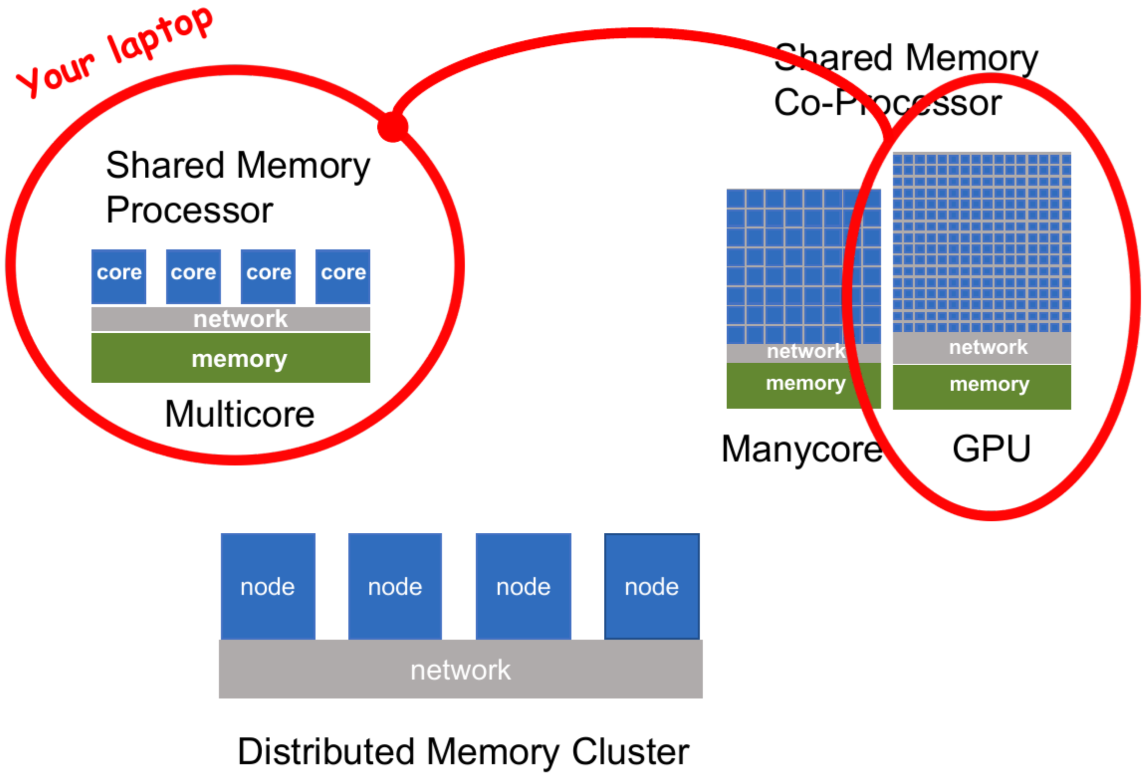
Data Science with R and pbdR at ORNL: From the CADES Cloud to the OLCF

Part 2: pbdR and the OLCF

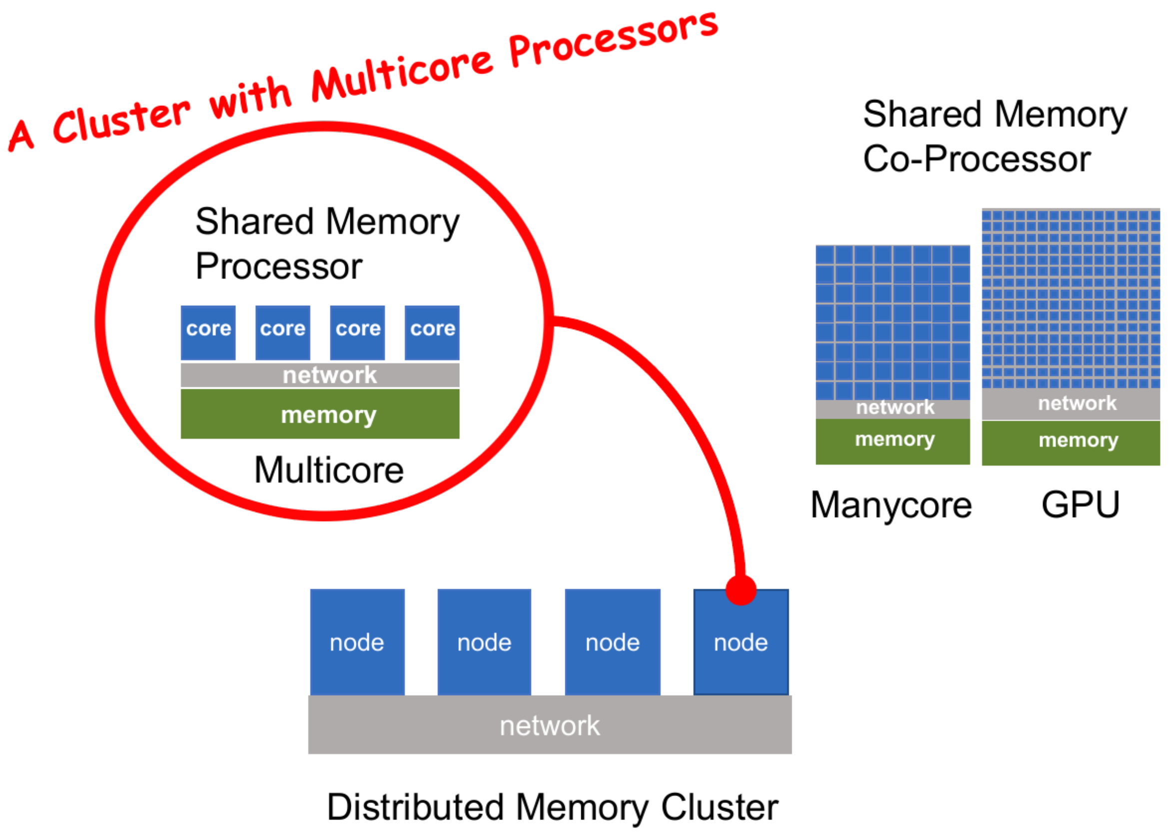
Drew Schmidt and George Ostrouchov
6/18/2018

Parallel Hardware

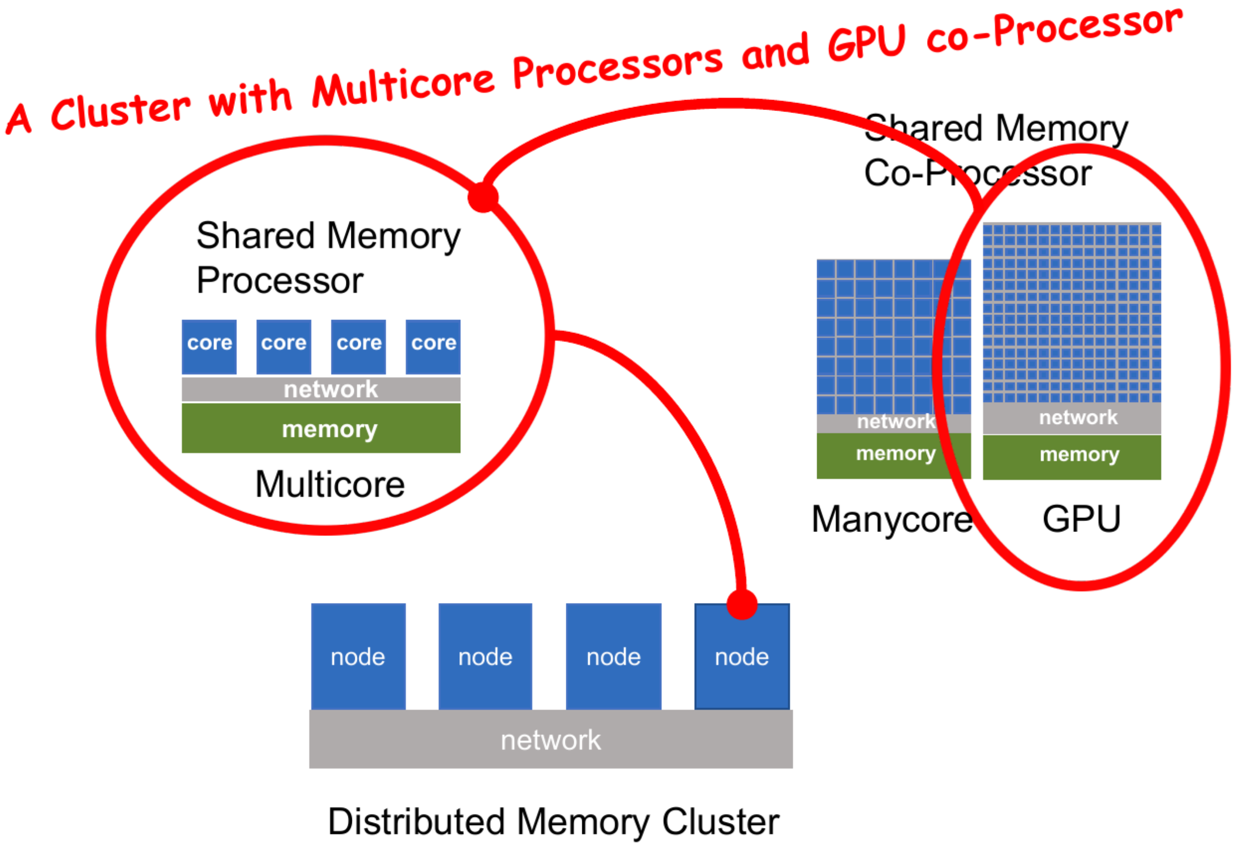
Three Flavors of Hardware



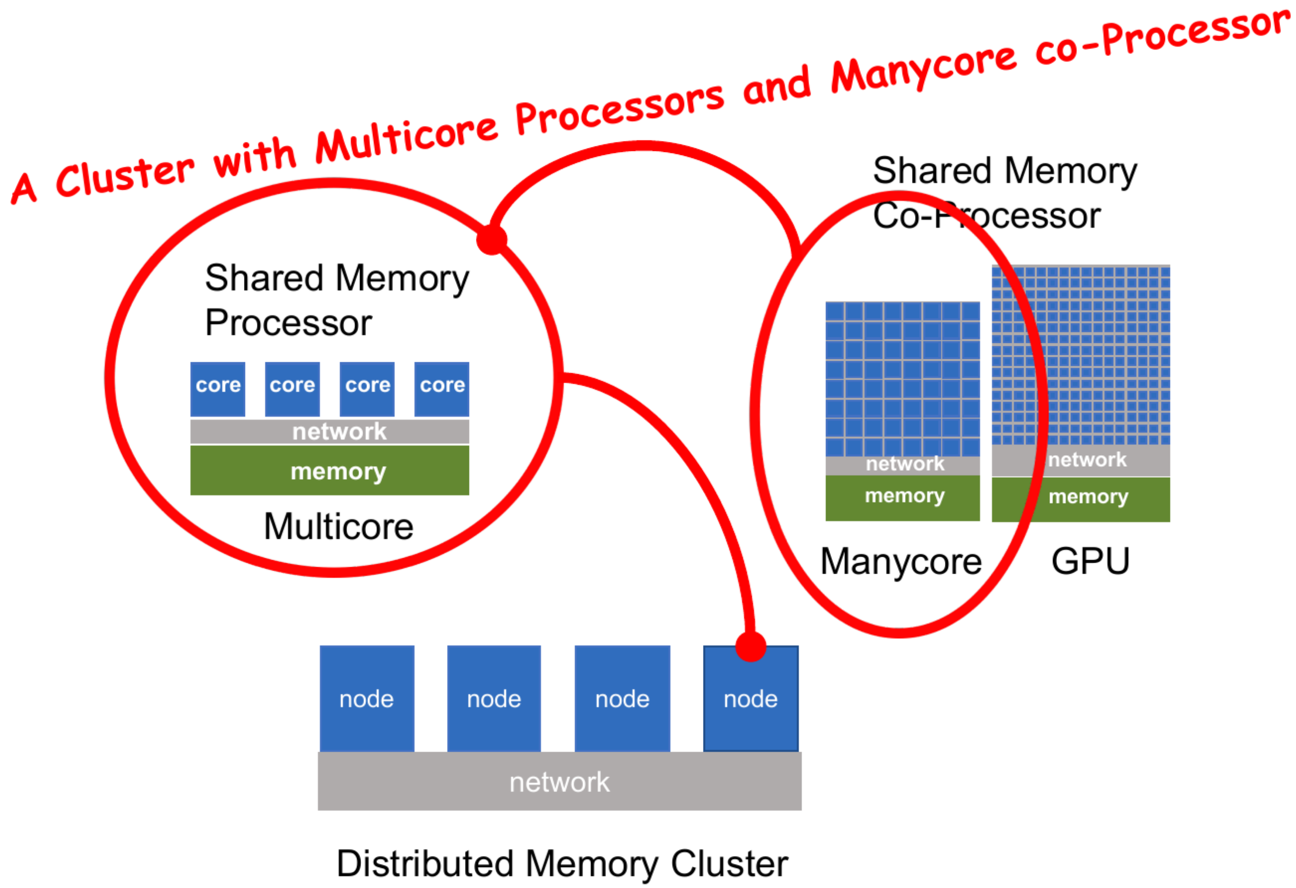
Three Flavors of Hardware



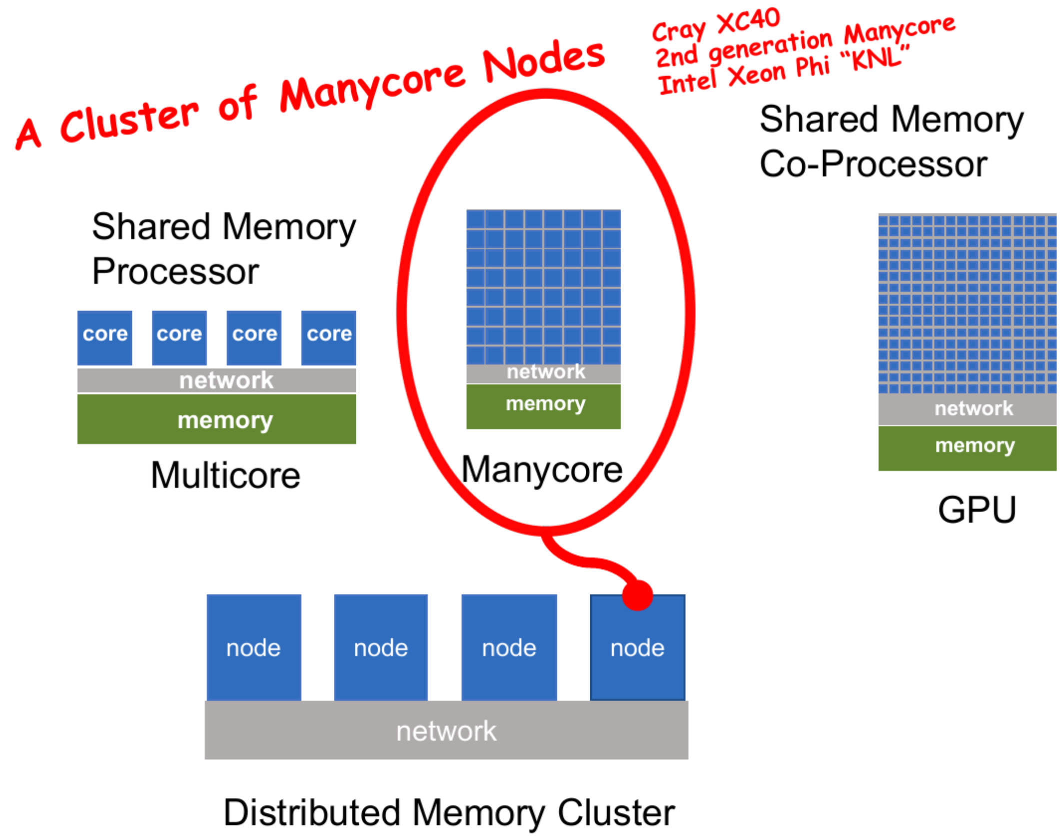
Three Flavors of Hardware



Three Flavors of Hardware

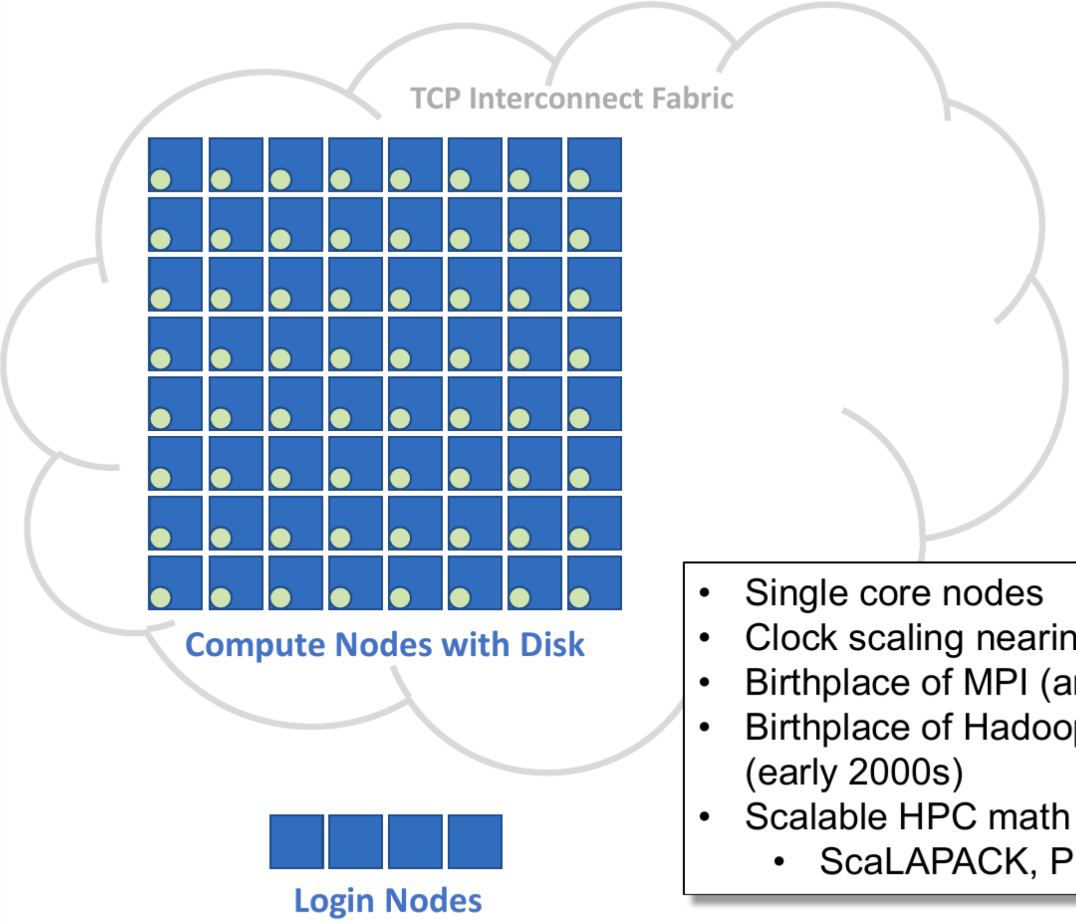


Three Flavors of Hardware



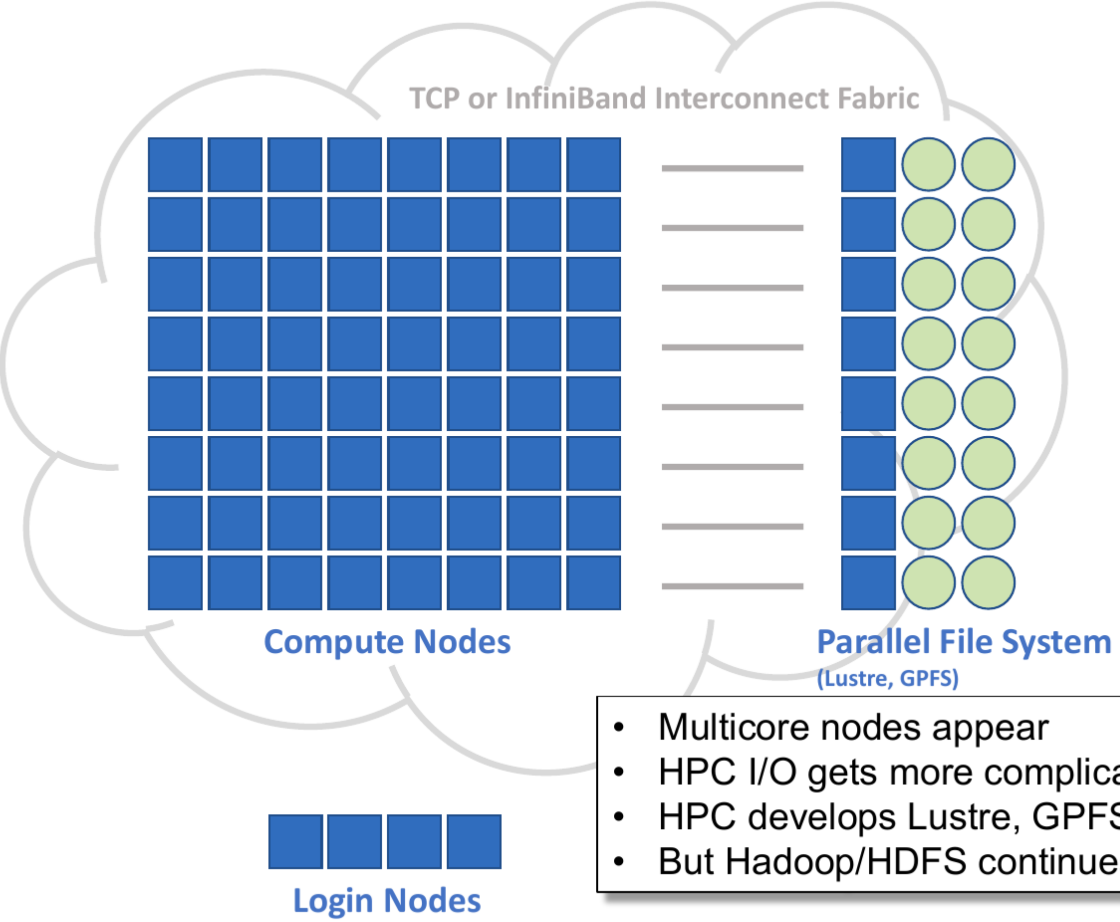
A Bit of Cluster History...

Commodity Cluster Before 2003



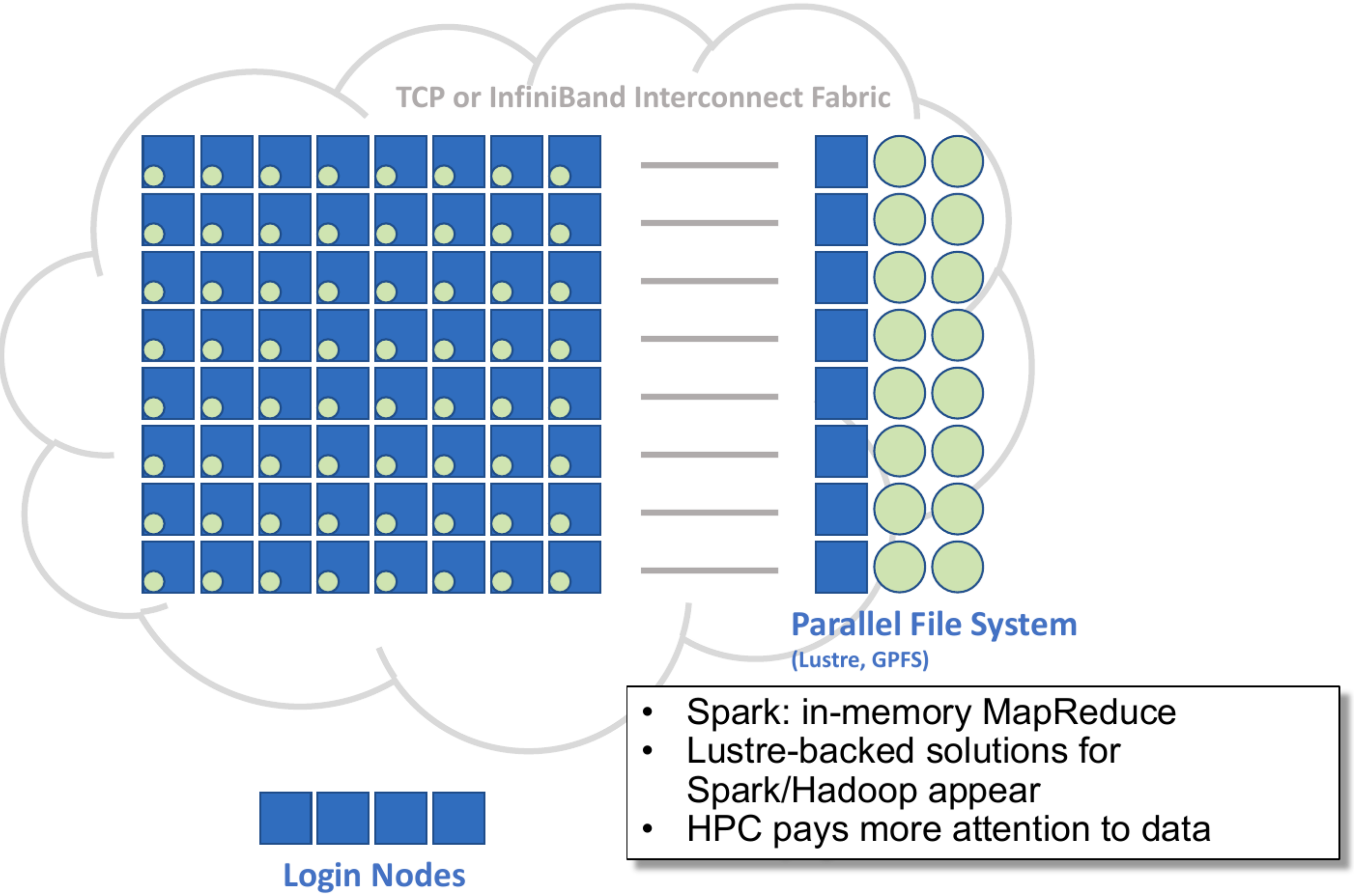
- Single core nodes
- Clock scaling nearing its end
- Birthplace of MPI (around 1990)
- Birthplace of Hadoop/HDFS/MapReduce (early 2000s)
- Scalable HPC math libraries are built
  - ScaLAPACK, PETSc, Trillinos, …

HPC Introduces Diskless Compute Nodes ~2003



- Multicore nodes appear
- HPC I/O gets more complicated
- HPC develops Lustre, GPFS, ADIOS
- But Hadoop/HDFS continues diskfull

Disk Comes Back as SSD ~2010



TCP or InfiniBand Interconnect Fabric

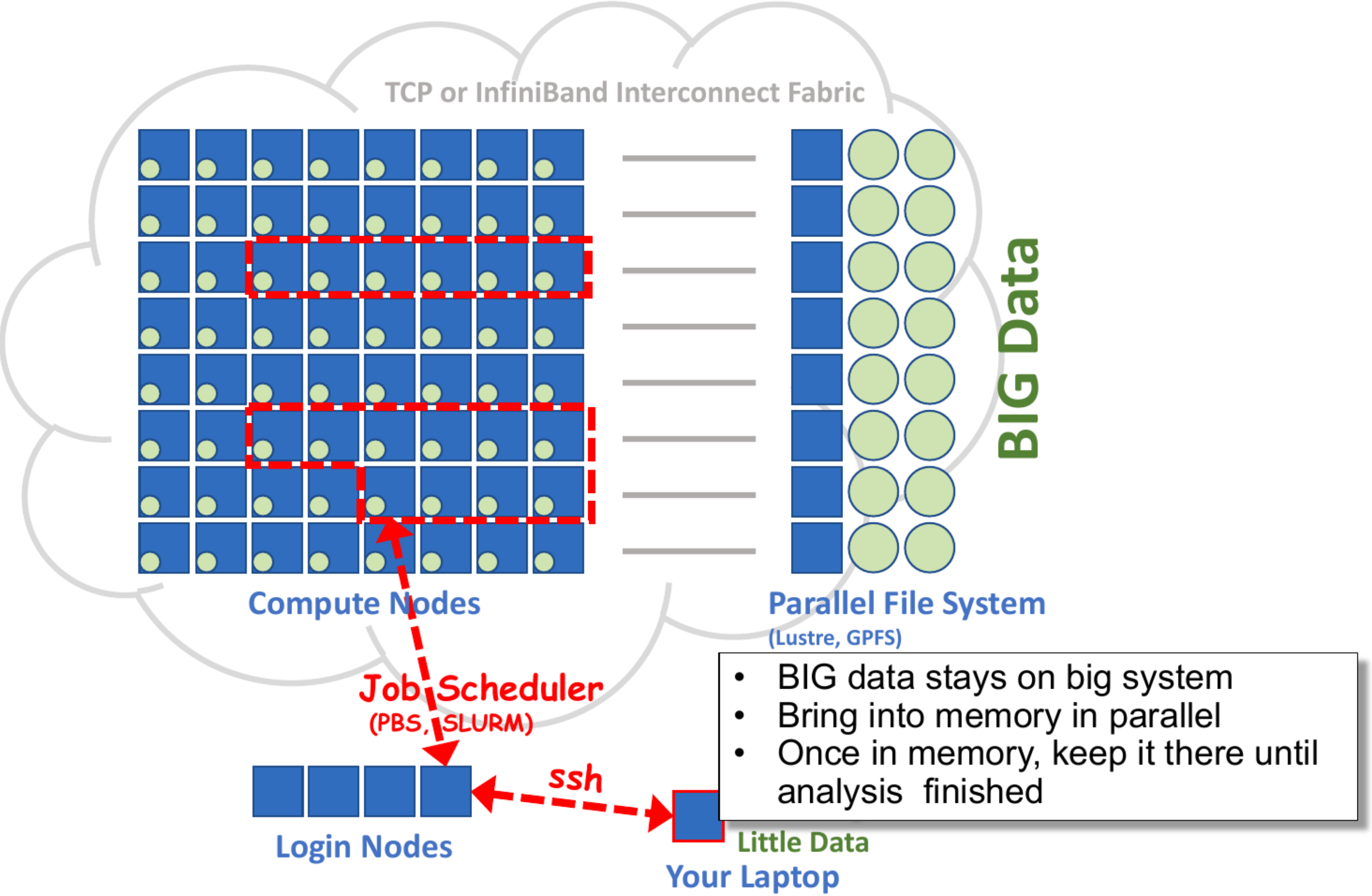
Parallel File System

(Lustre, GPFS)

Login Nodes

- Spark: in-memory MapReduce
- Lustre-backed solutions for Spark/Hadoop appear
- HPC pays more attention to data

Working with Today's HPC Systems



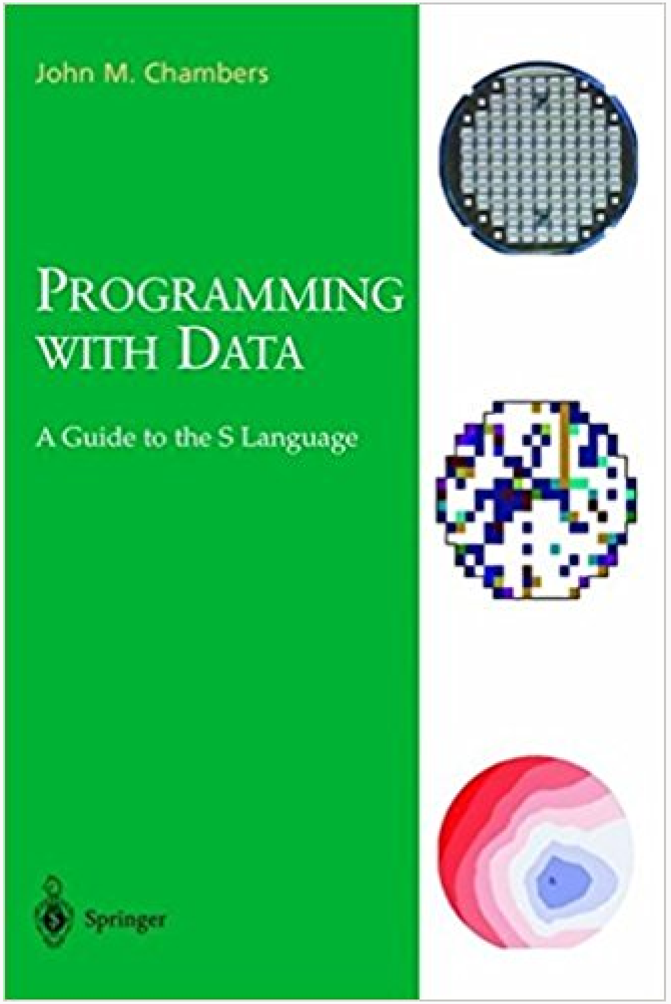
- BIG data stays on big system
- Bring into memory in parallel
- Once in memory, keep it there until analysis finished

pbdR

pbdR

What is it?

- An acronym
  - *Programming with Big Data in R*
  - *Parallel Big Data R*
  - *Pretty Bad for Dyslexics*
- A set of R packages
- Core Team: Wei-Chen Chen, George Ostrouchov, Drew Schmidt



John M. Chambers

PROGRAMMING WITH DATA

A Guide to the S Language

Springer

pbdR

Strive for *Productivity, Portability, Performance*

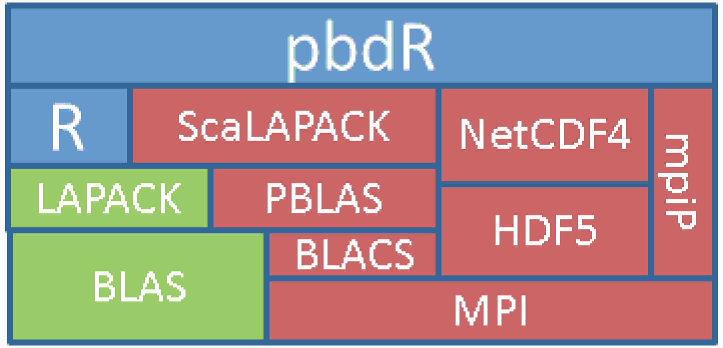
- Bridge high-performance computing with high-productivity of R language
- Keep syntax *identical* to R, when possible.
- Software reuse philosophy:
  - Don't reinvent the wheel when possible
  - Introduce HPC standards with R flavor
  - Use scalable HPC libraries with R convenience
- Simplify and use R intelligence where possible

pbdR

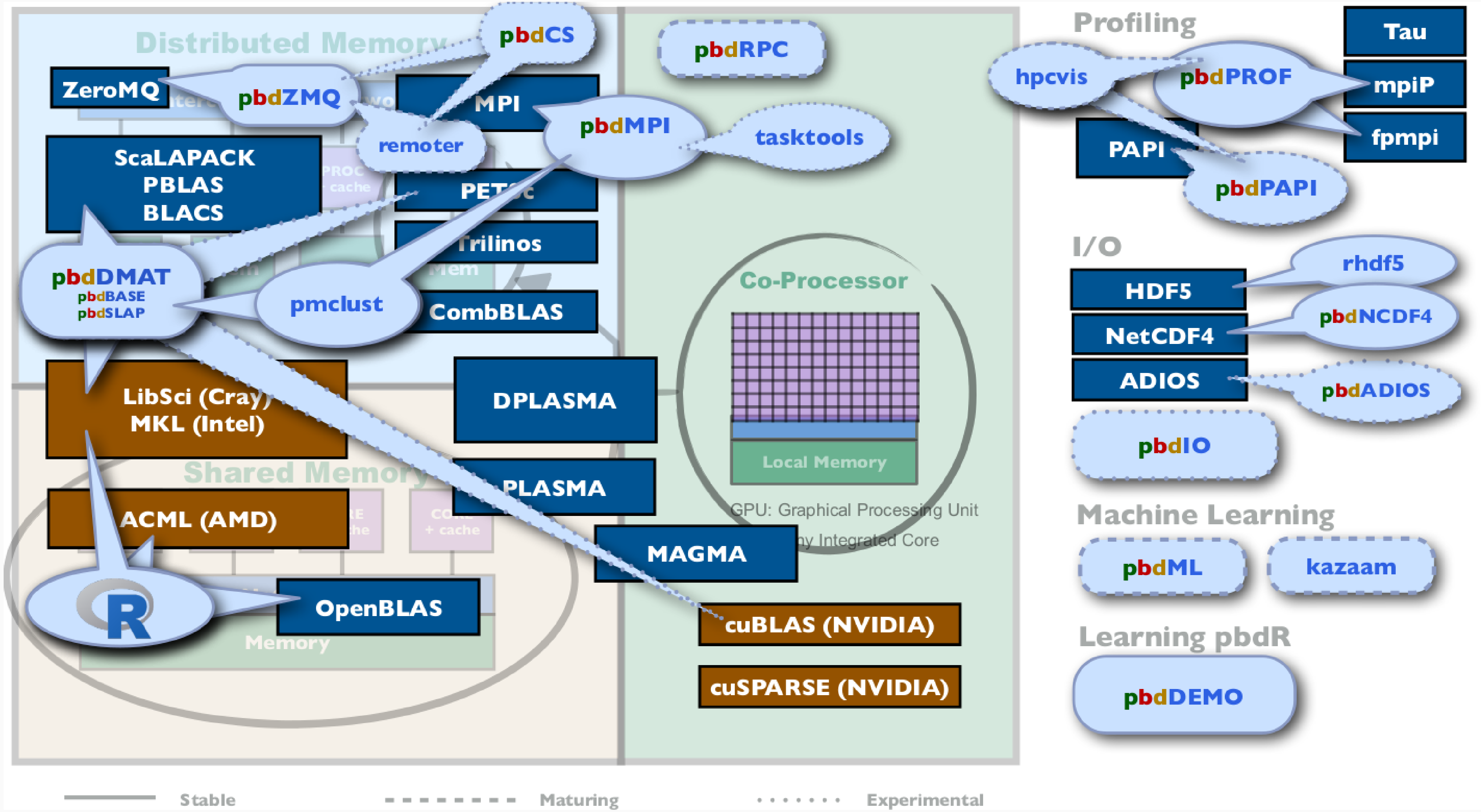
- MPI packages
    - pbdMPI
    - pbdSLAP, pbdBASE, pbdDMAT, pbdML, pmclust
    - kazaam
    - tasktools
- Communication tools
    - pbdZMQ
    - remoter
    - pbdCS
- Profilers
    - pbdPROF
    - pbdPAPI
    - hpcvis
- I/O packages
    - pbdIO
    - pbdNCDF4
    - pbdADIOS
    - hdfio (soon)

pbdR

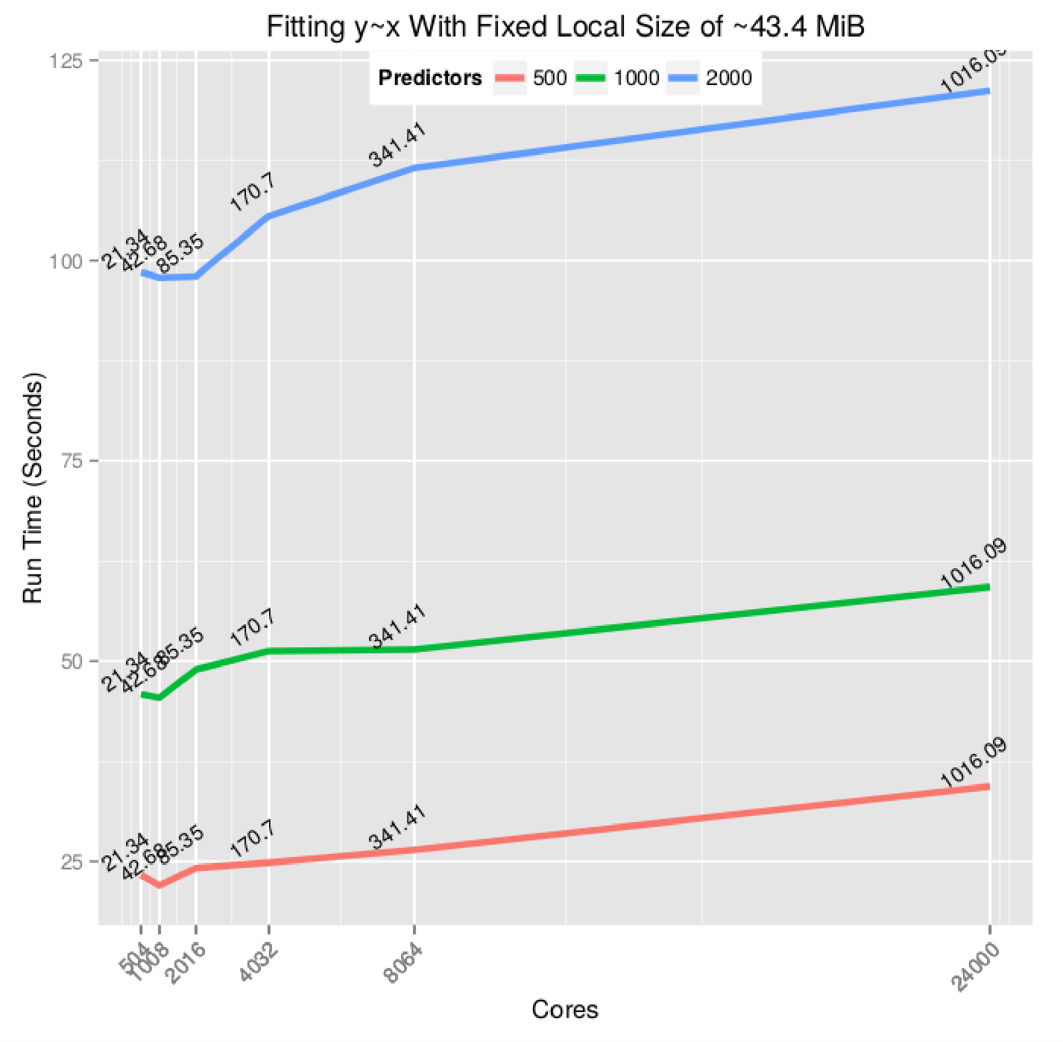
Programming with Big Data in R



HPC Libraries and Their R/pbdR Connections

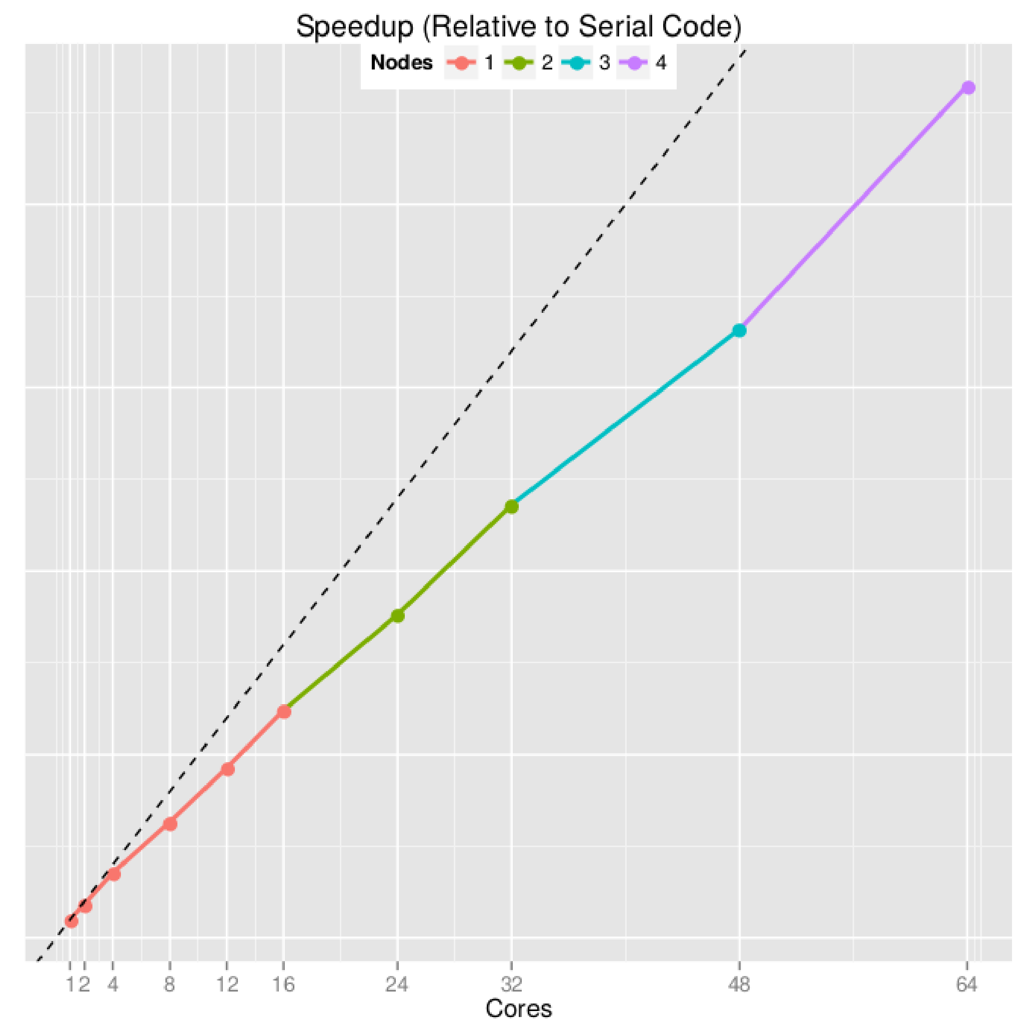


Least Squares Benchmark



```r
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()
```

Matrix Exponentiation Benchmark



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

dx = ddmatrix("rnorm", 5000, 5000)
expm(dx)

finalize()
```

Other (distributed) HPC Packages for R

- Rmpi
- A handful of hadoop/spark packages
- That's about it...

Rmpi vs pbdMPI

- Rmpi can be used interactively. pbdMPI is batch (when used without the client/server)
- pbdMPI often easier to install
- pbdMPI has simpler syntax

Rmpi

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

pbdMPI

```r
allreduce(x)
```

Types in R

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(2)
```

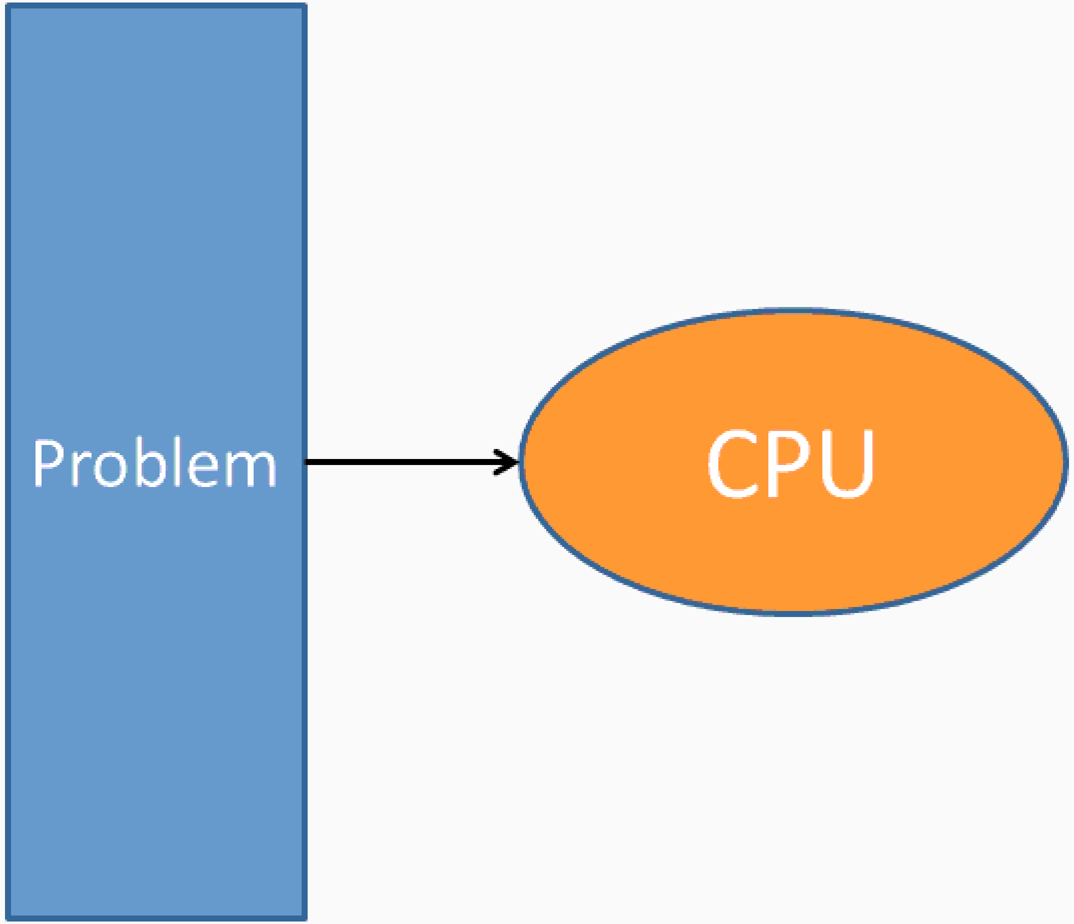
```
## [1] "double"
```

```
typeof(1:2)
```

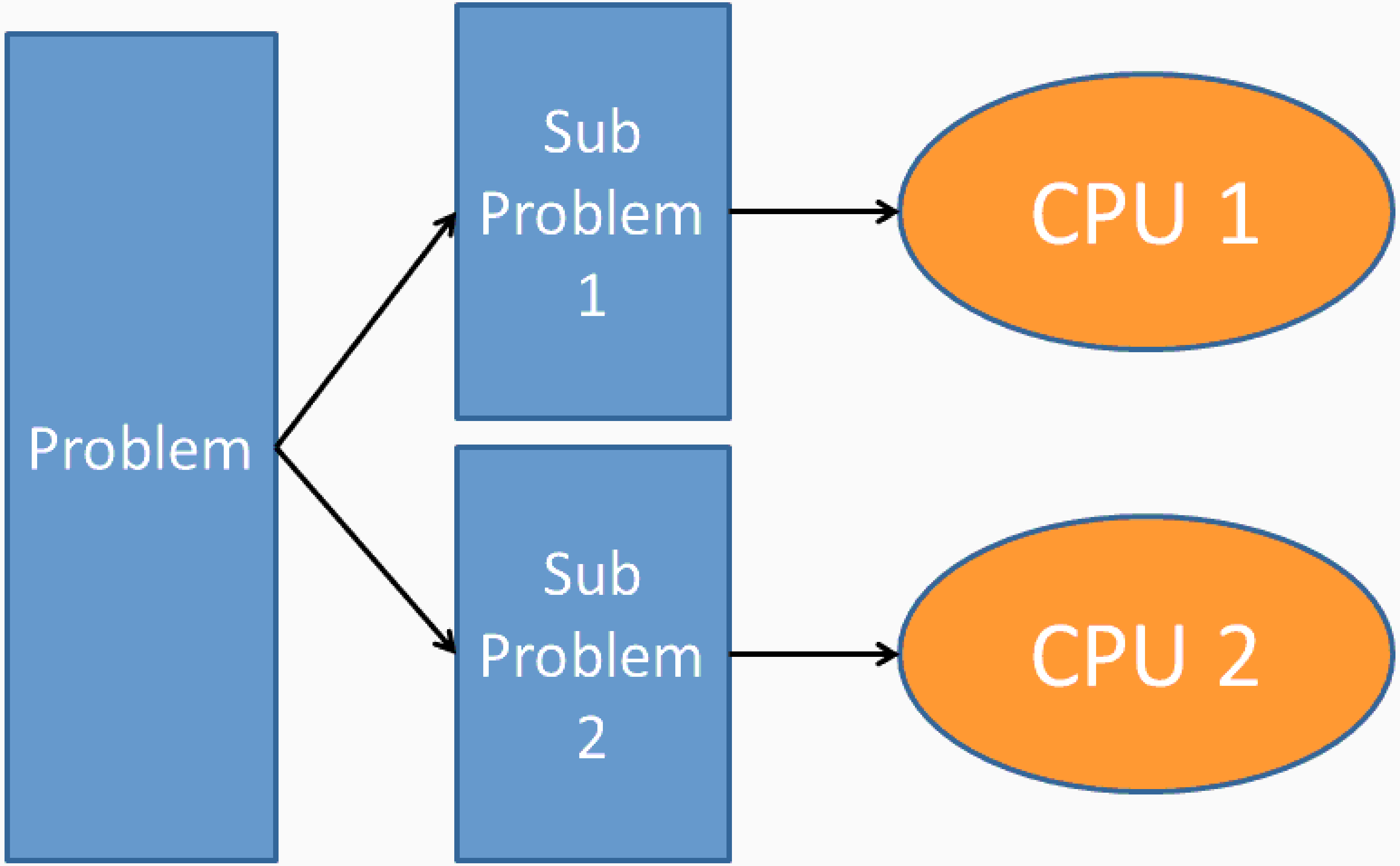
```
## [1] "integer"
```

MPI with pbdMPI

Parallelism



Parallelism



Problem

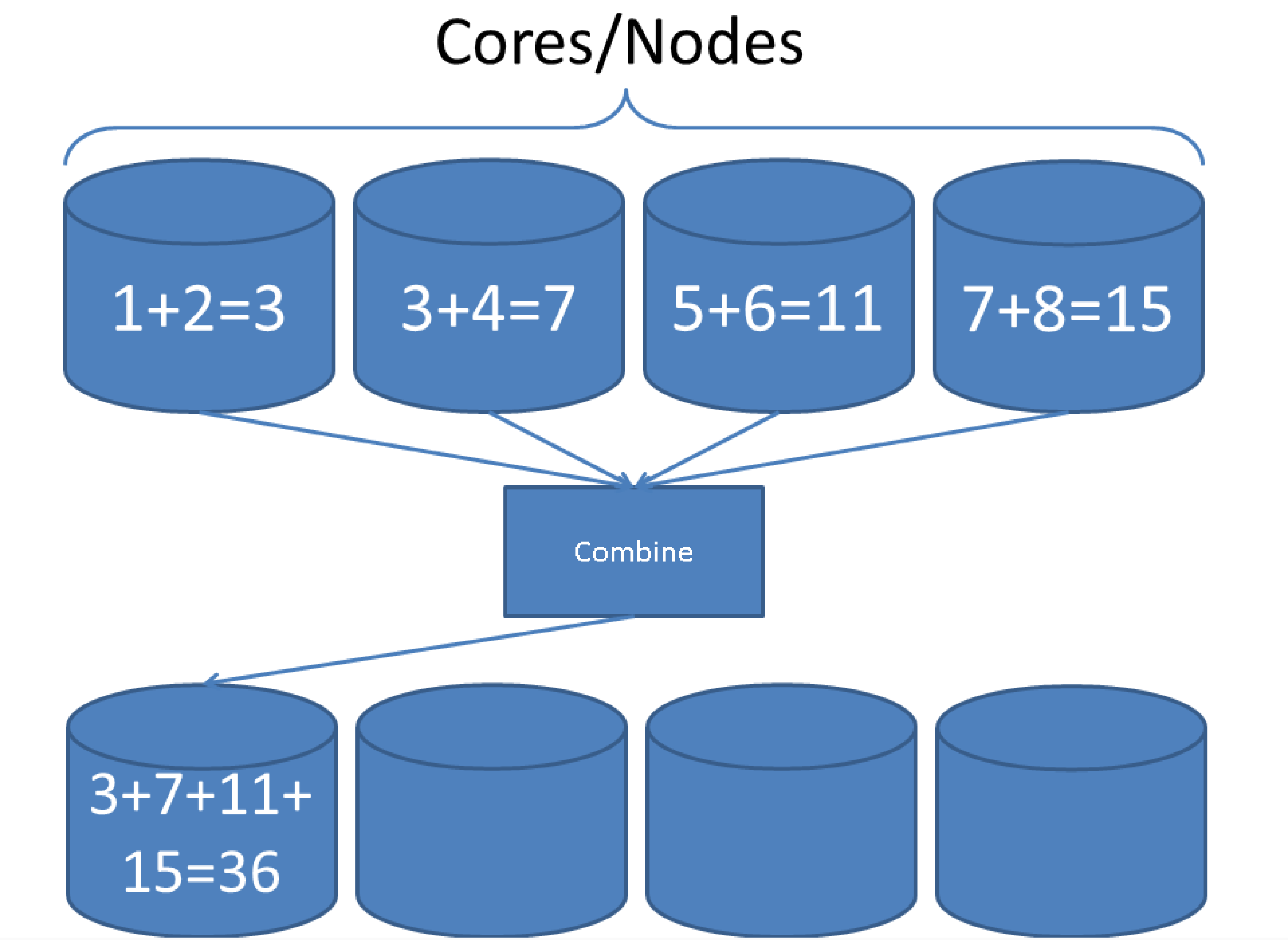
Sub Problem 1

Sub Problem 2

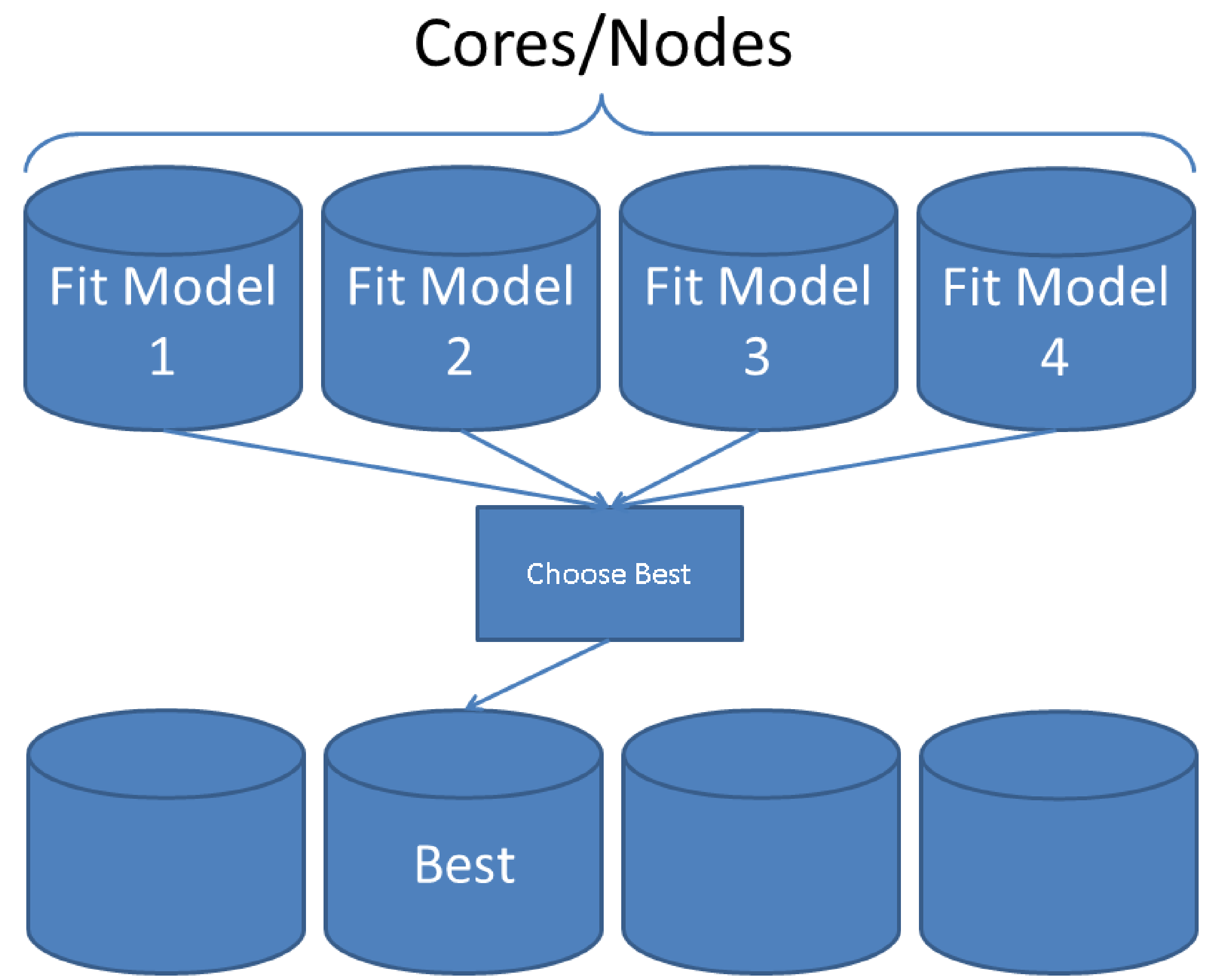
CPU 1

CPU 2

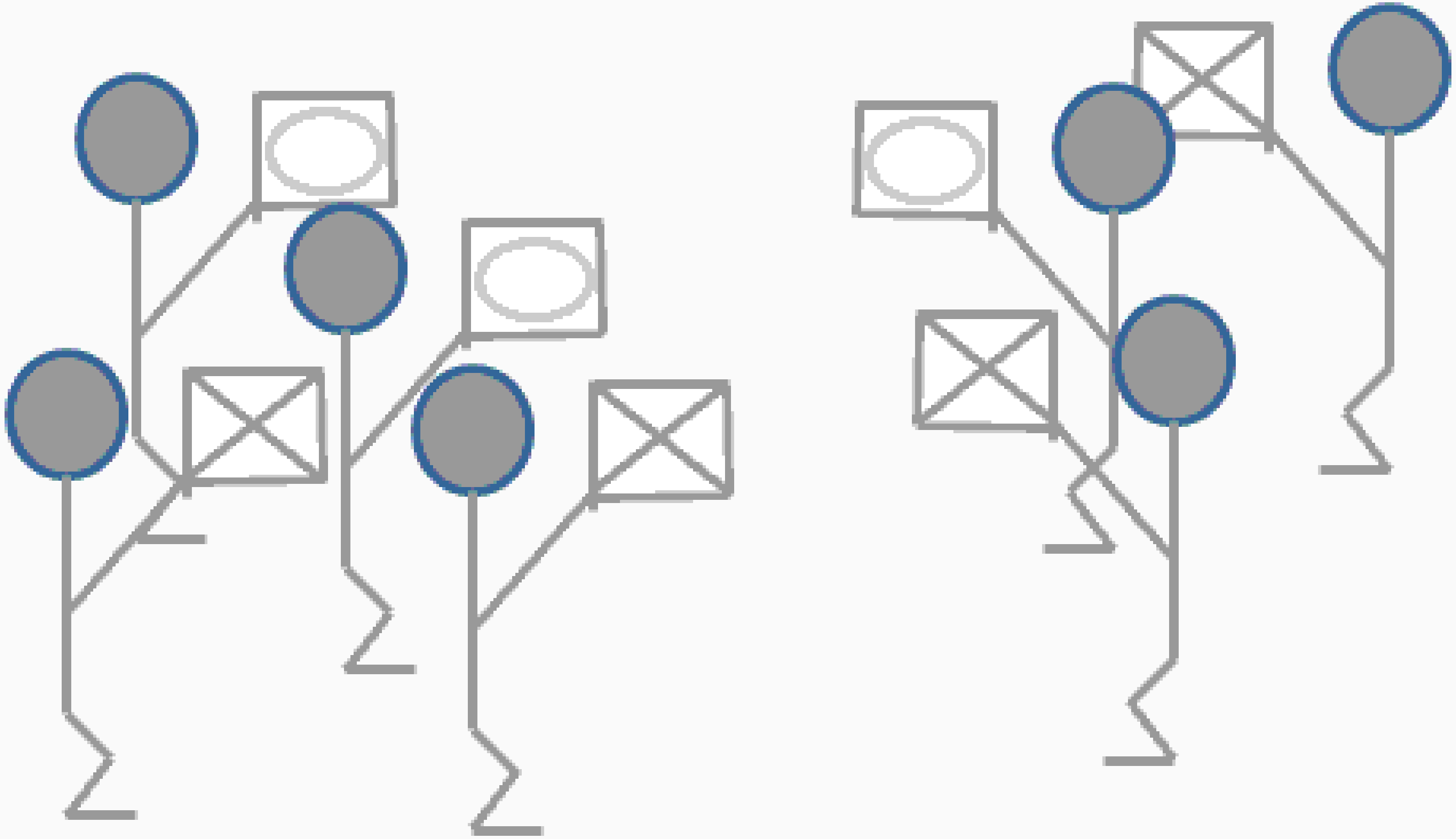
Parallelism



Parallelism



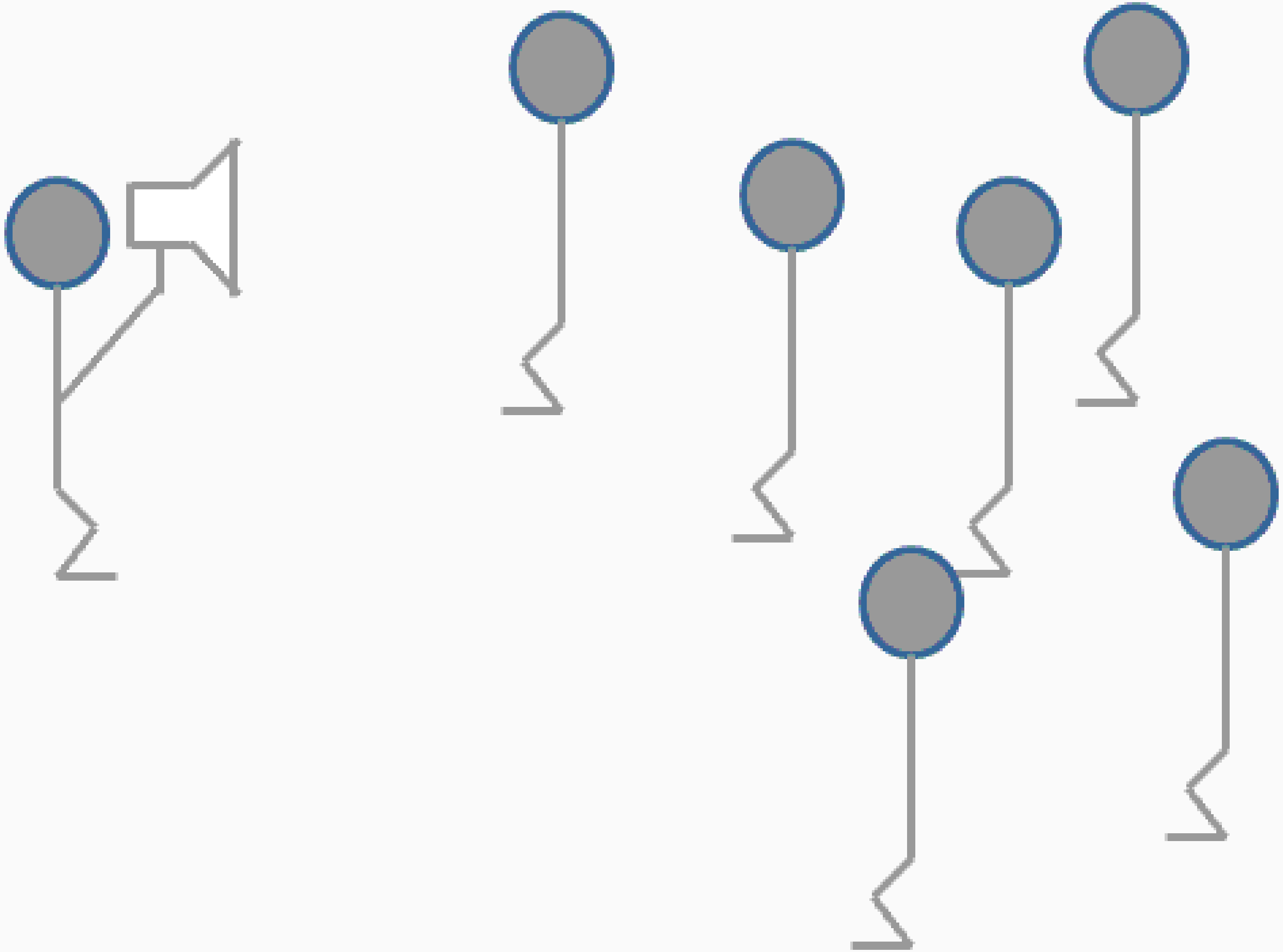
MPI Operations: Reduce



MPI Operations: Reduce

```r
library(pbdMPI)

nranks = comm.size()
ret = allreduce(1)
comm.print(nranks)
comm.print(ret)

finalize()
```

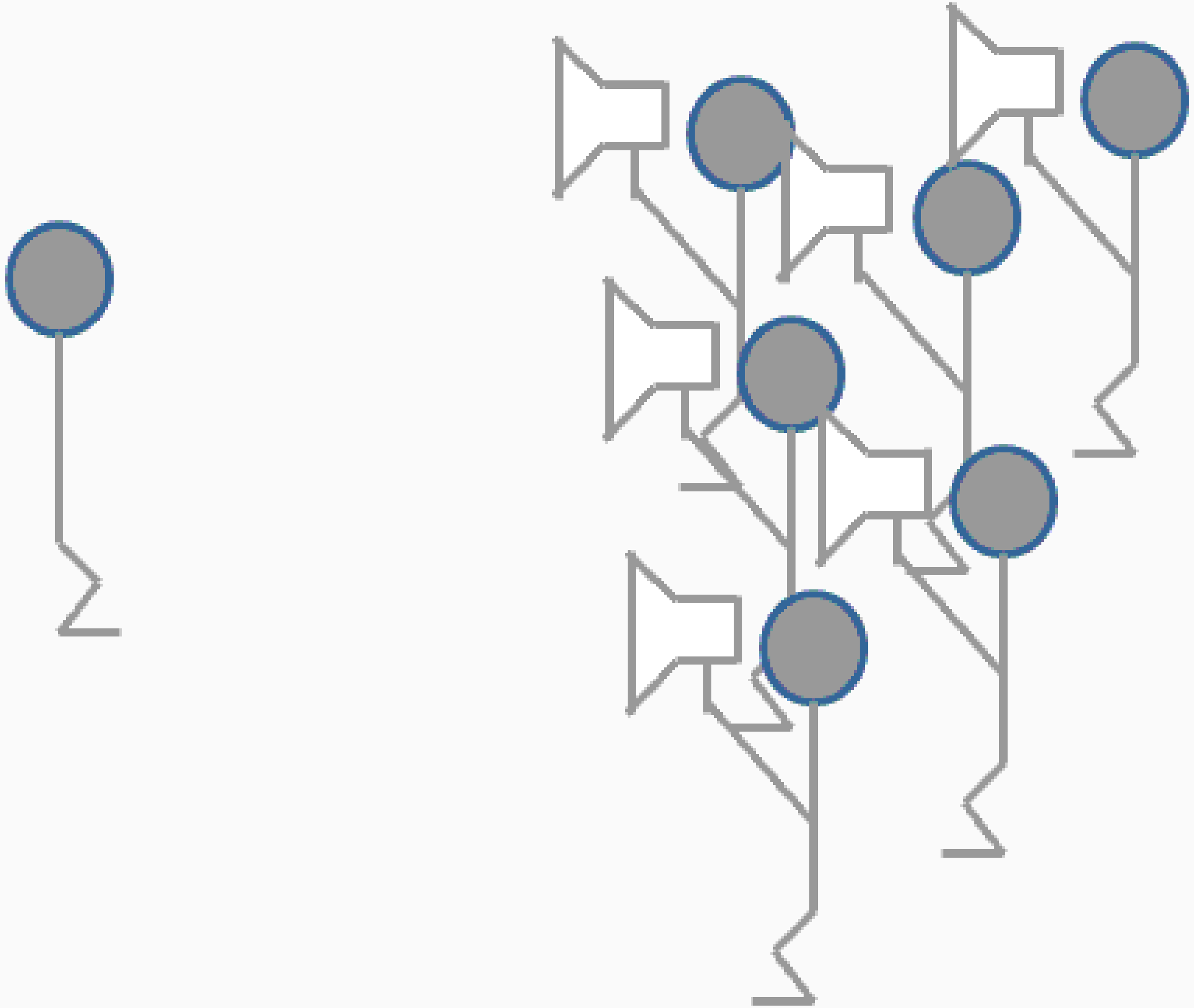
MPI Operations: Broadcast



MPI Operations: Broadcast

```r
library(pbdMPI)

if (comm.rank() == 0){
  important_value = 1+1
} else {
  important_value = NULL
}

ret = bcast(important_value)
comm.print(ret, all.rank=TRUE)

finalize()
```

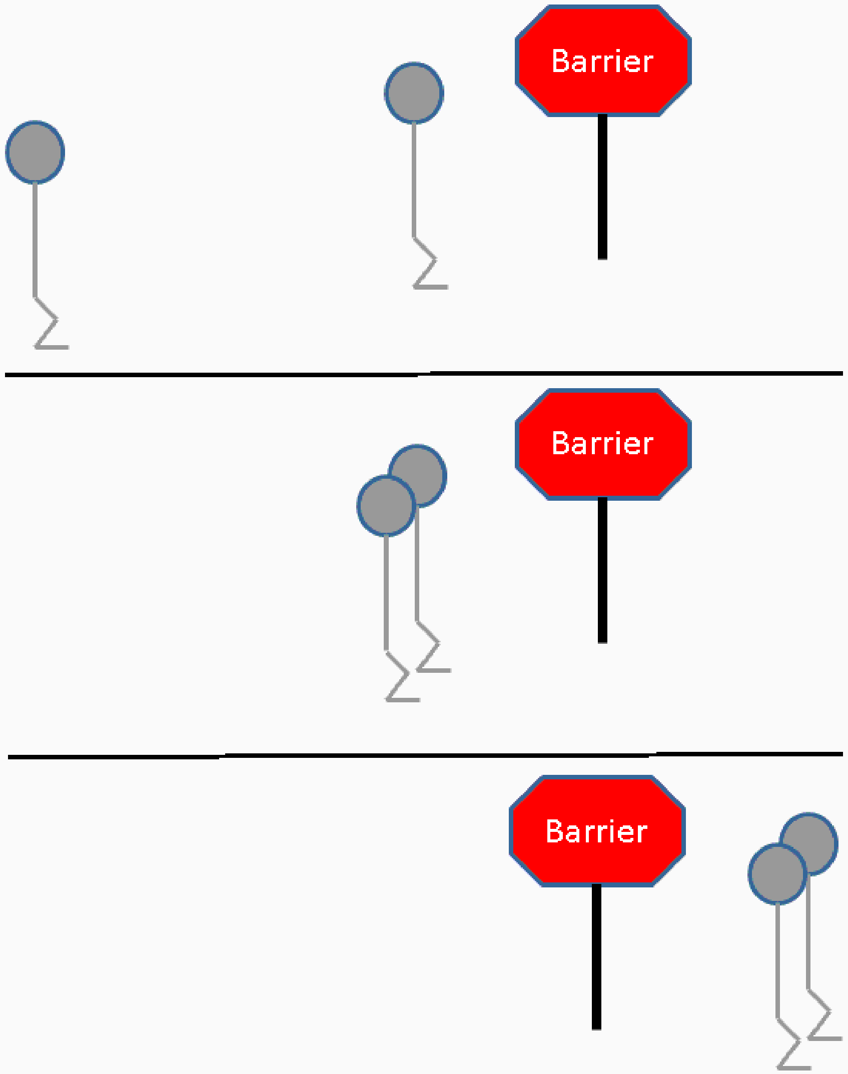
MPI Operations: Gather



MPI Operations: Gather

```
library(pbdMPI)

val.local = comm.rank()
vals = gather(val.local)
comm.print(vals)

finalize()
```

MPI Operations: Barrier



MPI Operations: Barrier

```r
library(pbdMPI)

comm.print("starting huge computation...")

if (comm.rank() == 0){
  Sys.sleep(5)
}

barrier()
comm.print("ok!")

finalize()
```

Task Parallelism

tasktools

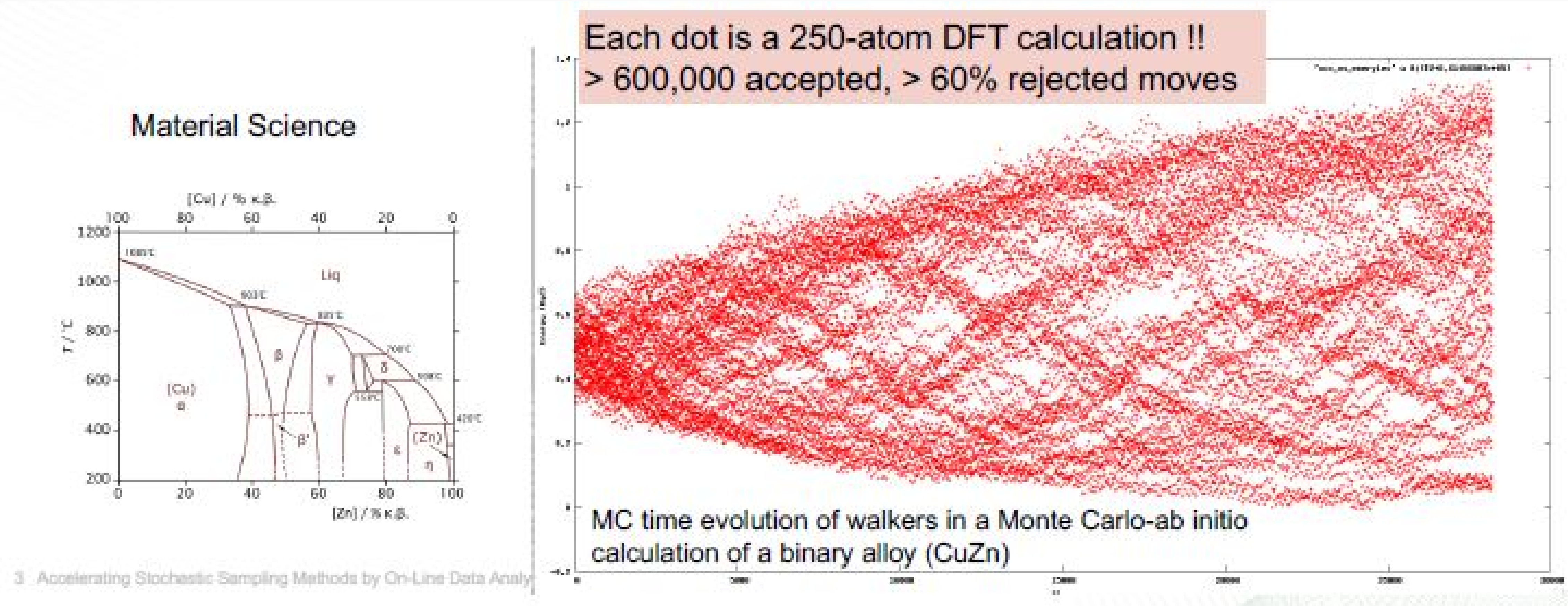
- Tools for task-based parallelism.
- Has an `lapply()`-like interface.
- Automatically handles input-checkpointing:
  - Have thousands of "jobs"
  - Run as many as you can in 2 hour run window
  - Keep running job until all tasks eventually complete.
- Can be used as a workflow tool for external programs.

tasktools

```r
costly = function(x, waittime)
{
  Sys.sleep(waittime)
  cat(paste("iter", i, "executed on rank", comm.

  sqrt(x)
}

ret = mpi_napply(10, costly, checkpoint_path="/t
comm.print(unlist(ret))
```

```
$ mpirun -np 3 r mpi_napply.r
iter 4 executed on rank 1
iter 7 executed on rank 2
iter 1 executed on rank 0
^Citer 2 executed on rank 0
iter 8 executed on rank 2
iter 5 executed on rank 1

$ mpirun -np 3 r mpi_napply.r
iter 9 executed on rank 2
iter 3 executed on rank 0
iter 6 executed on rank 1
iter 10 executed on rank 2

 [1] 1.000000 1.414214 1.732051 2.000000 2.23606
 [5] 2.449490 2.645751 2.828427 3.000000 3.16227
```

tasktools



Material Science

Each dot is a 250-atom DFT calculation !!

> 600,000 accepted, > 60% rejected moves

MC time evolution of walkers in a Monte Carlo-ab initio calculation of a binary alloy (CuZn)

3 Accelerating Stochastic Sampling Methods by On-Line Data Analy

Titan

- Cray XK7
- 18,688 nodes
- 299,008 cores
- 693.5 TiB of RAM

tasktools

Parameters

```r
eta_set = c(0.01, 0.05, 0.1, 0.5, 1)
gamma_set = 0:3
max_depth_set = c(6, 10, 15)
min_child_weight_set = c(1, 3, 5)

combos = expand.grid(eta=eta_set, gamma=gamma_se
NROW(combos)
```

```
## [1] 180
```

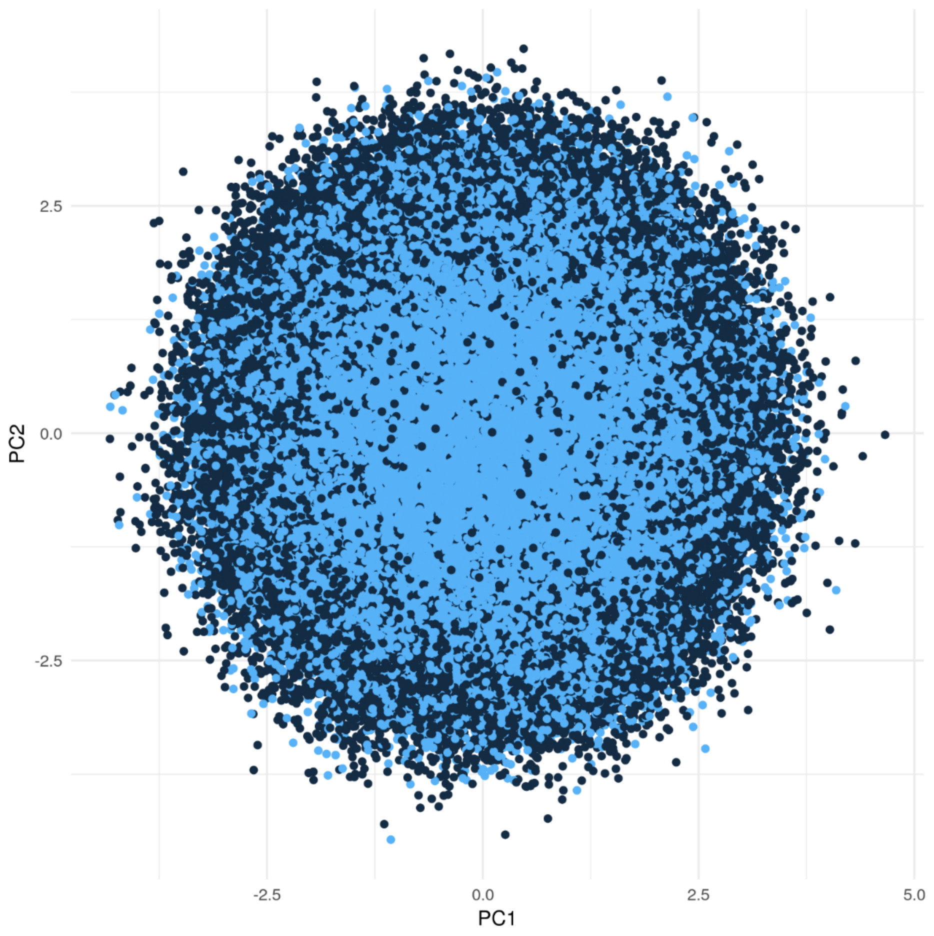
Launch

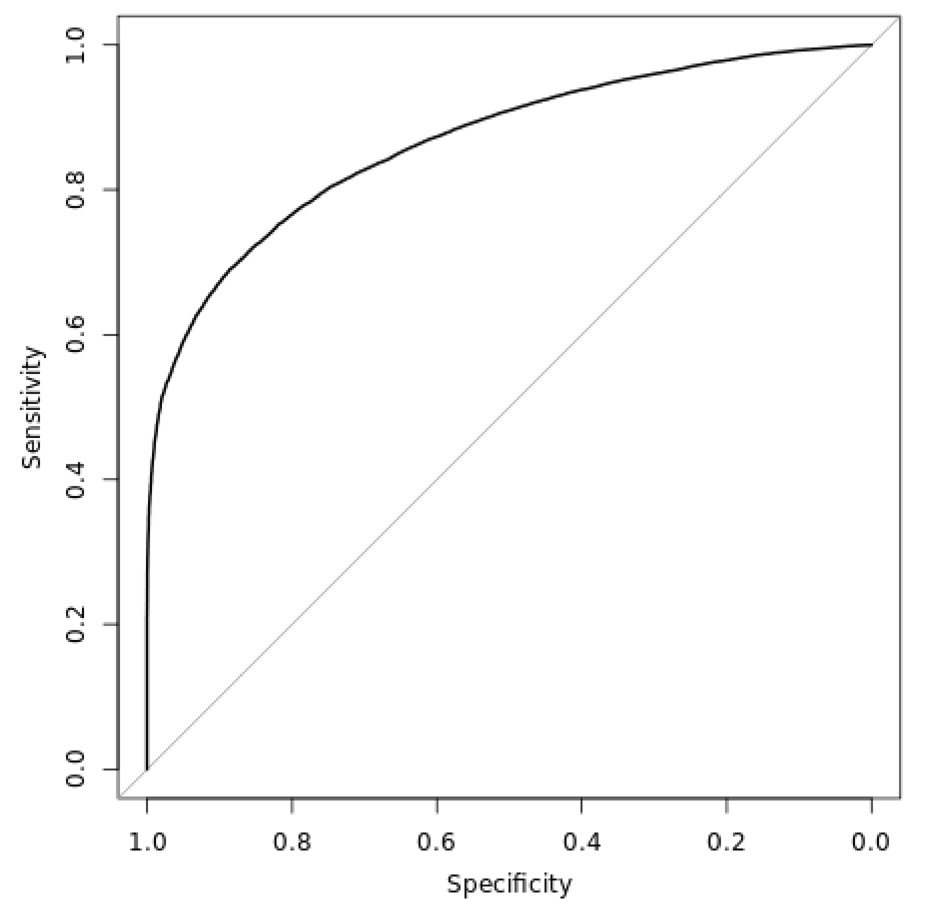
```
aprun -n 30 -d 16 xgb.r
```

Script

```r
run_one_cv = function(i)
{
  eta = combos[i, 1]
  gamma = combos[i, 2]
  max_depth = combos[i, 3]
  min_child_weight = combos[i, 4]

  params = list(...)
  cv = xgb.cv(params=params, ...)
  it = which.max(cv$evaluation_log$test_auc_mean)
  best = cv$evaluation_log[it]

  list(params=combos[i, , drop=FALSE], rating=bes
}

results = mpi_napply(n, run_one_cv, checkpoint_pa
```

tasktools


2.5
0.0
-2.5
PC2
-2.5
0.0
2.5
5.0
PC1


1.0
0.8
0.6
0.4
0.2
0.0
Sensitivity
1.0
0.8
0.6
0.4
0.2
0.0
Specificity

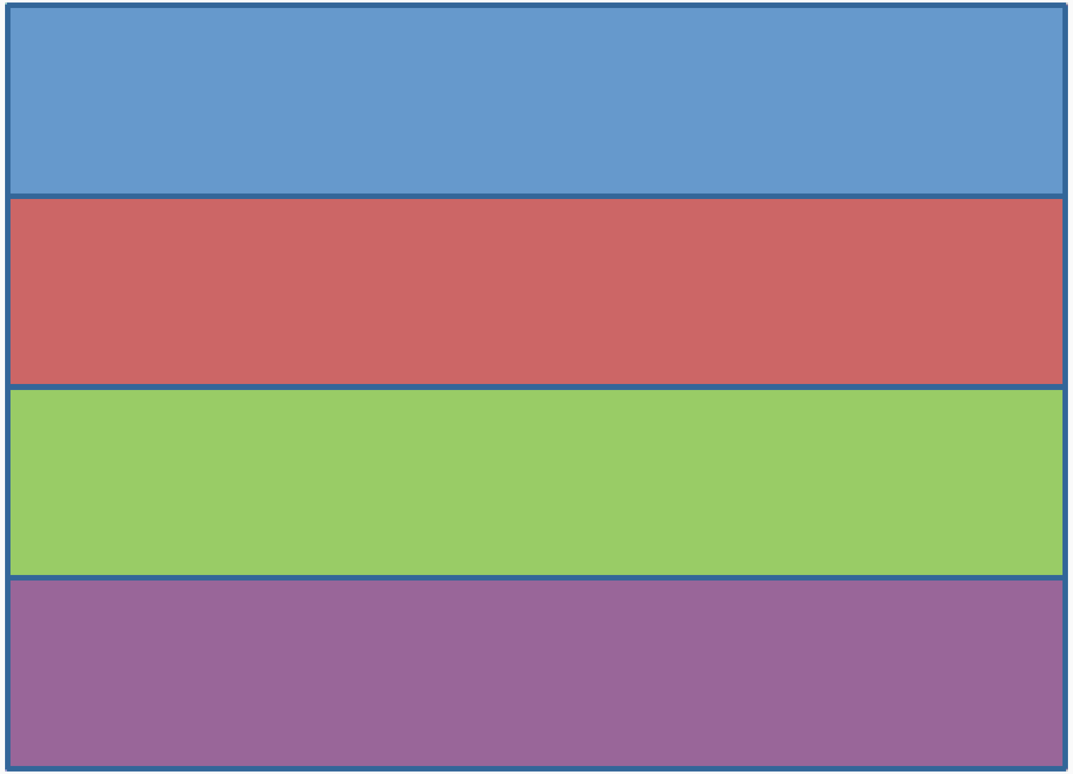
Distributed Matrices

pbdDMAT

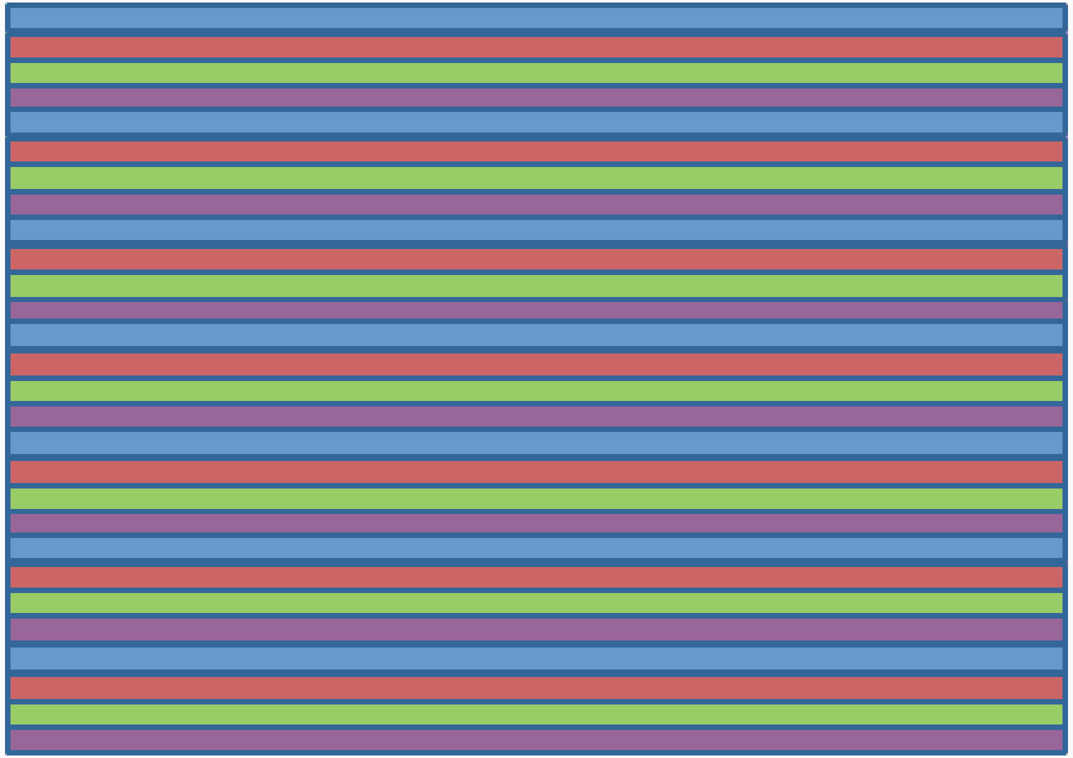
- High-level framework for distributed linear algebra and statistics
- Uses block-cyclic data decomposition (ScaLAPACK)
- Makes computing easy, but reading data still hard
- Syntax often identical to base R
    - Helpers: `[`, `rbind()`, `apply()`, …
    - Linear algebra: `%*%`, `svd()`, `qr()`, …
    - Basic statistics: `median()`, `mean()`, `rowSums()`, …
    - Multivariate statistics: `lm.fit()`, `prcomp()`, `cov()`, …

Block-cyclic

1-d Block



1-d cyclic

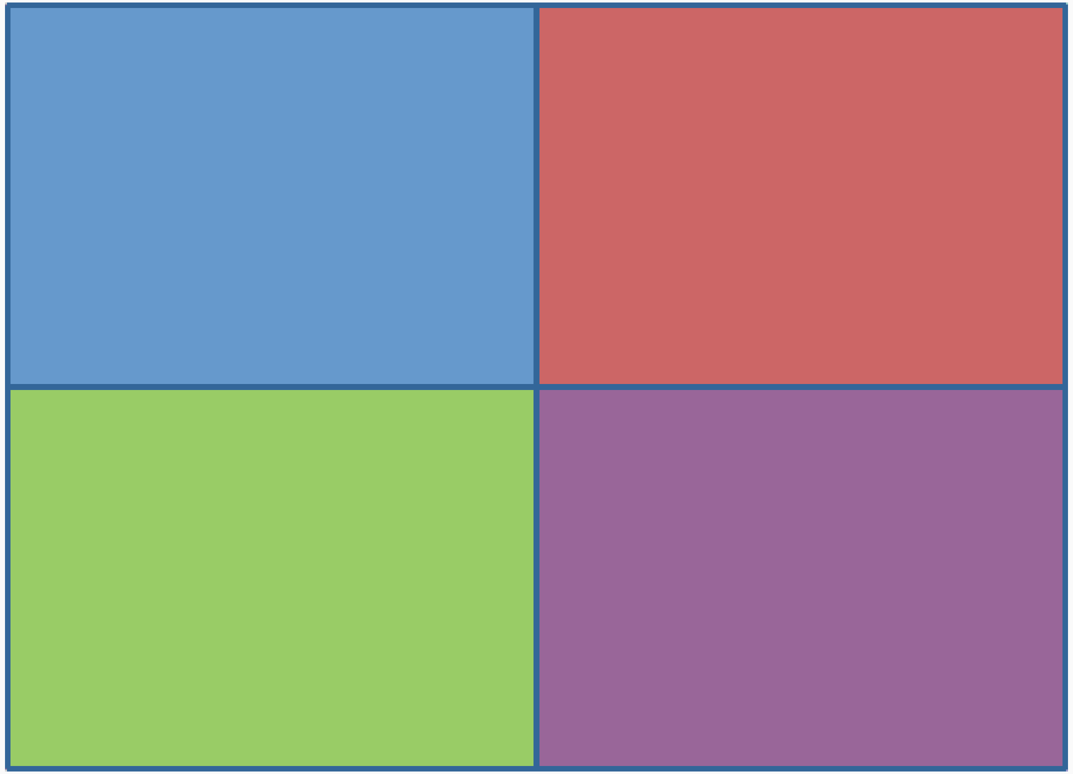


Block-cyclic

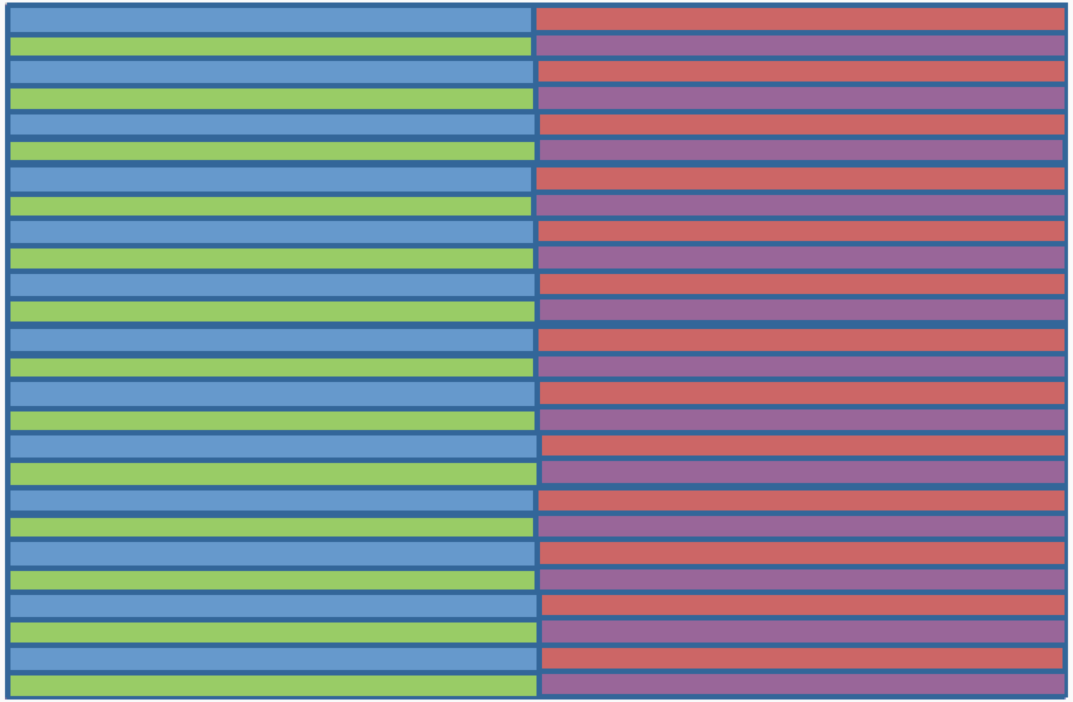
1-d block-cyclic

Block-cyclic

2-d Block

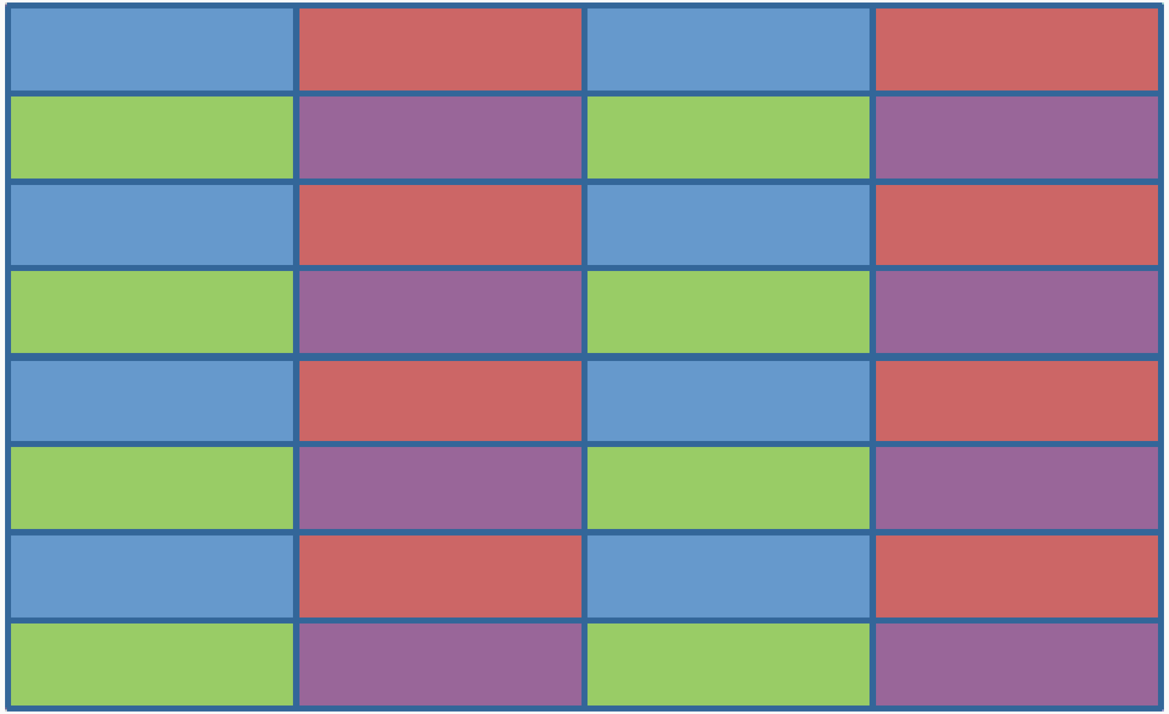


2-d cyclic



Block-cyclic

2-d block-cyclic



kazaam

- High-level framework for distributed linear algebra and statistics
- Optimized for very tall matrices with comparatively few columns ("shaq")
- Many linear algebra and machine learning methods
- Data distributed by row blocks (however you want)
- Some capability for very wide matrices distributed by column blocks ("tshaq")

pbdDMAT vs kazaam

- pbdDMAT much more thorough (sort of has to be…)
- Both have similar analytics capabilities (clustering, classifiers, dimension reduction, …)
- kazaam presently works better on GPU's
    - ECP slate may change this
    - both DIY right now
- Can redistribute from one layout to the other fairly easily

Benchmarks

Percival

- Cray XC40
- 168 nodes
- 10,752 cores
- 21 TiB of RAM

pbdDMAT

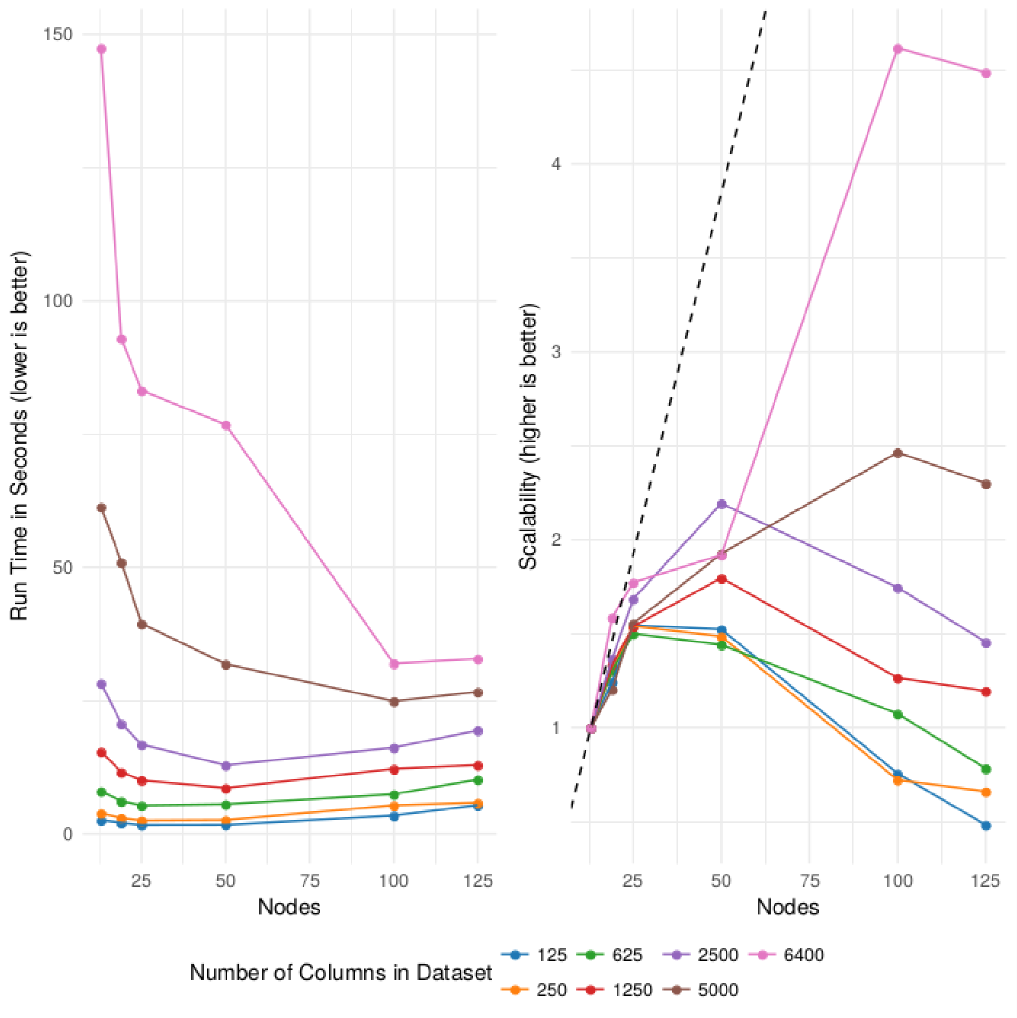
```r
x = ddmatrix("rnorm", m, n, ICTXT=2)

time = comm.timer({
  cp = crossprod(x)
  eigen(cp, symmetric=TRUE, only.values=TRUE)
})
```

kazaam

```r
x = ranshaq(rnorm, m.local, n, local=TRUE)

time = comm.timer(svd(x, nu=0, nv=0))
```

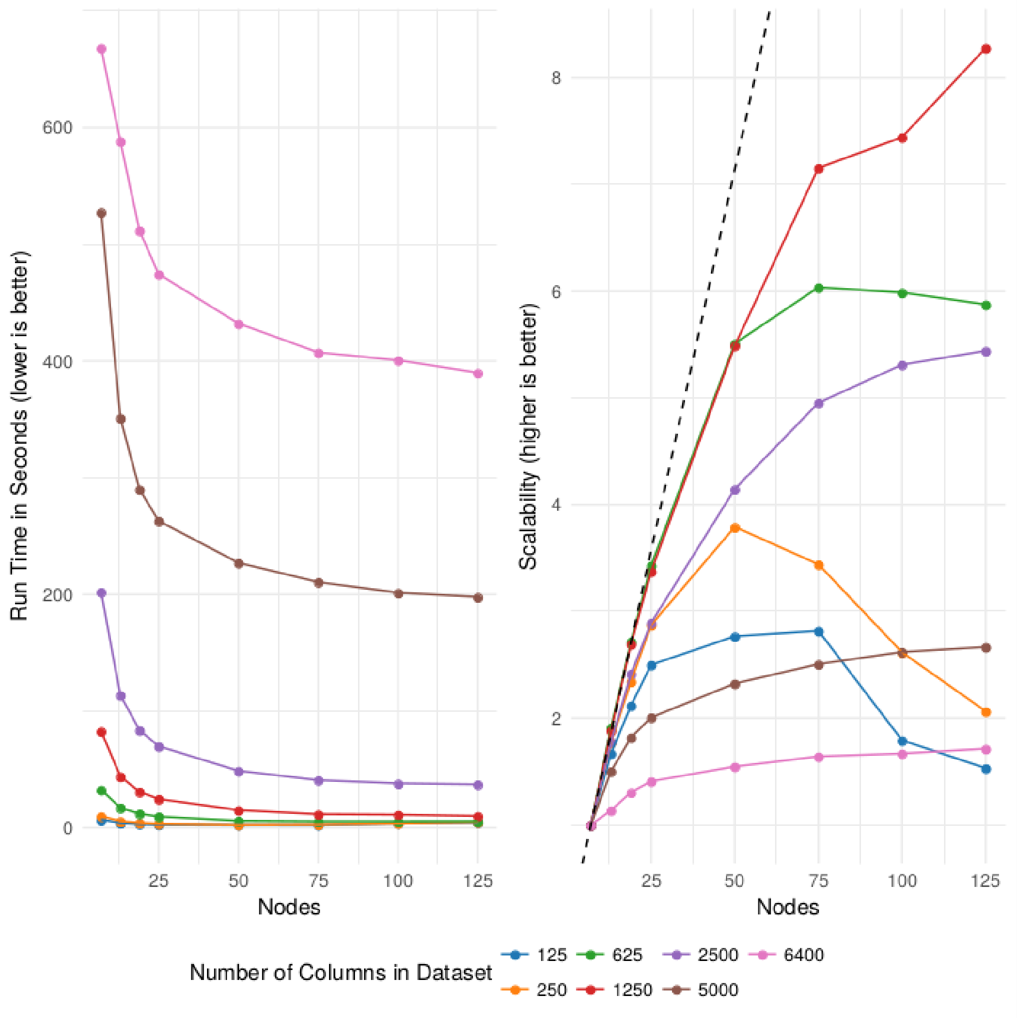
Benchmarks

pbdDMAT



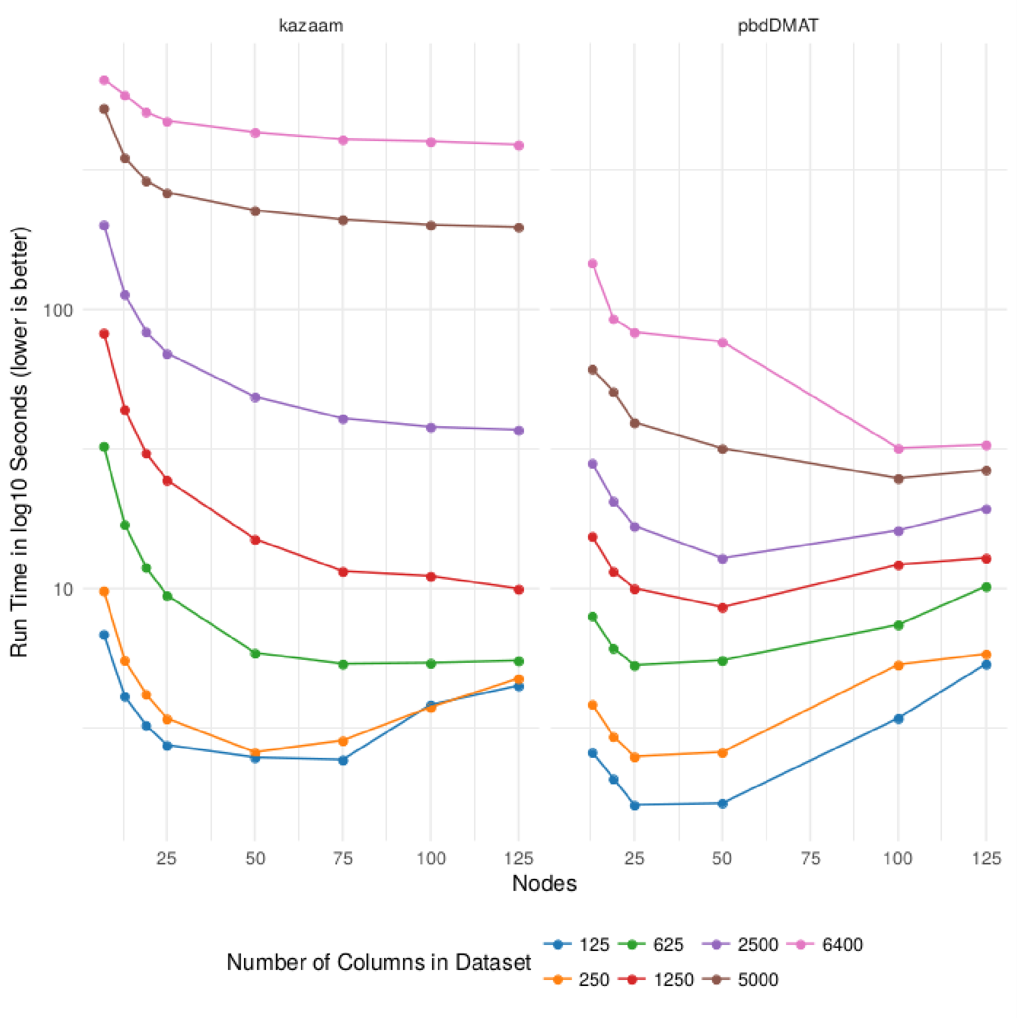
Benchmarks

kazaam



Benchmarks

pbdDMAT vs kazaam



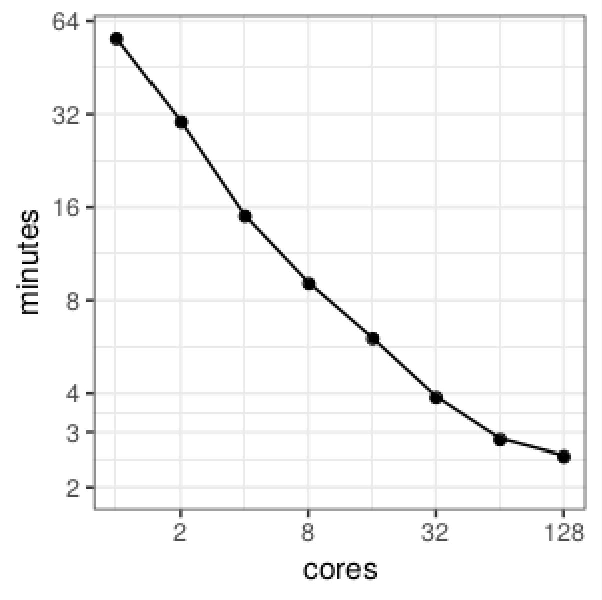
An Application

Outline

- Why you need performance for data analysis
- HOSVD workflow example
  - The math
  - The Algorithm
  - Removing covariance structure
- Workflow and scaling results
- Code notes

Why you need performance for data analysis

- Data analysis is a discovery process
- Iterate many times with different parameters, transformations, or models
- Context is lost if an iteration takes more than few minutes to compute
- Recovering context can take hours of researcher time



HOSVD: The Math

Primary source (including figures)

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl*, 21:1253-1278, 2000.

The SVD (2d tensor HOSVD)

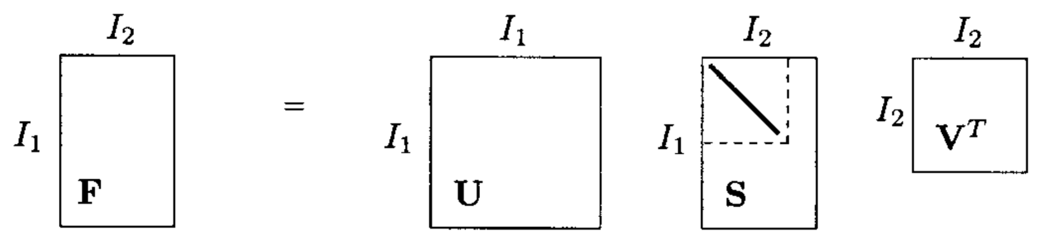


FIG. 3. *Visualization of the matrix SVD.*

- $\mathbf{F} = \mathbf{U}\mathbf{S}\mathbf{V}^T$
- $\mathbf{F} = \mathbf{S} \times_1 \mathbf{U} \times_2 \mathbf{V}$
- $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, that is $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$
- $\mathbf{S}$ positive, diagonal, ordered
- $\mathbf{U}$ and $\mathbf{V}$ are unique up to sign
- Consequently $\mathbf{F}^T\mathbf{F} = \mathbf{U}\mathbf{S}^2\mathbf{U}^T$

HOSVD: The Math

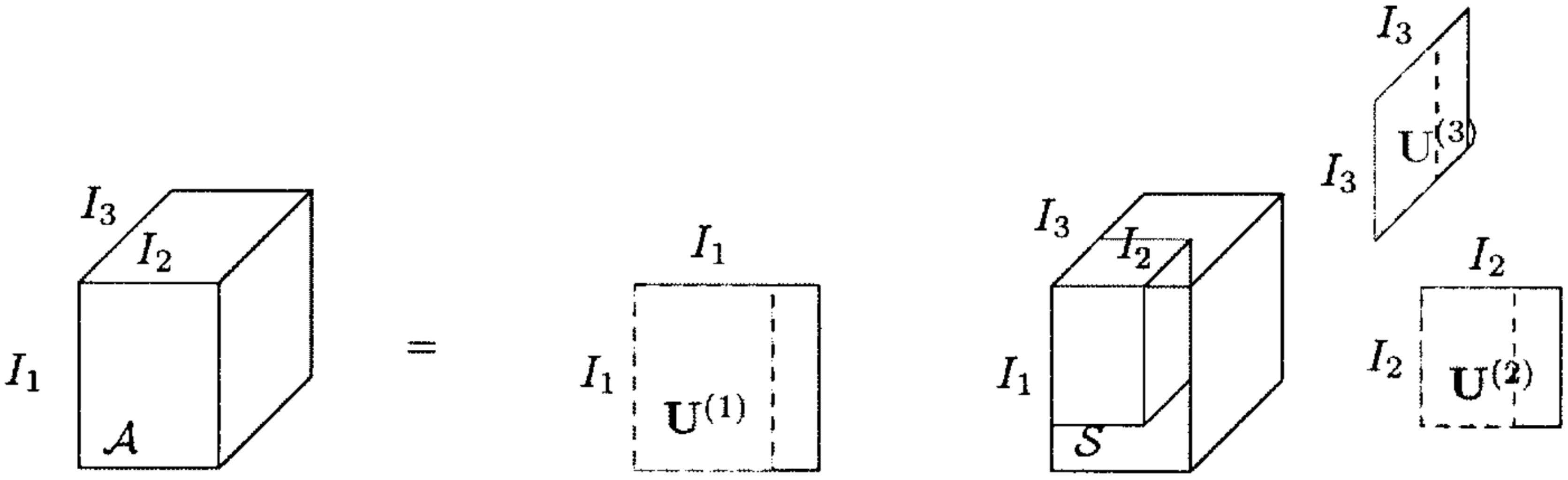


FIG. 4. *Visualization of the HOSVD for a third-order tensor.*

- $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{S}} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}$
- $\mathbf{U}^{(i)}$ are orthogonal $\mathbf{U}^{(i)^T}\mathbf{U}^{(i)} = I$, for $i = 1, 2, 3$
- The $\boldsymbol{\mathcal{S}}$ tensor is all-orthogonal (all its slice matrices are orthogonal)

HOSVD: The Algorithm

Unfolding a 3d tensor: Data wrangling!! Skinny matrices for kazaam package!

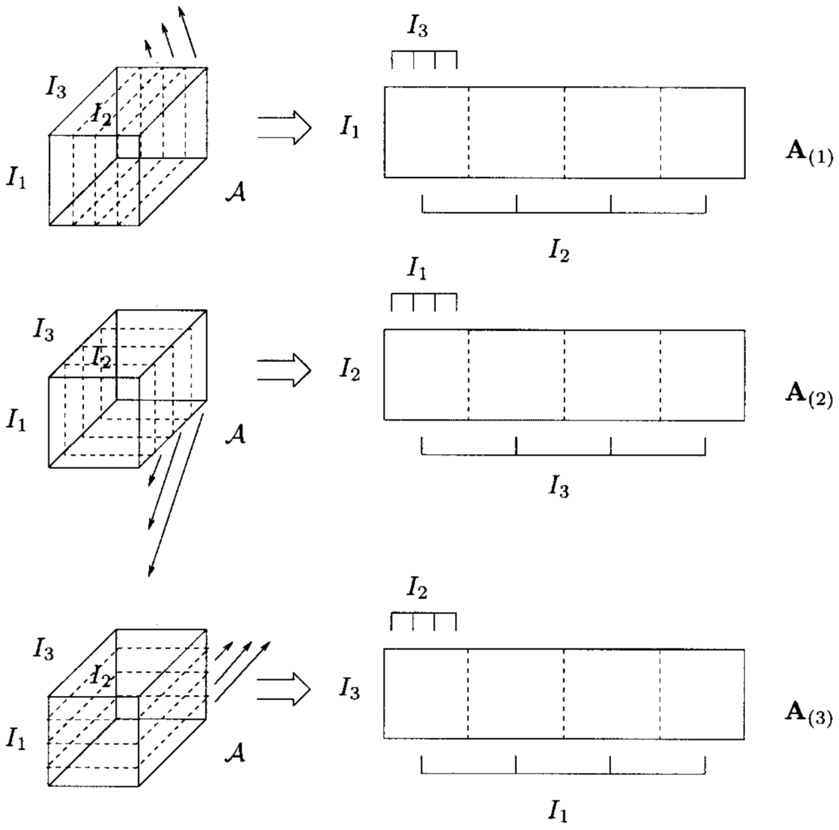


FIG. 1. *Unfolding of the* $(I_1 \times I_2 \times I_3)$*-tensor* $\mathcal{A}$ *to the* $(I_1 \times I_2 I_3)$*-matrix* $\mathbf{A}_{(1)}$*, the* $(I_2 \times I_3 I_1)$*-matrix* $\mathbf{A}_{(2)}$ *and the* $(I_3 \times I_1 I_2)$*-matrix* $\mathbf{A}_{(3)}$ $(I_1 = I_2 = I_3 = 4)$.

HOSVD: The Algorithm

3d HOSVD computes 3 SVDs, one for each unfolding

- $\mathbf{A}_{(1)} = \mathbf{U}^{(1)}\mathbf{\Sigma}^{(1)}{\mathbf{V}^{(1)}}^T$
- $\mathbf{A}_{(2)} = \mathbf{U}^{(2)}\mathbf{\Sigma}^{(2)}{\mathbf{V}^{(2)}}^T$
- $\mathbf{A}_{(3)} = \mathbf{U}^{(3)}\mathbf{\Sigma}^{(3)}{\mathbf{V}^{(3)}}^T$

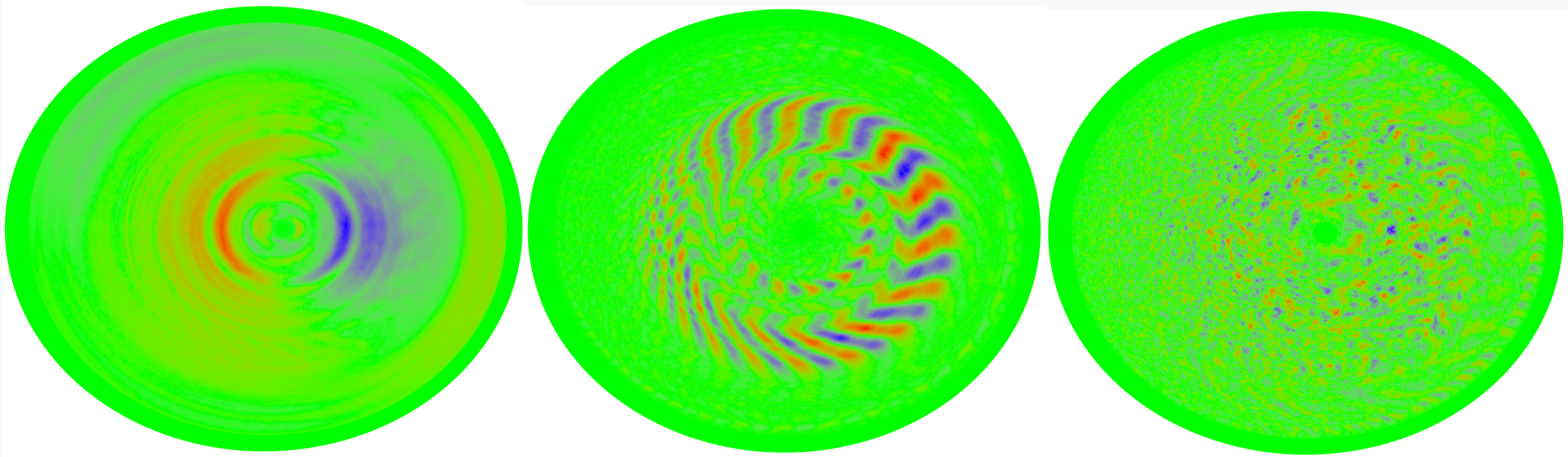
Keep $\mathbf{U}^{(1)}$, $\mathbf{U}^{(2)}$, $\mathbf{U}^{(3)}$ and compute the $\boldsymbol{\mathcal{S}}$

- $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{A}} \times_1 {\mathbf{U}^{(1)}}^T \times_2 {\mathbf{U}^{(2)}}^T \times_3 {\mathbf{U}^{(3)}}^T$
- Note that $(\boldsymbol{\mathcal{A}} \times_i \mathbf{U}^{(i)})_{(i)} = {\mathbf{U}^{(i)}}^T \mathbf{A}_{(i)}$
- So $\mathbf{S}_{(3)} = {\mathbf{U}^{(3)}}^T({\mathbf{U}^{(2)}}^T({\mathbf{U}^{(1)}}^T\mathbf{A}_{(1)})_{(2)})_{(3)}$

Removing Covariance Structure: Whitening

Displaying an unstructured mesh dimension adjusted for the other two dimensions

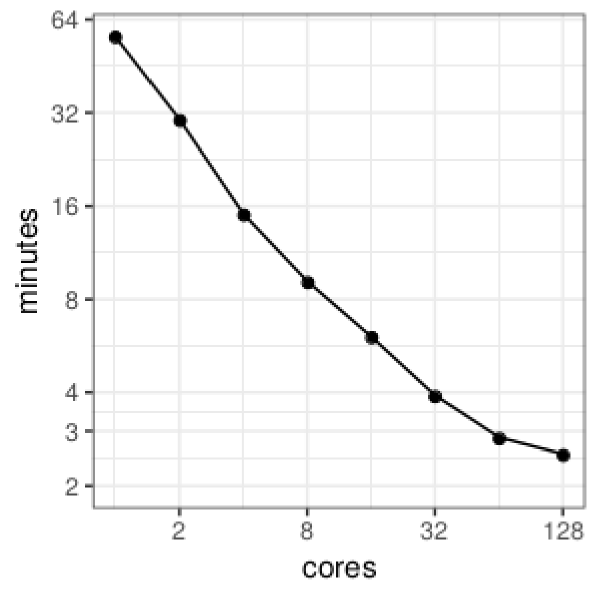
- Suppose $\boldsymbol{\mathcal{A}}$ is dim1 * dim2 * mesh
- $\mathbf{A}_{(3)} = \mathbf{U}^{(3)}\mathbf{\Sigma}^{(3)}\mathbf{V}^{(3)^T}$
    - Each row of $\mathbf{V}^{(3)}$ maps to mesh
    - Want to visualize this adjusted for dim1 and dim2 covariance structure
- $(\mathbf{U}^{(2)^T}(\mathbf{U}^{(1)^T}\mathbf{A}_{(1)})_{(2)})_{(3)} = \tilde{\mathbf{U}}^{(3)}\tilde{\mathbf{\Sigma}}^{(3)}\tilde{\mathbf{V}}^{(3)T}$



HOSVD Workflow Code for a Fusion XGC1 Simulation Output

- Tensor dimensions:
  - 41 (time)
  - 32 (toroidal angle)
  - 232,011 (unstructured poloidal mesh)
  - Total size ~ 2.3 GB
- Workflow computations:
  - Read data in 41 HDF5 files
  - 5 SVD computations in series
  - 6 unfoldings
  - 80 pdf plots (~3 MB each)
- Full workflow code in *scripts* directory
  - *tensor.r* the R code
  - *function_def.r* additional functions used
  - *tensor.pbs* the PBS submission shell script

- Strong scaling for complete workflow



  - Not bad, considering no optimization was done
  - The goal is to reach interactive speed

Code Notes: Reading HDF5 in Parallel

```r
library(rhdf5)
library(pbdIO)

# . . .

iopair_n = comm.chunk(mesh$n_n, form = "iopair")

## rhdf5 needs to add 1 to C/Python written data!
buffer = h5read(file, "coordinates/values",
                start=c(0 + 1, iopair_n[1] + 1),
                count=c(2, iopair_n[2]))

rz = do.call(c, allgather(unlist(buffer)))
```

Code Notes: Multiple Plots in Parallel

```
library(pbdIO)

#. . .

nplots = min(maxplots, length(d))
myPCs = comm.chunk(nplots, form="vector", type="balance")
my.d = d[myPCs]

#. . .

for (i in seq_along(myPCs)){
    spacePlot(mesh, Vc[, i]*my.d[i], sprintf(ftag, myPCs[i]))
}
```

Code Notes: Creating Directories

```
## Crate output directories
screedir = paste0(ref_dir, "scree/")
ref = sprintf("%0.5d", w_center)
plotdir = paste0(ref_dir, "plots", ref, "/")
if(myrank == 0) { # only one rank should be creating a directory
    dir.create(ref_dir, showWarnings=FALSE)
    dir.create(screedir, showWarnings=FALSE)
    dir.create(plotdir, showWarnings=FALSE)
}

barrier() # must be reachable by all ranks
```

Code Notes: Unfolding a 3d tensor and shaq/tshaq SVD

```
tens = read_xgc_window(file_var, var, w_center, window)$Data
tdim = dim(tens) # tensor dimensions (1, 2, 3d) = (toro, time, mesh)

## u1 toro: want dimensions (1, 3d*2) - need tshaq
u1tens = as.vector(tens) # (1, 2, 3d)
dim(u1tens) = c(tdim[1], tdim[2]*tdim[3]) # (1, 2*3d)
u1tens.s = tshaq(u1tens) # (1, 2*3d) tshaq
u1svd = svd(u1tens.s)

## u3 mesh: want dimensions (3d, 1*2) - need shaq of transpose
u3tens = as.vector(tens) # (1, 2, 3d)
dim(u3tens) = c(tdim[1]*tdim[2], tdim[3]) # dim (1*2, 3d)
u3tens.s = shaq(t(u3tens)) # transposed so dim (3d, 1*2) shaq
u3svd = svd(u3tens.s)
```

Code Notes: Core tensor data wrangling

```
## Core tensor computation
u1core1 = crossprod(u1svd$u, u1tens) # dim (1, 2*3d), all local op

u3core1 = as.vector(u1core1) # (1, 2, 3d)
dim(u3core1) = c(tdim[1]*tdim[2], tdim[3]) # dim (1*2, 3d)
u2core1 = u3core1[rindex, ] # reordered to (2, 1, 3d) dim (2*1, 3d)
dim(u2core1) = c(tdim[2], tdim[1]*tdim[3]) # dim (2, 1*3d)
u2core21 = crossprod(u2svd$u, u2core1) # dim(2, 1*3d), all local

u3core21 = as.vector(u2core21) # (2, 1, 3d)
dim(u3core21) = c(tdim[2]*tdim[1], tdim[3]) # dim (2*1, 3d)
u3core321 = crossprod(u3svd$u, shaq(t(u3core21))) # dim (3td, 2*1)
## Note that 3td = 2*1 as 3 > 2*1 leads to 3 - 2*1 zero eigenvalues
##      so that leading dimension of u3core321 is 2*1 instead of 3.
##      u3core ends up a local matrix that is replicated. The shaq
##      crossprod collapses the long dimension.
```

Thanks!