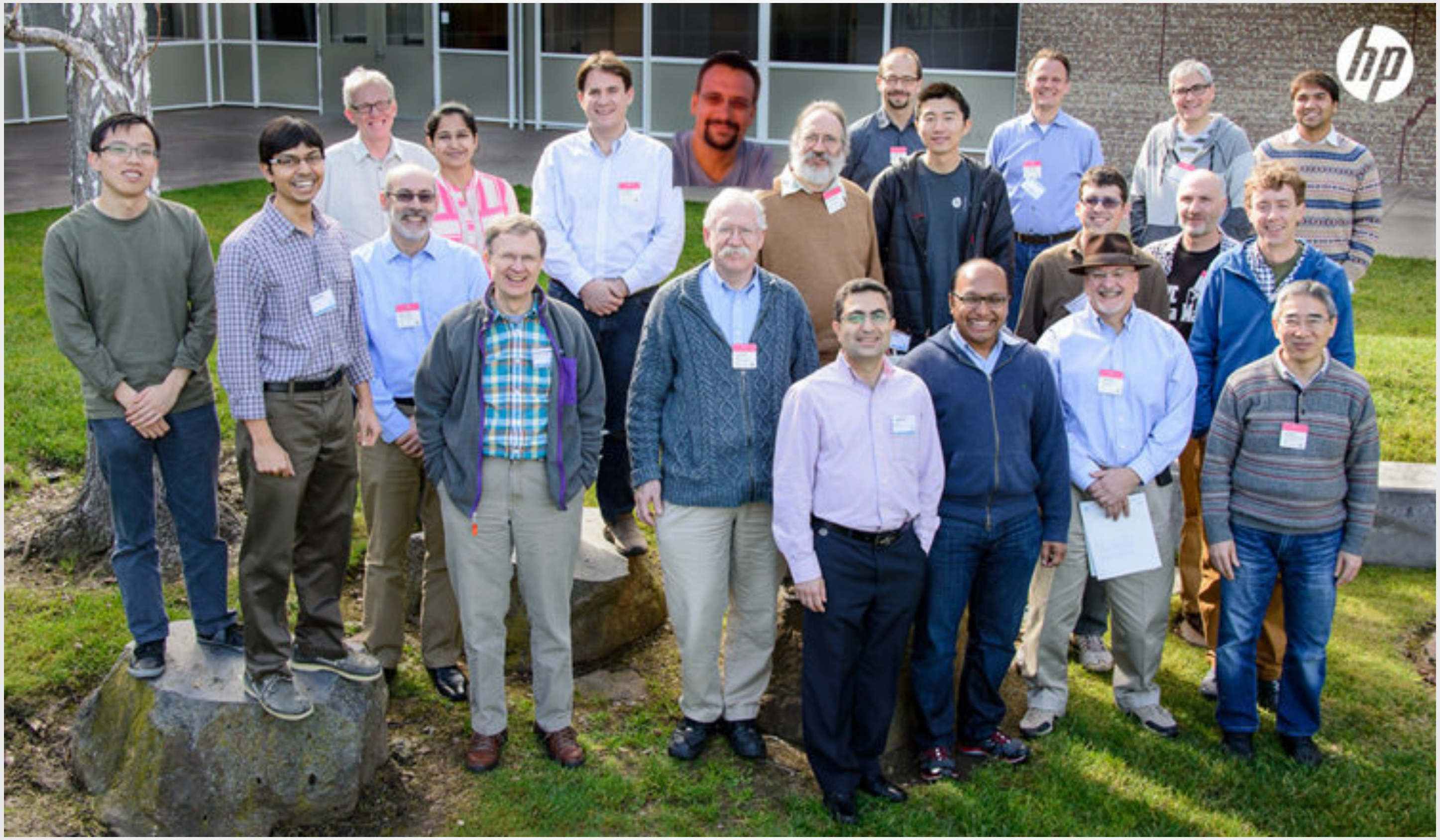
Moving Toward a Concurrent Computing Grammar

Michael Kane and Bryan Lewis

"Make the easy things easy and the hard things possible"

- Larry Wall

Workshop on Distributed Computing in R



Indrijit and Michael Identified a Few Themes

- For high-performance computing writing MPI at a low-level is a good option.
- For in-memory processing, adding some form of distributed objects in R can potentially improve performance.
- Using simple parallelism constructs, such as *lapply*, that operate on distributed data structures may make it easier to program in R.
- Any high level API should support multiple backends, each of which can be optimized for a specific platform, much like R's snow and foreach package run on any available backend.

Evaluating a Distributed Computing Grammar

A grammar provides a sufficient set of composable constructs for distributed computing tasks.

R has great grammars for

- for connecting to storage
- manipulating data
- data visualization
- models

It should be agnostic to the underlying technology but it needs to be able to use it in a way that's consistent.

	Map/Reduce Comm. Only	Integrated Data Storage	Fault Tolerance
hmr	x	x	x
RHIPE	x	x	x
rmr	x	x	x
sparkr	x*	x	x
parallel	x		
Rdsm		x	
Rmpi			
pdbMPI			
scidb		x	x
distributedR		x	x

High-Level Distributed Computing Packages

snow: Tierney, Rossini, Li, and Sevcikova (2003)

foreach: Revo and Steve Weston (2009)

ddR: Ma, Roy, and Lawrence (2015)

datadr: Hafen and Sego (2016)

Is "good enough" good enough?

We built pirls and irlba on top of ddR to answer:

1. Can we do it?
2. Does ddR provide the right constructs to do this in a natural way?

"Cheap" irls implementation

```
irls =
function(x, y, family=binomial, maxit=25, tol=1e-08)
{
  b = rep(0, ncol(x))
  for(j in 1:maxit)
  {
    eta    = drop(x %*% b)
    g      = family()$linkinv(eta)
    gprime = family()$mu.eta(eta)
    z      = eta + (y - g) / gprime
    W      = drop(gprime^2 / family()$variance(g))
    bold   = b
    b      = solve(crossprod(x, W), crossprod(x, W * z), tol)
    if(sqrt(drop(crossprod(b - bold))) < tol) break
  }
  list(coefficients=b, iterations=j)
}
```

Notation

model matrix: $X \in \mathcal{R}^{n \times p}$

dependent data: $y \in \mathcal{R}^n$

slope coefficient estimates: $\hat{\beta} \in \mathcal{R}^p$

penalty parameter: $\lambda \in [0, \infty)$

elastic net parameter: $\alpha \in [0, 1]$

a weight matrix: $W \in R^{n \times n}$

a link function: g

pirls

$$\min_{\hat{\beta}} ||W^{1/2}(y - g(X\hat{\beta}))||^2 + (1-\alpha)\frac{\lambda}{2}||\hat{\beta}||^2 + \alpha\lambda||\hat{\beta}||_1$$

The Ridge Part

The Least Absolute Shrinkage and Selection Operator (LASSO) Part

Data are Partitioned by Row

If you have too many columns there's:

SAFE - discard the jth variable if

$$|x_j^T y|/n > \lambda - \frac{||x_2||||y_2||}{n} \frac{\lambda_{\max} - \lambda}{\lambda_{\max}}$$

STRONG - discard if KKT conditions are met and

$$|x_j^T y|/n > 2\lambda - \lambda_{\max}$$

Simple ddR implementation almost the same

```
dirls = function(x, y, family=binomial, maxit=25, tol=1e-08)
{
  b = rep(0, ncol(x))
  for(j in 1:maxit)
  {
    eta    = drop(x %*% b)
    g      = family()$linkinv(eta)
    gprime = family()$mu.eta(eta)
    z      = eta + (y - g) / gprime
    W      = drop(gprime^2 / family()$variance(g))
    bold   = b
    b      = solve(wcross(x, W), cross(x, W * z), tol)
    if(sqrt(crossprod(b - bold)) < tol) break
  }
  list(coefficients=b, iterations=j)
}
```

```
cross = function(a, b) {
  Reduce(`+`,
         Map(function(j) {
           collect(dmapply(function(x, y) crossprod(x, y),
                   parts(a),
                   split(b, rep(1:nparts(a)[1], psize(a)[,1])),
                   output.type="darray",
                   combine="rbind", nparts=nparts(a)), j)
         },
         seq(1,totalParts(a))))
}

wcross = function (a, w) {
  Reduce(`+`,
         Map(function(j) {
           collect(dmapply(function(x, y) crossprod(x, y*x),
                   parts(a),
                   split(w, rep(1:nparts(a)[1], psize(a)[,1])),
                   output.type="darray",
                   combine="rbind",
                   nparts=nparts(a)), j)
         },
         seq(1,totalParts(a))))
}
```

A Toy Example

```
> x = dmapply(function(x) matrix(runif(4), 2, 2),
+             1:4,
+             output.type="darray",
+             combine="rbind",
+             nparts=c(4, 1))
> y = 1:8
>
> print(coef(dirls(x, y, gaussian)))
          [,1]
[1,] 6.2148108
[2,] 0.4186009
> print(coef(lm.fit(collect(x), y)))
       x1        x2
6.2148108 0.4186009
```

Another Algorithm: IRLBA

```
setMethod("%*%", signature(x="ParallelObj", y="numeric"), function(x ,y)
{
  stopifnot(ncol(x) == length(y))
  collect(
    dmapply(function(a, b) a %*% b,
            parts(x),
            replicate(totalParts(x), y, FALSE),
            output.type="darray", combine="rbind", nparts=nparts(x)))
})

setMethod("%*%", signature(x="numeric", y="ParallelObj"), function(x ,y)
{
  stopifnot(length(x) == nrow(y))
  colSums(
    dmapply(function(x, y) x %*% y,
            split(x, rep(1:nparts(y)[1], psize(y)[, 1])),
            parts(y),
            output.type="darray", combine="rbind", nparts=nparts(y)))
})
```

```
> x = dmapply(function(x) matrix(runif(4), 25, 25), 1:4,
+              output.type="darray", combine="rbind", nparts=c(4, 1))

> print(irlba(x, nv=3)$d)
[1] 32.745771  6.606732  6.506343

> print(svd(collect(x))$d[1:3])
[1] 32.745771  6.606732  6.506343
```

Applied this approach to compute 1st three principal components of the 1000 Genomes variant data:

2,504 x 81,271,844 (about 10^9 nonzero elements)

< 10 minutes across 16 R processes (on EC2)

What we like about ddR

The idea of distributed list, matrix, data.frame containers feels right.

Pretty easy to use productively.

Ideas we're thinking about

Separation of data and container API from execution

Simpler and more general data and container API

- data provider "backends" belong to chunks
- freedom to work chunk by chunk or more generally
- containers that infer chunk grid from chunk metadata (allowing non-uniform grids)

Modified chunk data API idea

Generalized/simplified ddR high-level containers

```
get_values(chunk, indices, ...)
get_attributes(chunk)
get_attr(chunk, x)
get_length(chunk)
get_object.size(chunk)
get_typeof(chunk)
new_chunk(backend, ...)
as.chunk(backend, value)
```

Simplified ddR container object ideas

```
chunk1 = as.chunk(backend, matrix(1, 10, 10))
chunk2 = as.chunk(backend, matrix(2, 10, 2))
chunk3 = as.chunk(backend, 1:12)

# A 20 x 12 darray
x = darray(list(chunk1, chunk1, chunk2, chunk2),
       nrow=2, ncol=2)

# A 12 x 1 darray
y = darray(list(chunk3), ncol=1)  # 12 x 1 array
```

Examples

Sequential (data API only)

```
z = darray(list(as.chunk(backend, x[1:10,] %*% y[]),
                as.chunk(backend, x[11:20,] %*% y[])),
           ncol=1)
```

With a parallel execution framework

```
z = darray(mclapply(list(1:10, 11:20),
           function(i) {
             as.chunk(backend, x[i, ] %*% y[])
           }), ncol=1)
```

If you want to know more...

cnidaria: A Generative Communication Approach to Scalable, Distributed Learning (2013)

Generative communication in Linda, Gelernter (1985)