Parallel Computing in R

BioC 2009, Seattle, July 2009

Who are REvolution Computing?

REvolution Computing:

- Is a commercial open-source company, founded in 2007
- Provides services and products based on R
  - The "Red Hat"® for R
- Produces free and subscription-based high-performance, enhanced distributions of R
- Offers support, training, validation and other services around R
- Has expertise in high-performance and distributed computing
- Is a financial and technical contributor to the R community
- Has operations in New Haven, Seattle, and San Francisco

REvolution computing
We do the math

The People of REvolution

- **Martin Schultz**, Chief Scientific Officer (Arthur K Watson Professor of Computer Science, Yale University; founder of Scientific Computing Associates; research in algorithm design, parallel programming environments and architectures)
- **David Smith**, Director of Community & R blogger (co-author of *An Introduction to R, ESS*)
- **Bryan Lewis** , Ambassador of Cool (aka Director of Systems Engineering; applied math interests in numerical analysis of inverse problems; former CEO of Rocketcalc)
- **Danese Cooper**, Open Source Diva (board of directors, Open Source Initiative; member, Apache Software Foundation; advisory board, Mozilla.org; previously senior director, open source strategies at Intel and Sun)
- **Steve Weston**, Senior Research Scientist, Director of Engineering (REvolution and Scientific Computing Associates; development of NetWorkSpaces – parallel programming with R, Python, Ruby, and Matlab – Network Linda, Paradise, and Piranha).
- **Jay Emerson**, Dept Statistics, Yale University (author of *bigmemory* package)

REvolution computing
We do the math

What is REvolution R?

- REvolution R is the free distribution of R
  - Optimized for speed
  - Uses multiple CPUs/cores for performance
  - For Windows and MacOS (soon: Ubuntu)
  - Support via community forums

- REvolution R Enterprise is our enhanced, subscription-only distribution of R
  - Telephone/email support from real R experts
  - Suitable for use in regulated/validated environments
  - Includes proprietary ParallelR packages for reliable distributed computing with R
    - on clusters or in the cloud
  - Supported on 64-bit Windows, Linux

REvolution computing
We do the math

Supporting the R Community

We are an open source company supporting the R community:

- **Benefactor of R Foundation**
- **Financial supporter of R conferences and user groups**
- **New functionality developed in core R to contributed under GPL**
  - **64-bit Windows support**
  - **Step-debugging support**
- **R Evangelism**

"Revolutions" Blog: blog.revolution-computing.com

Daily News about R, statistics, and open-source

In today's lab:

- Introduction to Parallel Processing
- Multi-Threaded Processing
  - Computing on the GPU
- Iterators
- The foreach loop
- Using multiple cores: SMP
- Cluster Computing
- Multi-Stratum parallelism
- Q & A / Exercises

Getting Started

- R 2.10.x, and 2.9.x on Windows:

```
install.packages("foreach",type="source")
install.packages("iterators",type="source")
```

- R 2.9.x, Mac/Linux only:

```
install.packages("doMC")
require(doMC)
```

- Windows/Mac:
  - Install REvolution R Enterprise 2.0 (R 2.7.2)

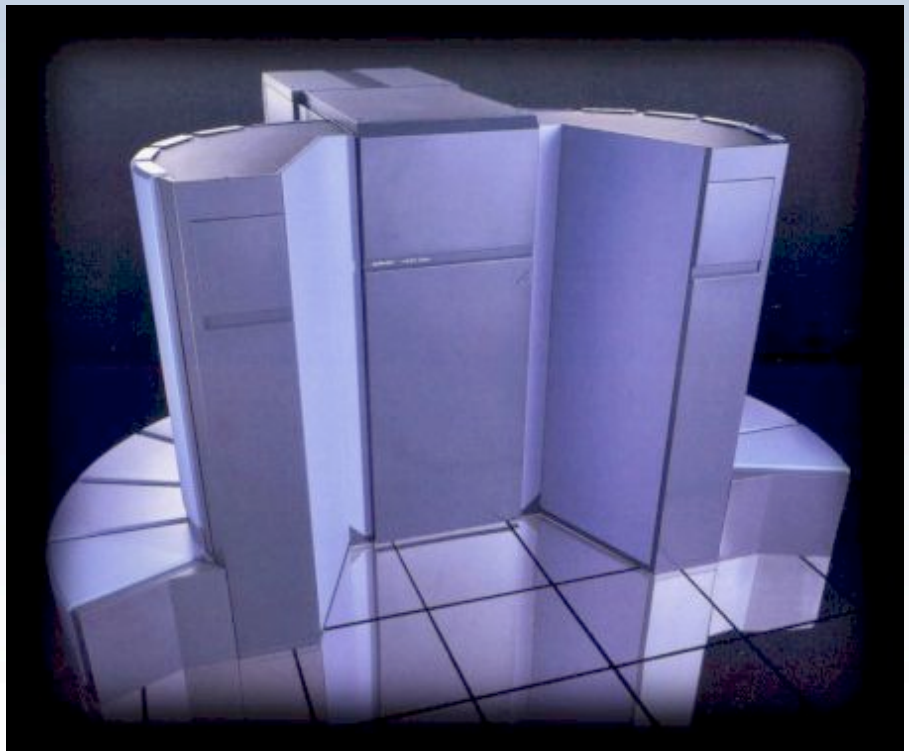
```
require(doNWS)
```

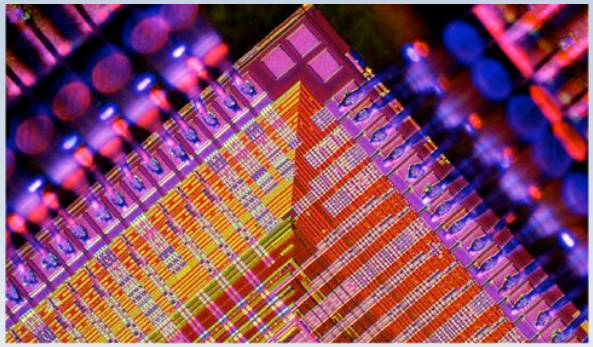
Introduction to Parallel Processing

With an aside to High-Performance Computing

What is High-Performance Computing (HPC)?

HPC often means efficiently exploiting specialized hardware





REvolution computing
We do the math

What is High-Performance Computing (HPC)?

- These days, HPC is frequently associated with COTS* cluster computing and with SIMD vectorization and pipelining (GPUs)

* Commodity, off the shelf

- New: cloud computing

Image from ncbr.sdsc.edu

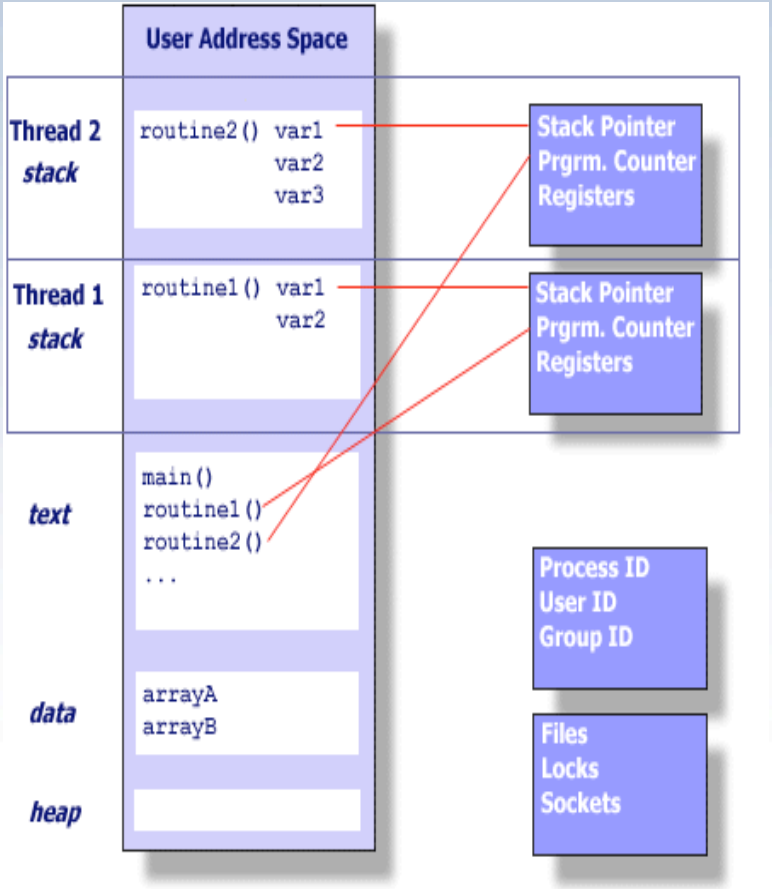
amazon web services™

Eucalyptus Systems

REvolution computing
We do the math

What is High-Performance Computing (HPC)?

- HPC is often concerned with multi-processing (parallel processing), the coördination of multiple, simultaneously running (sub)programs
  - Threads
  - Processes
  - Clusters



What is High-Performance Computing (HPC)?

HPC often involves effectively managing huge data sets

- Parallel file systems (GPFS, PVFS2, Lustre, GFS2, S3…)
- Parallel data operations (map-reduce)
- Working with high-performance databases
- `bigmemory` package in R

Image Copyright HP (a multi-petabyte storage system)

REvolution computing
We do the math

A Taxonomy of Parallel Processing

- **Multi-node / cluster / cloud computing** (heavyweight processes)
  - Memory distributed across network
  - Examples: foreach, SNOW, Rmpi, batch processing
- **Multi-core / multi-processor computing** (heavyweight processes)
  - SMP: Symmetric Multi-Processing
  - Independent memory in shared space
  - Naturally scales to multi-node processing
  - Examples: multicore (Windows/Unix), foreach
- **Multi-threaded processing** (lightweight processes)
  - Usually shared memory
  - Harder to scale out across networks
  - Examples: threaded linear-algebra libraries for R (ATLAS, MKL); GPU processors (CUDA/NVIDIA ; ct/INTEL)

Multi-Threaded Processing

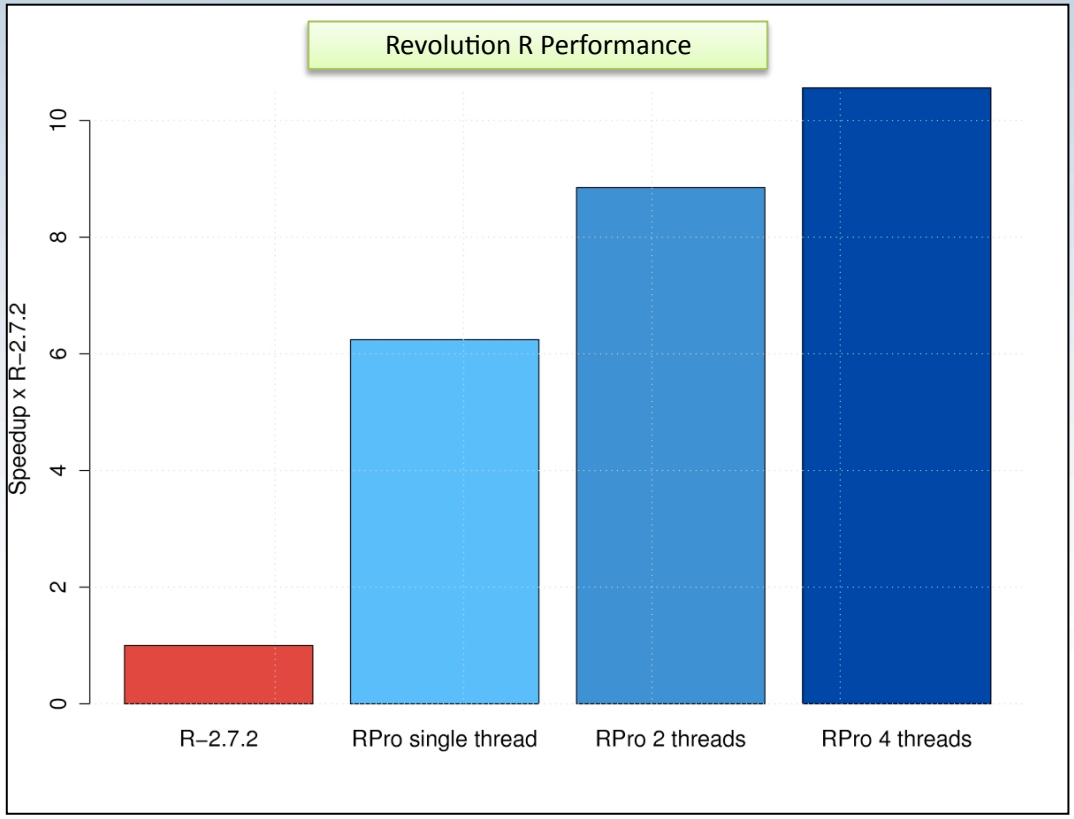
What is threaded programming?

- A thread is a kind of process that shares address space with its parent process
- Created, destroyed, managed and synchronized in C code
  - POSIX threads
  - OpenMP threads
- Fast, but difficult to program
  - Easy to overwrite variables
  - Need to worry about synchronization

Exploiting threads with R

- R links to BLAS (Basic Linear Algebra Subprograms) libraries for efficient vector/ matrix operations
- Linux: Need to compile and link with threaded BLAS (ATLAS)
- Windows/Mac: REvolution R linked to Intel MKL libraries, uses as many threads as cores
  - Many higher-level operations optimized as well
- MacOS: CRAN binary uses veclib BLAS
  - threaded, pretty good performance

REvolution R SVD Performance



Example data matrix

150,000 x 500

fast.svd

Quad-core Intel Core2 CPU,

Windows Vista 64-bit Workstation

REvolution computing

We do the math

GPU Programming

What is a GPU?

- Dedicated processing chip (or card) dedicated to fast floating-point operations
  - Originally for 3-D graphics calculations
- Highly parallel: 100's of processors on a single chip, capable of running 1000's of threads
- Usually includes dedicated high-speed RAM, accessible only by GPU
  - Need to transfer data in/out
- Programmed directly using custom C dialect / compilers
- > 90% of new desktops/laptops have an integrated GPU

GeForce 8800GT

- Launched Oct 29, 2007
- 512 Mb of 256-bit memory
- 128 processors
- 512 simultaneous threads
- < $200

- Download NVIDIA CUDA Tools:
  - http://www.nvidia.com/object/cuda_home.html
- Tutorial
  - http://www.ddj.com/architect/207200659

Performance Comparison

- Convolve 2 vectors of length 2^22
  - 60 Mb of data
- Quad dual-core processor / GPU

Method	Time (seconds)
Base R convolve function	9.89
AMD ACML	6.29
FFTW (8 threads)	3.75
CUDA on GeForce 8800GT	1.88 (single precision)

Introducing Iterators

Thought Experiment: Drawing Cards

- You are the teacher of 10 grade-school pupils.
- Class project: draw each of the 52 playing cards as a poster.
- Each child has supplies of poster paper and crayons, but requires a reference card to copy.

- How to organize the pupils?

REvolution computing
We do the math

Iterators

```
> require(iterators)
```

- Generalized loop variable
- Value need not be atomic
  - Row of a matrix
  - Random data set
  - Chunk of a data file
  - Record from a database
- Create with: `iter`
- Get values with: `nextElem`
- Used as indexing argument with `foreach`

Iterators are memory friendly

- Allow data to be split into manageable pieces on the fly
- Helps alleviate problems with processing large data structures
- Pieces can be processed in parallel

Iterators act as adaptors

- Allows your data to be processed by foreach without being converted
- Can iterate over matrices and data frames by row or by column:

```
it <- iter(Boston, by="row")
nextElem(it)
```

Numeric Iterator

```
> i <- iter(1:3)
> nextElem(i)
[1] 1
> nextElem(i)
[1] 2
> nextElem(i)
[1] 3
> nextElem(i)
Error: StopIteration
```

REvolution computing
We do the math

Long sequences

```
> i <- icount(1e9)
> nextElem(i)
[1] 1
> nextElem(i)
[1] 2
> nextElem(i)
[1] 3
> nextElem(i)
[1] 4
> nextElem(i)
[1] 5
```

REvolution computing
We do the math

Matrix dimensions

```
> M <- matrix(1:25,ncol=5)
> r <- iter(M,by="row")
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    7   12   17   22
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    8   13   18   23
```

REvolution computing
We do the math

Data File

```
> rec <- iread.table("MSFT.csv",sep=",", header=T, row.names=NULL)
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.91     30.25     29.4      29.86    76935100         28.73
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1      29.7     29.97    29.44      29.81    45774500         28.68
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.63     29.75    29.45      29.64    44607200         28.52
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.65      30.1    29.53      29.93    50220200          28.8
```

Database

```
> library(RSQLite)
> m <- dbDriver('SQLite')
> con <- dbConnect(m, dbname="arrests")
> it <- iquery(con, 'select * from USArrests', n=10)
> nextElem(it)
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
Connecticut   3.3     110       77 11.1
Delaware      5.9     238       72 15.8
Florida      15.4     335       80 31.9
Georgia      17.4     211       60 25.8
```

Infinite & Irregular sequences

```
iprime <- function() {
 lastPrime <- 1
 nextEl <- function() {
  lastPrime <<- as.numeric(nextprime(lastPrime))
  lastPrime
 }
 it <- list(nextElem=nextEl)
 class(it) <- c('abstractiter','iter')
it}

> require(gmp)
> p <- iprime()
> nextElem(p)
[1] 2
> nextElem(p)
[1] 3
```

REvolution computing
We do the math

Looping with foreach

Looping with `foreach`

```
foreach (var=iterator) %dopar% { statements }
```

- Evaluate `statements` until `iterator` terminates
- `statements` will reference variable `var`
- Values of `{ … }` block collected into a list

- Runs sequentially (by default) (or force with `%do%`)

```
> foreach (j=1:4) %dopar% sqrt (j)

[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

Warning message:
executing %dopar% sequentially: no
parallel backend registered
```

Combining Results

```
> foreach(j=1:4, .combine=c) %dopar% sqrt(j)
[1] 1.000000 1.414214 1.732051 2.000000

> foreach(j=1:4, .combine='+', .inorder=FALSE)
  %dopar% sqrt(j)
[1] 6.146264
```

- When order of evaluation is unimportant, use `.inorder=FALSE`

Referencing global variables

```
> z <- 2
> f <- function (x) sqrt (x + z)
> foreach (j=1:4, .combine='+') %dopar% f(j)
[1] 8.417609
```

- foreach automatically inspects code and ensures unbound objects are propagated to the evaluation environment

Nested foreach execution

- `foreach` operations can be nested using `%:%` operator
- Allows parallel execution across multiple loops levels, “unrolling” the inner loops

```
foreach(i=1:3, .combine=cbind) %:%
   foreach(j=1:3, .combine=c) %dopar%
     (i + j)
```

Speeding up code with foreach

SMP Processing

Quick review of parallel and distributed computing in R

- NetWorkSpaces (package `nws`; SMP, distributed)
  - GPL, also commercially supported by REvolution Computing
  - Very cross-platform, distributed shared-memory paradigm
  - Fault-tolerant
- MultiCore (package `multicore`; SMP only)
  - Linux / MacOS (requires POSIX)
  - Uses fork to create new R processes
- Rmpi (package `Rmpi`; SMP, distributed)
  - Fine-grained control allows very high-performance calculations
  - Can be tricky to configure
  - Limited Windows and heterogeneous cluster support
- SNOW (package `snow`; SMP, distributed*)
  - Limited Windows support (*single machine only)
  - Meta-package: supports MPI, sockets, NWS, PVM

Parallel backends for foreach

- %dopar% behaviour depends on current "registered" parallel backend

- *Modular* parallel backends

- registerDoSEQ (default)
- registerDoNWS (NetWorkSpaces)
- registerDoMC (multicore, MacOS/Windows)
  - From Terminal/ESS only! (R.app GUI will crash.)
- registerDoSNOW
- registerDoRMPI

Getting Started: Multi-core Processing

- R 2.10.x – wait until official release
- R 2.9.x

```
require(doMC)
registerDoMC(cores=2)
```

- REvolution R Enterprise

```
require(doNWS)
s <- sleigh(workerCount=2)
registerDoNWS()
```

A simple simulation:

```
birthday <- function(n) {
  ntests <- 1000
  pop <- 1:365
  anydup <- function(i)
      any(duplicated(
    sample(pop, n, replace=TRUE)))
  sum(sapply(seq(ntests), anydup)) / ntests
}

x <- foreach (j=1:100) %dopar% birthday (j)
```

Birthday Example - timings

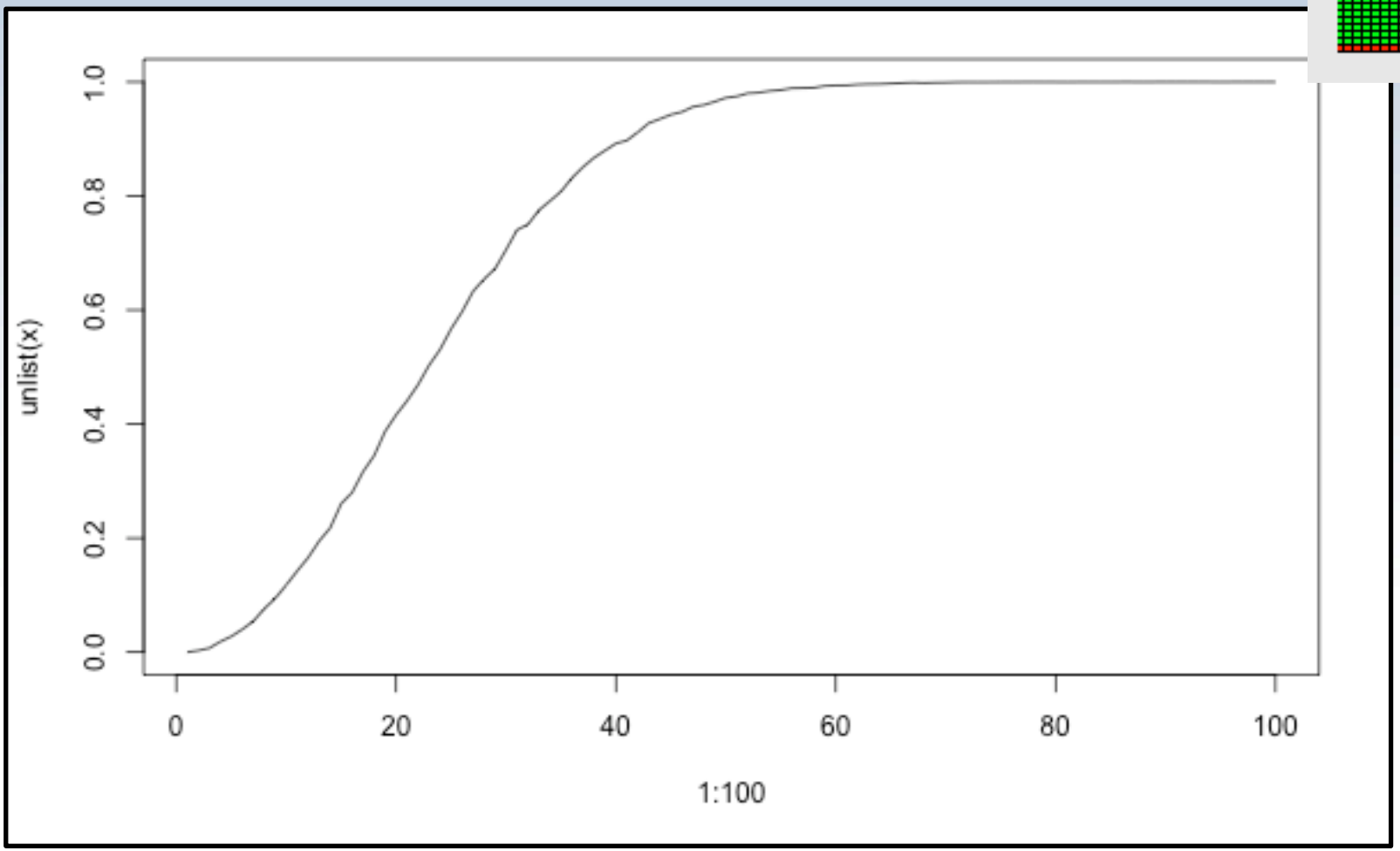
Dual-core 2.4GHz Intel MacBook:

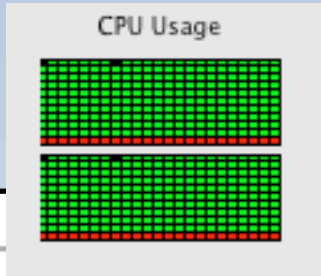
```
system.time{
 x <- foreach (j=1:100) %dopar%
   birthday (j)
} # Elapsed
```

Backend	Time (s)
`registerDoSEQ()`	41s
`registerDoMC()` # 2 cores	28s
`registerDoNWS()` # 2 workers	26s(*)

Birthday Simulation: Multicore / NWS

```
> x <- foreach (j=1:100) %dopar% birthday (j)
> plot(1:100, unlist(x),type="l")
```





REvolution computing
We do the math

Using clusters with NetworkSpaces

Setting Up a Cluster

1. Identify machines to form nodes on cluster
   - Easiest with Linux / MacOS
   - Possible with Windows
2. Select a server machine
   - OK for this one to be on Windows
3. Make sure passwordless ssh enabled on each worker node
   - `ssh` *`nodename`* `Revo --version` should work

Setting up a cluster, part 2

4. Log in to server, start REvolution R
5. Create a sleigh

```
require(doNWS)
s <- sleigh(nodeList=c(
   rep("localhost",2),
   rep("thor",8),
   rep("loki",4)),
     launch=sshcmd)
registerDoNWS(s)
```

6. Use foreach as before
7. (optional) use joinSleigh to add new nodes

Parallel Random Forest

```
# a simple parallel random forest
library(randomForest)
x <- matrix(runif(500), 100)
y <- gl(2, 50)
wc <- 2
n <- ceiling(1000 / wc)
registerDoNWS(s)
foreach(ntree=rep(n, wc), .combine=combine,
  .packages='randomForest') %dopar%
    randomForest(x, y, ntree=ntree)
```

- Easier: `randomShrubberyNWS()`

Converting existing code

- Convert these loops to `foreach`:
  - `for`: make body return iteration value and `.combine`
  - `apply`: use `iter(X, by="row")` and `.combine`
    - Or `iapply(X,1)`
  - `lapply`: use `iter(mylist)`

ppiStats Example

- Sequential:

```
bpMats1 <- lapply(bpList, function(x) {
  bpMatrix(x, symMat = TRUE, homodimer = FALSE,
  baitAsPrey = FALSE, unWeighted = FALSE,
  onlyRecip = FALSE, baitsOnly = FALSE)
})
```

- Parallel:

```
bpMats1 <- foreach(x=iter(bpList),
              .packages = "ppiStats") %dopar% {
  bpMatrix(x, sysMat = TRUE, homodimer = FALSE,
    baitAsPrey = FALSE, unWeighted = FALSE,
    onlyRecip = FALSE, baitsOnly = FALSE)
}
```

ppiStats Example

- Sequential:

```
bpGraphs <- lapply(bpMats1, function(x) {
  genBPGraph(x, directed = TRUE, bp = FALSE)
})
```

- Parallel:

```
bpGraph <- foreach(x=iter(bpMat1),
     .packages = "ppiStats") %dopar% {
   genBPGraph(x, directed = TRUE, bp = FALSE)
}
```

Excercise

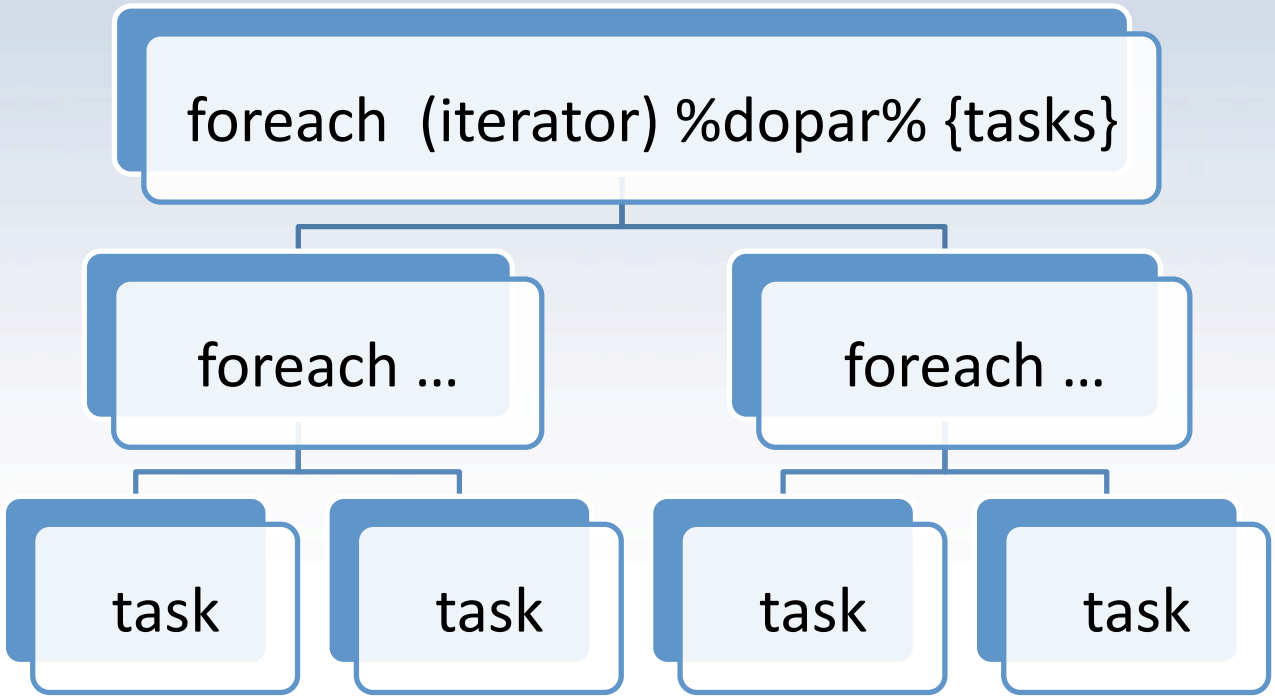
- Find other “embarassingly parallel” BioConductor examples, and convert to parallel with `foreach`.

Multi-Stratum Parallelism

An example of explicit multi-stratum||ism

CLUSTER

SMP



REvolution computing
We do the math

Multi-stratum template: NWS / Multicore

```
require ("doNWS")
require ("foreach")
require ("doMC")

s <- sleigh(nodelist=c(rep("localhost",2),
                       rep("bladeserver",8))
registerDoNWS(s)

foreach (iterator_i,
          .packages=c("foreach", "doMC"))%dopar%
   {
     registerDoMC()
     foreach (iteratorj_) %dopar% {
        tasks…
     }
   }
```

Pitfalls to avoid

- Sequential vs Parallel Programming
- Random Number Generation
  - `library(sprngNWS)`
  - `sleigh(workerCount=8,`
    `rngType='sprngLFG')`
- Node failure
- Cosmic Rays

Conclusions

- Parallel computing is easy!
- Write loops with foreach / %dopar%
  - Works fine in a single-processor environment
  - Third-party users can register backends for multiprocessor or cluster processing
  - Speed benefits without modifying code
- Easy performance gains on modern laptops / desktops
- Expand to clusters for meaty jobs

Thank You!

- David Smith
  - david@revolution-computing.com, @revodavid
- REvolution Computing
  - www.revolution-computing.com
- *Revolutions*, the R blog
  - blog.revolution-computing.com
- Downloads:
  - Slides: http://tinyurl.com/R-Bioc-slides
  - Script: http://tinyurl.com/R-Bioc-script

REvolution computing
We do the math