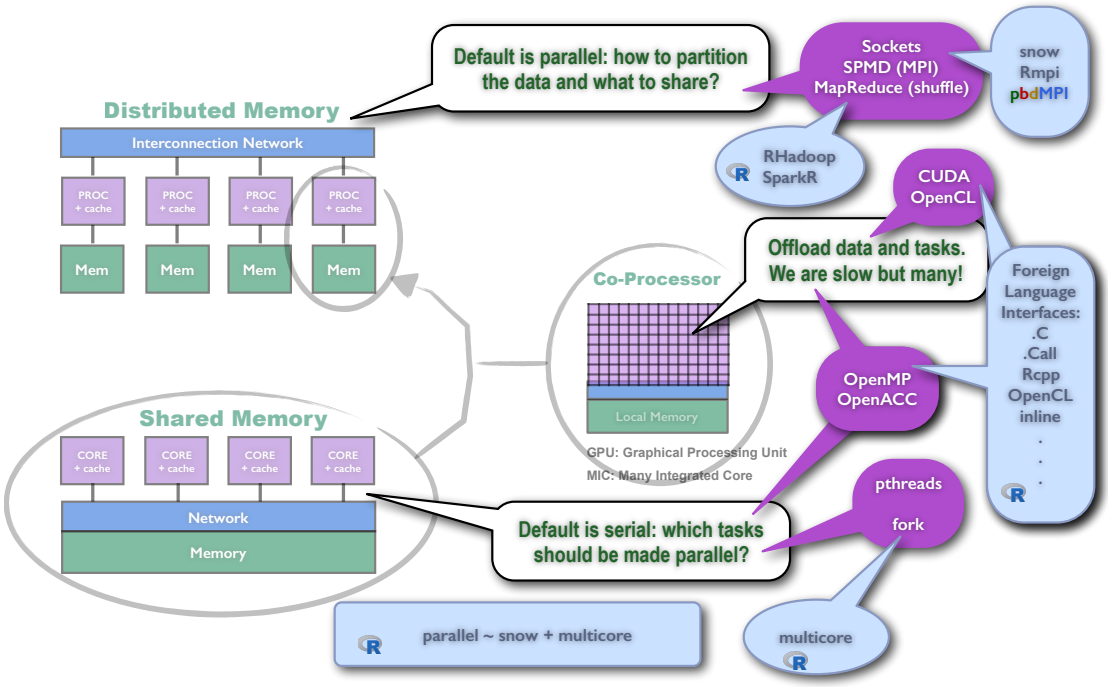
Last 10 years of Advances



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

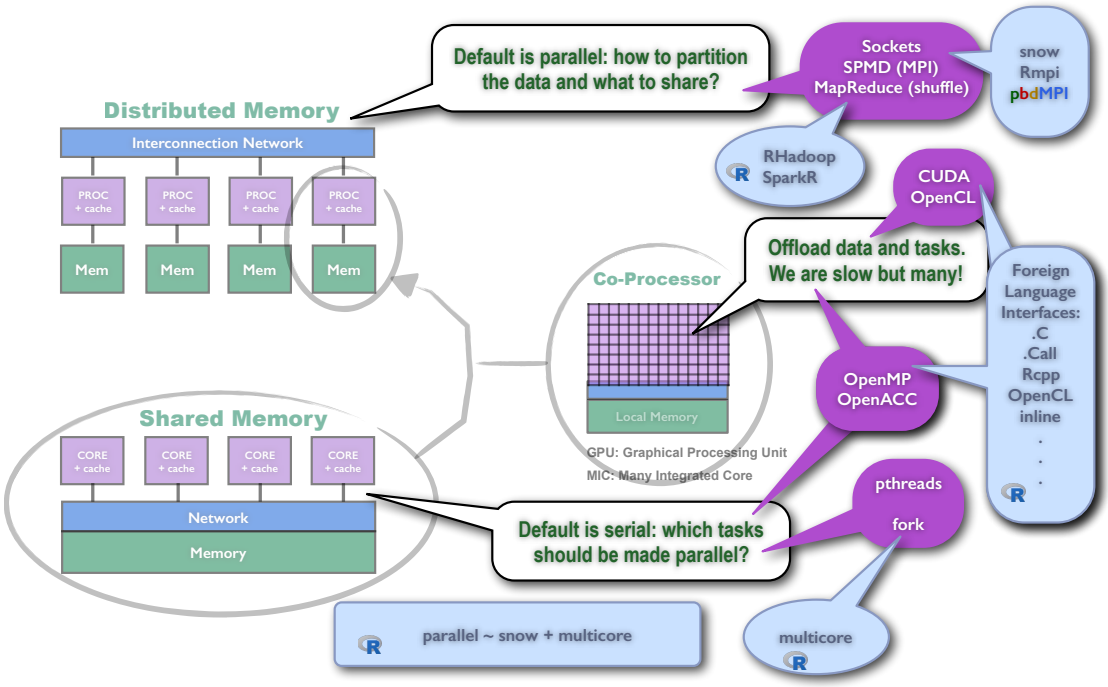
- Workers get their own data - parallel
- Workers can run complex SPMD codes - parallel
- Workers combine results - parallel
  - Associative and commutative reductions with recursive doubling

SPMD: Single Program Multiple Data

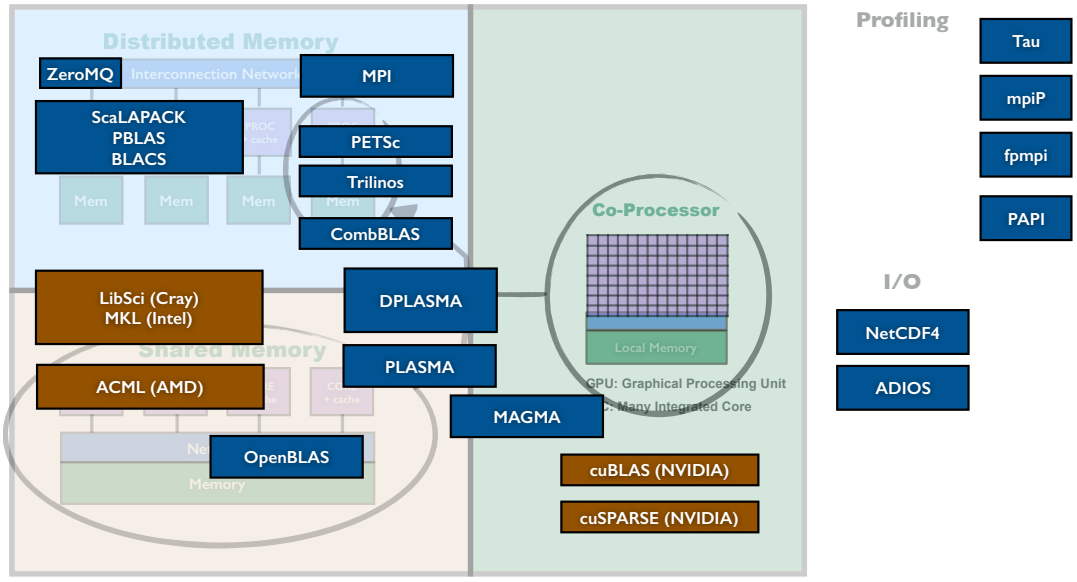
Write one general program so many copies of it can run asynchronously and cooperate (usually via MPI) to solve the problem.

- 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

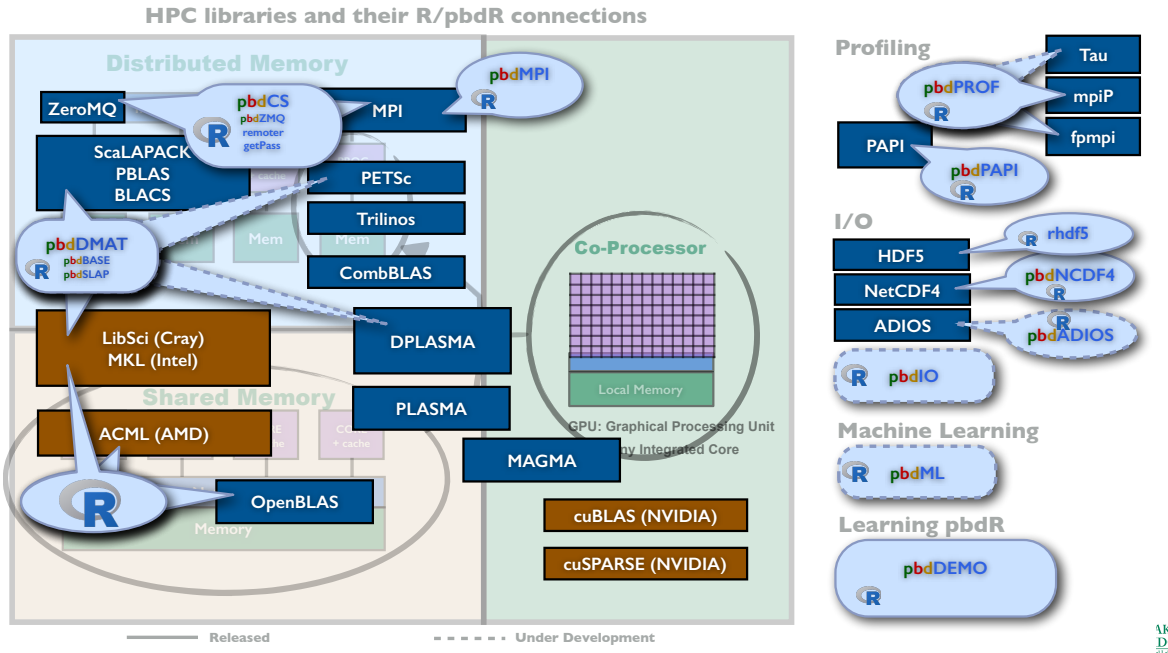
R Interfaces to Low-Level Native Tools



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**
allreduce	mpi.allreduce
allgather	mpi.allgather, mpi.allgatherv, mpi.allgather.Robj
bcast	mpi.bcast, mpi.bcast.Robj
gather	mpi.gather, mpi.gatherv, mpi.gather.Robj
recv	mpi.recv, mpi.recv.Robj
reduce	mpi.reduce
scatter	mpi.scatter, mpi.scatterv, mpi.scatter.Robj
send	mpi.send, mpi.send.Robj

Hello World

code/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

code/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

code/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

code/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?

code/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

code/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()

code/timer.r

```
library( pbdMPI, quiet = TRUE )

comm.set.seed( seed = 1234567, diff = T )

test <- function( timed )
{
  ltime <- system.time( timed )[ 3 ]

  mintime <- allreduce( ltime, op='min' )
  maxtime <- allreduce( ltime, op='max' )
  meantime <- allreduce( ltime, op='sum' ) / comm.size()

  return( data.frame( min = mintime, mean = meantime, max = maxtime ) )
}

# generate 10,000,000 random normal values (total)
times <- test( rnorm( 1e7/comm.size() ) ) # ~76 MiB of data
comm.print( times )

finalize()
```

What's My Job?

code/jid.r

```
library( pbdMPI, quiet = TRUE )

k <- get.jid( 10 )   # note: pbdIO has comm.chunk()
my.rank <- comm.rank()
comm.cat( my.rank, ":", k, "\n", all.rank = TRUE, quiet = TRUE )

finalize()
```

Reduce a Matrix

code/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

code/mcsim.r

```
1  ### Compute pi by simulaton
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

code/cov.r

```
library( pbdMPI, quiet = TRUE )

comm.set.seed( seed = 1234567, diff = TRUE )

## Generate 10 rows and 3 columns of data per process
my.X <- matrix( rnorm(10*3), ncol = 3 )

## Compute mean
N <- allreduce( nrow( my.X ), op = "sum" )
mu <- allreduce( colSums( my.X ) / N, op = "sum" )

## Sweep out mean and compute crossproducts sum
my.X <- sweep( my.X, STATS = mu, MARGIN = 2 )
Cov.X <- allreduce( crossprod( my.X ), op = "sum" ) / ( N - 1 )

comm.print( Cov.X )

finalize()
```

Least Squares Fit via Normal Equations

code/ols.r

```
### Least Squares Fit via Normal Equations (see lm.fit for a better way)
library( pbdMPI, quiet = TRUE )

comm.set.seed( seed = 12345, diff = TRUE )

## 10 rows and 3 columns of data per process
my.X <- matrix( rnorm(10*3), ncol = 3 )
my.y <- matrix( rnorm(10*1), ncol = 1 )

## Form the Normal Equations components
my.Xt <- t( my.X )
XtX <- allreduce( my.Xt %*% my.X, op = "sum" )
Xty <- allreduce( my.Xt %*% my.y, op = "sum" )

## Everyone solve the Normal Equations
ols <- solve( XtX, Xty )

comm.print( ols )

finalize()
```

Bootstrapping k-means on Iris data

code/bootstrap–kmp.r

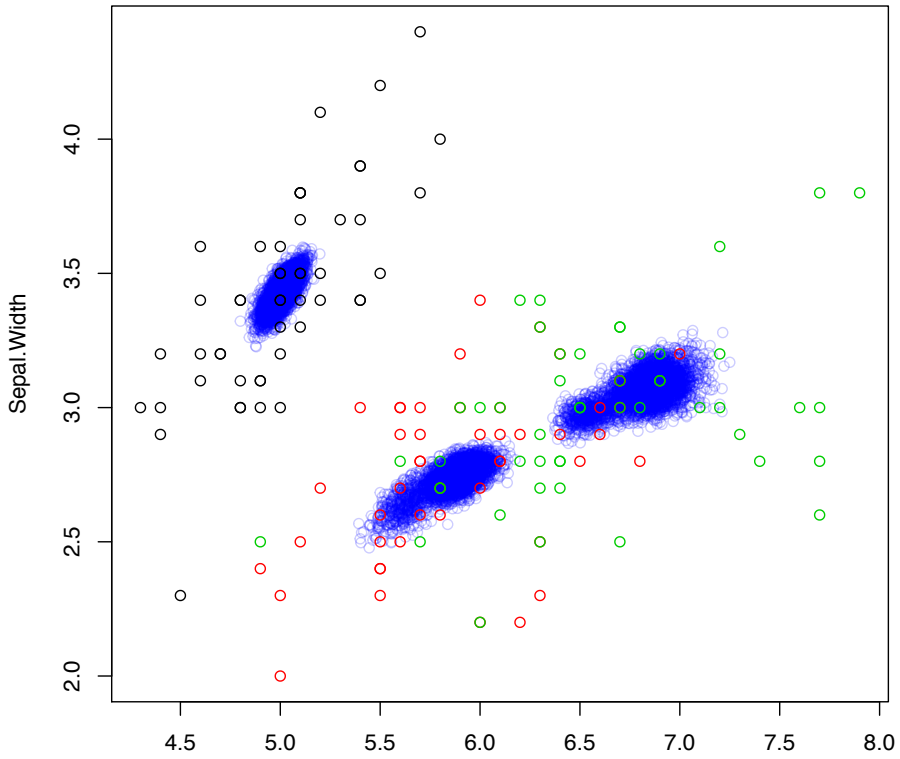
```
library( pbdMPI, quietly = TRUE )
X <- as.matrix( iris[ , 1:4 ] )
k <- 3
comm.set.seed( seed = 123, diff = TRUE )
FUN <- function( n ) {
  isample <- sample( nrow(X), nrow(X), replace = TRUE )
  kmeans( X[ isample, ], k, iter.max = 100, nstart = 100 )$centers
}

my.means <- do.call( rbind, lapply( get.jid( 5000 ), FUN ) )
means <- gather( my.means )

if( comm.rank( ) == 0 ){
  means <- do.call( rbind, means )
  pdf( "means.pdf" )
    plot( X[ , c(1, 2) ], type = "n" )
    points( means[ , c(1, 2) ], col = rgb( 0, 0, 1, 0.2 ) )
    points( X[ , c(1, 2) ], col = iris[ , 5 ] )
  dev.off()
}
finalize()
```

Bootstrapping k-means on Iris data + parallel plots

code/bootstrap–kmpp.r

```
### file "bootstrap_kmpp.r"
library(pbdMPI, quietly = TRUE)
X <- as.matrix(iris[, 1:4])
k <- 3
comm.set.seed(seed = 123, diff = TRUE)
FUN <- function(n) {
  isample <- sample(nrow(X), nrow(X), replace = TRUE)
  kmeans(X[isample, ], k, iter.max = 100, nstart = 100)$centers
}
my.samples <- get.jid(5000)
my.means <- do.call(rbind, lapply(my.samples, FUN))
means <- do.call(rbind, allgather(my.means))

plots <- combn(ncol(X), 2)
my.plots <- plots[, get.jid(ncol(plots)), drop = FALSE]
plotFUN <- function(cols) {
  pdf(paste(k, "means", paste(cols, collapse = "-"), ".pdf", sep = ""))
    plot(X[, cols], type = "n")
    points(means[, cols], col = rgb(0, 0, 1, 0.2))
    points(X[, cols], col = iris[, 5])
  dev.off()
}
bits <- apply(my.plots, 2, plotFUN)
finalize()
```

Bootstrapping k-means on Iris data



Crossvalidation: Sombrero Function Data Setup

code/crossvalidate–setup.r

```
1  library( locfit )
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
       grid
10 for ( i in 1:length( x.seq ) ) {
11     for ( j in 1:length( y.seq ) ) {
12         z[ i, j ] <- f( c( x.seq[ i ], y.seq[ j ] ) )
13     }
14 }
15
16 sigma.true <- 0.1                # Generate observations with error
17 n <- 1000
18 x1 <- runif( n, -pi, pi )
19 x2 <- runif( n, -pi, pi )
20 y <- numeric( n )
21 f.true <- numeric( n )
22 for ( i in 1:n ) {
23    y[ i ] <- f( c( x1[ i ], x2[ i ] ) ) + rnorm( 1, 0, sigma.true )
24    f.true[ i ] <- f( c( x1[ i ], x2[ i ] ) )
25 }
26 dat <- data.frame( y = y, x1 = x1, x2 = x2 )
```

Crossvalidation: Nearest (alpha) Neighbor Fitting

code/crossvalidate.r

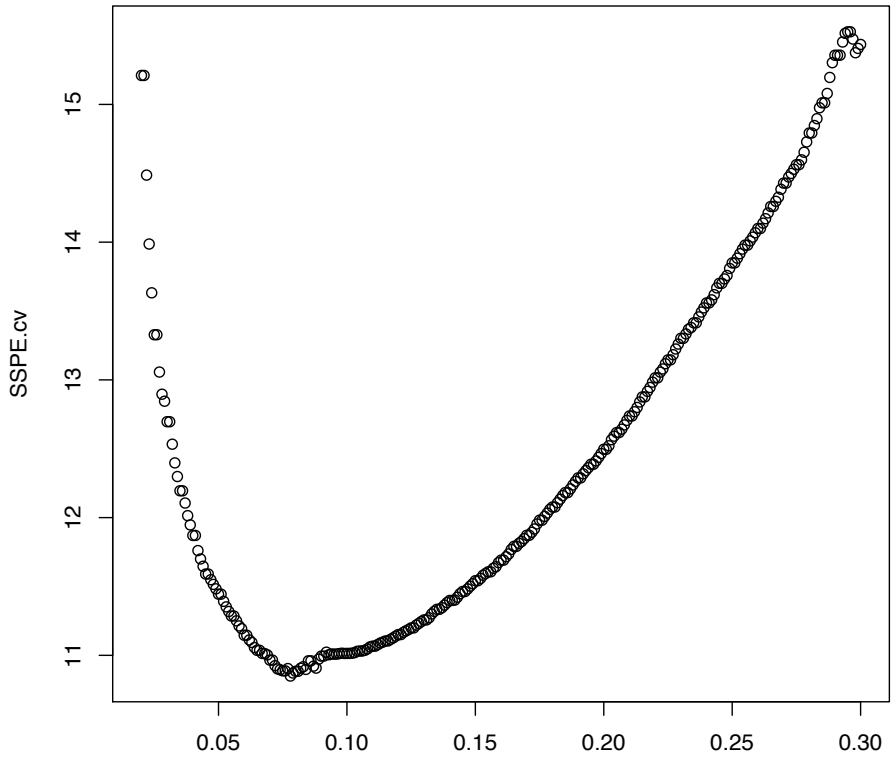
```
source( "crossvalidate-setup.r" )

K <- 5
fold.grp <- ( 1:n - 1 ) %% K + 1
folds <- split( 1:n, f = fold.grp )
y.hat.cv <- numeric( n )
alpha <- seq( 0.02, 0.3, 0.001 )

SSPE.cv <- numeric( length( alpha ) )

for ( j in 1:length( alpha ) ) {
    for ( k in 1:K ) {
        idx <- folds[ [ k ] ]
        fit.fold.out <- locfit( y ~ lp( x1, x2, nn = alpha[ j ] ), data
            = dat[ -idx, ] )
        y.hat.cv[ idx ] <- predict( fit.fold.out, newdata = dat[ idx, ] )
    }
  SSPE.cv[ j ] <- sum( ( y - y.hat.cv )^2 )
}

plot( alpha, SSPE.cv )
```

Crossvalidation: Parallel Nearest (alpha) Neighbor Fitting

code/crossvalidate.pbdR.r

```
source( "crossvalidate-setup.r" )

K <- 5
fold.grp <- ( 1:n - 1 ) %% K + 1
folds <- split( 1:n, f = fold.grp )
y.hat.cv <- numeric( n )
alpha <- seq( 0.02, 0.3, 0.001 )

my.alpha.i <- get.jid( length( alpha ) ) # pbdR
SSPE.cv <- numeric( length( my.alpha.i ) ) # pbdR
for ( j in my.alpha.i ) { # pbdR
    for ( k in 1:K ) {
        idx <- folds[ [ k ] ]
        fit.fold.out <- locfit( y ~ lp( x1, x2, nn = alpha[ j ] ), data
            = dat[ -idx, ] )
        y.hat.cv[ idx ] <- predict( fit.fold.out, newdata = dat[ idx, ] )
    }
  SSPE.cv[ j - my.alpha.i[1] + 1 ] <- sum( ( y - y.hat.cv )^2 ) # pbdR
}

SSPE.cv <- do.call( c, allgather( SSPE.cv ) ) # pbdR

if( comm.rank() == 0 ) { # pbdR
    plot( alpha, SSPE.cv )
} # pbdR

finalize()  # pbdR
```

Crossvalidation: alpha SSPE curve



Machine Learning Example: Random Forest

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

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

Serial Code rf-serial.r

```
library(randomForest)
library(mlbench)
data(LetterRecognition) # 26 Capital Letters Data 20,000 x 17
set.seed(seed=123)
n <- nrow(LetterRecognition)
n_test <- floor(0.2*n)
i_test <- sample.int(n, n_test) # Use 1/5 of the data to test
train <- LetterRecognition[-i_test, ]
test <- LetterRecognition[i_test, ]

## train random forest
rf.all <- randomForest(lettr ~ ., train, ntree=500, norm.votes=FALSE)

## predict test data
pred <- predict(rf.all, test)
correct <- sum(pred == test$lettr)
cat("Proportion Correct:", correct/n_test, "\n")
```

Example: Random Forest Code

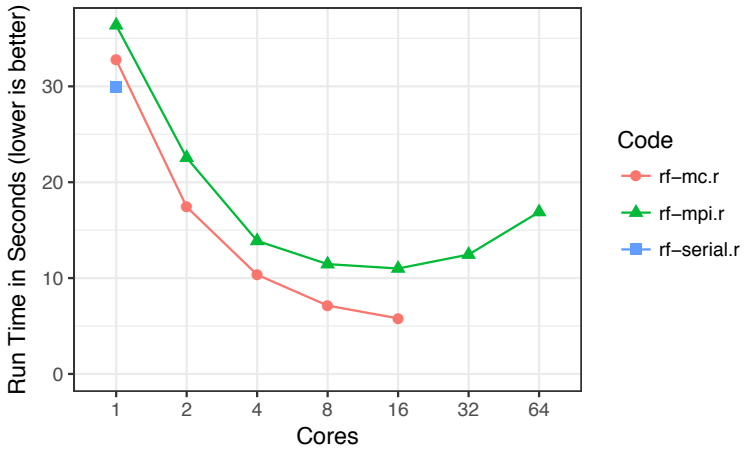
Split learning by blocks of trees. Split prediction by blocks of rows.

Parallel Code rf-mpi.r

```
library(randomForest)
library(mlbench)
data(LetterRecognition)
comm.set.seed(seed=123, diff=FALSE) # same training data
n <- nrow(LetterRecognition)
n_test <- floor(0.2*n)
i_test <- sample.int(n, n_test) # Use 1/5 of the data to test
train <- LetterRecognition[-i_test, ]
test <- LetterRecognition[i_test, ][get.jid(n_test), ]

comm.set.seed(seed=1e6*runif(1), diff=TRUE)
my.rf <- randomForest(lettr ~ ., train, ntree=500%/%comm.size(),
    norm.votes=FALSE)
rf.all <- do.call(combine, allgather(my.rf))

pred <- predict(rf.all, test)
correct <- allreduce(sum(pred == test$lettr))
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

```
> summaryRprof()
$by.total
                    total.time  total.pct  self.time  self.pct
"min.basis"              12.32     100.00       0.00      0.00
"type.CV"                 6.54      53.08       0.02      0.16
"S.basis"                 5.76      46.75       0.00      0.00
"drop"                    4.20      34.09       0.00      0.00
"norm.fdata"              4.20      34.09       0.00      0.00
"metric"                  4.18      33.93       1.04      8.44
"%*%"                     3.98      32.31       3.98     32.31
"getbasispenalty"         2.72      22.08       0.02      0.16
"bsplinepen"              2.68      21.75       0.36      2.92
"int.simpson2"            2.54      20.62       1.96     15.91
"t"                       2.10      17.05       0.10      0.81
"ppBspline"               1.60      12.99       0.82      6.66
. . .
```

min.basis() 110 lines SPMD: Add 5, change 3

```
min.basis <- function(fdataobj, type.CV = GCV.S, . . ., ...)
{
    . . . 13 lines
    library(pbdMPI)
    init()
    my.k <- get.jid(lenlambda)
    my.gcv <- array(Inf, dim = c(lenbasis, length(my.k)))
    . . . 36 lines
    for (i in 1:lenbasis) {
        . . . 3 lines
        for (k in my.k) {
            S2 <- S.basis(tt, base, lambda[k])
            my.gcv[i, k - my.k[1] + 1] <-
                type.CV(fdataobj, S = S2, W = W, trim = par.CV$trim,
                    draw = par.CV$draw, ...)
        }
    }
    gcv <- do.call(cbind, allgather(my.gcv))
    finalize()
    . . . 48 lines
```

Running parallel min.basis()

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

```
-bash-4.1$ time Rscript fda-s.r
real 0m28.906s   user 0m28.587s   sys 0m0.138s

-bash-4.1$ time Rscript fda-pbdR.r
real 0m31.090s   user 0m30.654s   sys 0m0.163s

-bash-4.1$ time mpirun -np 1 Rscript fda-pbdR.r
real 0m31.435s   user 0m30.611s   sys 0m0.205s

-bash-4.1$ time mpirun -np 2 Rscript fda-pbdR.r
real 0m18.398s   user 0m35.346s   sys 0m0.444s

-bash-4.1$ time mpirun -np 4 Rscript fda-pbdR.r
real 0m10.590s   user 0m39.252s   sys 0m1.023s

-bash-4.1$ time mpirun -np 8 Rscript fda-pbdR.r
real 0m7.203s   user 0m52.165s   sys 0m1.843s

-bash-4.1$ time mpirun -np 16 Rscript fda-pbdR.r
real 0m5.800s   user 1m18.676s   sys 0m5.876s

-bash-4.1$ time mpirun -np 32 Rscript fda-pbdR.r
real 0m4.965s   user 1m3.870s   sys 0m6.571s
```

kazaam package: "shaq" class for tall and skinny matrices

code/shaq.r

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

kazaam for tall and skinny matrices

code/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 Ostrouchov[1] and Wei-Chen Chen[2]

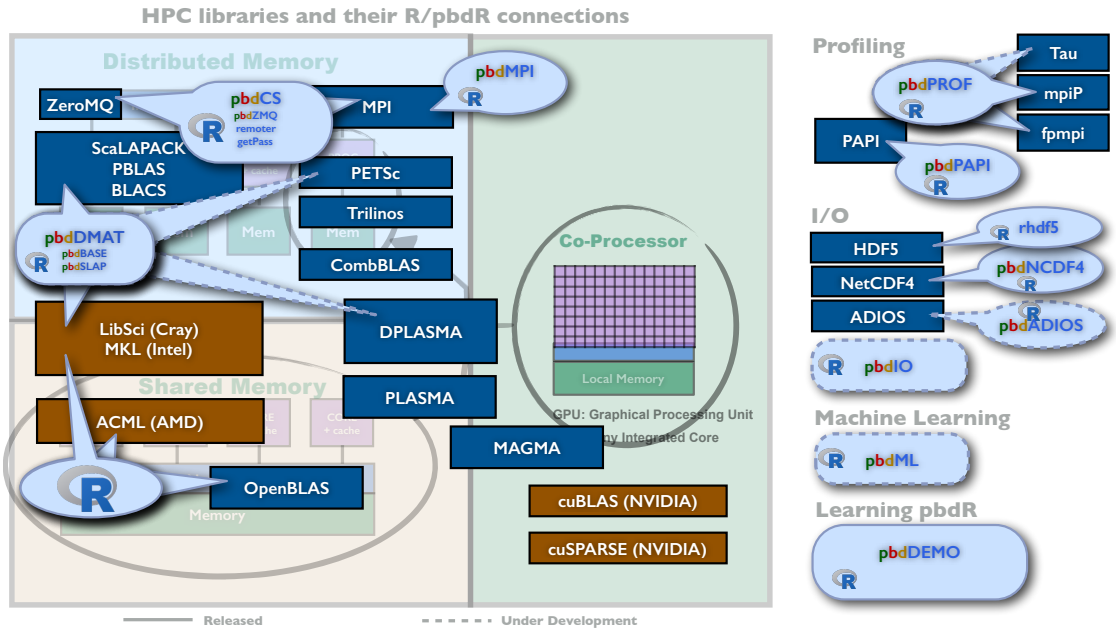
[1] Oak Ridge National Laboratory and University of Tennessee
[2] U.S. Food and Drug Administration

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

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

(a) 1×6 (ICTXT=1) (b) 2×3 (User defined) (c) 3×2 (ICTXT=0) (d) 6×1 (ICTXT=2)

Table: Processor Grid Shapes with 6 Processors

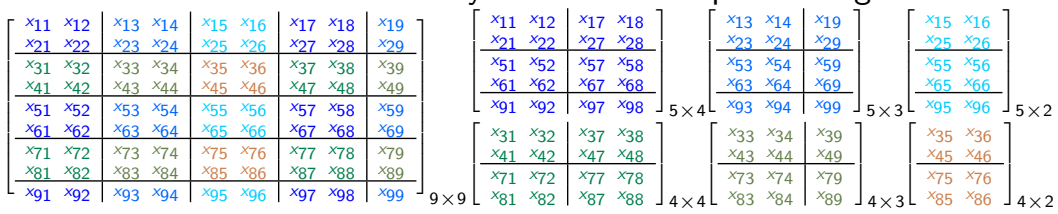
Distributed Matrix Operations

pbdDMAT

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

- 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

```
x <- x[ -1, 2:5 ]
x <- log( abs( x ) + 1 )
x.pca <- prcomp( x )
xtx <- t( x ) %*% x
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

```
> showClass( "ddmatrix" )
Class "ddmatrix" [package "pbdDMAT"]
Slots:
Name:      Data       dim      ldim     bldim     ICTXT
Class:   matrix   numeric   numeric   numeric   numeric
```

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

Generate a Matrix and Distribute

code/gen–dist.r

```
suppressPackageStartupMessages( library( pbdDMAT ) )
init.grid()

## Construct on 0 and convert to distributed (communicates)
if ( comm.rank() == 0 ){
  x <- matrix( 1:100, nrow = 10, ncol = 10 )
} else {
  x <- NULL
}
x.dmat <- as.ddmatrix( x, bldim = c( 2, 2 ) )

x.dmat
comm.print( submatrix( x.dmat ), all.rank = TRUE )

## More conveniently (no communication)
y.dmat <- ddmatrix( 1:100, nrow = 10, ncol = 10, bldim = c( 2, 2 ) )

all.equal( x.dmat, y.dmat )

## Some ddmatrix defaults
comm.cat(".pbd_env$BLDIM =", .pbd_env$BLDIM, "\n")
comm.cat(".pbd_env$ICTXT =", .pbd_env$ICTXT, "\n")

finalize()
```

Generate Identity or Zero ddmatrix

code/gen–01.r

```
library( pbdDMAT, quiet = TRUE )
init.grid()

zero.dmat <- ddmatrix( 0, nrow = 10, ncol = 10 )
zero.dmat

id.dmat <- diag( 1, nrow = 10, ncol = 10, type = "ddmatrix" )
id.dmat

comm.print( submatrix( id.dmat ), all.rank = TRUE )

diag.dmat <- diag( 1:5, nrow = 10, ncol = 10, type = "ddmatrix" )
diag.dmat

comm.print( submatrix( diag.dmat ), all.rank = TRUE )

comm.set.seed( seed = 1234567, diff = TRUE )
rand.dmat <- ddmatrix( "rnorm", nrow = 100, ncol = 100, mean = 10, sd =
    100 )
rand.dmat

finalize()
```

Distributed Construction of a ddmatrix

code/ddmatrix.r

```
suppressPackageStartupMessages( library( pbdDMAT ) )
init.grid()

## construct local piece of a row-block distributed matrix
x <- matrix( 100*comm.rank() + 1:(3*5), ncol = 5 )
comm.print( x, all.rank = TRUE )

## get correct row-block dimensions and glue into ddmatrix
gdim <- c( allreduce( nrow( x ) ), ncol( x ) )
bd <- c( allreduce( nrow( x ), op = "max" ), ncol( x ) )
xnew <- new( "ddmatrix", Data = x, dim = gdim, bldim = bd, ldim = dim( x
    ), ICTXT = 2 )
print( xnew )
comm.print( submatrix( xnew ), all.rank = TRUE )

xt.cb <- as.colblock( xnew )
print( xt.cb )
comm.print( submatrix( xt.cb ), all.rank = TRUE )

xt.bc <- as.blockcyclic( xnew, bldim = c( 2, 2 ) )
print( xt.bc )
comm.print( submatrix( xt.bc ), all.rank = TRUE )

xt <- as.rowblock( t( xnew ) )
print( xt )
comm.print( submatrix( xt ), all.rank = TRUE )
finalize()
```


Extract Rows and Columns of a ddmatrix

code/extract.r

```
library( pbdDMAT, quiet = TRUE )
init.grid()

.BLDIM <- c( 1, 1 )

x.dmat <- ddmatrix( 1:30, nrow = 10 )
y.dmat <- x.dmat[ c( 1, 3, 5, 7, 9 ), -3 ]

comm.print( submatrix( y.dmat ), all.rank = TRUE )

comm.print( y.dmat )
y <- as.matrix( y.dmat )
comm.print( y )

finalize()
```

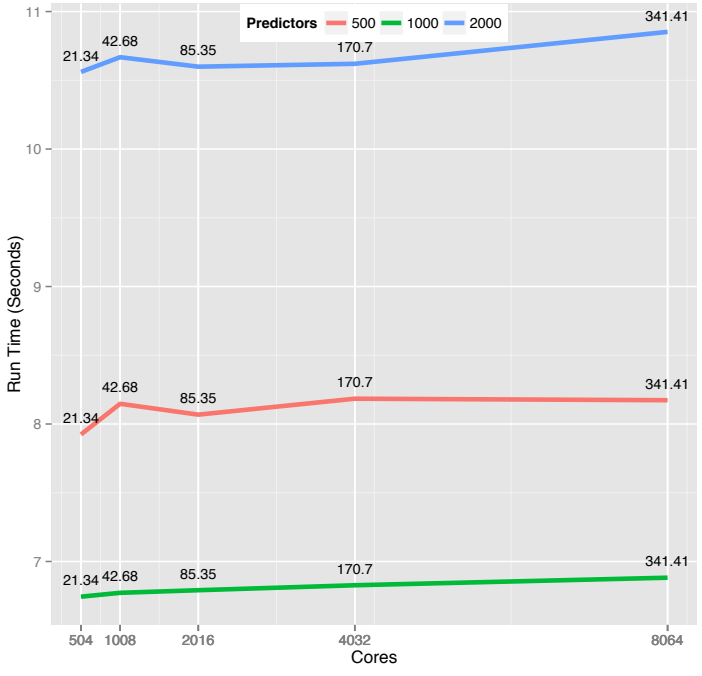
apply() functions on a ddmatrix

ddmatirx.r

```
suppressPackageStartupMessages( library( pbdDMAT ) )
init.grid()

x.dmat <- ddmatrix( "rnorm", nrow = 10, ncol = 3 )
comm.print( x.dmat )

colsd <- apply( X = x.dmat, FUN = sd, MARGIN = 2 )
colsd.loc <- as.matrix( colsd )

comm.print( colsd )
comm.print( colsd.loc )

finalize ()
```

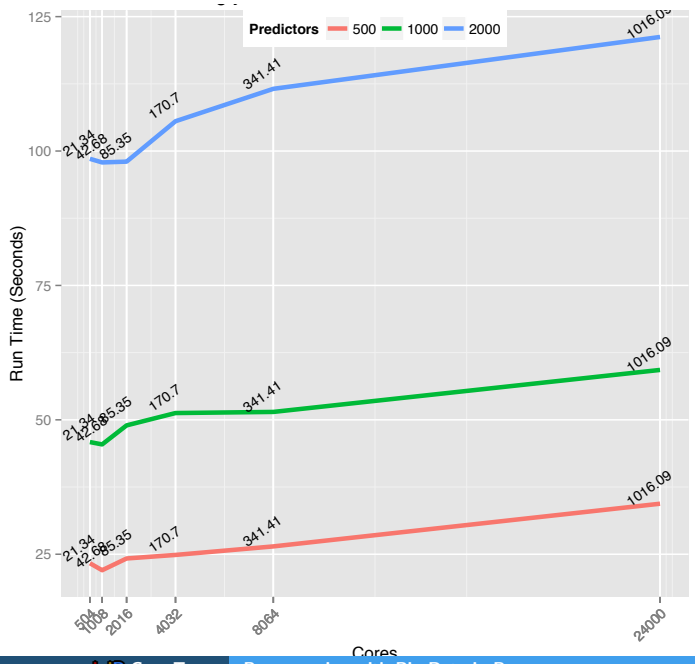

Covariance

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



Regression

```
beta.true <- ddmatrix("runif", nrow=p, ncol=1)
y <- x %*% beta.true
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

code/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 projections[1]

PROTOTYPE FOR RANDOMIZED SVD

Given an $m \times n$ matrix $\boldsymbol{A}$, 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 $\boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is nonnegative and diagonal.

Stage A:

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

Stage B:

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

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

ALGORITHM 4.4: RANDOMIZED SUBSPACE ITERATION

Given an $m \times n$ matrix $\boldsymbol{A}$ and integers ℓ and q, this algorithm computes an $m \times \ell$ orthonormal matrix $\boldsymbol{Q}$ whose range approximates the range of $\boldsymbol{A}$.

1 Draw an $n \times \ell$ standard Gaussian matrix $\boldsymbol{\Omega}$.
2 Form $\boldsymbol{Y}_0 = \boldsymbol{A\Omega}$ and compute its QR factorization $\boldsymbol{Y}_0 = \boldsymbol{Q}_0\boldsymbol{R}_0$.
3 **for** $j = 1, 2, \ldots, q$
4 Form $\tilde{\boldsymbol{Y}}_j = \boldsymbol{A}^*\boldsymbol{Q}_{j-1}$ and compute its QR factorization $\tilde{\boldsymbol{Y}}_j = \tilde{\boldsymbol{Q}}_j\tilde{\boldsymbol{R}}_j$.
5 Form $\boldsymbol{Y}_j = \boldsymbol{A}\tilde{\boldsymbol{Q}}_j$ and compute its QR factorization $\boldsymbol{Y}_j = \boldsymbol{Q}_j\boldsymbol{R}_j$.
6 **end**
7 $\boldsymbol{Q} = \boldsymbol{Q}_q$.

Serial R

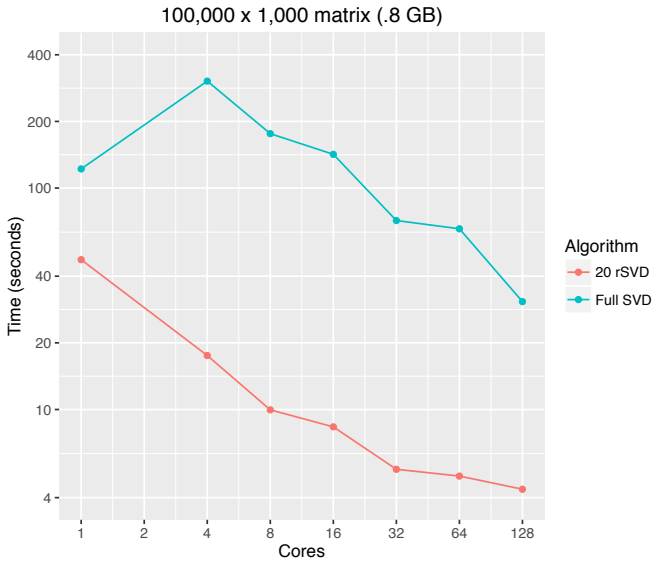
```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <- matrix(rnorm(n*2*k),
                 nrow=n, ncol=2*k)
5      Y <- A %*% Omega
6      Q <- qr.Q(qr(Y))
7      At <- t(A)
8      for(i in 1:q)
9        {
10         Y <- At %*% Q
11         Q <- qr.Q(qr(Y))
12         Y <- 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   }
```

[1] Halko, 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

```
randSVD <- function(A, k, q=3)
  {
    ## Stage A
    Omega <- ddmatrix("rnorm", 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]
  }
```



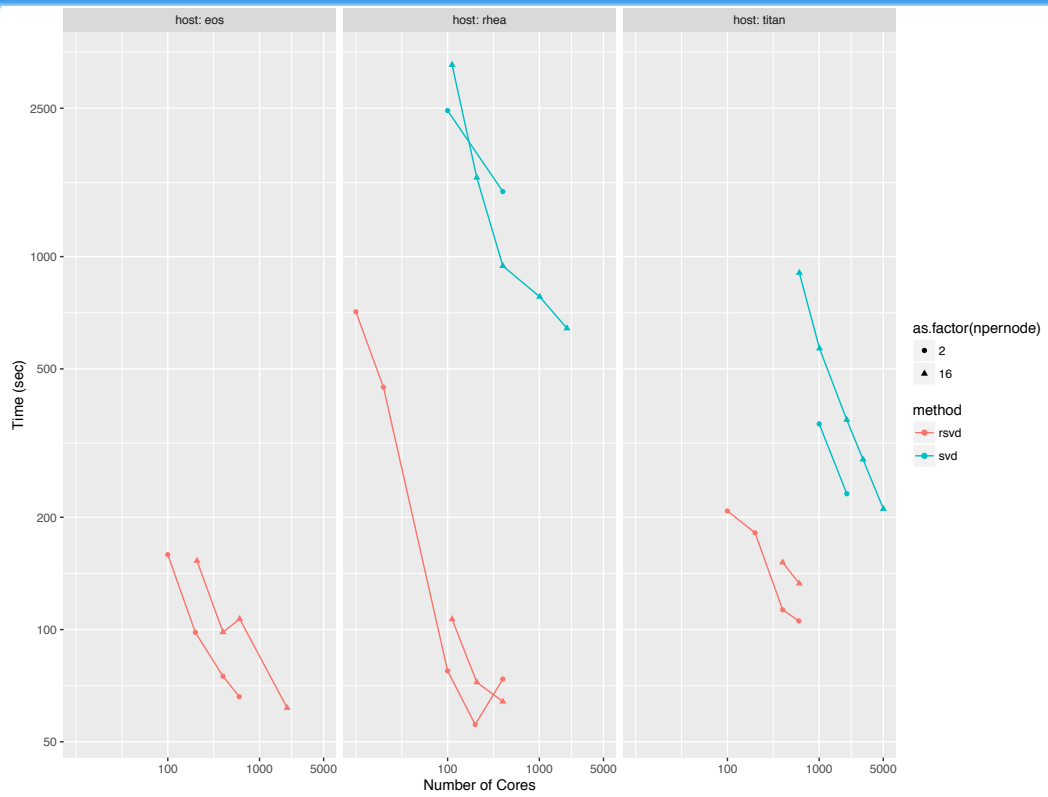
Big Benchmark (134 GB) on Big Clusters

rpca centers, optionally scales, and uses rsvd

fastpca.R

```
suppressPackageStartupMessages(library(rhdf5))
suppressPackageStartupMessages(library(pbdIO))
suppressPackageStartupMessages(library(pbdML))
init.grid( )

var  <- "your_hdf5_full_variable_name"
h5f <- H5Fopen( "your_hdf5_file_name" )
h5d <- H5Dopen( h5f, var )
h5s <- H5Dget_space( h5d )
dims <- H5Sget_simple_extent_dims( h5s )$size
rows <- dims[2]    # row-major to column-major
cols <- dims[1]    # because data written by C/Py

## read C/Py-written blocks of rows into R blocks of columns
my_rows <- comm.chunk(rows, form="vector", type="equal")
A <- t(h5read(h5f, var, index=list(NULL, my_rows)))

## add glue to make a global column-block ddmatrix
A <- new("ddmatrix", Data=A, dim=c(rows, cols), ldim=dim(A), bldim=dim(A), ICTXT=2)

## rearrange into block-cyclic
A <- as.blockcyclic( A )

## get 32 top singular values and vectors
Res <- rpca( A, k = 32 )

## print the singular values
comm.print( Res$d )

finalize()
```

Big Benchmark (134 GB) on Big Clusters



Big Benchmark (134 GB) on HPC Wire

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 Ostrouchov[1] and Wei-Chen Chen[2]

[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

pbdZMQ + 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, ...

pbdZMQ + remoter + pbdCS

- 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.