Parallel Programming with Big Data in R: pbdR

George Ostrouchov
University of Tennessee and Oak Ridge National Laboratory

International Statistical Institute 60^{th} World Statistics Congress
July 2015, Rio de Janeiro, Brazil

The pbdR Core Team

Wei-Chen Chen[1]
George Ostrouchov[2,3]
Pragneshkumar Patel[3]
Drew Schmidt[3]

[1]FDA
Washington, DC, USA

[2]Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge TN, USA

[3]Joint Institute for Computational Sciences
University of Tennessee, Knoxville TN, USA

Support

This material is based upon work supported by the National Science Foundation Division of Mathematical Sciences under Grant No. 1418195. This work 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 under Award No. ARRA-NSF-OCI-0906324 for NICS-RDAV center. This work also 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.

About This Presentation

This Presentation is available at

Installation Instructions

Installation instructions for setting up a pbdR environment are available:

http://r-pbd.org/install.html

This includes instructions for installing R, MPI, and pbdR.

Windows binaries for pbdR are available at

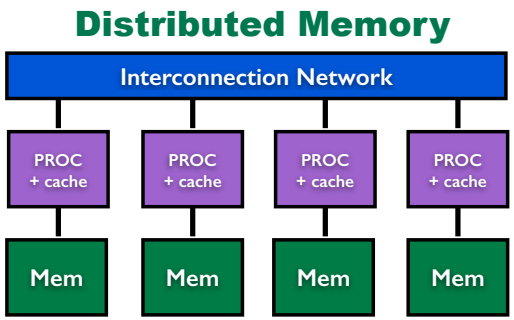
http://thirteen-01.stat.iastate.edu/snoweye/pbdr/?item=download

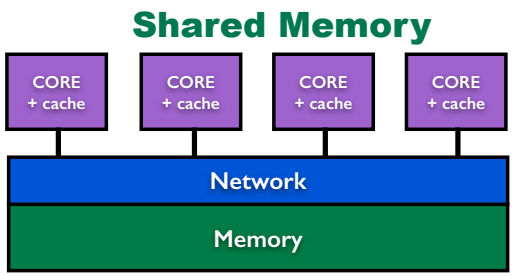
Example scripts are available at:

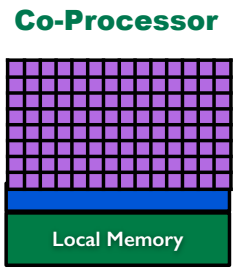
Contents

Contents

Three Basic Flavors of Hardware



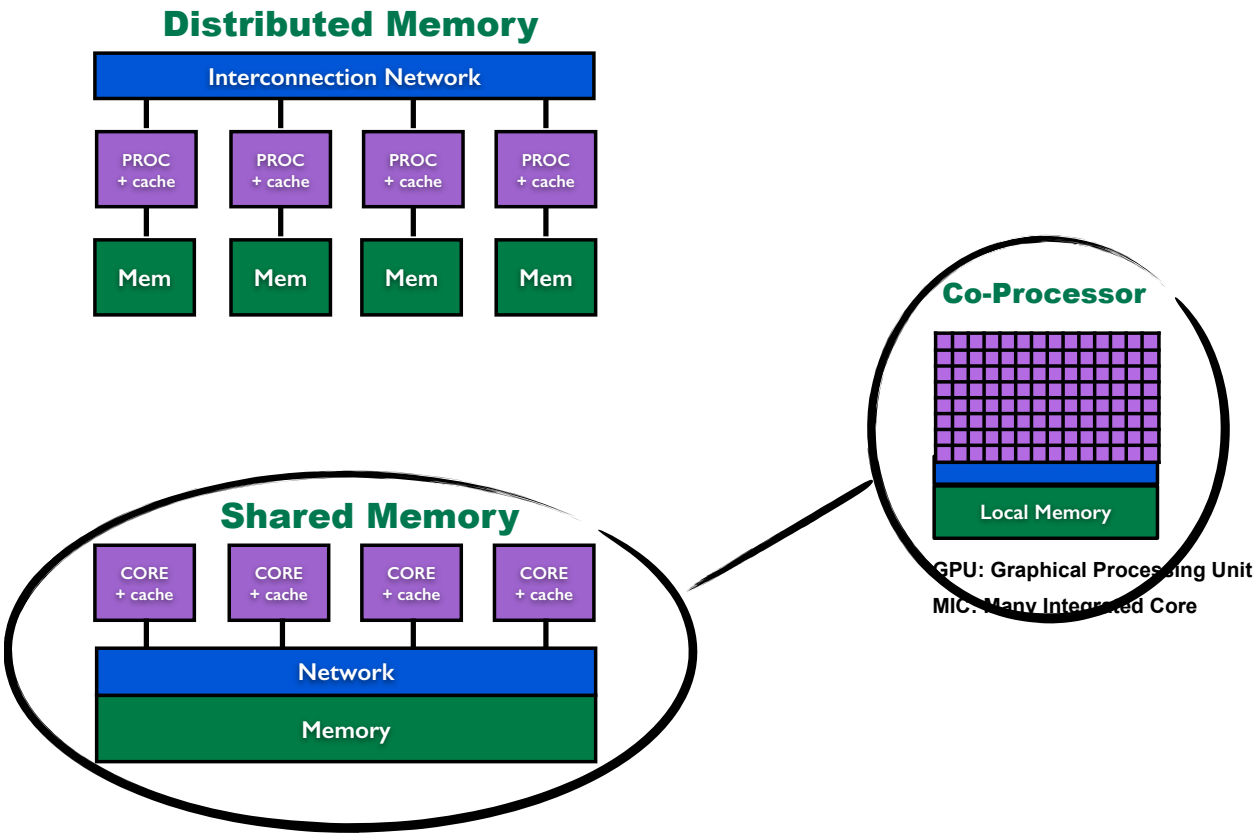




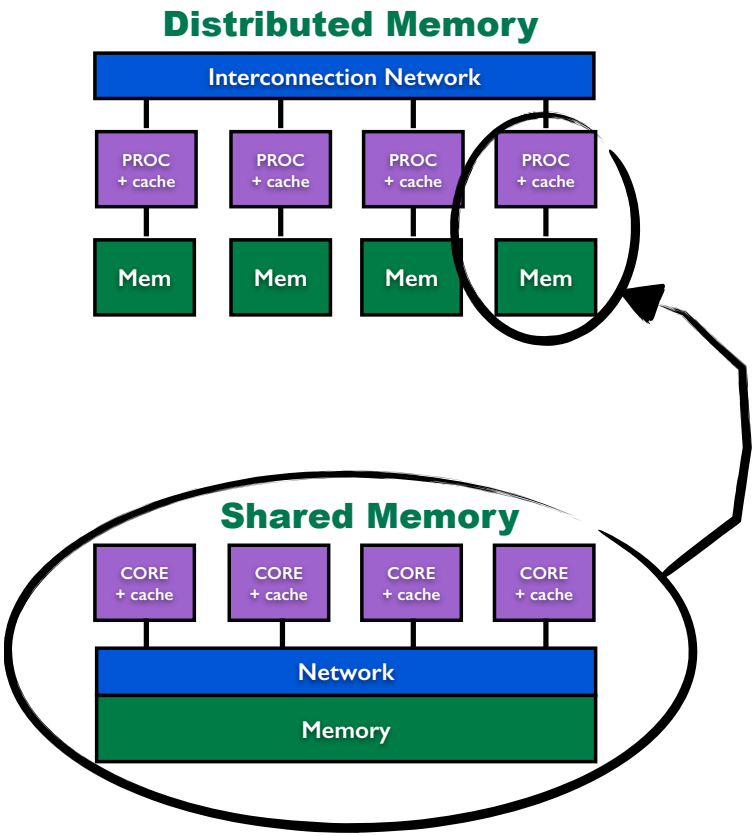
GPU: Graphical Processing Unit

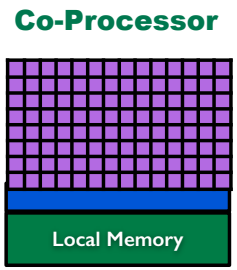
MIC: Many Integrated Core

Your Laptop or Desktop



A Server or Cluster

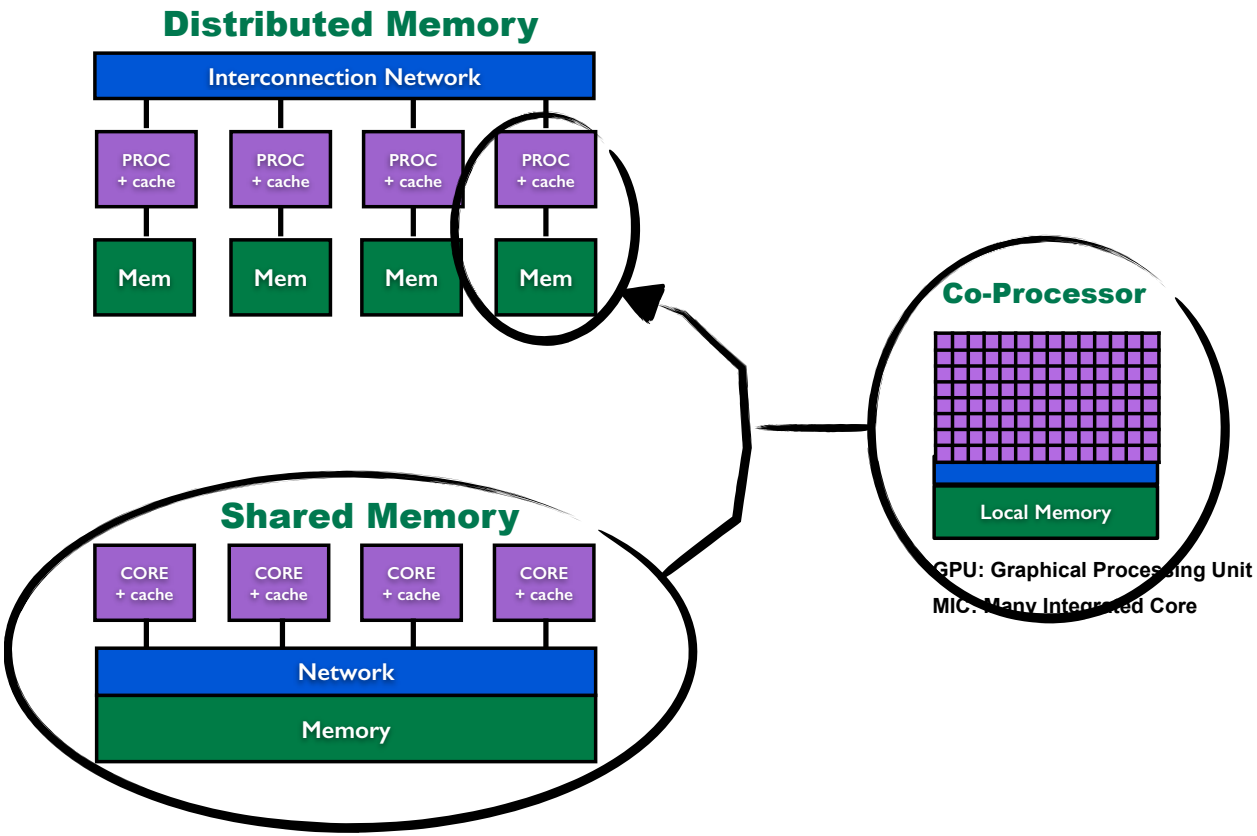




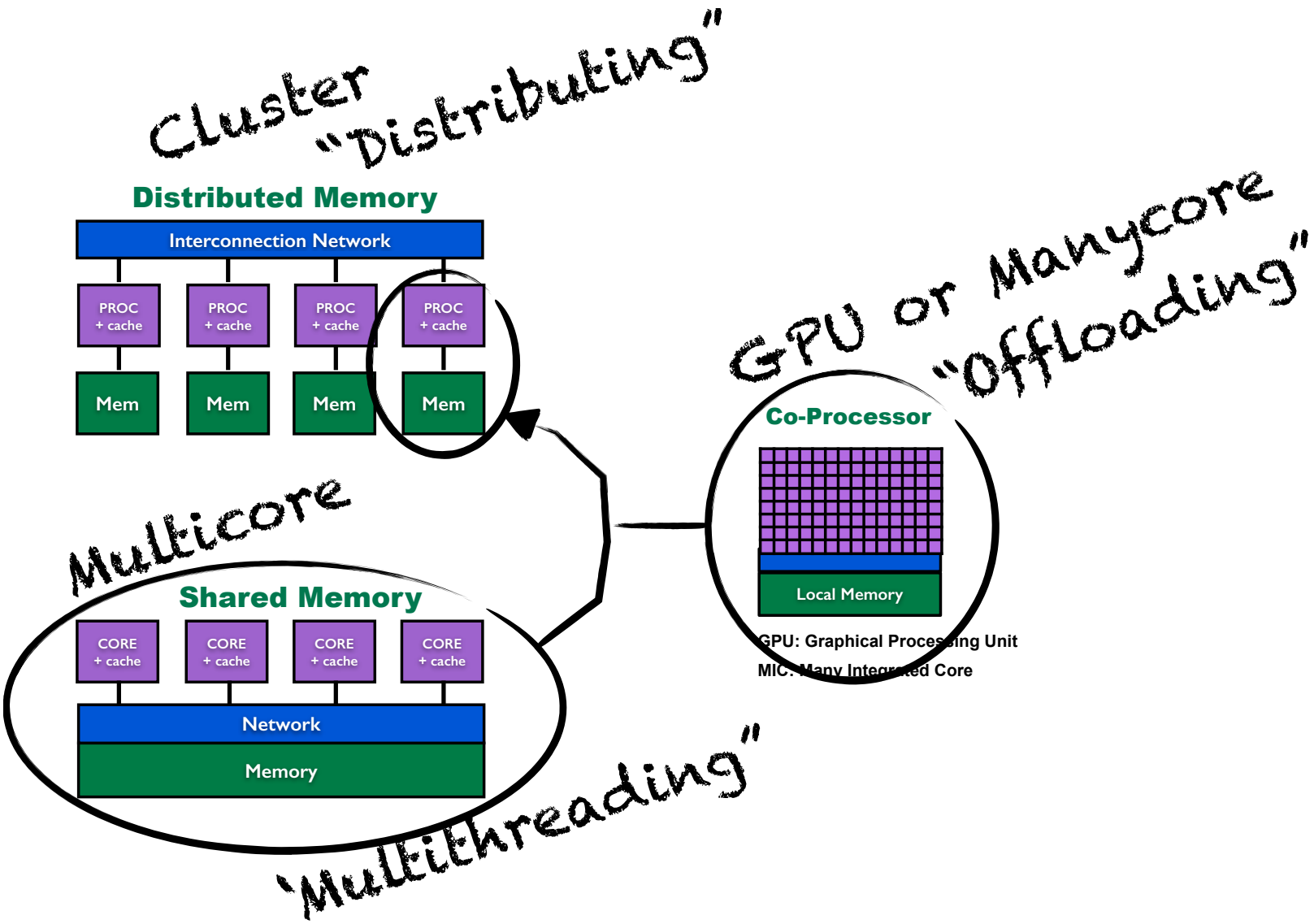
GPU: Graphical Processing Unit

MIC: Many Integrated Core

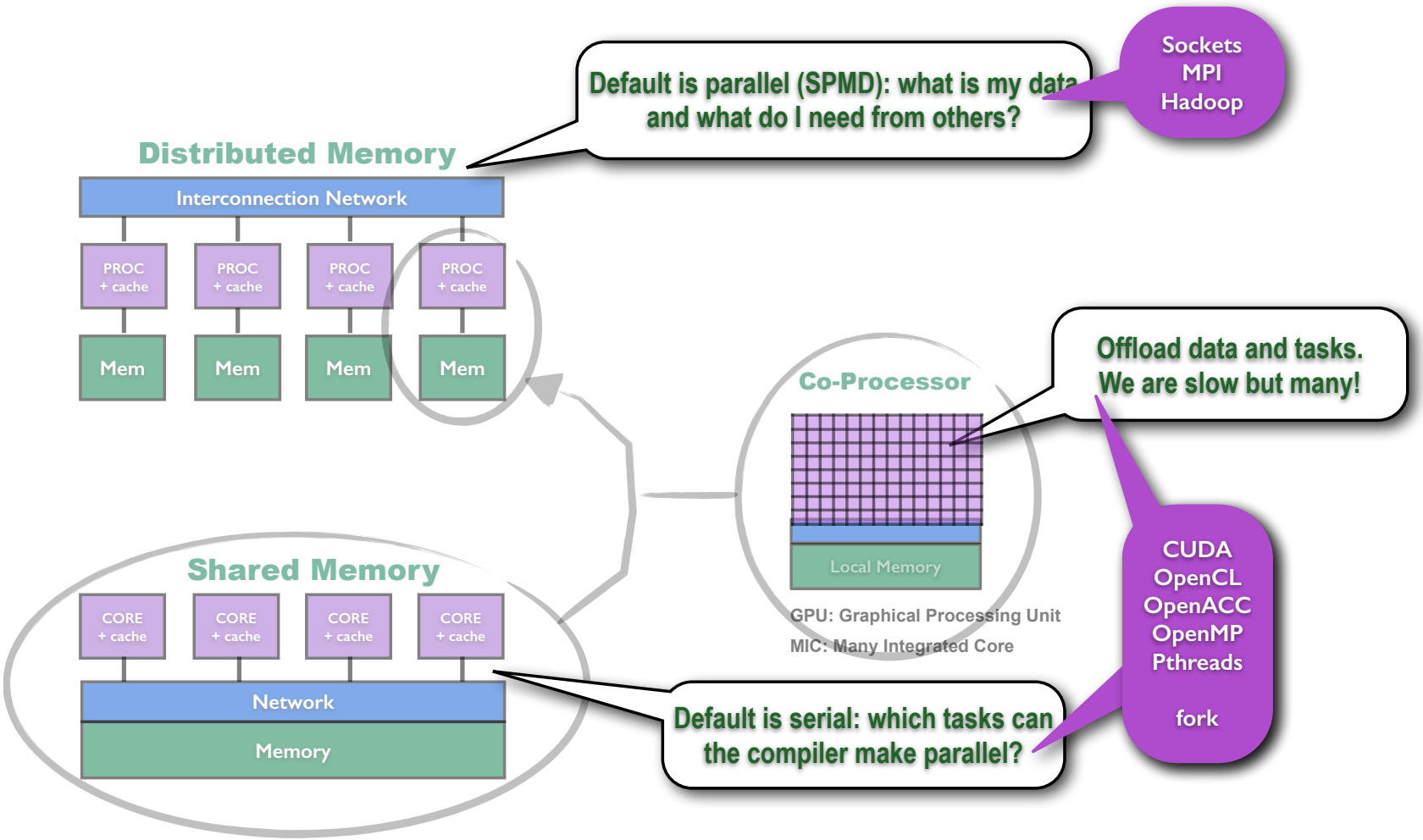
Server to Supercomputer



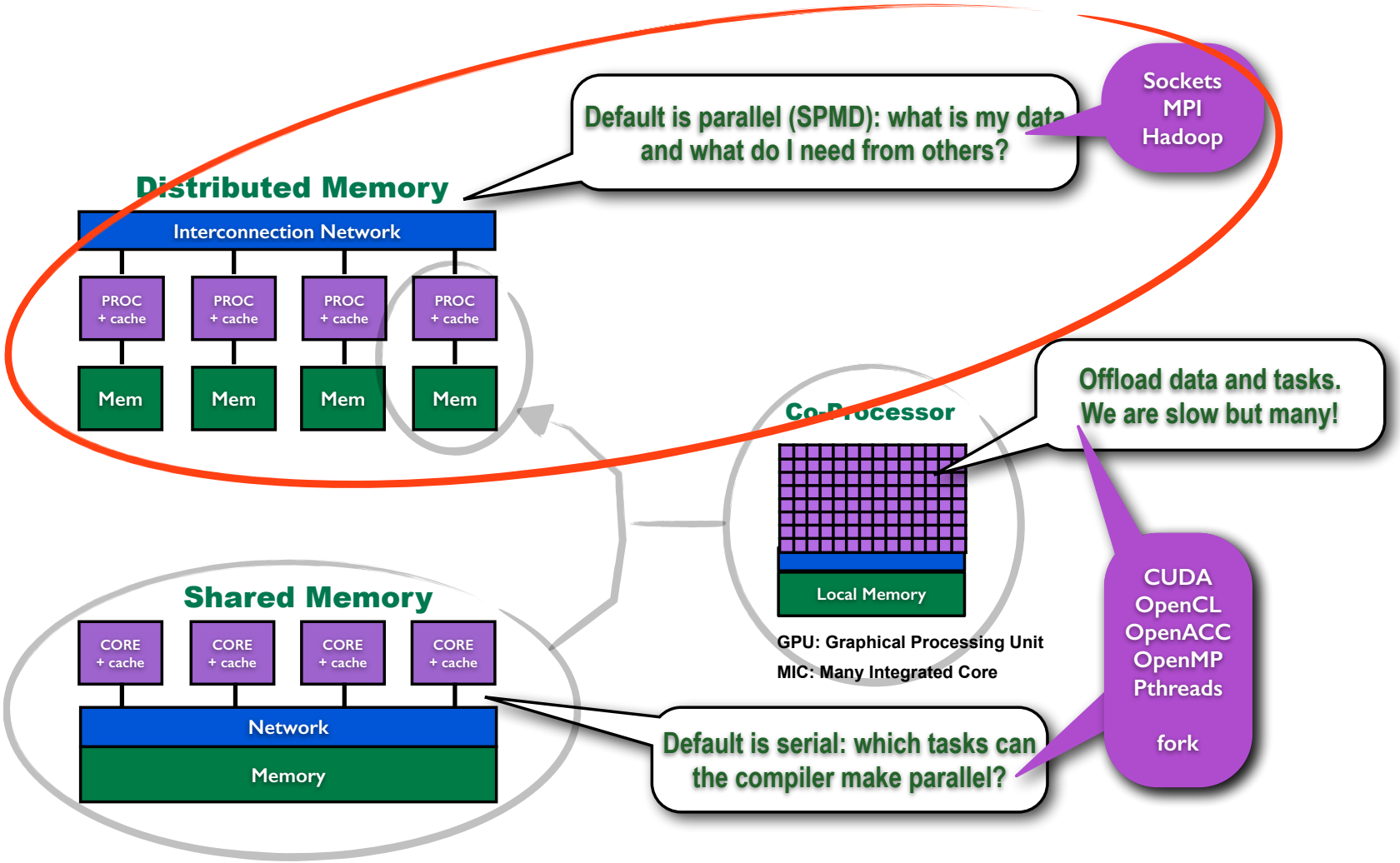
Knowing the Right Words



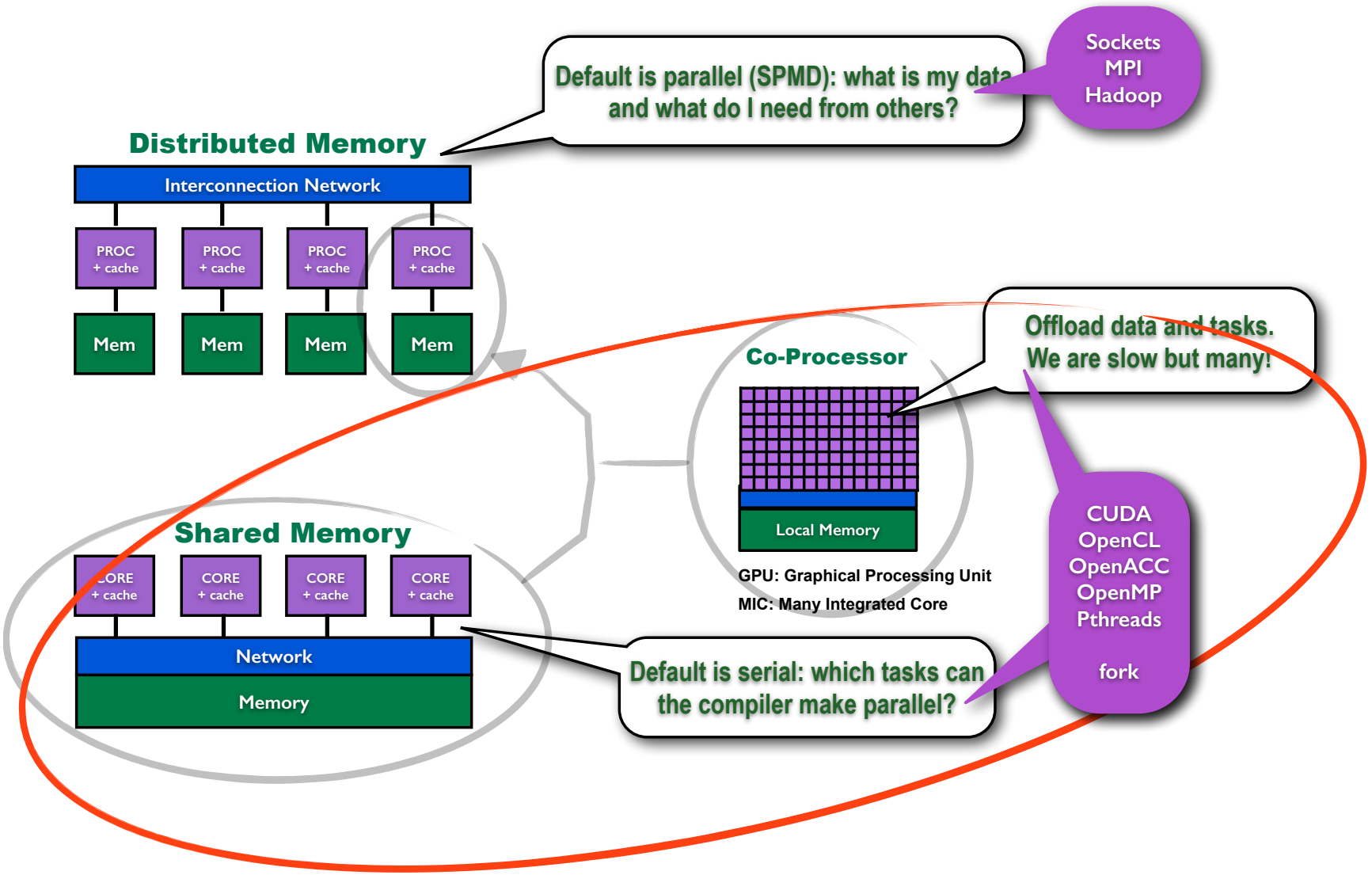
"Native" Programming Models and Tools



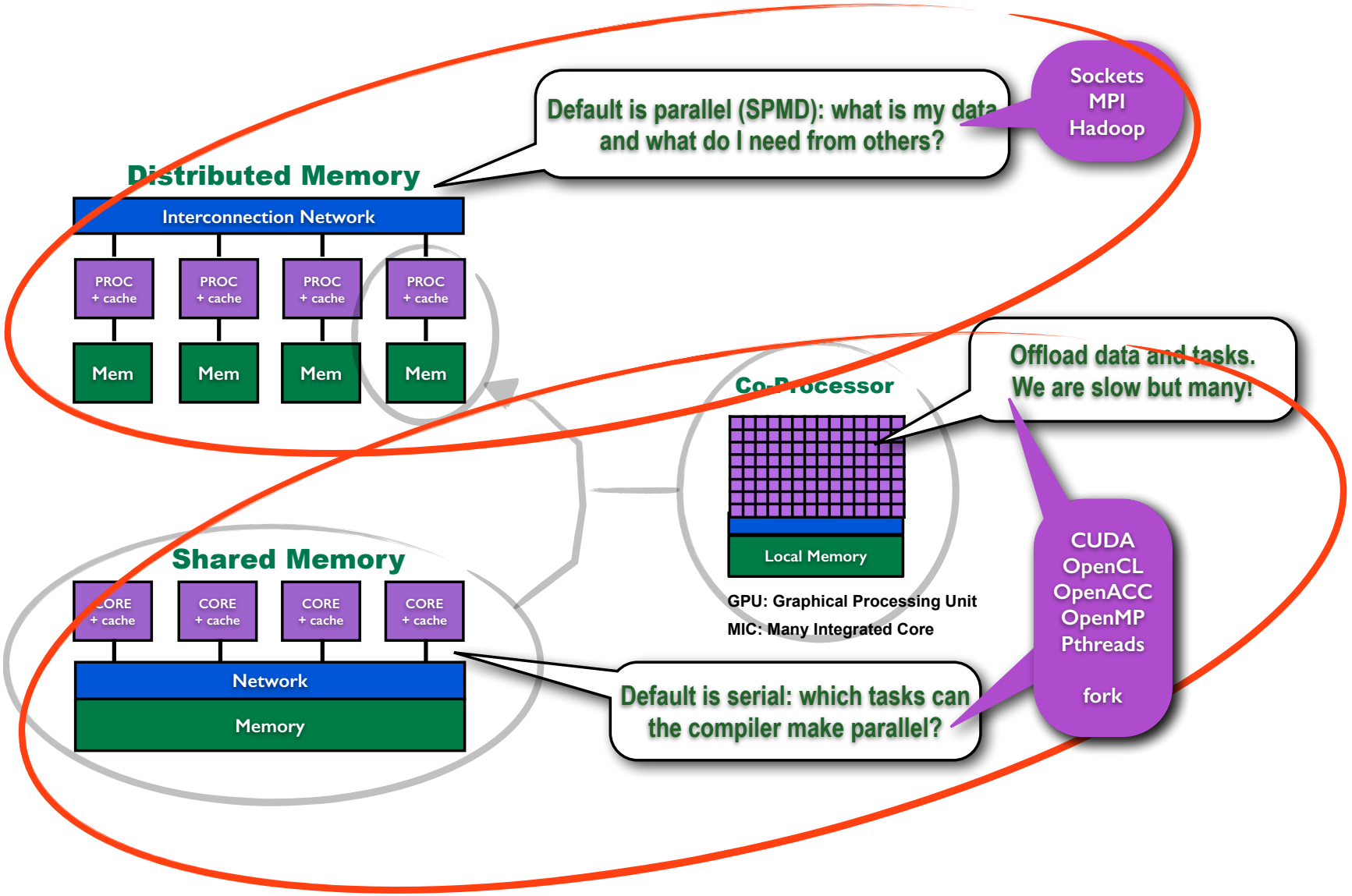
30+ Years of Parallel Computing Research



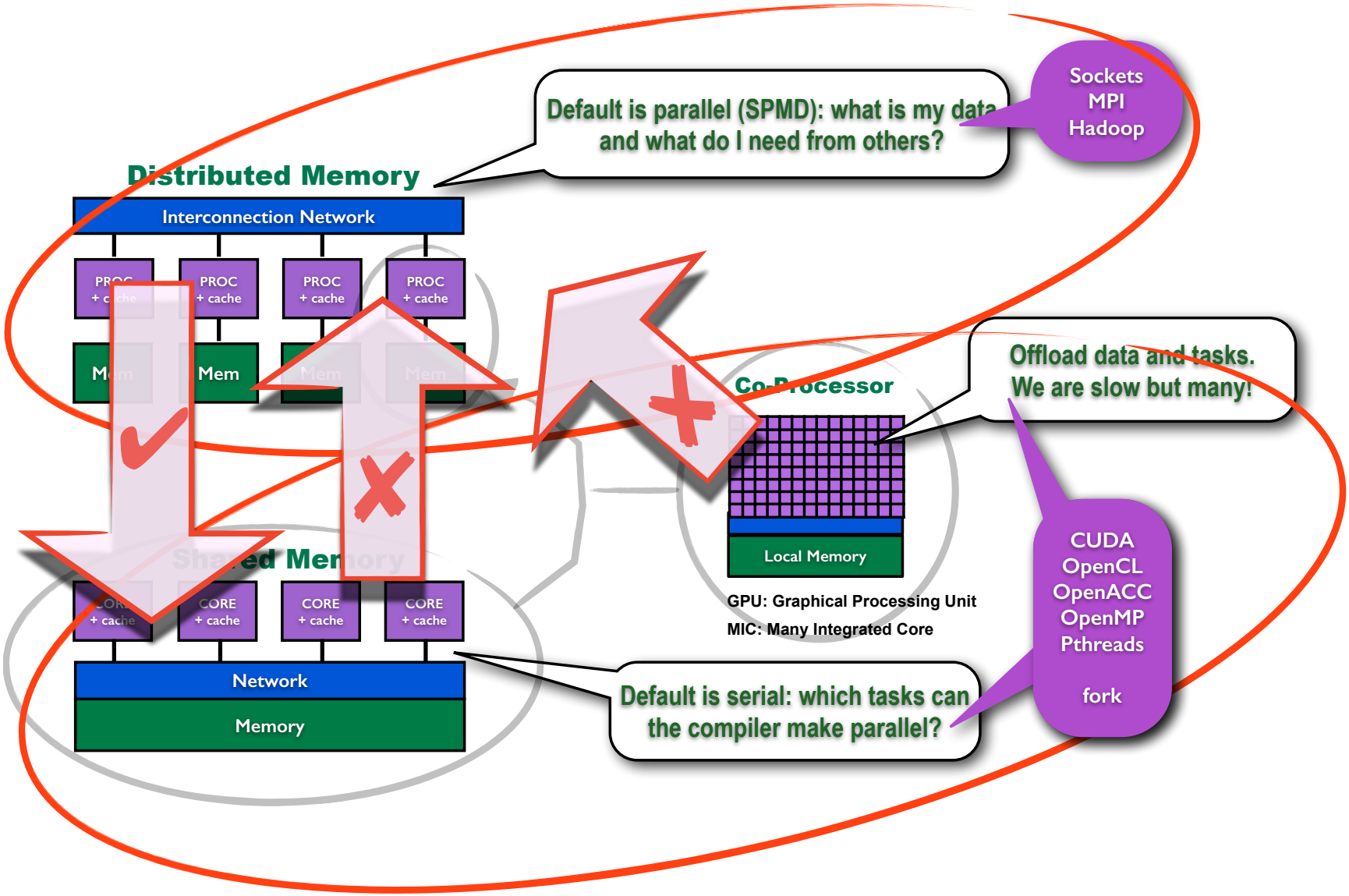
Last 10 years of Advances



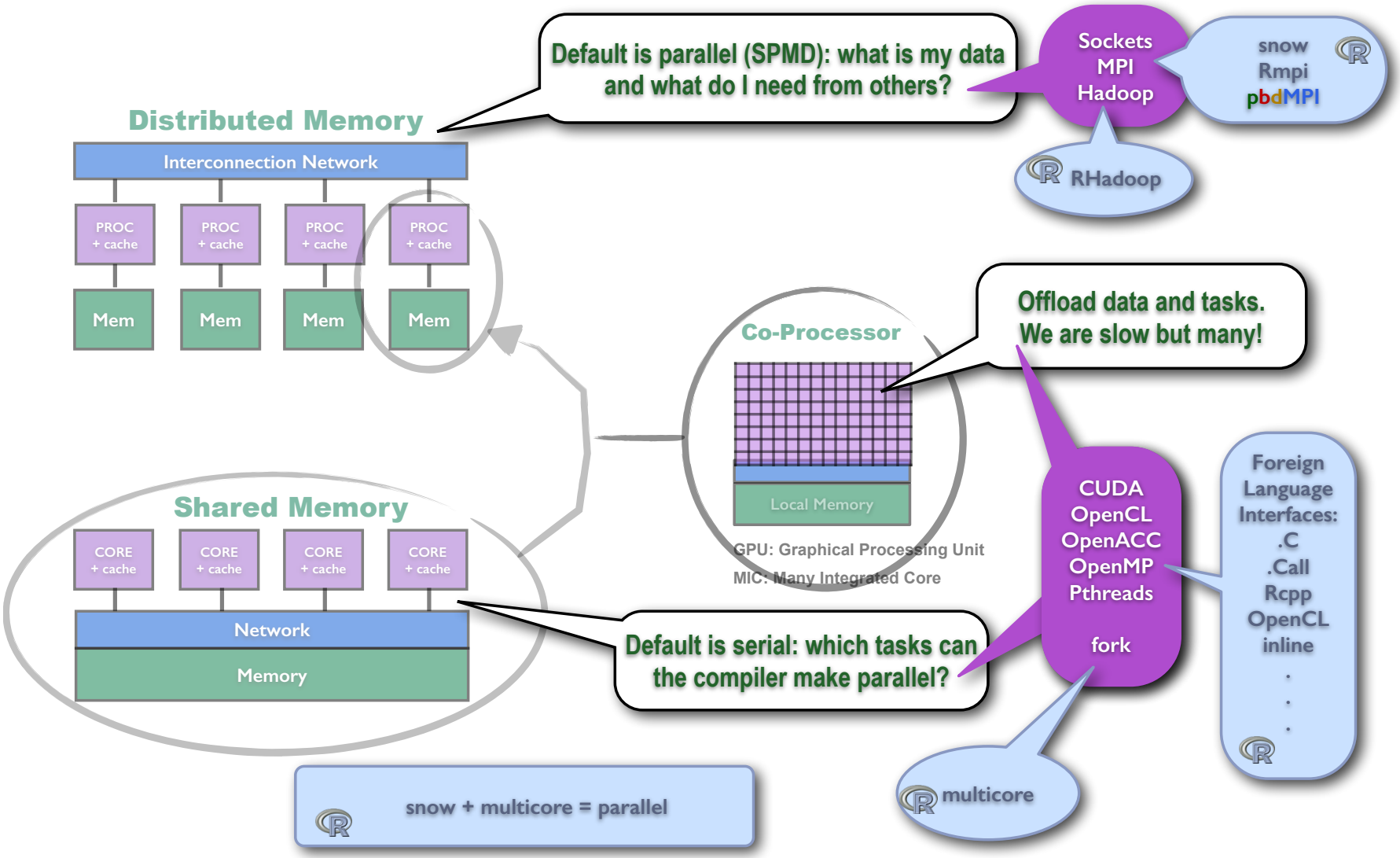
Putting It All Together Challenge



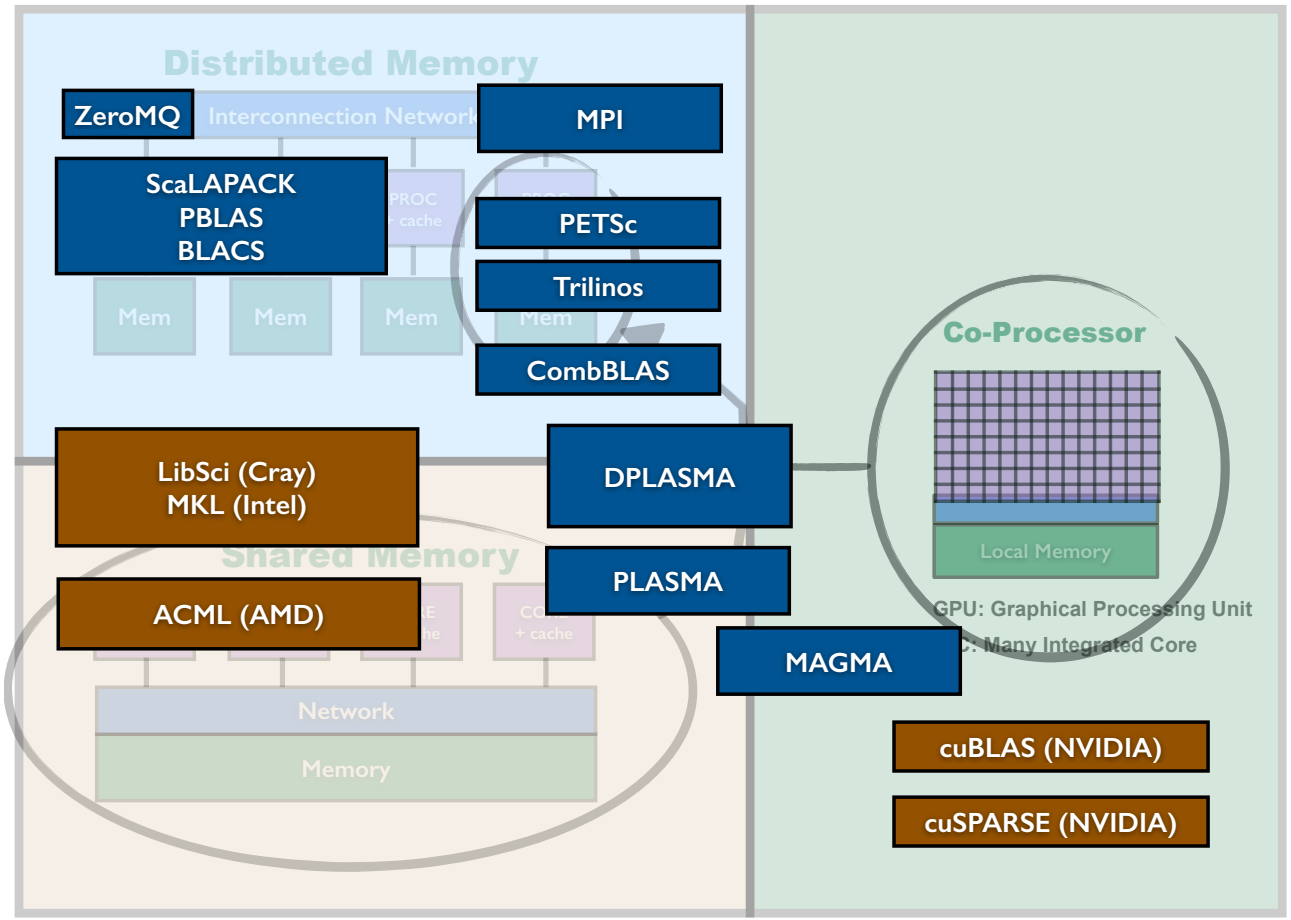
Distributed Programming Works in Shared Memory



R Interfaces to Low-Level Native Tools



Scalable Libraries



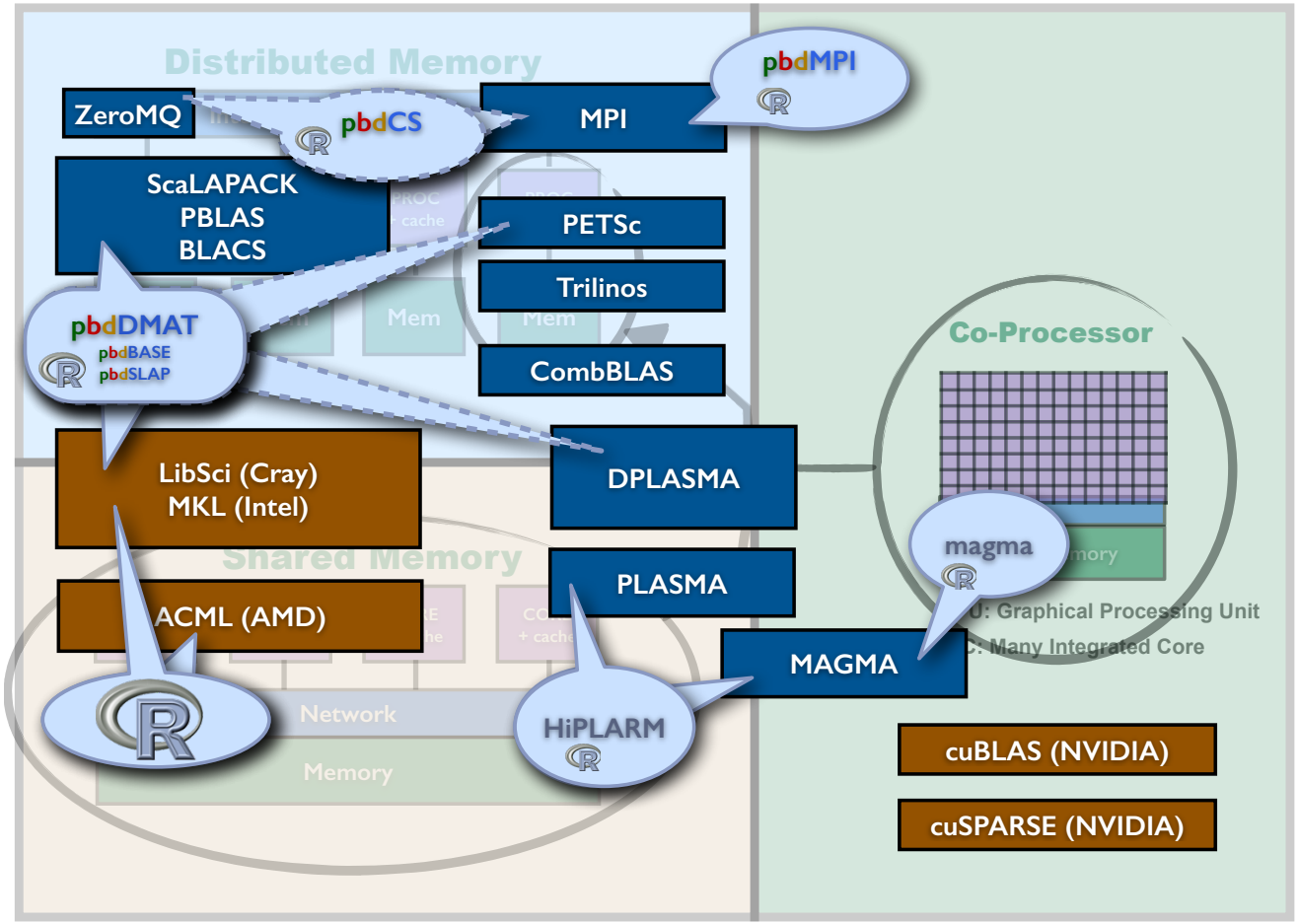
Profiling

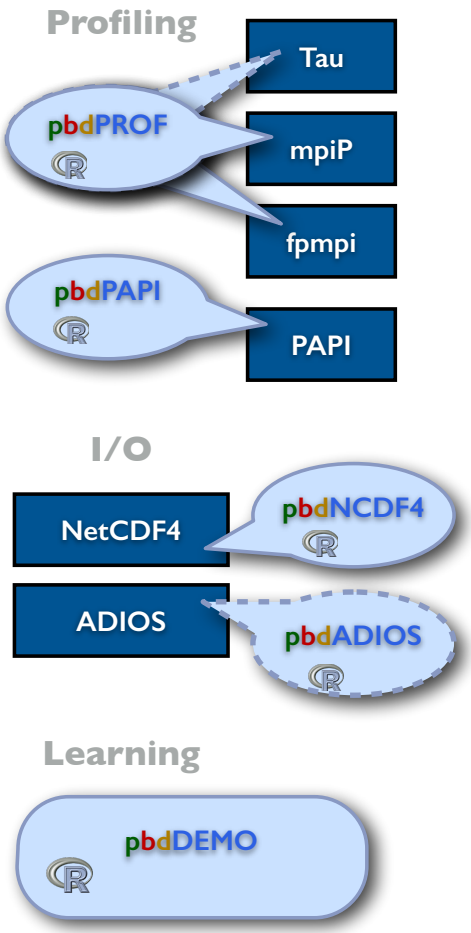
- Tau
- mpiP
- fpmpi
- PAPI

I/O

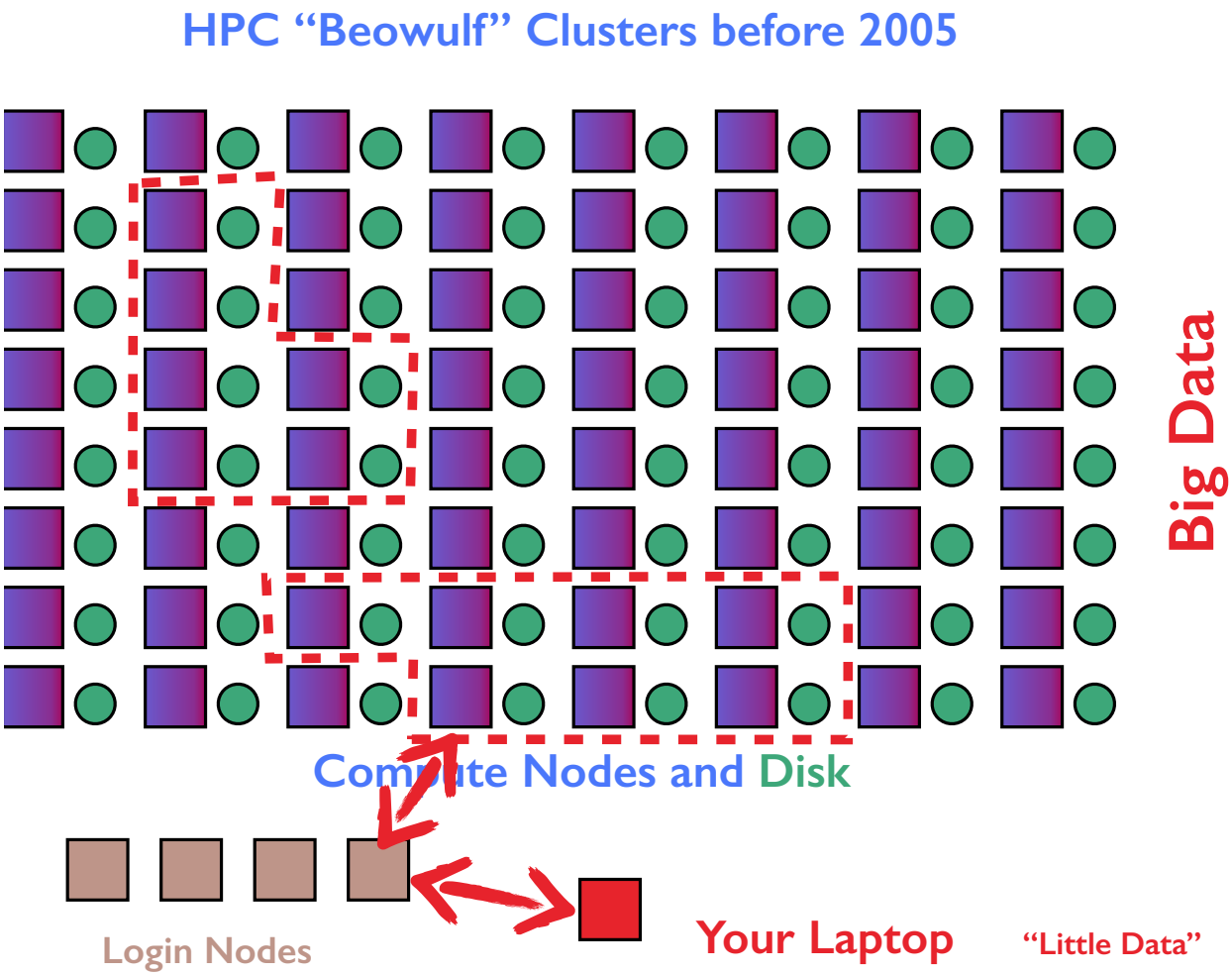
- NetCDF4
- ADIOS

R and pbdR Interfaces to HPC Libraries





Parallel Computing before Multicore



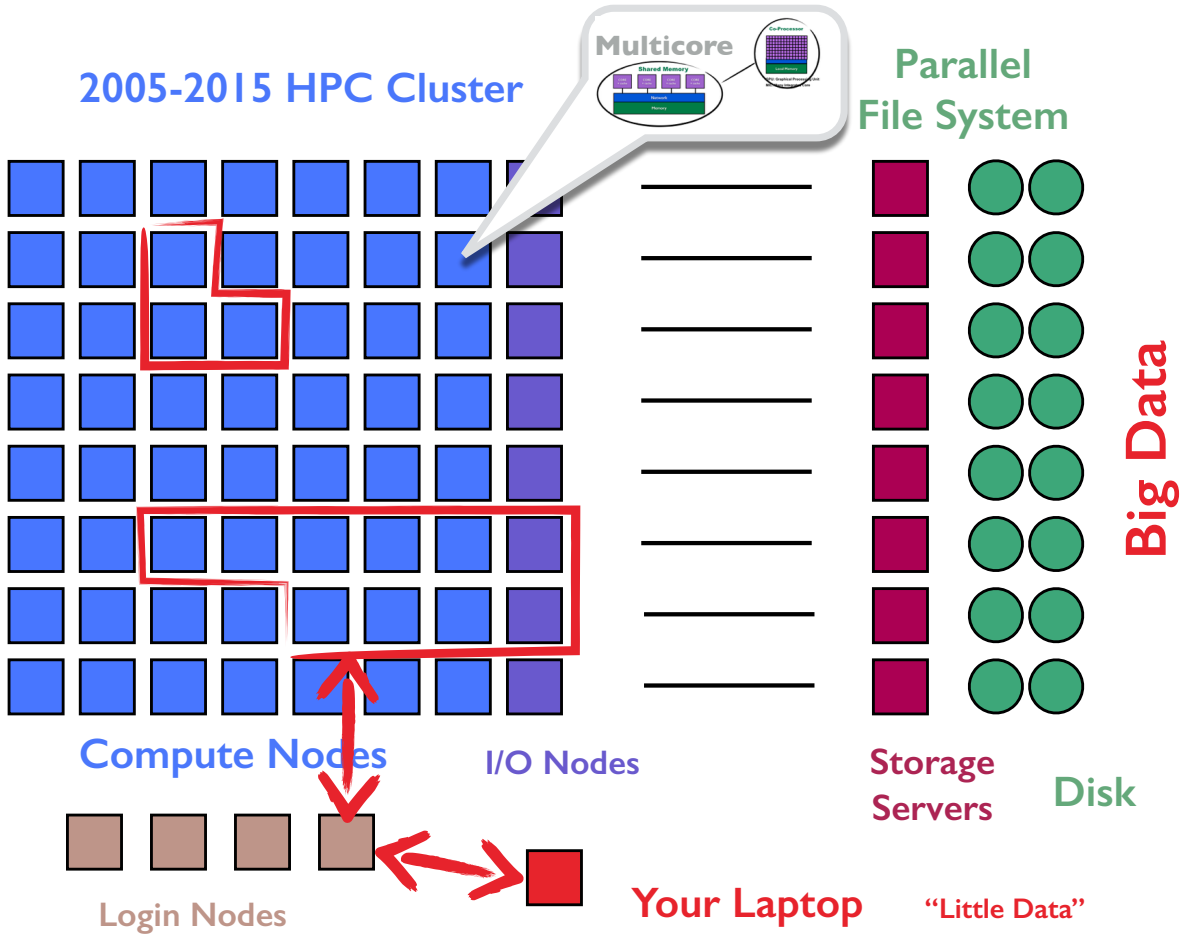
Software Developments:

MPI is mature, Map-Reduce emerges

Parallel Libraries: PBLAS, ScaLAPACK, PETSc, etc.

Resource Manager: PBS mature, HADOOP emerges

Multicore Emerges and Clusters become Diskless

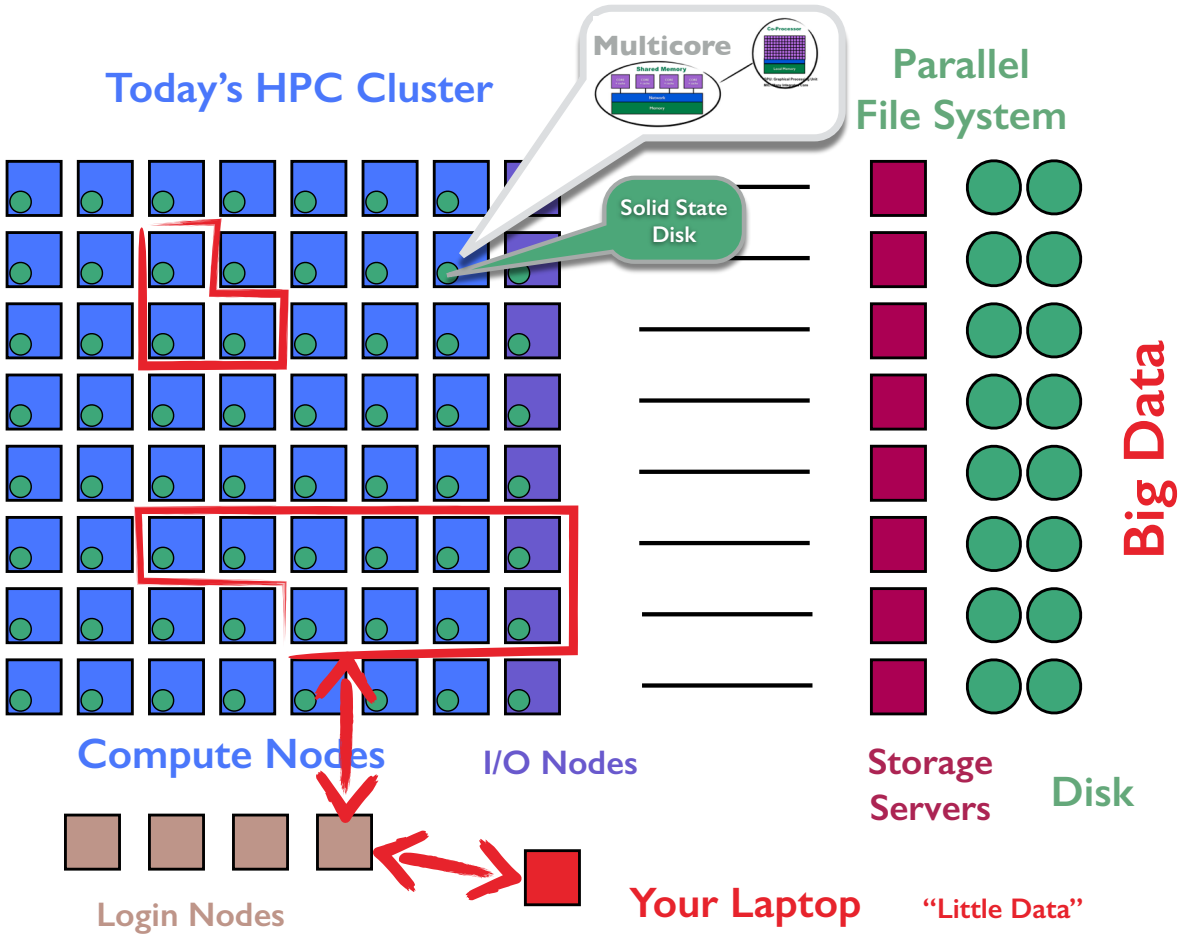


Software Developments

OpenMP, CUDA, OpenCL, OpenACC

Libraries: PLASMA, MAGMA, CuBLAS

Adding NVRAM to New HPC Systems



Software Developments

Libraries: DPLASMA, CombBLAS

HADOOP fades, Spark emerges

Data analysis is interactive!

- Data reduction to knowledge
- Iterative process with same data
  - Exploration, model construction
  - Diagnostics of fit and quantification of uncertainty
  - Interpretation
- S (and R) interactive "answer" to batch data analysis
- Efficient use of expensive people

Big platform computing is batch!

- Libraries built for batch computing
- Traditionally data generation by simulation science
- Efficient use of expensive platforms

High-Level Language: Batch and Interactive Distinction Blurred.

- A function is a "batch" script
- R "An interactive environment to use batch scripts"

Ideal solution: Interactive Client with a Batch Server

- Parallel visualization systems (VisIt and ParaView) are client-server (batch on server)
- Current pbdR packages address server side (batch)
- pbdR client under development
  - SPMD interactive (pbdCS, alpha on GitHub)
  - Bridge laptop to login node to resource manager to cluster
  - Site configuration file
  - Manage relationship of big data (server side) to little data (client side)

Summary

- Three flavors of hardware
  - Distributed is stable
  - Multicore and co-processor are evolving
  - Two memory models
  - Distributed works in multicore, but not the other way around
- Medium to big machines have all three
- HPC Libraries can span hierarchy
- A confusing diversity of parallel R packages exists
- pbdR currently batch, eventually client-server

Contents

Why Profile?

- Because performance matters.
- Bad practices scale up!
- Your bottlenecks may surprise you.
- Because R is dumb.

Compilers often correct bad behavior...

A Really Dumb Loop

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

clang -O3 example.c

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

clang example.c

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

R will not!

Dumb Loop

```
1
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
```

Better Loop

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

Example from 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×** !

Some Thoughts

- R is slow.
- Bad programs are slower.
- High-level language: one line can touch a lot of data
- R will not fix bad programming

Performance Profiling Tools: system.time()

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

```
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()

Rprof() times the execution of all R functions:

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

```
1  x <- matrix(rnorm(10000*250), nrow=10000, ncol=250)
2
3  Rprof()
4  invisible(prcomp(x))
5  Rprof(NULL)
6
7  summaryRprof()
8
9  Rprof(interval=.99)
10 invisible(prcomp(x))
11 Rprof(NULL)
12
13 summaryRprof()
```

Performance Profiling Tools: Rprof()

```
$by.self
                self.time self.pct total.time total.pct
"La.svd"             0.68    69.39       0.72     73.47
"%*%"                0.12    12.24       0.12     12.24
"aperm.default"      0.04     4.08       0.04      4.08
"array"              0.04     4.08       0.04      4.08
"matrix"             0.04     4.08       0.04      4.08
"sweep"              0.02     2.04       0.10     10.20
### output truncated by presenter

$by.total
                 total.time total.pct self.time self.pct
"prcomp"               0.98    100.00       0.00      0.00
"prcomp.default"       0.98    100.00       0.00      0.00
"svd"                  0.76     77.55       0.00      0.00
"La.svd"               0.72     73.47       0.68     69.39
### output truncated by presenter

$sample.interval
[1] 0.02

$sampling.time
[1] 0.98
```

Performance Profiling Tools: Rprof()

```
$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:

```
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
```

2 Profiling and Benchmarking

- Why Profile?
- How to Profile and Benchmark
- Summary

Summary

- *Profile, profile, profile.*
- Use `system.time()` to get a general sense of a method.
- Use **rbenchmark**'s `benchmark()` to compare 2 methods.
- Use `Rprof()` for more detailed profiling.
- Other tools exist for more hardcore applications (**pbdPAPI** and **pbdPROF**).

Contents

Common Terminology

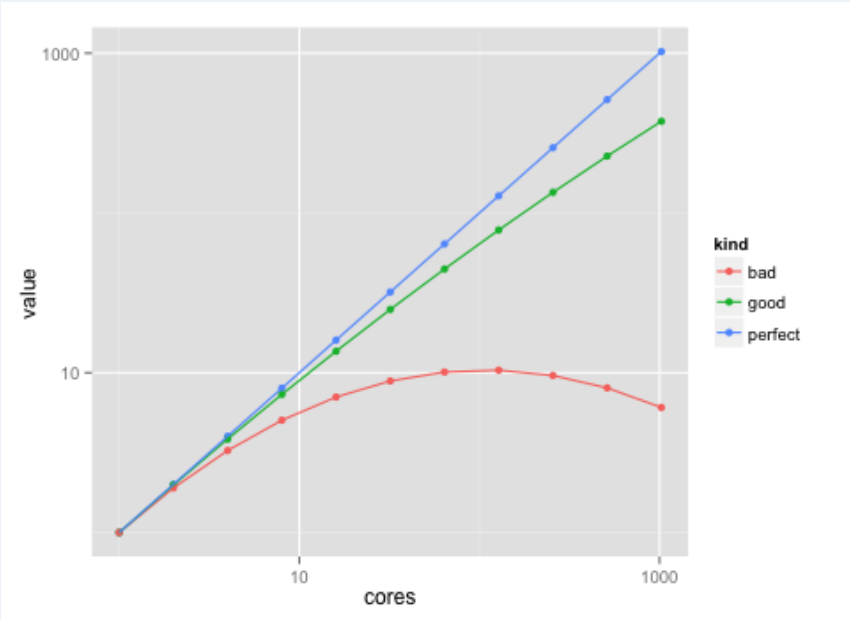
- *Implicit parallelism*: Parallel details hidden from user
- *Explicit parallelism*: Some assembly required...
- *Embarrassingly Parallel*: Also called *loosely coupled*. Obvious how to make parallel; lots of independence in computations.
- *Tightly Coupled*: Opposite of embarrassingly parallel; lots of dependence in computations.
- *Wallclock Time*: Time of the clock on the wall from start to finish
- *Speedup*: unitless measure of improvement; more is better.

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

n_1 is often taken to be 1, comparing parallel algorithm to serial algorithm

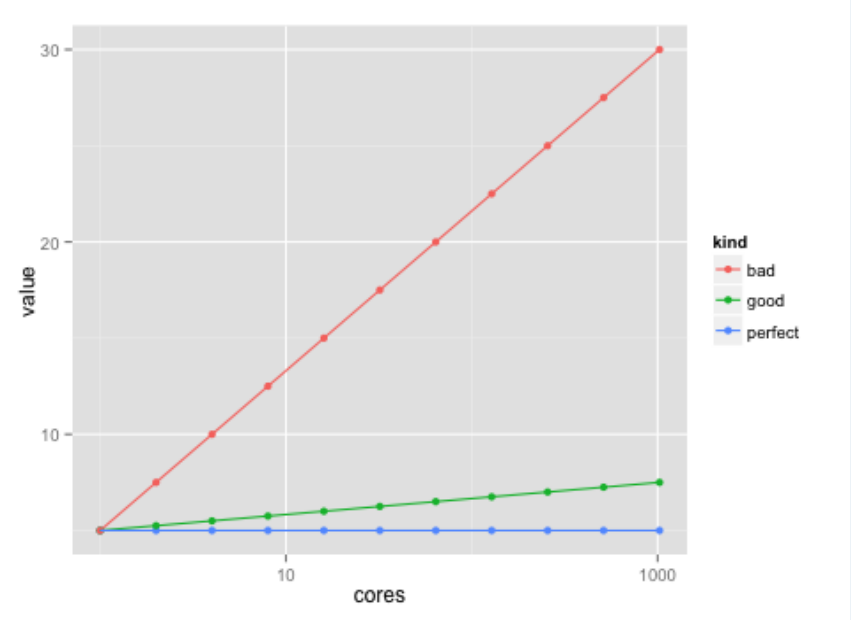
Measuring Parallelism: Scalability

Strong Scaling: Speedup



Fixed **total** problem size.

Weak Scaling: Overhead



Fixed **local** (per core) problem size.

Parallel Programming Paradigms

Mostly Shared Memory (view from serial)

- Manager-Workers (Master-Slaves)
  - One process hands out work and merges results
- Fork-Join
  - One process splits into many then results are joined

Mostly Distributed Memory (view from parallel)

- Map-Reduce (on a virtual matrix)
  - Map = Row operations
  - Reduce = Column operations
  - Shuffle = Transpose (Hidden from user)
- Single Program Multiple Data (SPMD)
  - Many copies of one program run in parallel
  - A 30+ year tradition in HPC
  - Appears harder than it is

Distributed Architectures Enable Larger Scale

Shared Memory Machines

Thousands of cores

Nautilus, University of Tennessee
1024 cores
4 TB RAM

Distributed Memory Machines

Hundreds of thousands of cores

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

Random Number Generators in Parallel

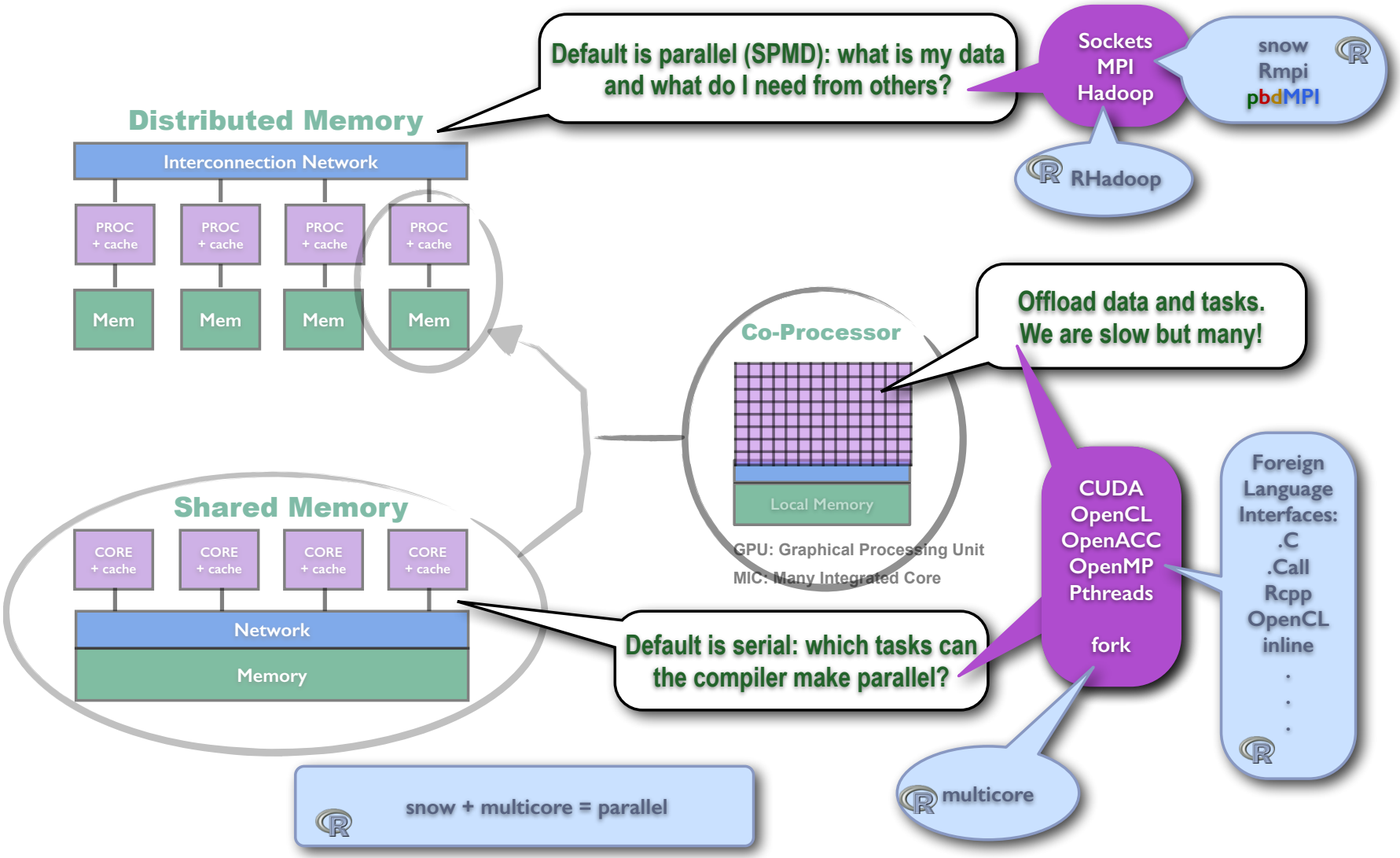
- Guarantee stream independence and reproducibility
- Aided by **rlecuyer**, **rsprng**, and **doRNG** packages.
- Be careful!

Parallel Programming: In Theory

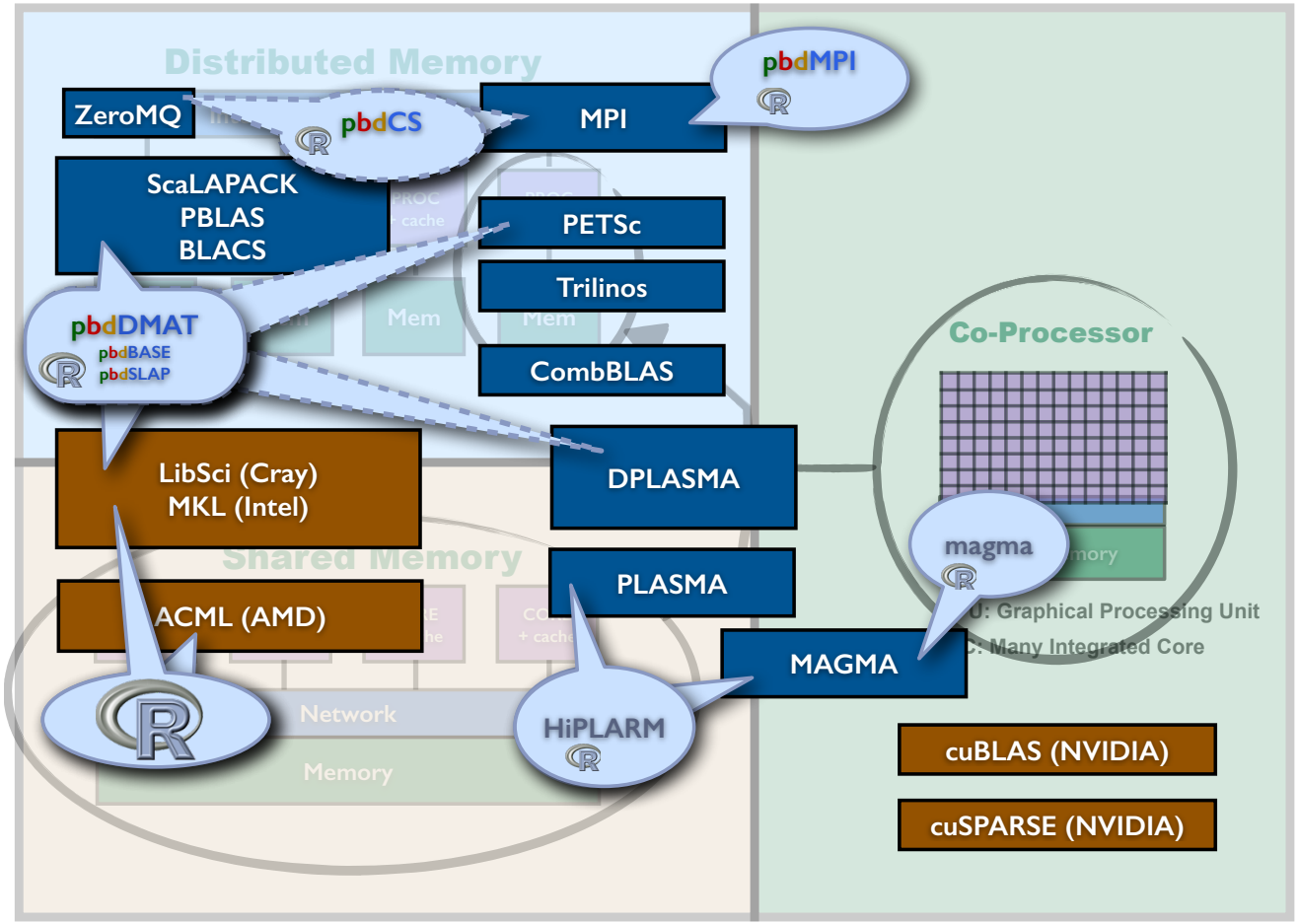
Parallel Programming: In Practice

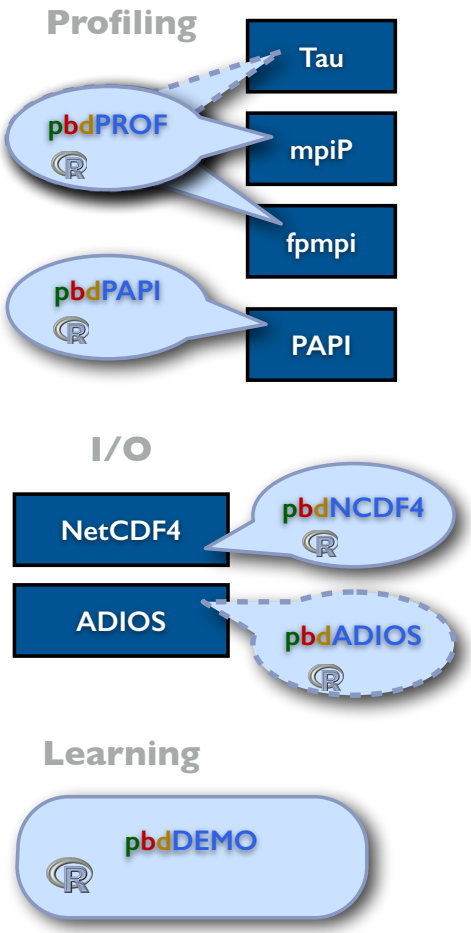
Contents

R Interfaces to Low-Level Native Tools



R and pbdR Interfaces to HPC Libraries





Parallel Programming Packages for R

CRAN HPC Task View

http://cran.r-project.org/web/views/HighPerformanceComputing.html

Shared Memory

parallel, **foreach**

Distributed works in Shared Memory

pbdR: **pbdMPI**, **pbdDMAT** (**pbdSLAP** and **pbdBASE**)

4 Shared Memory Parallel Packages in R

- The parallel Package
- The foreach Package

The parallel Package

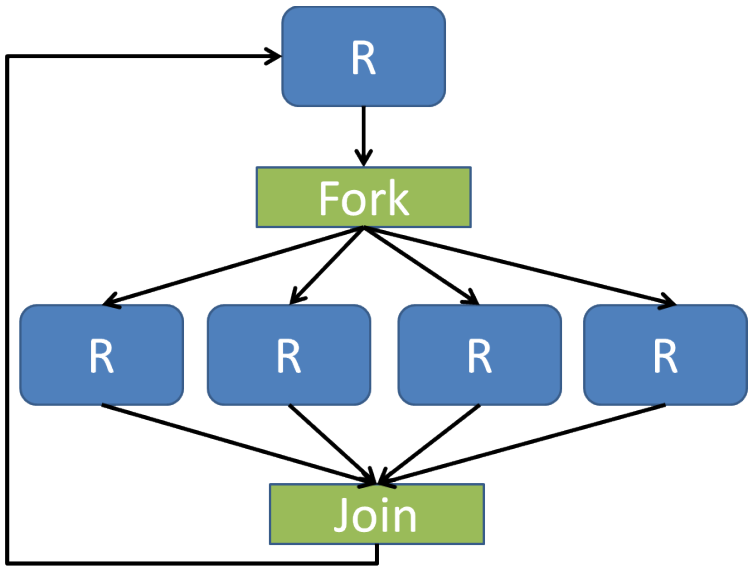
- Comes with R $\geq$ 2.14.0
- Has 2 disjoint interfaces.

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

The parallel Package: multicore

A simple Fork-Join parallel programming paradigm

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



The parallel Package: multicore

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

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

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

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

The parallel Package: multicore

```
1 unlist(mclapply(1:10, function(i) Sys.getpid(), mc.cores=4))
2 # [1] 13390 13391 13392 13393 13390 13391 13392 13393 13390
      13391
3
4 unlist(mclapply(1:10, function(i) Sys.getpid(), mc.cores=2))
5 # [1] 13394 13395 13394 13395 13394 13395 13394 13395 13394
      13395
```

Check process id's involved.

The parallel Package: snow

? Uses sockets.

\+ Works on all platforms.

– More fiddly than `mclapply()`.

– Not as efficient as forks.

The parallel Package: snow

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

The parallel Package: Summary

All

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

multicore

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

snow

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

4 Shared Memory Parallel Packages in R

- The parallel Package
- The foreach Package

The foreach Package

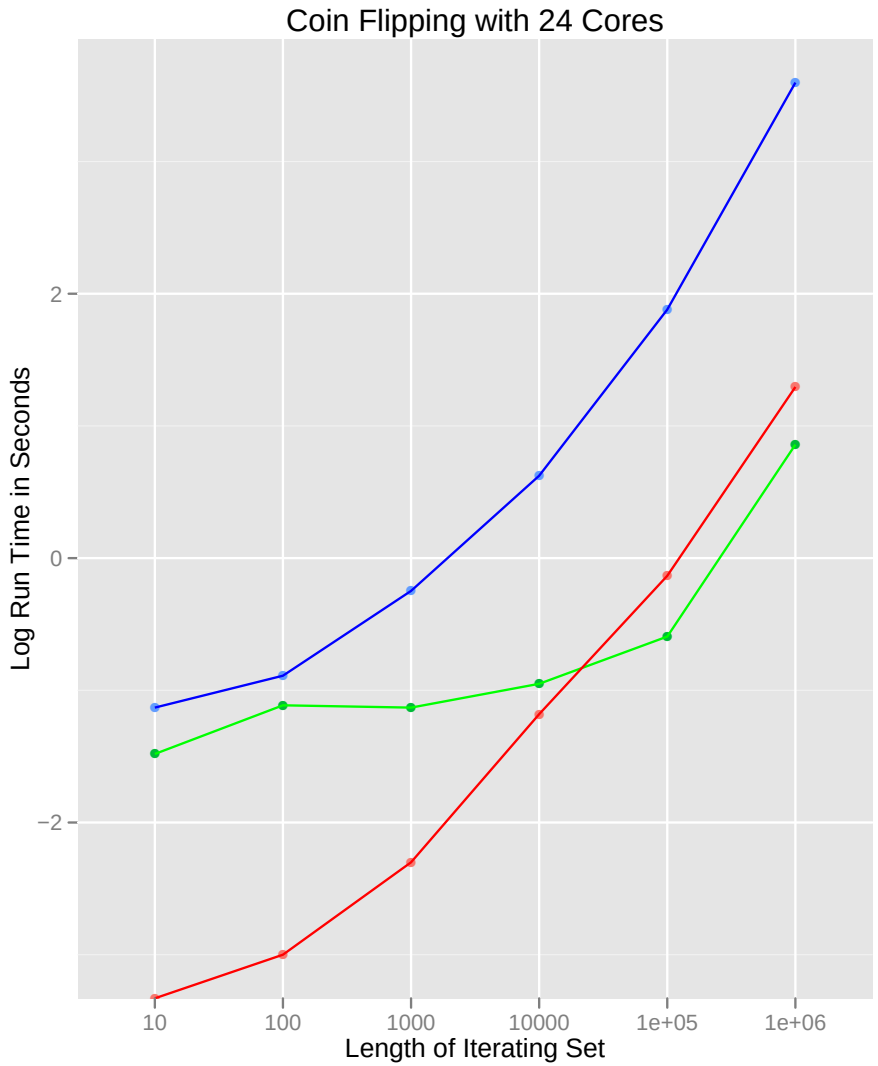
- On Cran (Revolution Analytics).
- **foreach** is a single interface for a number of "backend" packages.
- Backends: **doMC**, **doMPI**, **doParallel**, **doRedis**, **doRNG**, **doSNOW**.
- A Manager-Workers approach

The foreach Package

The Idea: Unify the disparate interfaces.

- \+ Works on all platforms (if backend does).
- \+ Can even work serial with minor notational change.
- \+ Write the code once, use whichever backend you prefer.
- – Binary operator, non-R-ish synatx.
- – Overhead issues if you aren't careful!

Overhead Issues




```
### Bad performance
foreach(i=1:len) %dopar%
    tinyfun(i)

### Expected performance
foreach(i=1:ncores)
    %dopar% {
  out <-
      numeric(len/ncores)
  for (j in 1:(len/ncores))
    out[i] <- tinyfun(j)
  out
}
```

The foreach Package: General Procedure

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

Using foreach

Serial

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

Parallel

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

foreach backends

multicore

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

snow

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

foreach Summary

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

Contents

5 The pbdR Project

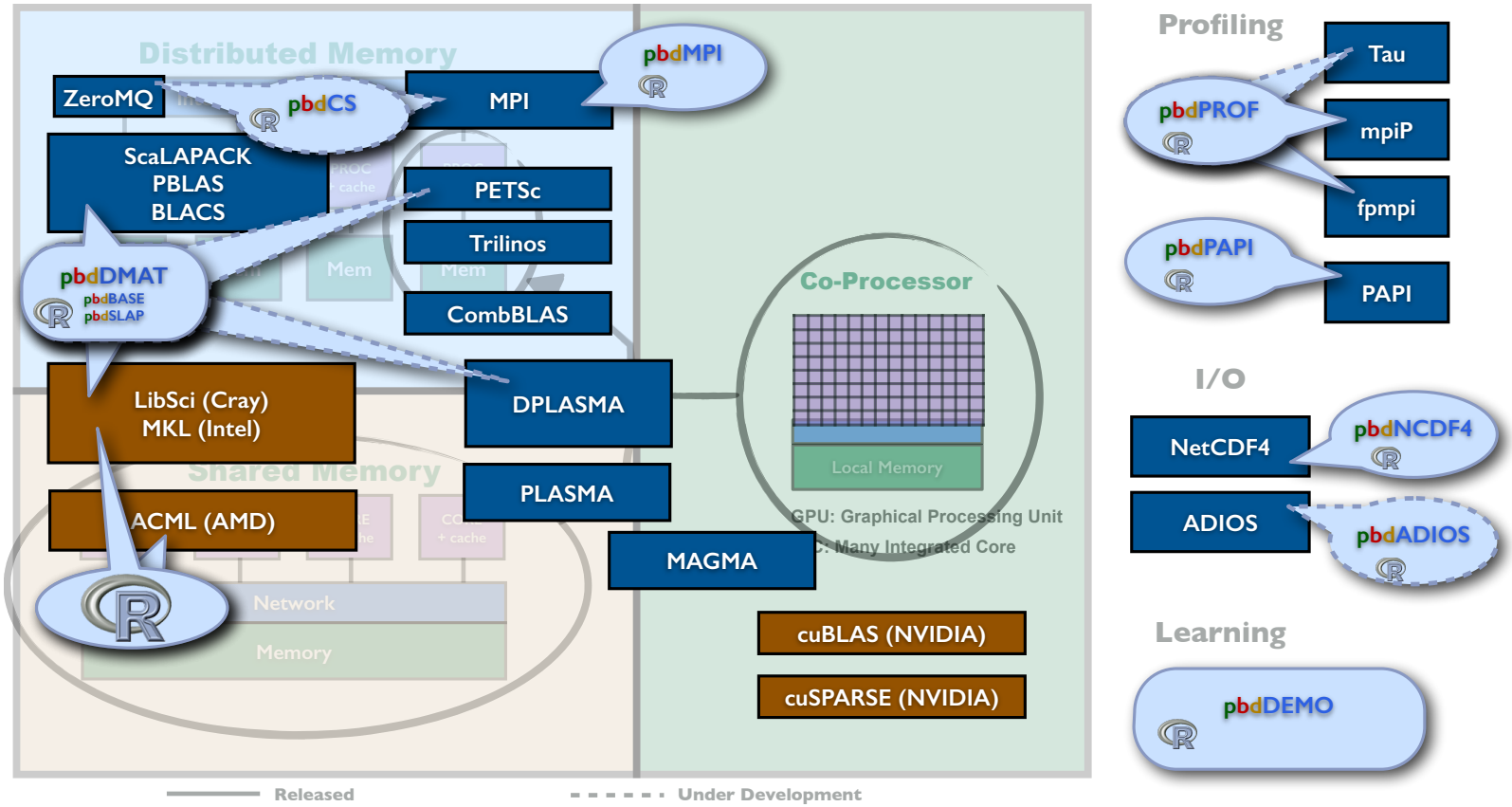
- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

Programming with Big Data in R (pbdR)

Strive for *Productivity, Portability, Performance*

- Bridge high-performance compiled code with high-productivity of R
- Keep syntax *identical* to R, when possible.
- Software reuse philosophy:
  - Don't make up new stuff if it's already solved in HPC
  - Introduce HPC standards with R flavor
  - Use scalable HPC libraries with R convenience
- Simplify and use R intelligence where possible

pbdR Interfaces to Libraries: Sustainable Path



Why use HPC libraries?

- The libraries represent 30+ years of research by the HPC community
- *They're tested. They're fast. They're scalable.*
- Many science communities are invested in their API
- HPC Simulation Science uses much of the same math as data analysis

Integer? Not always obvious in R.

```
> is.integer(1)
[1] FALSE
> is.integer(2)
[1] FALSE
> is.integer(1:2)
[1] TRUE
```

pbdMPI lets R figure it out

Rmpi

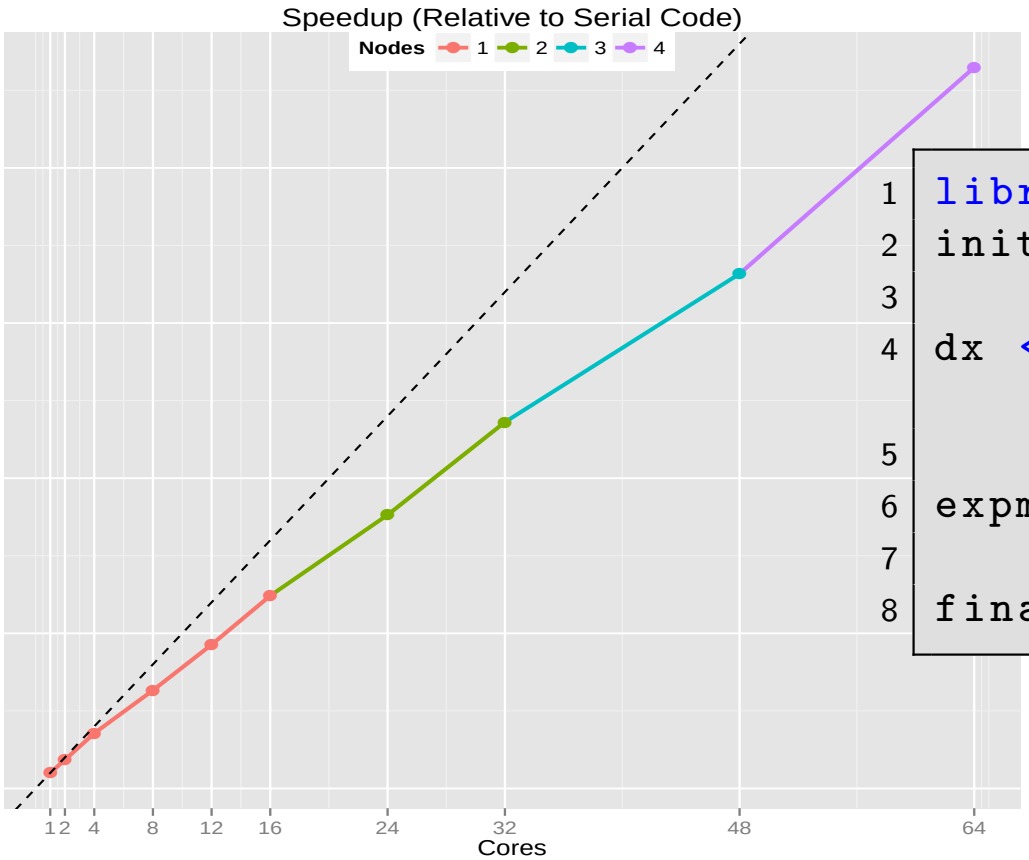
```
# int
mpi.allreduce(x, type=1)
# double
mpi.allreduce(x, type=2)
```

pbdMPI

```
allreduce(x)
```

Distributed Matrices and Statistics with **pbdDMAT**

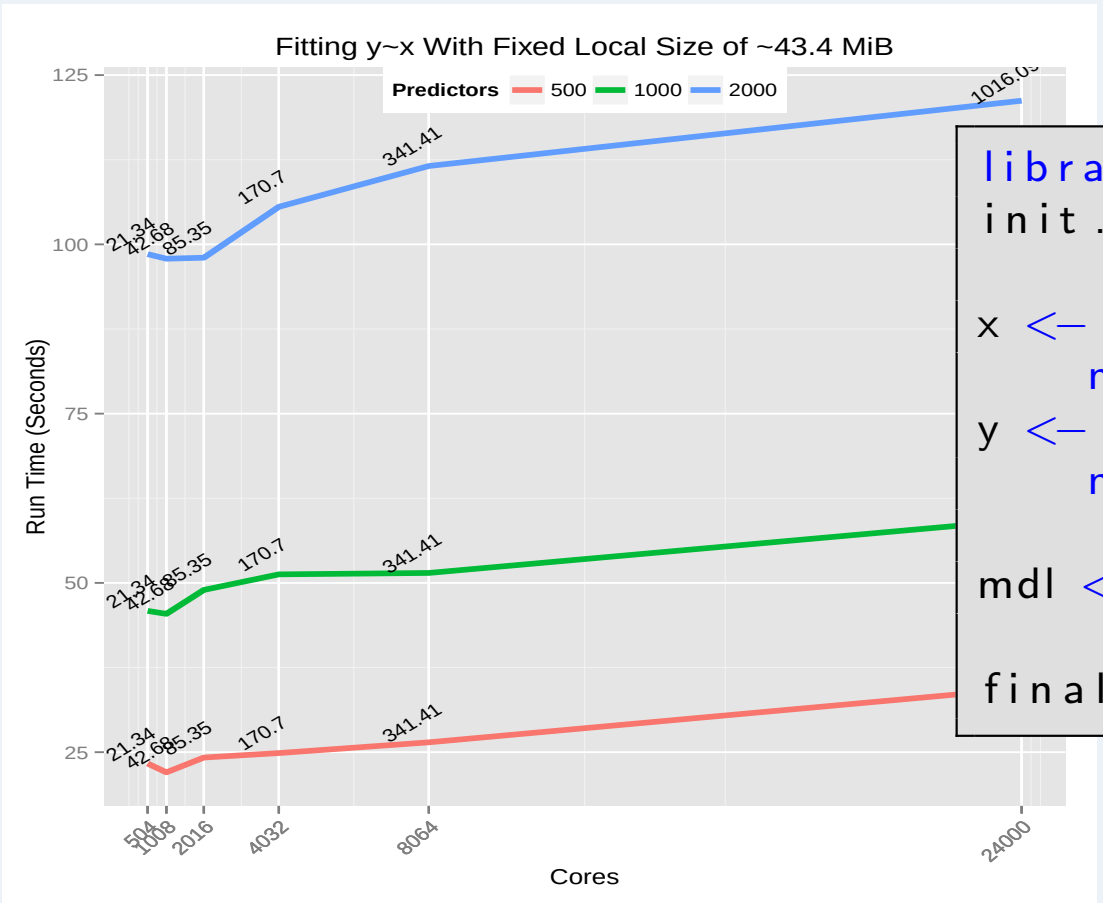
Matrix Exponentiation




```
1 library(pbdDMAT)
2 init.grid()
3
4 dx <- ddmatrix("rnorm",
      5000, 5000)
5
6 expm(dx)
7
8 finalize()
```

Distributed Matrices and Statistics with **pbdDMAT**

Least Squares Benchmark



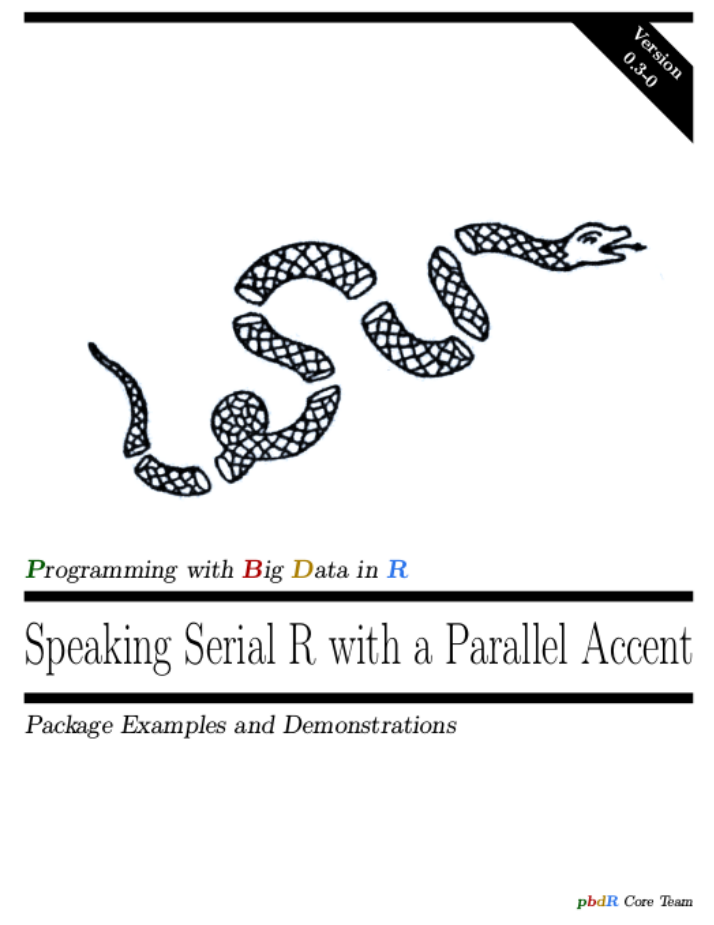

```
library(pbdDMAT)
init.grid()

x <- ddmatrix("rnorm",
    nrow=m, ncol=n)
y <- ddmatrix("rnorm",
    nrow=m, ncol=1)

mdl <- lm.fit(x=x, y=y)

finalize()
```

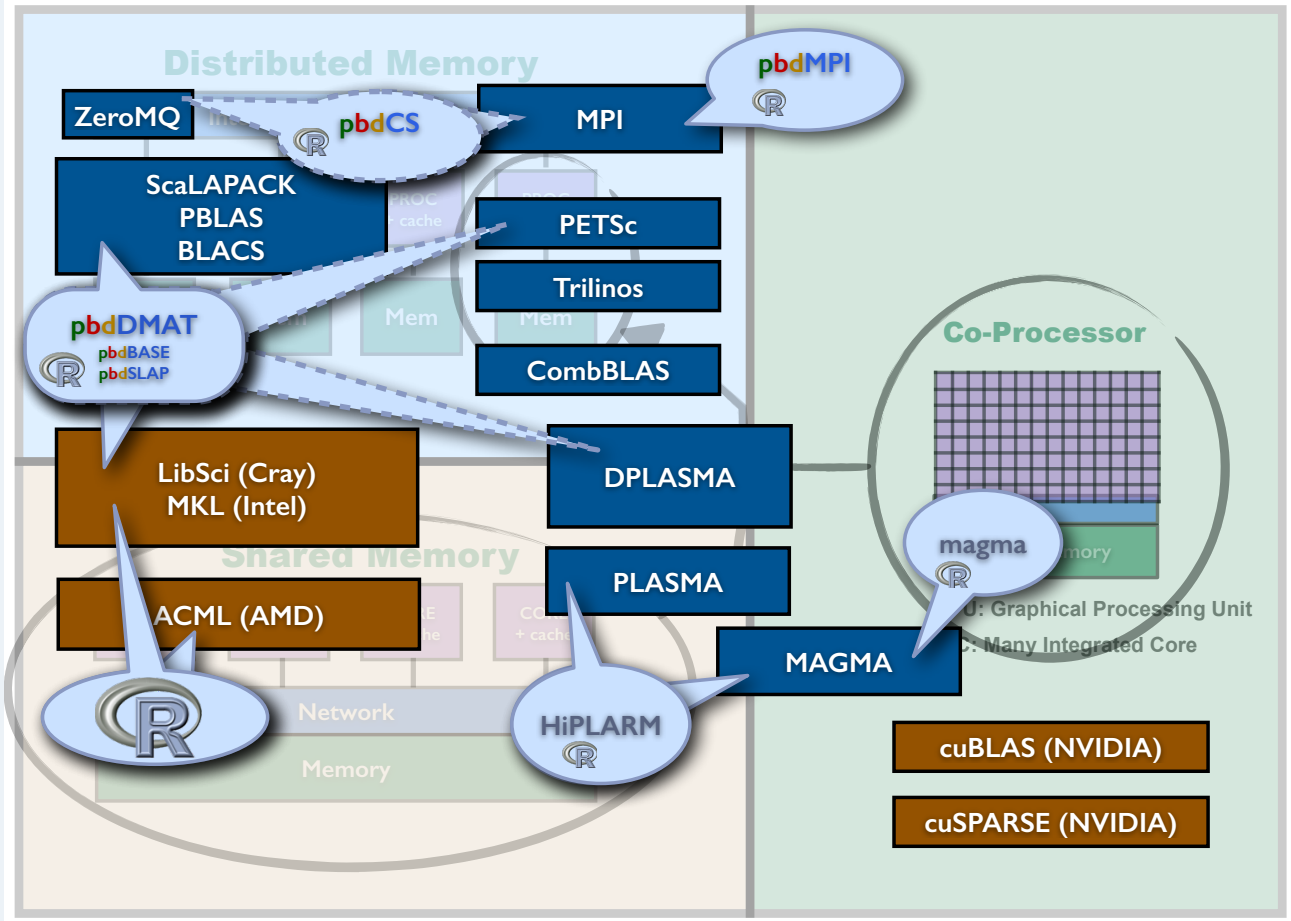
Getting Started with HPC for R Users: **pbdDEMO**

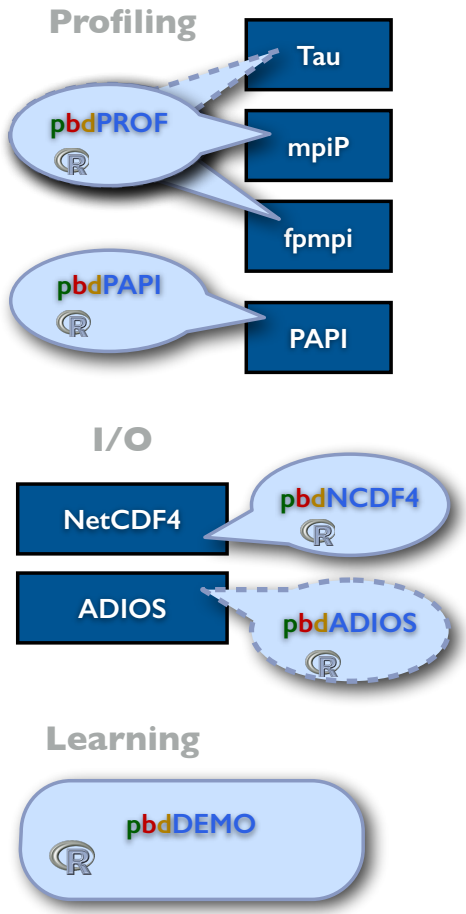


- 140 page, textbook-style vignette.
- Over 30 demos, utilizing all* packages.
- Not just a "hello world"!
- Demos include:
  - PCA
  - Regression
  - Parallel data input
  - Model-based clustering
  - Simple Monte Carlo simulation
  - Bayesian MCMC

5 The pbdR Project

R and pbdR Interfaces to HPC Libraries





5 The pbdR Project

- The pbdR Project
- pbdR Connects R to HPC Libraries
- Using pbdR
- Summary

pbdR Paradigms

pbdR programs are R programs!

Differences:

- Batch execution (non-interactive).
- Parallel code utilizes Single Program/Multiple Data (SPMD) style
- Emphasizes data parallelism.

Batch Execution

- Running a serial R program in batch:

```
1 Rscript my_script.r
```

or

```
1 R CMD BATCH my_script.r
```

- Running a parallel (with MPI) R program in batch:

```
1 mpirun -np 2 Rscript my_par_script.r
```

Single Program/Multiple Data (SPMD)

- SPMD is a programming *paradigm*.
- Not to be confused with SIMD.

Paradigms

Programming models
OOP, Functional, SPMD, ...

SIMD

Hardware instructions
MMX, SSE, ...

Single Program/Multiple Data (SPMD)

SPMD is arguably the simplest extension of serial programming.

- Only one program is written, executed in batch on all processors.
- Different processors are autonomous; there is no manager.
- Dominant programming model for large machines for 30 years.

Summary

- pbdR connects R to scalable HPC libraries.
- The **pbdDEMO** package offers numerous examples and explanations for getting started with distributed R programming.
- pbdR programs are R programs.

Contents

Message Passing Interface (MPI)

- *MPI*: Standard for managing communications (data and instructions) between different nodes/computers.
- *Implementations*: OpenMPI, MPICH2, Cray MPT, ...
- Enables parallelism (via communication) on distributed machines.
- *Communicator*: manages communications between processors.

MPI Operations (1 of 2)

- **Managing a Communicator**: Create and destroy communicators.
  `init()` — initialize communicator
  `finalize()` — shut down communicator(s)

- **Rank query**: determine the processor's position in the communicator.
  `comm.rank()` — "who am I?"
  `comm.size()` — "how many of us are there?"

- **Printing**: Printing output from various ranks.
  `comm.print(x)`
  `comm.cat(x)`
  WARNING: only use these functions on *results*, never on yet-to-be-computed things.

Quick Example 1

Rank Query: 1_rank.r

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

Execute this script via:

```
1 mpirun -np 2 Rscript 1_rank.r
```

Sample Output:

```
1 COMM.RANK = 0
2 [1] 0
3 COMM.RANK = 1
4 [1] 1
```

Quick Example 2

Hello World: 2_hello.r

```
1  library(pbdMPI, quietly=TRUE)
2  init()
3
4  comm.print("Hello, world")
5
6  comm.print("Hello, world too")
7
8  comm.print("Hello again", all.rank=TRUE, quiet=TRUE)
9
10 finalize()
```

Execute this script via:

```
1  mpirun -np 2 Rscript 2_hello.r
```

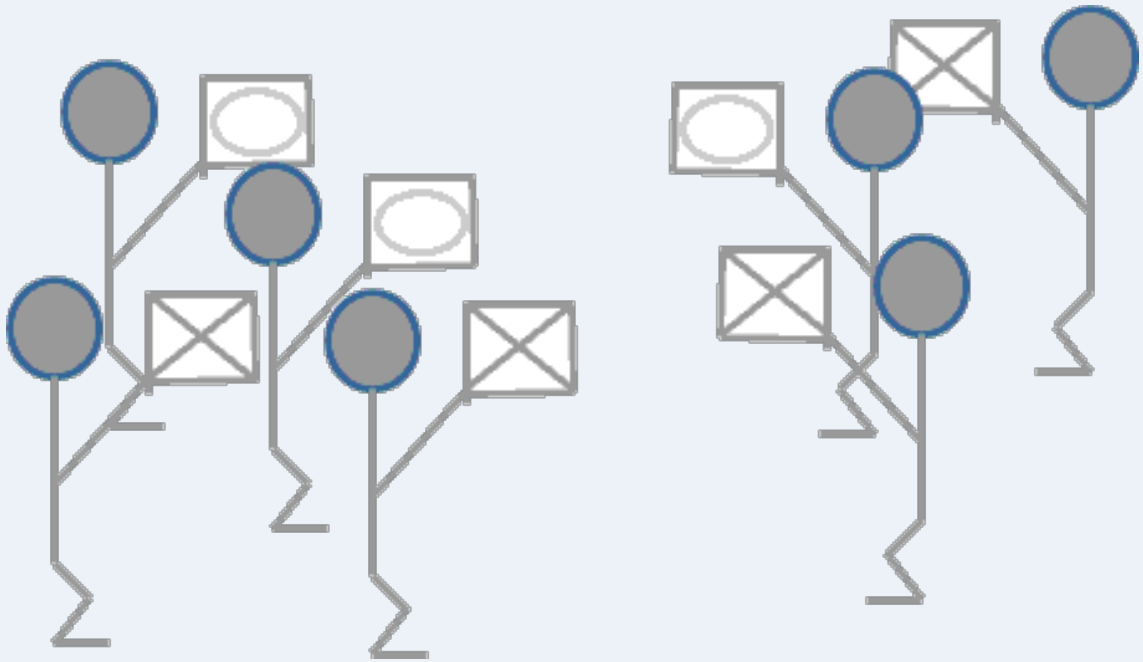
Sample Output:

```
1  COMM.RANK = 0
2  [1] "Hello, world"
3  [1] "Hello again"
4  [1] "Hello again"
```

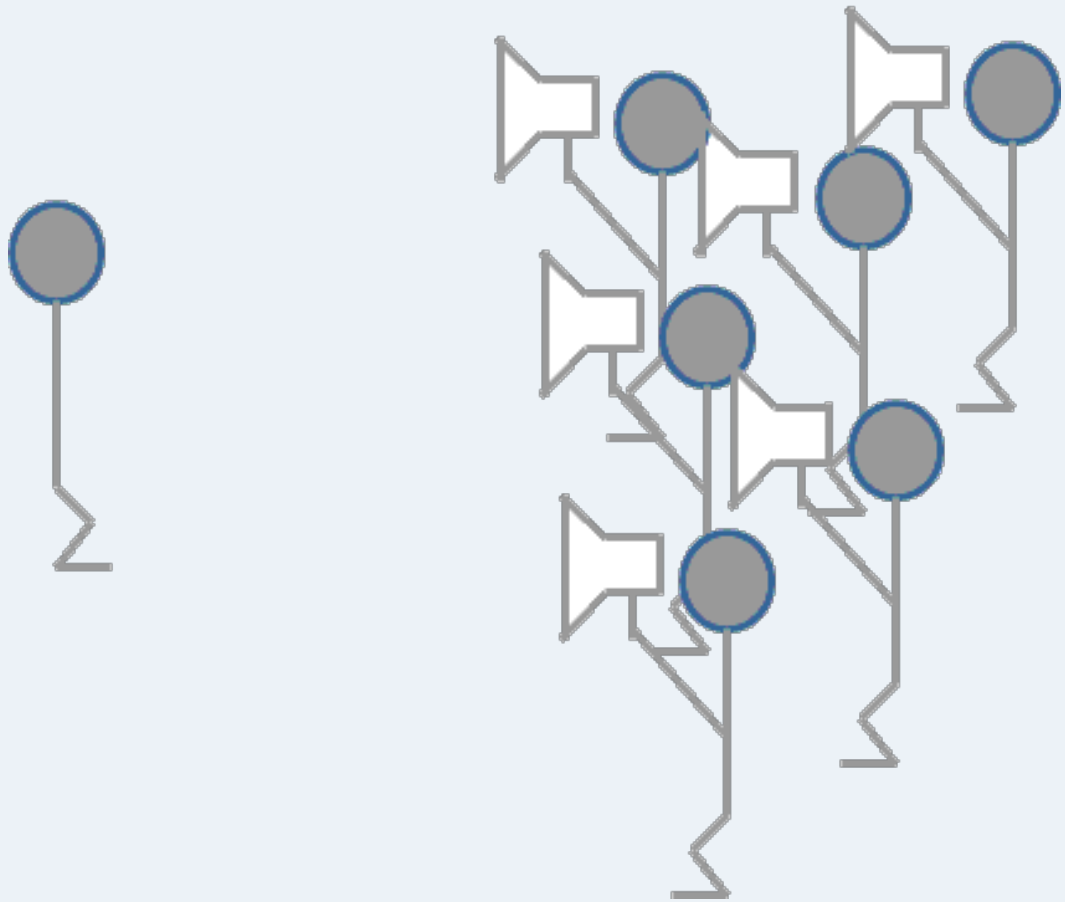
MPI Operations

1. Reduce
2. Gather
3. Broadcast
4. Barrier

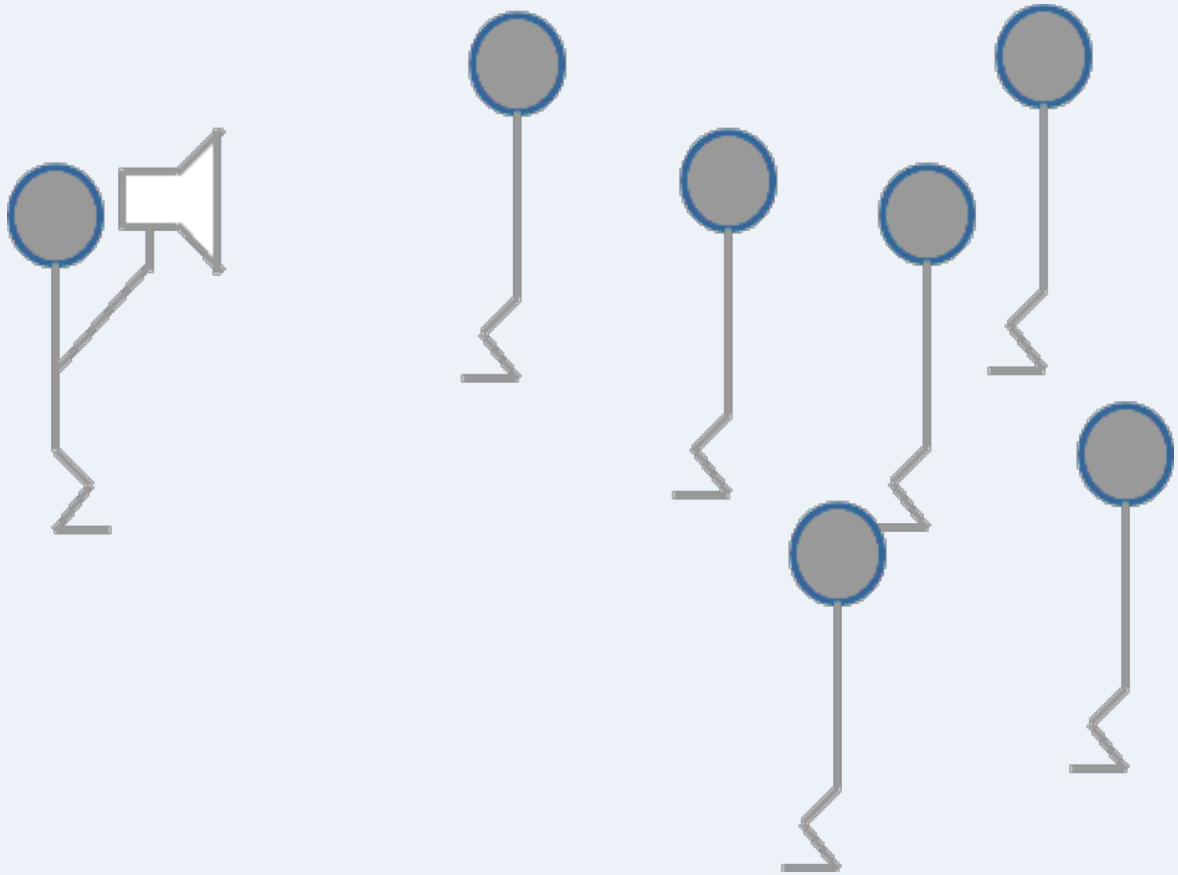
Reductions — Combine results into single result



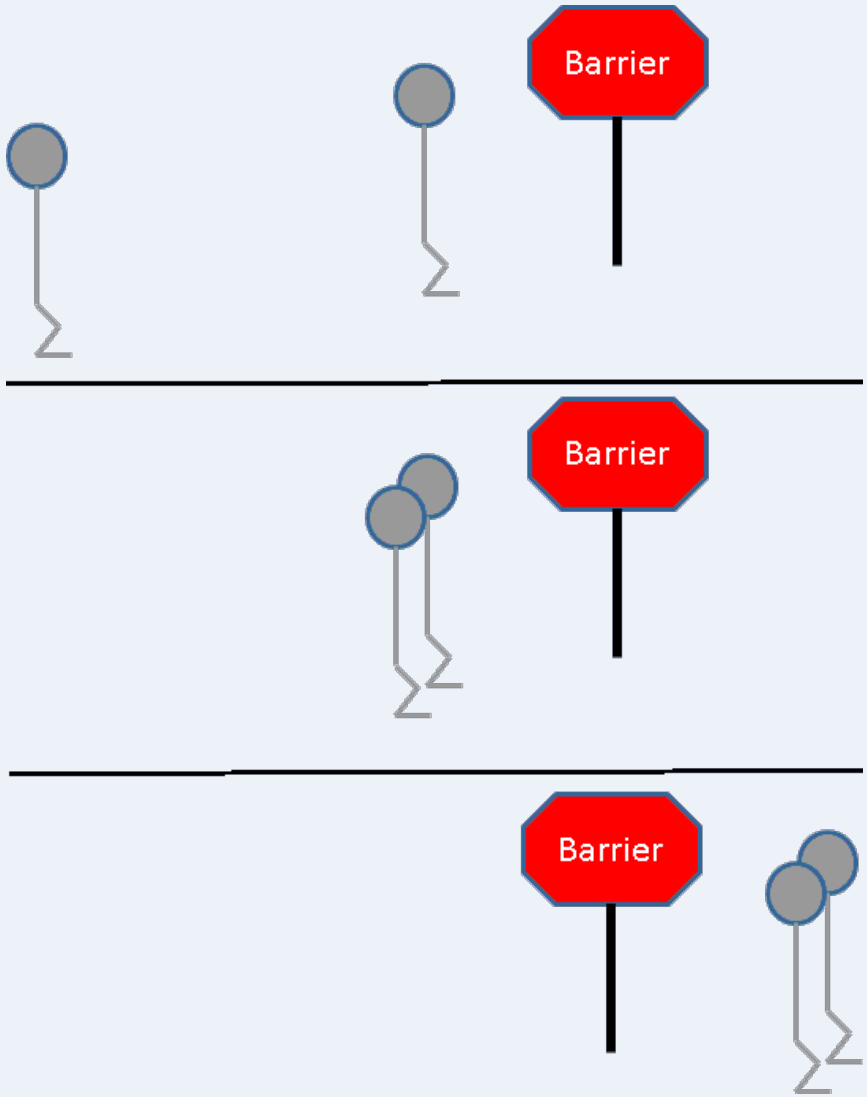
Gather — Many-to-one



Broadcast — One-to-many



Barrier — Synchronization



MPI Operations (2 of 2)

- **Reduction**: each processor has a number `x`; add all of them up, find the largest/smallest, ...
  `reduce(x, op='sum')` — reduce to one
  `allreduce(x, op='sum')` — reduce to all

- **Gather**: each processor has a number; create a new object on some processor containing all of those numbers.
  `gather(x)` — gather to one
  `allgather(x)` — gather to all

- **Broadcast**: one processor has a number `x` that every other processor should also have.
  `bcast(x)`

- **Barrier**: "computation wall"; no processor can proceed until *all* processors can proceed.
  `barrier()`

Quick Example 3

Reduce and Gather: 3_gt.r

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

Execute this script via:

```
1  mpirun -np 2 Rscript 3_gt.r
```

Sample Output:

```
1  COMM.RANK = 0
2  [1] 2 8
3  COMM.RANK = 0
4  [1] 10
5  COMM.RANK = 1
6  [1] 10
```

Quick Example 4

Broadcast: 4_bcast.r

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

Execute this script via:

```
1  mpirun -np 2 Rscript 4_bcast.r
```

Sample Output:

```
1  COMM.RANK = 0
2        [,1] [,2]
3  [1,]    1    3
4  [2,]    2    4
5  . . .
```

Random Seeds

pbdMPI offers a simple interface for managing random seeds:

- `comm.set.seed(seed=1234, diff=TRUE)` — All processors generate different streams.
- `comm.set.seed(seed=1234, diff=FALSE)` — All processors generate same streams.

Other Helper Tools

pbdMPI Also contains useful tools for Manager/Worker and task parallelism codes:

- **Task Subsetting**: Distributing a list of jobs/tasks
  `get.jid(n)`
- ***ply**: Functions in the *ply family.
  `pbdApply(X, MARGIN, FUN, ...)` — analogue of `apply()`
  `pbdLapply(X, FUN, ...)` — analogue of `lapply()`
  `pbdSapply(X, FUN, ...)` — analogue of `sapply()`

Distributing Data

Problem: How to distribute the data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ x_{9,1} & x_{9,2} & x_{9,3} \\ x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10 \times 3}$$

?

Distributing a Matrix Across 4 Processors: Block Distribution

Data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ x_{9,1} & x_{9,2} & x_{9,3} \\ \hline x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10\times 3}$$

Processors

0
1
2
3

Distributing a Matrix Across 4 Processors: Local Load Balance

Data

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \\ \hline x_{4,1} & x_{4,2} & x_{4,3} \\ x_{5,1} & x_{5,2} & x_{5,3} \\ x_{6,1} & x_{6,2} & x_{6,3} \\ \hline x_{7,1} & x_{7,2} & x_{7,3} \\ x_{8,1} & x_{8,2} & x_{8,3} \\ \hline x_{9,1} & x_{9,2} & x_{9,3} \\ x_{10,1} & x_{10,2} & x_{10,3} \end{bmatrix}_{10\times 3}$$

Processors

0
1
2
3

Summary

- Start by loading the package:

```
1 library(pbdMPI, quiet = TRUE)
```

- Always initialize before starting and finalize when finished:

```
1 init()
2
3 # ...
4
5 finalize()
```

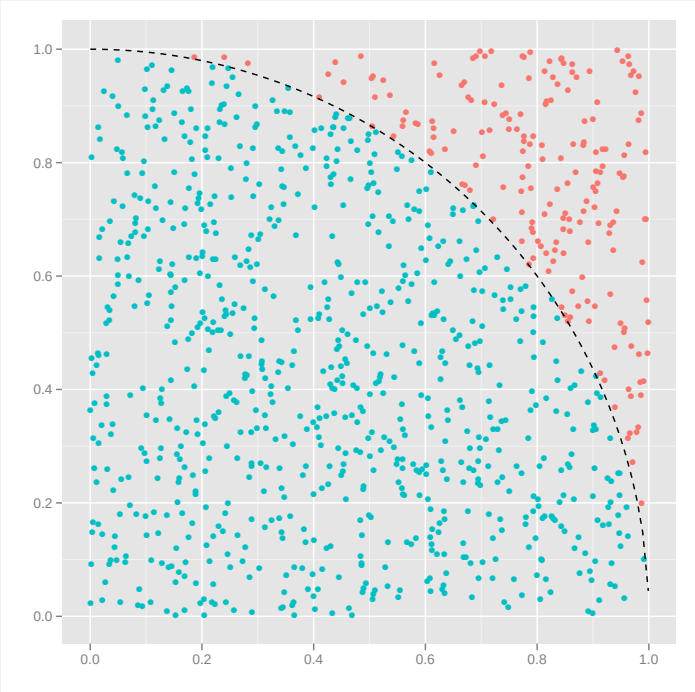
- Need to distribute your data? Try splitting by blocks of rows or by blocks of columns.

Contents

Example 1: Monte Carlo Simulation

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

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



Example 1: Monte Carlo Simulation Algorithm

1. Let n be big-ish; we'll take $n = 50,000$.
2. Generate an $n \times 2$ matrix x of standard uniform observations.
3. Count the number of rows satisfying $x^2 + y^2 \leq 1$.
4. Sum everyone's count.
5. Use area formula to estimate π.
6. If my rank is 0, print the result.

Example 1: Monte Carlo Simulation Code

Serial Code

```
1 set.seed(seed=1234567)
2 N <- 1e7
3 X <- matrix(runif(N * 2), ncol=2)
4 r <- sum(rowSums(X^2) <= 1)
5 PI <- 4*r/N
6 print(PI)
```

Parallel Code

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

Note

For the remainder, we will exclude loading, init, and finalize calls.

7 Basic Statistics Examples

- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- pbdMPI Example: Random Forest Prediction
- Summary

Example 2: Sample Covariance

$$cov(x_{n\times p}) = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mu_x\right)^T\left(x_i - \mu_x\right),$$

where x_i is row i of x, and
μ_x is a vector of the column means of x.

Note that the sum can be broken up by blocks of rows.

Example 2: Sample Covariance Algorithm (split by blocks of rows)

1. Determine the total number of rows N.
2. Compute the vector of column means of the full matrix.
3. Subtract each column's mean from that column's entries in each local matrix.
4. Compute the crossproduct locally and reduce.
5. Divide by $N - 1$.

Example 2: Sample Covariance Code (Split by blocks of rows)

Serial Code

```
1 N <- nrow(X)
2 mu <- colSums(X) / N
3
4 X <- sweep(X, STATS=mu, MARGIN=2)
5 Cov.X <- crossprod(X) / (N-1)
6
7 print(Cov.X)
```

Parallel Code

```
1 N <- allreduce(nrow(X.loc), op="sum")
2 mu <- allreduce(colSums(X.loc) / N, op="sum")
3
4 X.loc <- sweep(X.loc, STATS=mu, MARGIN=2)
5 Cov.X <- allreduce(crossprod(X.loc), op="sum") / (N-1)
6
7 comm.print(Cov.X)
```

7 Basic Statistics Examples

- pbdMPI Example: Monte Carlo Simulation
- pbdMPI Example: Sample Covariance
- pbdMPI Example: Linear Regression
- pbdMPI Example: Random Forest Prediction
- Summary

Example 3: Linear Regression

Find β such that

$$\boldsymbol{y} = \boldsymbol{X}\beta + \boldsymbol{\epsilon}$$

When $\boldsymbol{X}$ is full rank,

$$\hat{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Example 3: Linear Regression Algorithm (Split by blocks of rows)

1. Locally, compute $tx = x^T$
2. Locally, compute $A = tx * x$. Add across all processors. (allreduce).
3. Locally, compute $B = tx * y$. Add across all processors (allreduce).
4. Locally, compute $A^{-1} * B$

Example 3: Linear Regression Code (Split by blocks of rows.)

Serial Code

```
1 tX <- t(X)
2 A <- tX %*% X
3 B <- tX %*% y
4
5 ols <- solve(A) %*% B
```

Parallel Code

```
1 tX.loc <- t(X.loc)
2 A <- allreduce(tX.loc %*% X.loc, op = "sum")
3 B <- allreduce(tX.loc %*% y.loc, op = "sum")
4
5 ols <- solve(A) %*% B
```

Letter Recognition Data

Example 4: 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 4: Random Forest Algorithm

1. Build simple regression trees from random subsets of columns
2. Use model averaging for prediction
3. Package **randomForest** has a `combine()` function that enables the following parallel approach:
   1. Everyone gets the same training data
   2. Split regression tree building among processors (**randomForest**)
   3. Use `allgather` to bring built predictors to all
   4. Everyone `combine` predictors
   5. Split prediction work by blocks of rows
   6. Use `allreduce` to assess prediction
4. Steps (3) and (4) can be improved with a custom reduce/combine to take advantage of MPI vendor optimizations

Example 4: Random Forest Code
(Split learning by blocks of trees. Split prediction by blocks of rows.)

Serial Code

```
1  data(LetterRecognition) # 26 Capital Letters Data 20,000 x 17
2  set.seed(seed=1234567)
3  n <- nrow(LetterRecognition)
4  ## get train data
5  n_train <- floor(0.8*n)
6  i_train <- sample.int(n, n_train) # Use 4/5 of the data to train
7  train <- LetterRecognition[i_train, ]
8  test <- LetterRecognition[-i_train, ]
9
10 ## train random forest
11 my.k <- 500
12 rf.all <- randomForest(lettr ~ ., train, ntree=my.k,
       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 - n_train), "\n")
```

Example 4: Random Forest Code
(Split learning by blocks of trees. Split prediction by blocks of rows.)

Parallel Code

```
1  data(LetterRecognition) # 26 Capital Letters Data 20,000 x 17
2  comm.set.seed(seed=123, diff=FALSE) # same training data
3  n <- nrow(LetterRecognition)
4  n_train <- floor(0.8*n)
5  i_train <- sample.int(n, n_train) # Use 4/5 of the data to train
6  train <- LetterRecognition[i_train, ]
7  my.test_rows <- get.jid(n - n_train)  # different test data
8  test <- LetterRecognition[-i_train, ][my.test_rows, ]
9
10 comm.set.seed(seed=1e6*runif(1), diff=TRUE)
11 my.k <- floor(500/comm.size())
12 my.rf <- randomForest(lettr ~ ., train, ntree=my.k,
       norm.votes=FALSE)
13
14 rf.each <- allgather(my.rf)
15 rf.all <- do.call(combine, rf.each)
16
17 pred <- predict(rf.all, test)
18 correct <- allreduce(sum(pred == test\$lettr))
19 comm.cat("Proportion Correct:", correct/(n - n_train), "\n")
```

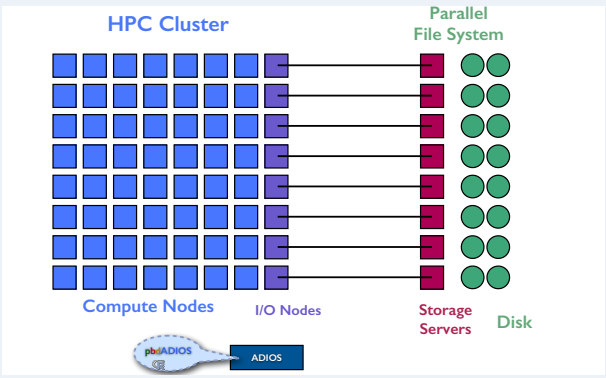
Summary

- SPMD programming is (often) a natural extension of serial programming.
- More **pbdMPI** examples in **pbdDEMO**.

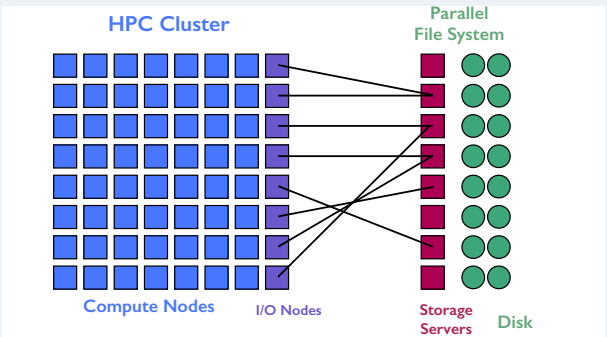
Contents

I/O Configurations

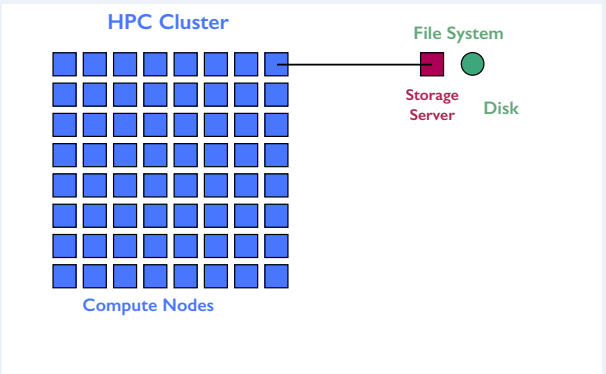
Best!



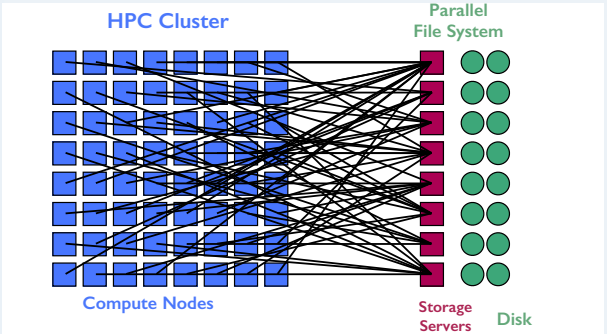
Good.



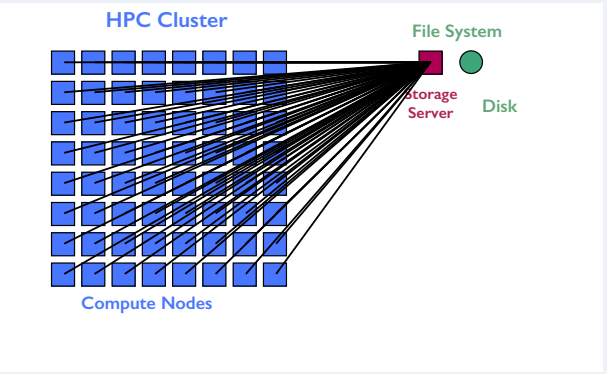
If you have to.



Can work.



Don't do this!



R has a separate manual for data import/export: http://r-project.org/

- **readr**: read_table(), read_fwf(), read_csv(), ...
- scan(), readLines(), read.table(), read.csv(), ...
- seek(), readBin()
- CSV, SQL, ...
- NetCDF4, HDF5, ADIOS bp, custom binary, ...

CSV Data: Read Serial then Distribute

0_readcsv.r

```
1 library(pbdDMAT)
2 if(comm.rank() == 0) { # only read on process 0
3   x <- as.matrix(read.csv("myfile.csv"))
4 } else {
5   x <- NULL
6 }
7
8 dx <- as.ddmatrix(x)
```

8 Data Input

- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

Parallel reading brings new issues

- How to partition data across nodes?
- How to structure for scalable libraries?
- Read directly into form needed or restructure?
- CSV, SQL, HDFS, NetCDF4, HDF5, ADIOS, custom binary
- ...
- Lot of work needed to make it intuitive!
- Currently under development.

CSV Data

Serial Code

```
1 x <- read.csv("x.csv")
2
3 x
```

Parallel Code

```
1 library(pbdDEMO, quiet = TRUE)
2 init.grid()
3
4 dx <- read.csv.ddmatrix("x.csv", header=TRUE, sep=",",
5           nrows=10, ncols=10, num.rdrs=2, ICTXT=0)
6
7 dx
8
9 finalize()
```

Hadoop HDFS, RHadoop

ravro - read and write files in avro format

plyrmr - higher level plyr-like data processing for structured data, powered by rmr

rmr - functions providing Hadoop MapReduce functionality in R

rhdfs - functions providing file management of the HDFS from within R

rhbase - functions providing database management for the HBase distributed database from within R

Binary Data: Vector

```
size <- 8 # bytes

my_ids <- get.jid(n)    # my index values from n

my_start <- (my_ids[1] - 1)*size     # my starting byte location
my_length <- length(my_ids)      # my number of bytes to read

con <- file("binary.vector.file", "rb")
seekval <- seek(con, where=my_start, rw="read")
x <- readBin(con, what="double", n=my_length, size=size)
```

Binary Data: Matrix

```
size <- 8 # bytes

my_ids <- get.jid(ncol)
my_ncol <- length(my_ids)
my_start <- (my_ids[1] - 1)*nrow*size
my_length <- my_ncol*nrow

con <- file("binary.matrix.file", "rb")
seekval <- seek(con, where=my_start, rw="read")
x <- readBin(con, what="double", n=my_length, size=size)

gdim <- c(nrow, ncol)
ldim <- c(nrow, my_ncol)
bldim <- c(nrow, allreduce(my_ncol, op="max"))
X <- new("ddmatrix", Data=matrix(x, nrow, my_ncol), dim=gdim,
    ldim=ldim, bldim=bldim, ICTXT=1)     # glue together as
    column-block ddmatrix

X <- redistribute(X, bldim=c(2, 2), ICTXT=0)  # redistribute as
    block-cyclic
Xprc <- prcomp(X)   # proceed as with serial code
```

NetCDF4 Data

```
### parallel read after determining start and length
nc <- nc_open_par(file_name)

nc_var_par_access(nc, "variable_name")
new.X <- ncvar_get(nc, "variable_name", start, length)
nc_close(nc)
```

ADIOS Data (.bp files)

pbdADIOS is under construction

```
### parallel read after determining start and length
file <- adios_open(file_name)

## Bounding box (start, length) access
## Staging capability with ADIOS configured simulation codes
## Streaming access

adios_close()
```

8 Data Input

- Considering I/O Configurations
- Serial Data Input
- Parallel Data Input
- Summary

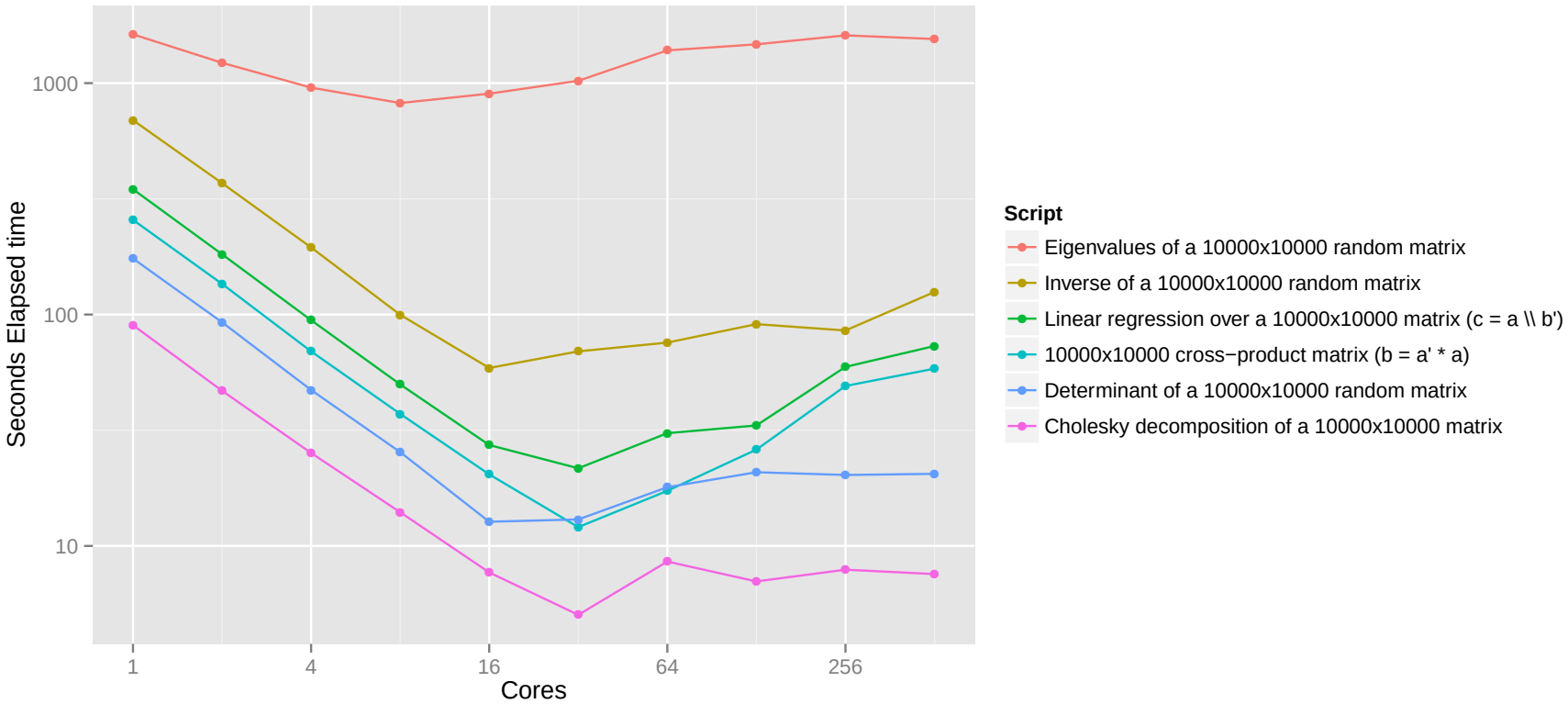
Summary

- Mostly "do it yourself" with bounding box
- Parallel file system for big data
  - Binary files for true parallel reads
  - Use correct number of readers vs number of storage servers
- Redistribution help from `ddmatrix` functions
- More intuitive input under development
- Staging and in situ capabilities via ADIOS soon

Contents

Distributed Matrices

You can only get so far with one node. . .



The solution is to distribute the data.

A Matrix is Mapped onto a Processor Grid

Processor Grid Shapes

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

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

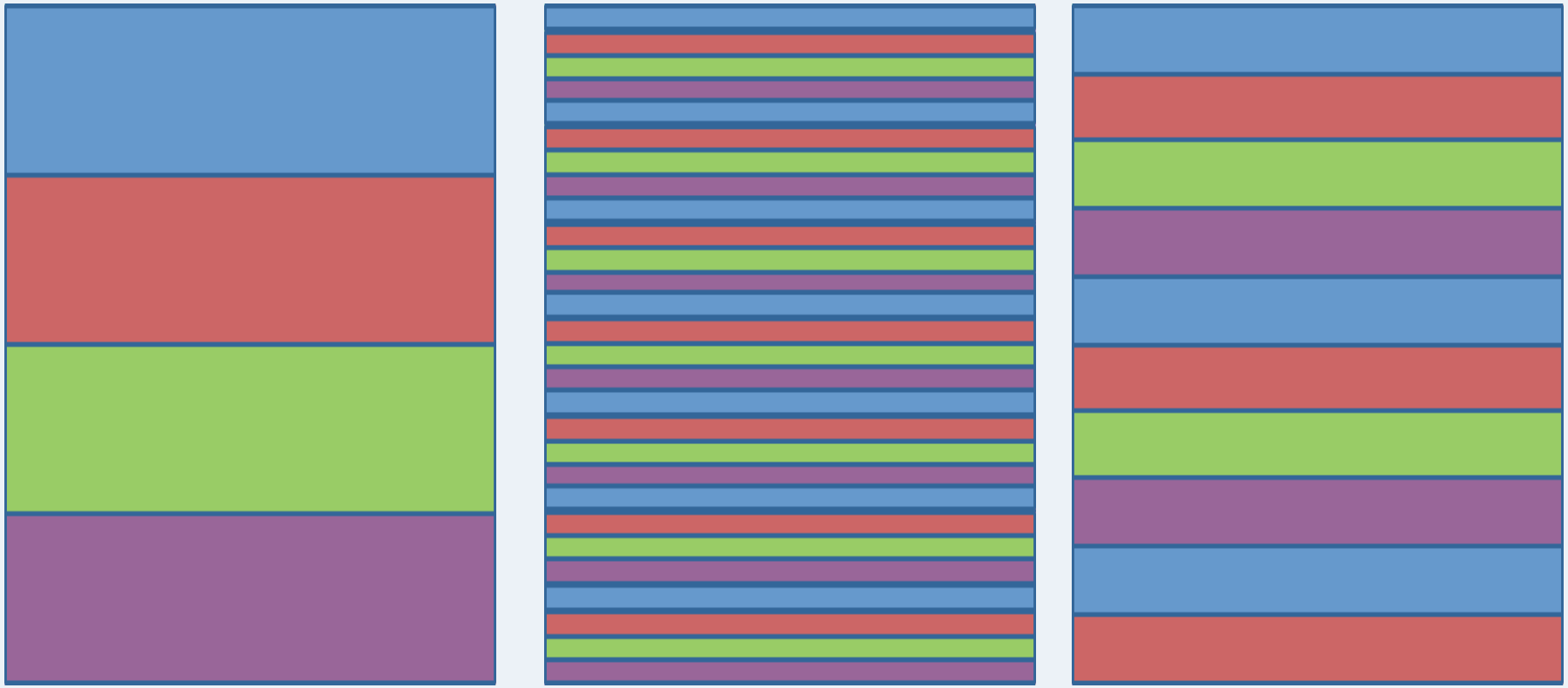
$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

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

(a) 1×6 (b) 2×3 (c) 3×2 (d) 6×1

Table: Processor Grid Shapes with 6 Processors

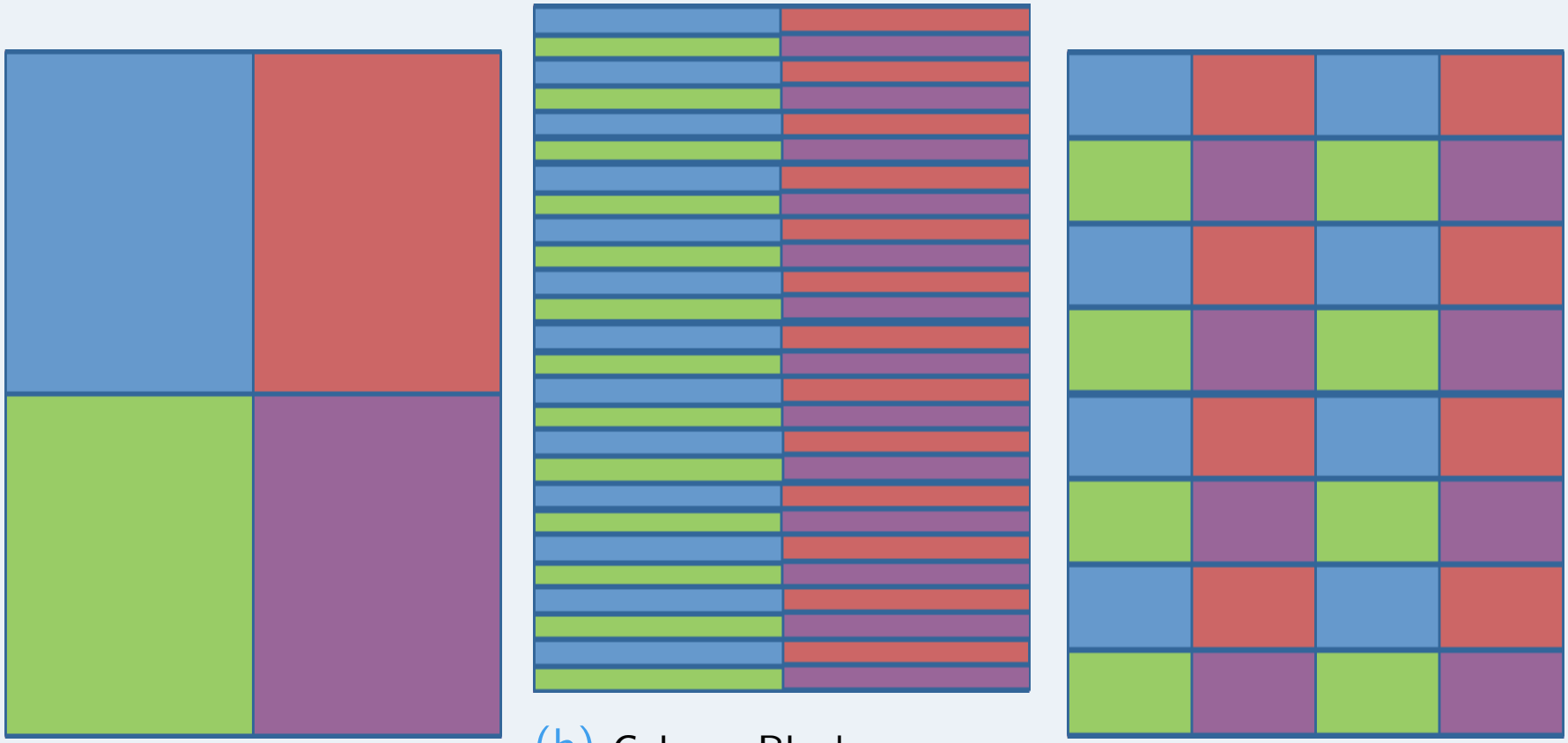
Distributed Matrices



(a) Row Block (b) Row Cyclic (c) Row-Block Cyclic

Figure: Matrix Distribution Schemes Onto a 4×1 Processor Grid

Distributed Matrices



(a) Block

(b) Column-Block Row-Cyclic

(c) Block-Cyclic

Figure: Matrix Distribution Schemes Onto a 2×2 Processor Grid

The ddmatrix Class

For **d**istributed **d**ense **matrix** objects, we use the special S4 class `ddmatrix`.

$$\texttt{ddmatrix} = \begin{cases} \textbf{Data} & \text{The local submatrix (an R matrix)} \\ \textbf{dim} & \text{Dimension of the global matrix} \\ \textbf{ldim} & \text{Dimension of the local submatrix} \\ \textbf{bldim} & \text{Dimension of the blocks} \\ \textbf{ICTXT} & \text{MPI Grid Context} \end{cases}$$

Predefined contexts:

ICTXT $= 0$ Block-Cyclic

ICTXT $= 1$ Row Block

ICTXT $= 2$ Column Block

Understanding ddmatrix: Global Matrix

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

ddmatrix: Row Block

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

$$\text{Processor grid} = \left|\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}\right|$$

ddmatrix: Row-Column Block

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

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

ddmatrix: Row Cyclic

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

$$\text{Processor grid} = \left|\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}\right|$$

ddmatrix: Row-Cyclic Column-Block

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

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

ddmatrix: Block-Cyclic

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

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

ddmatrix: Block-Cyclic on a 2×3 Processor Grid

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

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

Understanding ddmatrix: Local View of Block-Cyclic on a 2×3 Processor Grid

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

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

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

Pros and Cons of This Data Structure

Pros

- Robust for matrix computations.

Cons

- Complex layout.

This is why we hide most of the distributed details.

The details are there if you want them (you don't want them).

Methods for class ddmatrix

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

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

Serial Code

```
1 cov(x)
```

Parallel Code

```
1 cov(x)
```

Comparing pbdMPI and pbdDMAT

pbdMPI:

- MPI + sugar.

pbdDMAT:

- Distributed matrices + statistics.
- The `ddmatrix` structure *must* be used for **pbdDMAT**.
- If the data is not 2d block-cyclic compatible, `ddmatrix` will *definitely* give the wrong answer.

9 Introduction to pbdDMAT and the ddmatrix Structure

- Distributed Matrices and Class `ddmatrix`
- Methods for class `ddmatrix`
- Summary

Summary

1. Start by loading the package:

```
1 library(pbdDMAT, quiet = TRUE)
```

2. Always initialize before starting and finalize when finished:

```
1 init.grid()
2
3 # ...
4
5 finalize()
```

Contents

10 Examples Using pbdDMAT

- RandSVD
- Summary

Randomized truncated SVD[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: $\boldsymbol{B} = \widetilde{\boldsymbol{U}} \boldsymbol{\Sigma} \boldsymbol{V}^*$.
6 Set $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.

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 $\widetilde{\boldsymbol{Y}}_j = \boldsymbol{A}^* \boldsymbol{Q}_{j-1}$ and compute its QR factorization $\widetilde{\boldsymbol{Y}}_j = \widetilde{\boldsymbol{Q}}_j \widetilde{\boldsymbol{R}}_j$.
5 Form $\boldsymbol{Y}_j = \boldsymbol{A}\widetilde{\boldsymbol{Q}}_j$ 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),
5          nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

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

Randomized truncated SVD

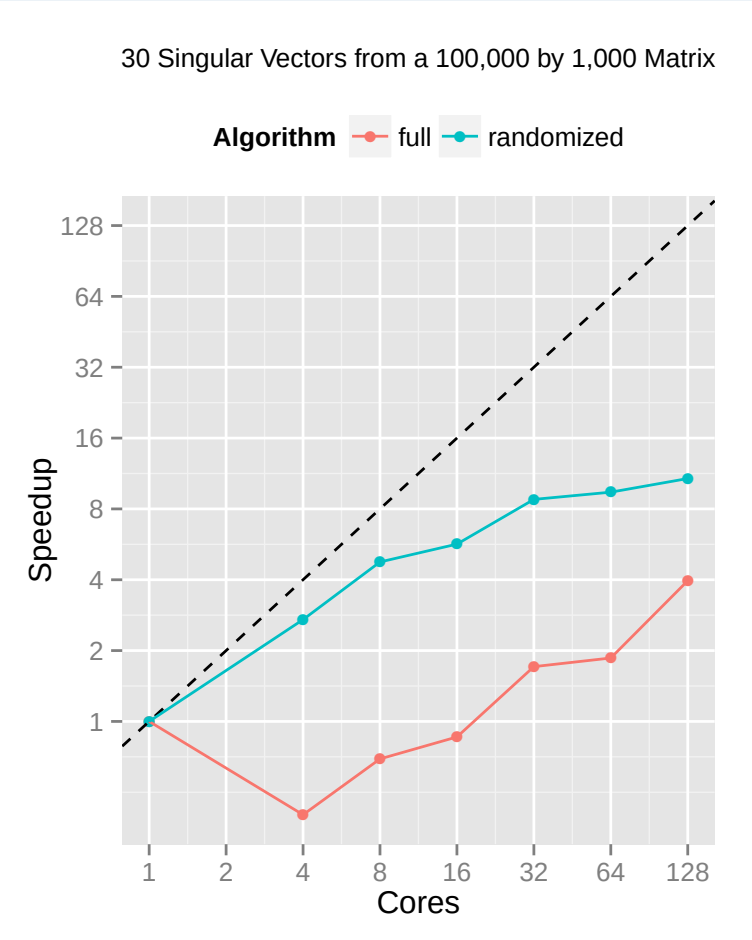
Serial R

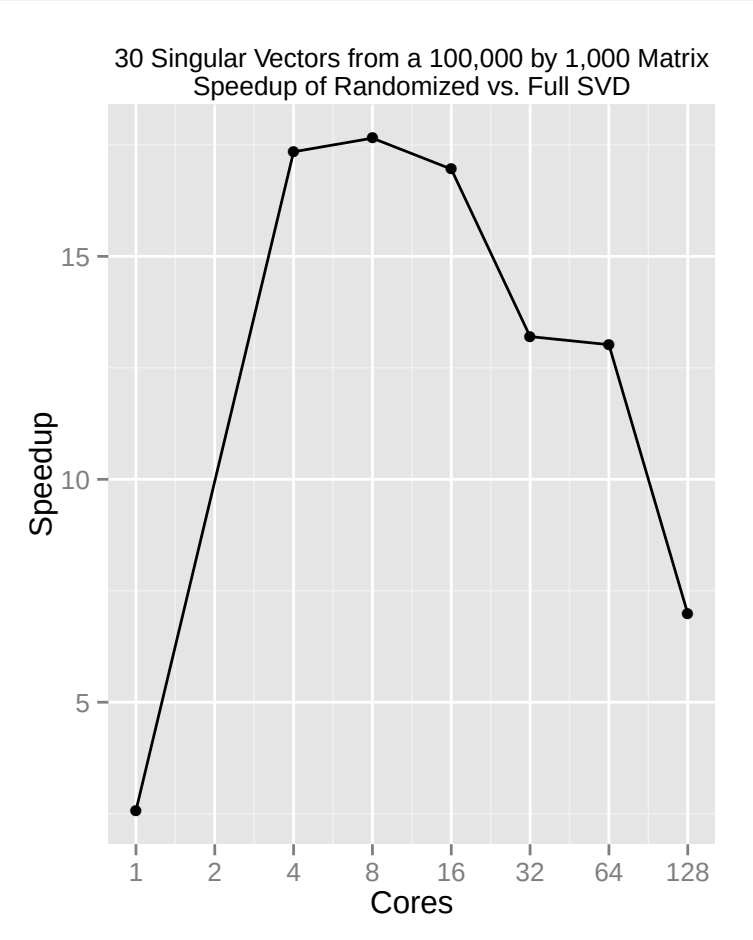
```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  matrix(rnorm(n*2*k),
5          nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

Parallel pbdR

```
1  randSVD <- function(A, k, q=3)
2    {
3      ## Stage A
4      Omega <-  ddmatrix("rnorm",
5          nrow=n, ncol=2*k)
6      Y <- A %*% Omega
7      Q <- qr.Q(qr(Y))
8      At <- t(A)
9      for(i in 1:q)
10       {
11         Y <- At %*% Q
12         Q <- qr.Q(qr(Y))
13         Y <- A %*% Q
14         Q <- qr.Q(qr(Y))
15       }
16
17     ## Stage B
18     B <- t(Q) %*% A
19     U <- La.svd(B)$u
20     U <- Q %*% U
21     U[, 1:k]
22   }
```

From journal to scalable code and scaling data in one day.





10 Examples Using pbdDMAT

- RandSVD
- Summary

Summary

- **pbdDMAT** makes distributed (dense) linear algebra easier
- Enables rapid prototyping at large scale

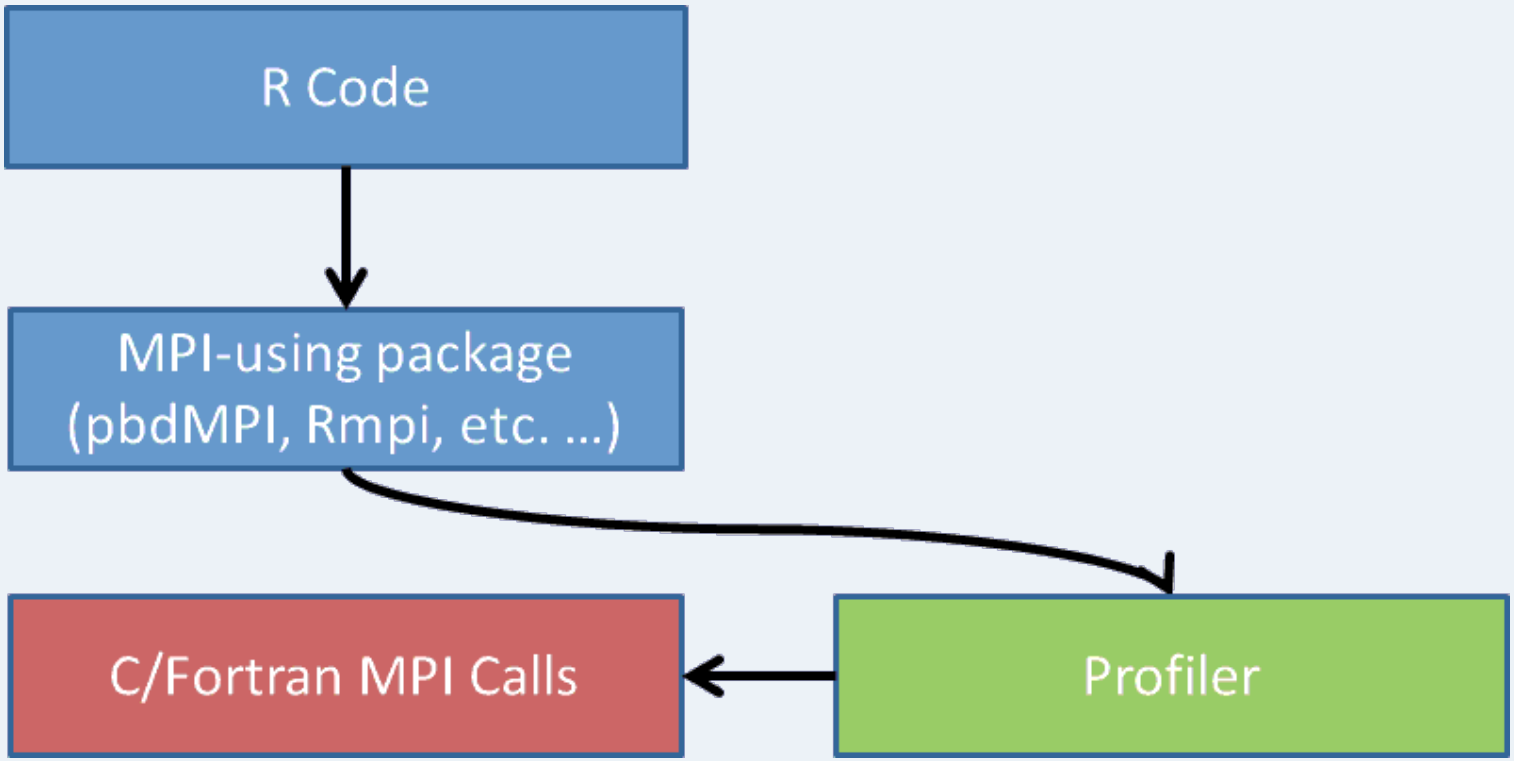
Contents

Introduction to **pbdPROF**

- Successful Google Summer of Code 2013 project.
- Available on the CRAN.
- Enables profiling of MPI-using R scripts.
- pbdR packages officially supported; can work with others...
- Also reads, parses, and plots profiler outputs.

How it works

MPI calls get hijacked by profiler and logged:



Introduction to **pbdPROF**

- Currently supports the profilers **fpmpi** and **mpiP**.
- **fpmpi** is distributed with **pbdPROF** and installs easily, but offers minimal profiling capabilities.
- **mpiP** is fully supported also, but you have to install and link it yourself.

Installing **pbdPROF**

1. Build **pbdPROF**.
2. Rebuild **pbdMPI** (linking with **pbdPROF**).
3. Run your analysis as usual.
4. Interactively analyze profiler outputs with **pbdPROF**.

This is explained at length in the **pbdPROF** vignette.

Rebuild **pbdMPI**

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

- Any package which explicitly links with an MPI library must be rebuilt in this way (**pbdMPI**, **Rmpi**, ...).
- Other pbdR packages link with **pbdMPI**, and so do not need to be rebuilt.
- See **pbdPROF** vignette if something goes wrong.

An Example from **pbdDMAT**

- Compute SVD in **pbdDMAT** package.
- Profile MPI calls with **mpiP**.

Example Script

my_svd.r

```
1  library(pbdMPI, quietly=TRUE)
2  library(pbdDMAT, quietly=TRUE)
3  init.grid()
4
5
6  n <- 1000
7  x <- ddmatrix("rnorm", n, n)
8
9  my.svd <- La.svd(x)
10
11
12 finalize()
```

Example Script

Run example with 4 ranks:

```
$ mpirun -np 4 Rscript my_svd.r
mpiP:
mpiP: mpiP: mpiP V3.3.0 (Build Sep 23 2013/14:00:47)
mpiP: Direct questions and errors to
    mpip-help@lists.sourceforge.net
mpiP:
Using 2x2 for the default grid size

mpiP:
mpiP: Storing mpiP output in [./R.4.5944.1.mpiP].
mpiP:
```

Read Profiler Data into R

Interactively (or in batch) Read in Profiler Data

```
1 library(pbdPROF)
2 prof.data <- read.prof("R.4.28812.1.mpiP")
```

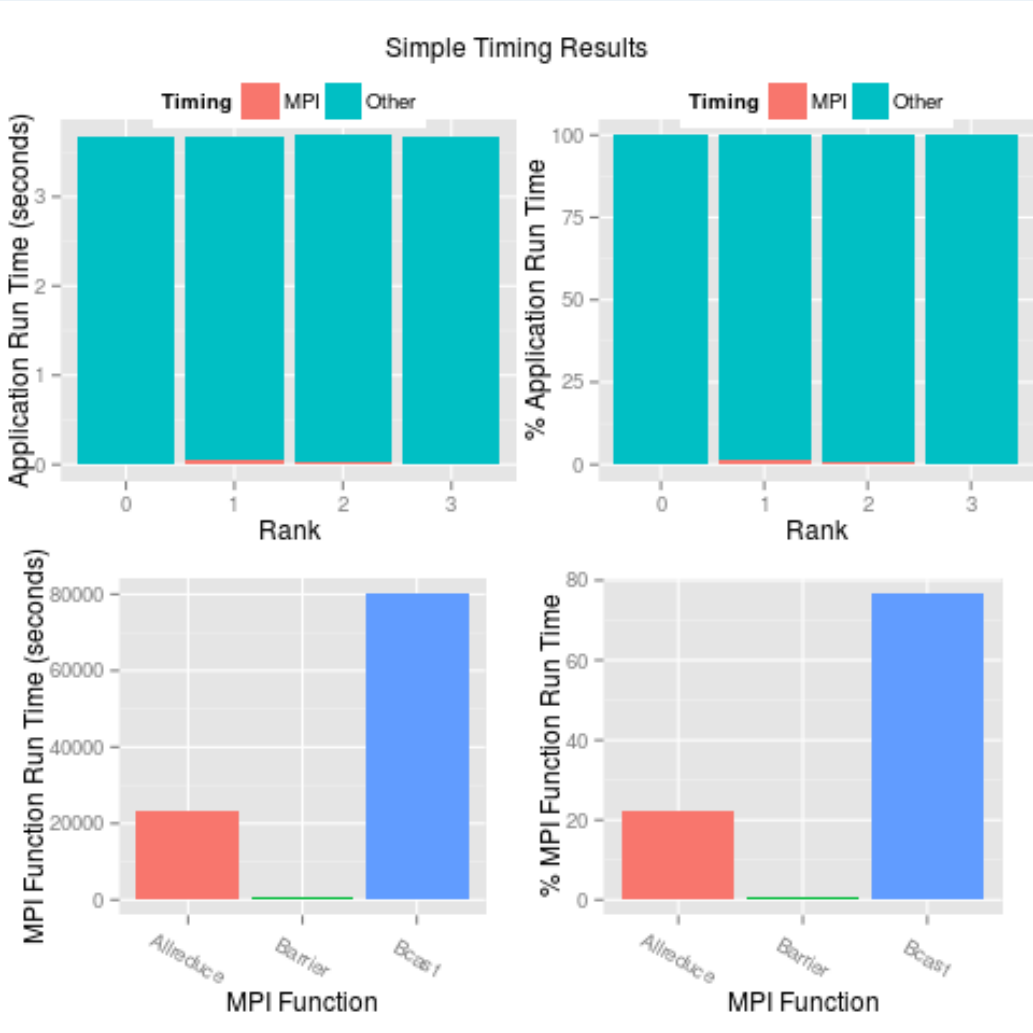
Partial Output of Example Data

```
> prof.data
An mpip profiler object:
[[1]]
  Task AppTime MPITime MPI.
1    0    5.71  0.0387 0.68
2    1    5.70  0.0297 0.52
3    2    5.71  0.0540 0.95
4    3    5.71  0.0355 0.62
5    *   22.80  0.1580 0.69

[[2]]
   ID Lev File.Address Line_Parent_Funct  MPI_Call
1   1   0 1.397301e+14          [unknown] Allreduce
2   2   0 1.397301e+14          [unknown]     Bcast
```

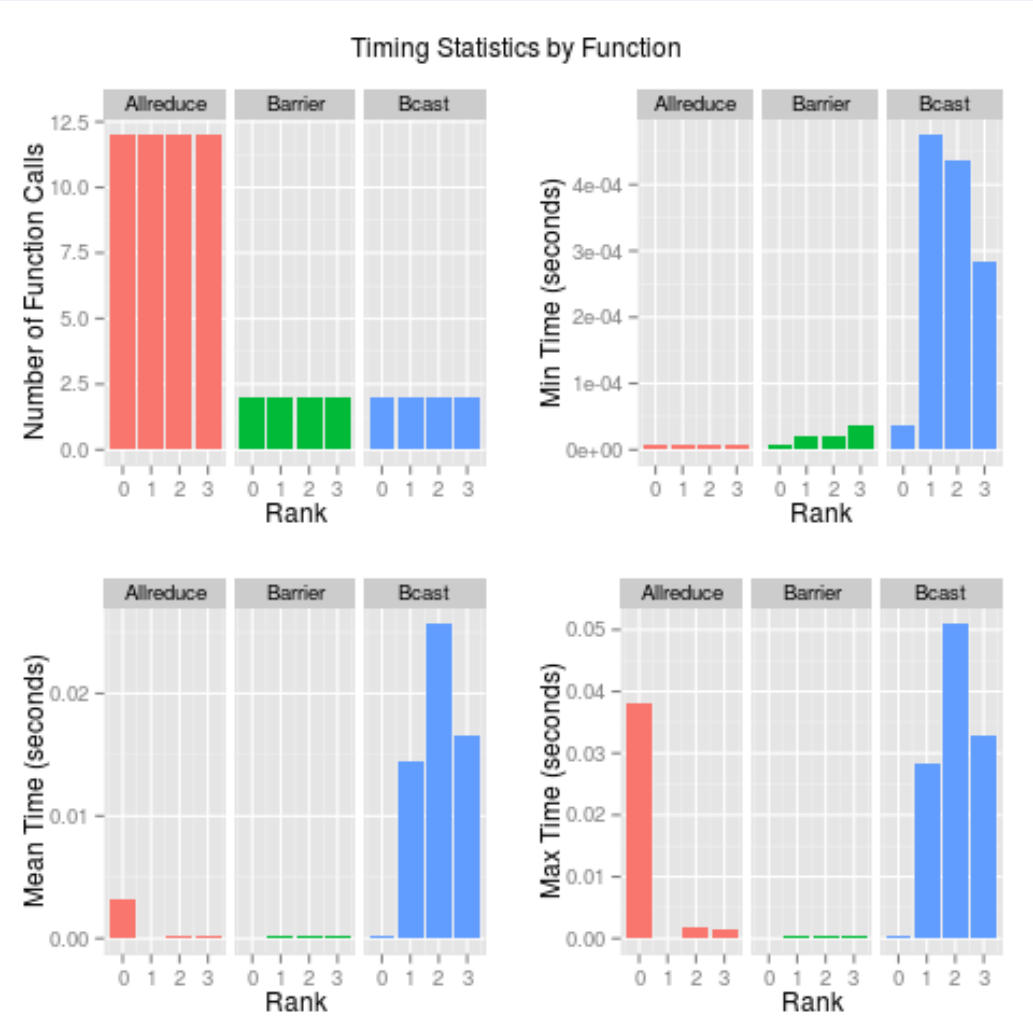
Generate plots

```
1 plot(prof.data)
```



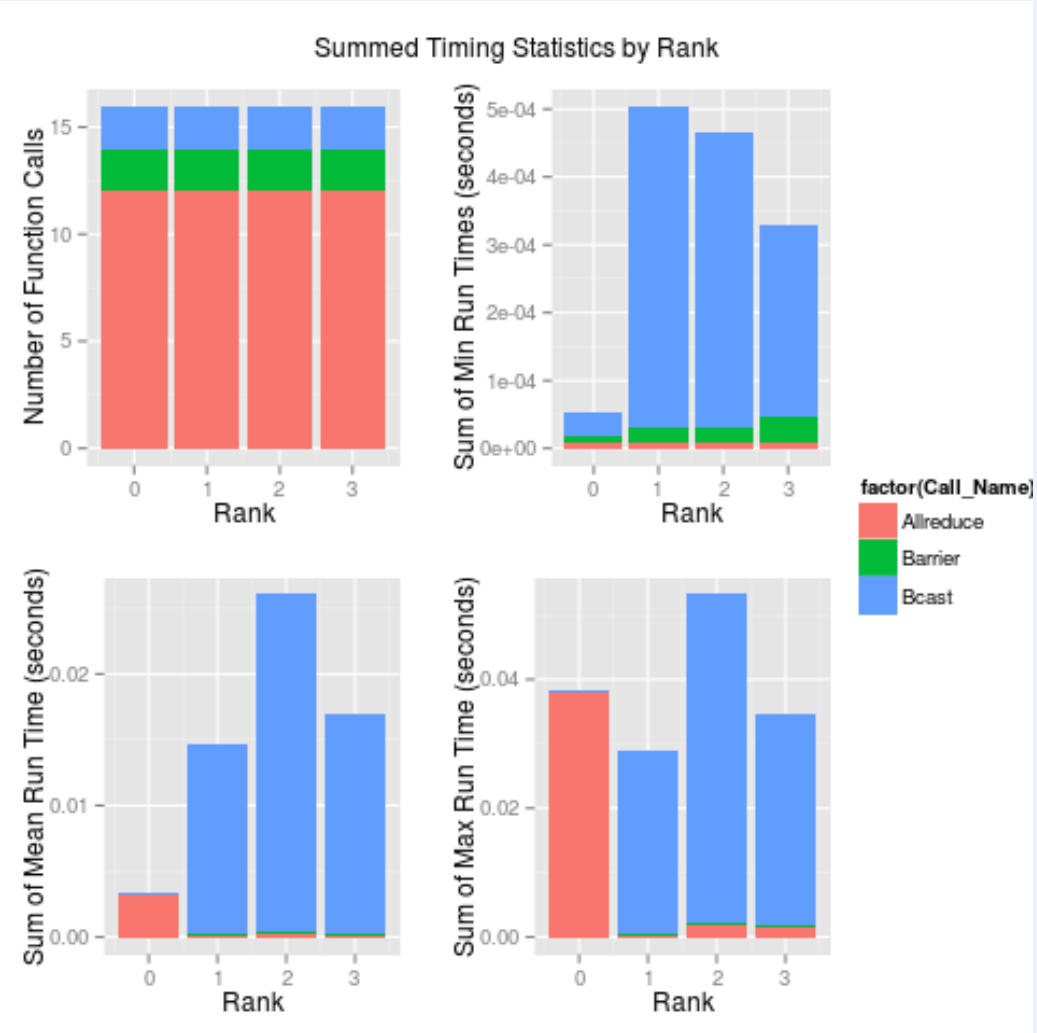
Generate plots

```
1 plot(prof.data, plot.type="stats1")
```



Generate plots

```
1 plot(prof.data, plot.type="stats2")
```

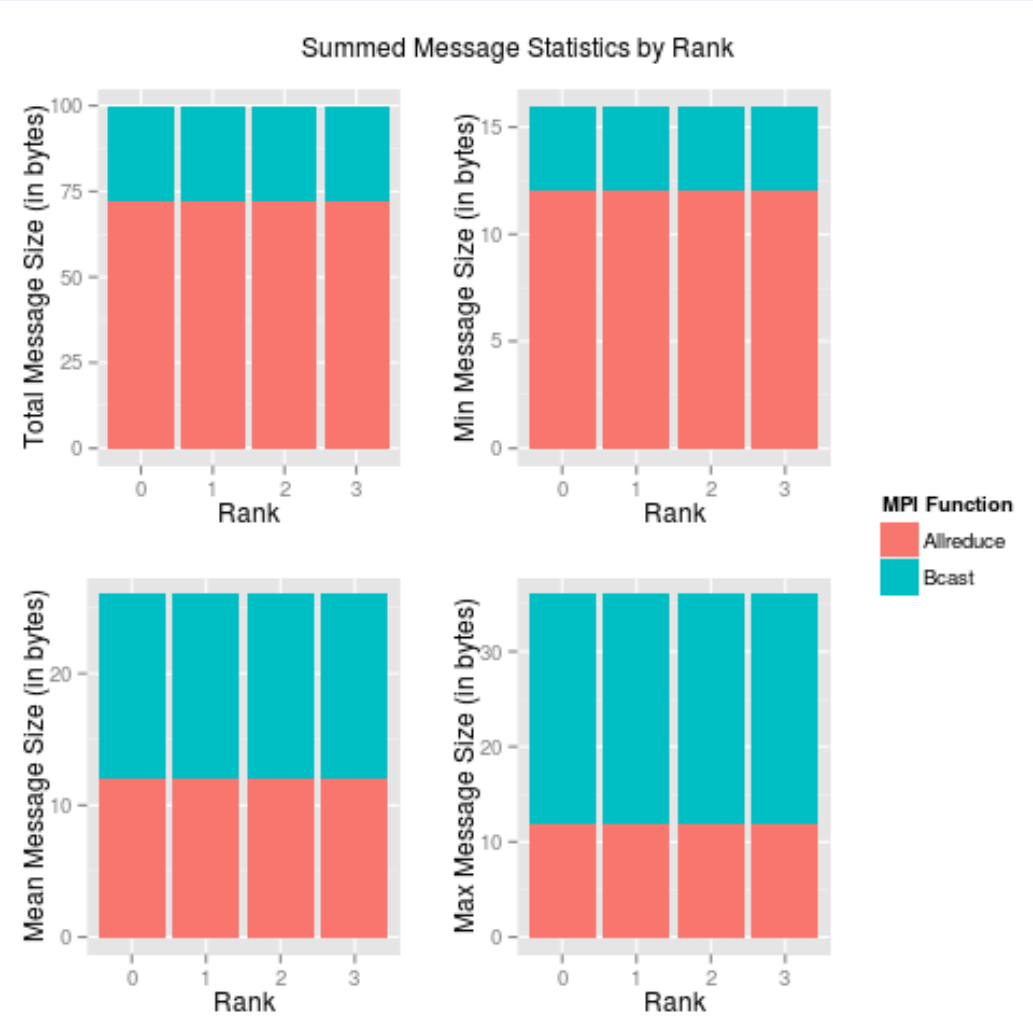


Generate plots

```
1 plot(prof.data, plot.type="messages1")
```

Generate plots

```
1 plot(prof.data, plot.type="messages2")
```



Summary

- **pbdPROF** offers tools for profiling R-using MPI codes.
- Easily builds **fpmpi**; also supports **mpiP**.

Contents

Summary

- Profile your code to understand your bottlenecks
- pbdR makes distributed parallelism with R easier
- pbdR connects R to HPC scalable libraries
- Distributing data to multiple nodes
- For truly large data, I/O must be parallel

The pbdR Project

- Our website: http://r-pbd.org/
- Email us at: RBigData@gmail.com
- Our google group: http://group.r-pbd.org/

Where to begin?

- The **pbdDEMO** package
  http://cran.r-project.org/web/packages/pbdDEMO/
- The **pbdDEMO** Vignette: http://goo.gl/HZkRt

Thanks for coming!

Questions?

http://r-pbd.org/