REVOLUTION ANALYTICS

Overview of Parallel Computing

Revolution Analytics

Outline

Introduction to Parallel Computing in R

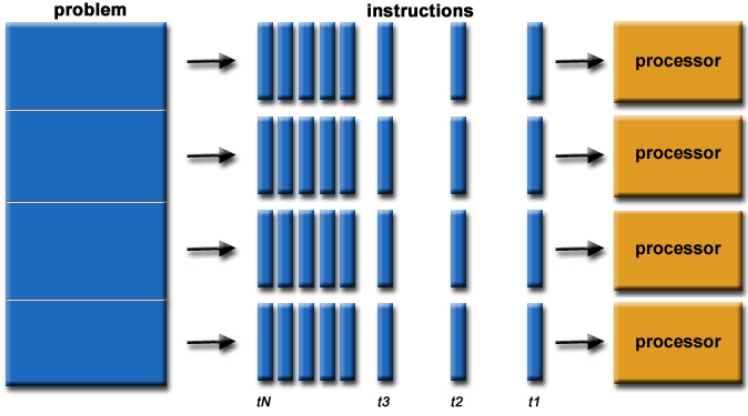
- We will cover parallel fundamentals (language independent)
  - What is parallelization?
  - Why parallelize?
  - What are some limitations?
- Identify different types of parallelism
- Focus on parallelism within R
- Discuss parallel support within Revolution R Enterprise

What is Parallel Computing?

Parallel computing is the simultaneous use of multiple compuptational resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- Results from different processors are then integrated in some fashion

Illustration



Why parallelize?

Why woud you want to do this?

Speed gains

- Reduce the time of a lenghty process (single job can take less time)
- Improve throughput (many jobs can be executed at once)
- Improve responsivity

Examples

Brain Multiple senses are processed simultaneously

Web search Many problems are too large to be solved with limited resources, in particular memory

Optimization / simulation Require many attempts – perform multiple attempts at a single point in time.

Commercial Applications of Parallelism

- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Advanced graphics and virtual reality
- Networked video and multi-media technologies
- Climate Models

Questions?

Outline

What does it cost?

We've told you why you would want to do it, why wouldn't you?

A number of constraints and limitations…

- Problem Constraints (Important)
- Hardware Constraints (Less important these days)
- Overhead / Algorithmic Constraints (Important)

Problem Constraints

The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously
- Be solved in less time with multiple compute resources than with a single compute resource.

Embarrassingly Parallel Computations

- No communication or very little communication between processes
- Each process can do its tasks without any interaction with other processes
- Examples
  - Monte-Carlo Calculations
  - Mandelbrot Set
  - Low level Image Processing
  - Bootstrapping

Computational Resources

The computational resources must be able to execute multiple program instructions at a given point in time.

Examples:

- A single computer with multiple processors/cores
- An arbitrary number of such computers connected by a network

There has been a huge shift in processor design in the past decade…

Computational Power (Historically)

Early approaches to increasing computational power. Increase the

- Clock speed
- Instruction throughput
- Cache

Computational Power (Current)

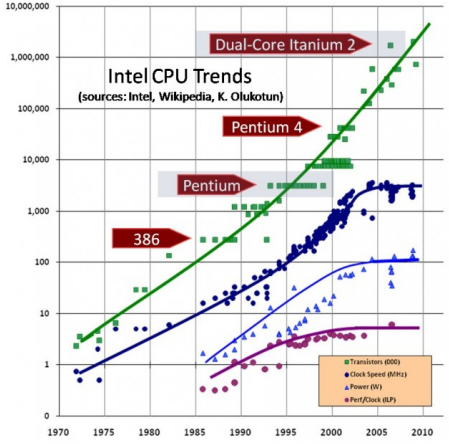
Currently, a different trend and set of optimization constraints:

- Multicore
- Hyperthreading
- Cache

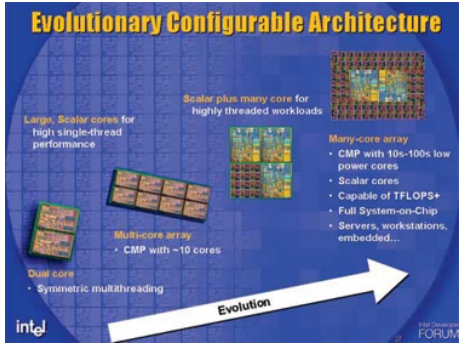
Moore's law

- Past: Processor speed doubles every 18 months
- Future: Number of cores doubles every 18 months(?)

Moore's law


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010

Intel Roadmap



(Image (c) Intel)

Ubiquitous Parallel Systems!

- Multicore systems offer 2-16 cores
- Multiprocessor workstations are affordable (<$500 for dual processor Dell workstation)
- Clusters are now common in labs, business
- Cloud-computing - you can rent a multi-core/cluster for a short period of use
- Inexpensive with cheap PCs (Beowulf systems)
  - Easy to create with modern networking HW, SW
  - Grids – collections of other systems
- Linux and Window have lots of parallel support

Types of Parallelism

There are many different kinds of parallelism!

Common Kinds of Parallelism

Bit level	(add, multiply, memory reference) – hardware, compiler dependent (nothing to do in R)
Instruction level	Example from C: a = b + c; d = e + f; g = h + i; (nothing to do in R)
Data level	Same operations over different data (ℝ's method)
Task level	Different tasks (I/O, graphics, etc)

Parallelism Doesn't Assure Scaling!

- Parallelization is often easy. Scaling is much harder.
- Amdahl's law governs scaling:

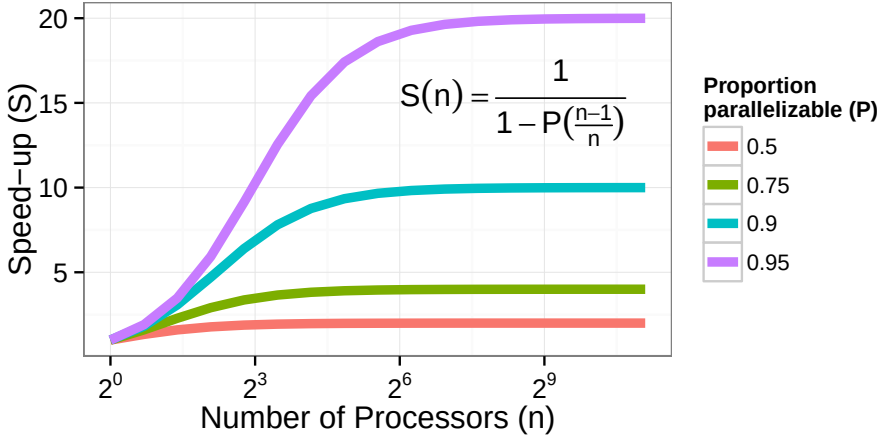
$$S(n) = \frac{1}{1 - P(\frac{N-1}{N})}$$

where $S(n)$ is the proportional speed-up for n processors, and P corresponds to the proportion of the program that is parallelizable.

What counts as non-parallelizable?

- Sequential part is anything that can't run in parallel
  - Memory bandwidth bounds (`a <- b + c`)
  - Invoking parallelism (kicking off processes is sequential)
  - But inefficiency can help scaling

Amdahl's law


20
15
10
5
Speed−up (S)
$S(n) = \frac{1}{1 - P(\frac{n-1}{n})}$
2^0
2^3
2^6
2^9
Number of Processors (n)
Proportion parallelizable (P)
0.5
0.75
0.9
0.95

Code May Run Slower in Parallel

- Parallelization can slow down execution time
- Example from SNOW documentation

```
set.seed(42)
A <- matrix(rnorm(1000000), 1000)

## Check CPU time for unparallel matrix multiplication:

system.time(A %*% A)
```

```
##    user  system elapsed
##   0.191   0.007   0.204
```

… And for parallel matrix multiplication:

```
library(snow)
cl <- makeSOCKcluster(names = 3L)
system.time(parMM(cl, A, A))
```

```
##    user  system elapsed
##   0.114   0.053   0.503
```

```
stopCluster(cl)
```

Why is it slower?

Communication will always be slower than computation!

Try to minimize data movement.

Other Constraints

- The algorithm may have inherent limits to scalability.
- Hardware factors play a significant role in scalability.
- Memory-cpu bus bandwidth on an SMP machine
- Communications network bandwidth
- Amount of memory available on any given machine or set of machines
- Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Designing Parallel Programs

First step is to decompose the problem into “chunks” of work that can be distributed to multiple tasks.

There are two basic ways to partition computational work among parallel tasks

Decompositions

Domain decomposition The data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.

Functional decomposition Focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

Outline

Parallelism in R – Many Avenues

CRAN package	Function
`snow`	Simple Network of Workstations
`multicore`	SMP-based parallelism for Unix-like operating systems
`rmpi`	MPI-based parallelism
`rpvm`	PVM-based parallelism

Historical Offerings

- `NetWorkSpaces` and `sleighs`
- `foreach` and `%dopar%` with parallel backends
  - `doSNOW`
  - `doNWS`
  - `doMC` (Uses the package `multicore`)
  - `doParallel` (Uses the package `parallel`)

Since version 2.14.0, open source R includes a new base package, `parallel`, combining elements of `snow` and `multicore`

Revolution R Enterprise - RevoScaleR

- A package of parallelized statistical functions ready to use capable of handling very large data sets and with excellent scaling properties
- Consists of functions for High-Performance Analytics that combine parallel operation with the efficient management of massive data sets
- Traditional High Performance Computing involves CPU-intensive computations on relatively small data sets.
- Support Parallelizing your own applications through parallel backends and parallel operations

High Performance Computing Approach

- All the initial parallel offerings in R were designed to support problems in High Performance Computing
- High Performance Computing is CPU centric
  - Lots of processing on small amounts of data
  - Focus is on cores

High Performance Analytics

- But, as we have seen, parallelism does not ensure scaling!
- Scalability requires not only parallelism, but efficient data management
- High Performance Analytics is data centric
  - Less processing per amount of data
  - Focus is on feeding data to the cores
  - On disk I/O
  - On efficient threading, data management in RAM

RevoScaleR: HPC and HPA Together

- RevoScaleR addresses both parallelism and data management concerns.
- Implements many popular statistical techniques as High Performance Analytics, providing scalable data analysis from in-memory data frames to massive, distributed disk files.
- Provides `rxExec()` function to facilitate traditional High Performance Computing.
- This course focuses on HPC programming

Outline

HPA Capabilities Quick Review

- Types of functionality
  - Data management
  - Analytics
  - Visualization
- Scales from small local data to huge distributed data
- Scales from laptop to server to cluster to cloud
- Portable – the same code works on small and big data, and on laptop, server, cluster, cloud

HPA: Data management capabilities

- Use data from a wide variety of sources (ODBC, SAS, SPSS, CSV, XDF)
- Transform and clean variables
- Code and recode factors
- Missing values
- Data validation
- Sort
- Merge
- Aggregate, summarize

HPA: Full-featured analysis algorithms

- Descriptive statistics, overall and by group
- Tables and cubes
- Correlations and covariances
- Linear regressions
- Logistic regressions
- Generalized linear models
- K means clustering
- Predictions (scoring), including std errors

HPA: Full-featured analysis algorithms

- PCA, Factor Analysis
- Stepwise regression
- Decision trees
- Random Decision forests
- Boosted Decision Trees

Sample code for logistic regression

Consider the kyphosis data set in the rpart package. Data set consists of 81 observations of four variables (Age, Number, Kyphosis, Start) in children following correctivespinal surgery. The variable Kyphosis reports the absence or presence of this deformity.

```
library(rpart)
rxLogit(Kyphosis ~ Age + Start + Number, data = kyphosis)
```

```
## Logistic Regression Results for: Kyphosis ~ Age + Start + Number
## Data: kyphosis
## Dependent variable(s): Kyphosis
## Total independent variables: 4
## Number of valid observations: 81
## Number of missing observations: 0
##
...
```

About Compute Contexts

- Allow you to easily change available compute computing resources.
- Key to RevoScaleR's parallel flexibility
- Default compute context is local, sequential
- Easy to switch to local, parallel compute context—no user setup required except to switch compute context:

```
rxSetComputeContext(RxLocalParallel())
```

Crosstabulation with rxCube

- One very useful thing to do with large data is to count it effectively— of all the 32-35 year olds who have purchased x, how many have also purchased y?
- `rxCube()` and `rxCrossTabs()` are two functions that do essentially the same thing (crosstabulations) but differ in the form of their output.
- All predictors need to be factor variables; they are inherently interactions, so are separated by `:` instead of + in formula

rxCube example

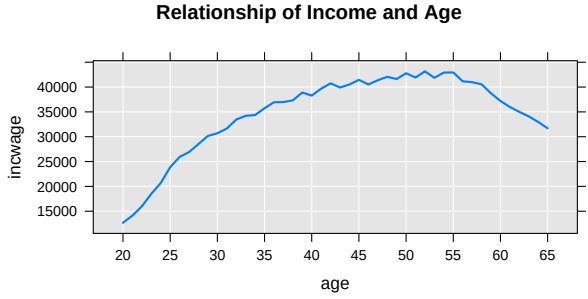
```
readPath <- system.file("SampleData", package = "RevoScaleR")
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
censusWorkersCube <- rxCube(incwage ~ F(age), data = censusWorkers)
censusWorkersCube
```

```
## Call:
## rxCube(formula = incwage ~ F(age), data = censusWorkers)
##
## Cube Results for: incwage ~ F(age)
## File name:
##    /usr/lib64/Revo-7.3/R-3.1.1/lib64/R/library/RevoScaleR/SampleData/CensusWorkers.xdf
## Dependent variable(s): incwage
...
```

```
censusWorkersCube$age <- as.integer(levels(censusWorkersCube$F_age))[censusWorkersCube$F_age]
censusWorkersCubeDF <- as.data.frame(censusWorkersCube)
```

Ploting

```
rxLinePlot(incwage ~ age, data = censusWorkersCubeDF, title = "Relationship of Income and Age")
```



Linear Models with rxLinMod

- Scalable function for linear models
- Easily supports hundreds of computed variables
- Returned object does not contain fitted values, residuals, or other components that are the length of the data (you can obtain these with rxPredict)
- The `cube` argument allows quicker solution when first predictor is a factor (allows use of partitioned inverse method)

rxLinMod examples

```
fourthgraders <- file.path(rxGetOption("sampleDataDir"), "fourthgraders.xdf")
fourthgradersLm <- rxLinMod(height ~ eyecolor, data = fourthgraders,
  fweights = "reps")
summary(fourthgradersLm)
```

```
## Call:
## rxLinMod(formula = height ~ eyecolor, data = fourthgraders, fweights = "reps")
##
## Linear Regression Results for: height ~ eyecolor
## Data: fourthgraders (RxXdfData Data Source)
## File name:
##    /usr/lib64/Revo-7.3/R-3.1.1/lib64/R/library/RevoScaleR/SampleData/fourthgraders.xdf
...
```

Generalized Linear Models with rxGlm

- Relaxes assumptions required of linear models
- Allows functional form (the “link” function) for conditional mean of the predictor
- Allows you to specify distribution of the response
- Logistic regression models are a special case (binomial/logit)

GLM example

```
library(robust)
data(breslow.dat, package = "robust")
myGlm <- rxGlm(sumY ~ Base + Age + Trt, dropFirst = TRUE, data = breslow.dat,
  family = poisson())
```

Glm Summary

```
library(robust)
summary(myGlm)
```

```
## Call:
## rxGlm(formula = sumY ~ Base + Age + Trt, data = breslow.dat,
##     family = poisson(), dropFirst = TRUE)
##
## Generalized Linear Model Results for: sumY ~ Base + Age + Trt
## Data: breslow.dat
## Dependent variable(s): sumY
...
```

Outline

High performance computing (HPC)

HPC Capabilities in Revolution R Enterprise

- Execute (essentially) any R function in parallel on nodes and cores
- Results from all runs are returned in a list to the user
- Extensive control over parameters
- Extensive control over nodes, cores, and times to run
- Ideal for simulations and for running R functions on small amounts of data
- Can run on single multicore workstation or large clusters/grids

Advantages over legacy tools

- With rxExec, you can replicate virtually any parallel computation performed with the legacy parallel tools mentioned earlier (`snow`, `NWS`, `rmpi`, `rpvm`).
- In our experience, the code required to do so will be easier to write and understand than the original.

A Simple rxExec Example

- Calculate square roots for sequence $1:4$

```
# First, set the compute context to local parallel
rxSetComputeContext(RxLocalParallel())
rxOptions(numCoresToUse = 4)

# Then, call rxExec with sqrt:
sqrt <- function(x) base::sqrt(x)
rxExec(sqrt, 1:4)
```

```
## [[1]]
## [1] 1.000000 1.414214 1.732051 2.000000
##
## [[2]]
## [1] 1.000000 1.414214 1.732051 2.000000
##
## [[3]]
...
```

A Simple rxExec Example

```
rxExec(sqrt, rxElemArg(1:4))
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
...
```

A Simple rxExec Example

```
# The helper function rxElemArg allows you to pass different
# arguments to each worker. Compare to
rxExec(sqrt, 1:4, timesToRun = 4)
```

```
## [[1]]
## [1] 1.000000 1.414214 1.732051 2.000000
##
## [[2]]
## [1] 1.000000 1.414214 1.732051 2.000000
##
## [[3]]
...
```

```
# which calculates the square roots of the entire sequence 1:4
# four times.
```

Outline

Mandelbrot Example

A number *c* is part of a Mandelbrot set if

- $z_{n+1} = z_n^2 + c$ remains bounded and *z* and *c* are complex numbers

Every parallel system has run Mandelbrot set:

- Embarrassingly parallel
- Expandable to most any size
- No communication (except final image)
- Computationally simple (easy to write in C, Fortran, R)
- And it creates pretty pictures

More on Mandelbrot Sets

- http://mathforum.org/advanced/robertd/mandelbrot-d.html
- http://en.wikipedia.org/wiki/Mandelbrot_set

Basic/Naive Mandelbrot R Code

```
mandelbrot <- function(x0, y0, lim) {
  x <- x0
  y <- y0
  iter <- 0
  while (x^2 + y^2 < 4 && iter < lim) {
    xtemp <- x^2 - y^2 + x0
    y <- 2 * x * y + y0
    x <- xtemp
    iter <- iter + 1
  }
  iter
}
```

Observations

- Generates a single point for a given X, Y.
- We need to execute this over a region (number of x, y positions)
- This simple code generates `iter` which is the output value. The color used to represent this point depends on the number of iterations needed to meet the while condition.

First step: “vectorize”

Need to “Vectorize” Mandelbrot, because this gets us to parallelization stage.

Note – this is a naive example and not the best way of vectorizing.

```
vmandelbrot <- function(xvec, y0, lim) {
  unlist(lapply(xvec, mandelbrot, y0 = y0, lim = lim))
}
```

- We are very close to being able to parallelize Mandelbrot

Now to parallelize

We Need to …

- Define input data
- Define the dimensions of the problem
- Create an output matrix
- Be sure we are using local parallel compute context
- Apply rxExec

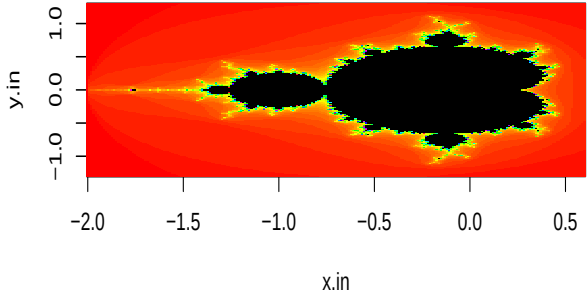
Let's see the code

```
size <- 150
x.in <- seq(-2, 0.6, length.out = size)
y.in <- seq(-1.3, 1.3, length.out = size)
m <- 100
z <- rxExec(vmandelbrot, x.in, y0 = rxElemArg(y.in), m, execObjects = "mandelbrot")
z <- matrix(unlist(z), ncol = size)  #order the data for the image
```

Once we had the vectorized version, going to the parallelized version is pretty straightforward. `x.in`, `y.in` and the limit to the number of iterations are needed as is the size.

Resulting Image

```
image(x.in, y.in, z, col = c(rainbow(m), "#000000"))
```



Exercise - Compare the peformance

Compare performance of the Mandelbrot Function for the following:

- Local Sequential
- Parallel with 2 cores
- Parallel with 4 cores
- Parallel with 8 cores

Additional rxExec arguments

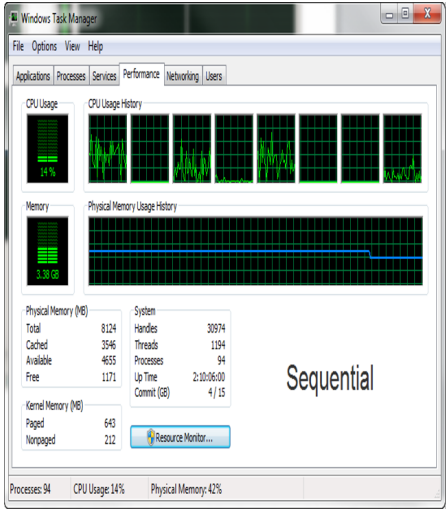
- Additional `rxExec()` arguments give control over worker
- Some of the options are
  - `execObjects` : allows you to pass needed objects to workers without passing entire global environment
  - `taskChunkSize` (tuning): number of tasks for workers. Reduces number of worker invocations; improves efficiency

Controlling Chunk Size

- As is, `rxExec()` works one value at a time
- Can reduce overhead by controlling `taskChunkSize`
- Usually, `taskChunkSize` should be chosen so that each worker is allocated multiple tasks with each dispatch.
- Setting `taskChunkSize = #tasks / #Workers` is shown here, but is seldom optimal

```
z2 <- rxExec(vmandelbrot, x.in, y0 = rxElemArg(y.in), m, taskChunkSize = 48,
  execObjects = "mandelbrot")
```

Illustrative Mandelbrot Workloads



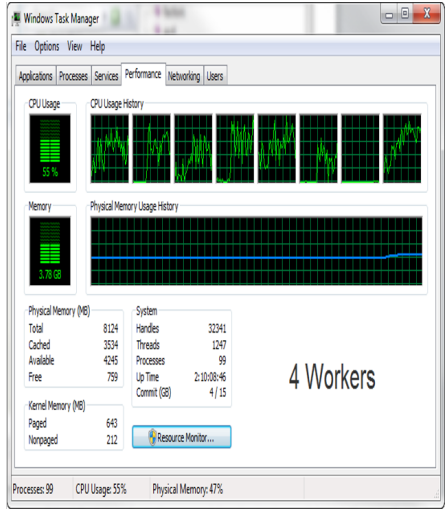


Figure: Processor Activity for sequential and parallel executions

Lab Exercise

- Goal: Create a program which will print out `Hello, world` on each worker.
- With `rxExec()`, this is quite trivial, since `RevoScaleR` is automatically loaded when you start Revolution R Enterprise.

Hint:

- Step 1: Register a compute context with parallel abilities
- Step 2: Call `rxExec()` to print “Hello, world” on each worker.

Lab Exercise 2: A Less Trivial Example

Goal: Create a program to plot the Mandelbrot set for a variable number of sizes and iterations

- Step 1: Start with the Mandelbrot example shown earlier.
- Step 2: Encapsulate the code into an R function
- Step 3: Run the code with `size=240` and `m=100`

Exercise 3…

Remember the earlier example with SNOW…

Why are the prior `mandelbrot()` and `vmandelbrot()` functions naive examples?

How could you improve their efficiency?

Session Summary

- Parallelism in general
  - Parallelization does not assure scaling
- Introduced `RevoScaleR` and the distinction between High Performance Computing and High Performance Analytics
- Saw example of HPA computing with RevoScaleR
- R parallelizing operator `rxExec()`
  - Applied `rxExec()` to Mandelbrot set

Questions?

REVOLUTION ANALYTICS

Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com
1.855.GET.REVO
Twitter: @RevolutionR