# High Performance Simulation and Computation in Biostatistics: Some Tutorial Examples

-lk

8th May 2014

# Outline

# Initialization

Initialize objects of increasing size at once and completely:

```
a <- NULL
for(i in 1:n) a <- c(a, foo(i)) # slow
a <- numeric(n)
for(i in 1:n) a[i] <- foo(i) # fast.
```

Don't apply error checking within loops! Check arguments before and/or results after the loop. Don't do things $n$ times that need to be applied just once (as it happens in loops, e.g., no need to count coverage probability within loop).

# read.table

The `read.table` function is one of the most commonly used functions for reading data into R. It has a few important arguemtns:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset
- `skip`, the number of lines to skip from the beginning

# read.table

For small ro moderately sized datasets, we can usually call `read.table` without specifying any other arguments

```
iris.data <- read.table("iris.txt")
```

R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table

In fact, telling R all these things directly makes R run faster and more efficiently

# read.table

With much larger datasets, e.g., microarry data with thousands of probes, doing the following will make life easier and will prevent R from choking.

- Read the help page for `read.table`, which contains many hints
- Make a rough calculation of the memory required to store the dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right there.

# read.table

By default, R goes through every column of the dataset and tries to figure out on its own what type of data it is.

- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. A quick and dirty way to figure out the classes of each column:

```
initial <- read.table("iris.txt", nrows=50)
classes <- sapply(initial, class)
tabAll <- read.table("iris.txt", colClasses=classes)
```

# Know the system

In general, when using R with larger datasets, it's useful to know a few things about our system.

- How much memory is available?
- What other applications are in use?
- What operating system?
- Is the OS 32 or 64 bit? (You are able to access more memory if the computer has a lot more memory on 64 bit)

# Calculating memory requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data, Roughly, how much memory is required to store this data frame?

$$
\begin{aligned}
1,500,000 \times 120 \times 8 \text{ bytes/numeric} &= 1440000000 \text{ bytes} \\
&= 1440000000/2^{20} \text{ bytes/MB} \\
&= 1,373.291 \text{ MB} \\
&= 1.34 \text{ GB}
\end{aligned}
$$

You need a little bit more memory to read this dataset into R.

# Dumping R objects

Multiple objects can be deparsed using the `dump` function and read back in using `source`. `dumping` is useful because the resulting textual format preserve the metadata, so that another user doesn't have to specify it all over again.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a b
1 1 a
> x
[1] "foo"
```

# data.R

```
x <- "foo"
y <- structure(list(a = 1, b = structure(1L, .Label = "a",
class = "factor")), .Names = c("a", "b"),
row.names = c(NA, -1L), class = "data.frame")
```

Vectorized operation is one of the features of the R language that makes it easy to use on the command line and makes it very kind of nice to write code without having to do lots of looping and things like that.

Many operations in R are vectorized making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

# Looping on the command line

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier and computing faster.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of lapply

# lapply

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

The actual looping is done internally in C code.

# sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length ($> 1$), a matrix is returned.
- If it can't figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1))
> sapply(x, mean)
         a          b          c
 2.5000000 -0.5677294  0.9900573
> x <- list(rnorm(100), runif(100), rpois(100, 1))
> sapply(x, quantile, probs = c(0.25, 0.75))
           [,1]      [,2] [,3]
25% -0.6702744 0.2064002    0
75%  0.5815927 0.8012238    2
```

# apply

`apply` is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

# apply

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be retained.
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

# apply

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, sum)
 [1] -3.23806413 -4.94035985  0.09638841 -2.80237376
 [5]  0.13526182 -4.78634046 -2.21373286 -2.44903169
 [9] -0.54052394  4.74367819 -1.22109580 -2.21044208
[13]  0.99714545 -5.59323015  0.17058743 -5.74101876
[17] -5.58225951 -2.96516927  3.72425820 -2.61576950
> apply(x, 2, mean)
 [1] -0.42655545 -0.20701347 -0.07781864 -0.25445159
 [5] -0.17354146 -0.13573665  0.05408274 -0.24067160
 [9] -0.10550660 -0.28439190
```

# row/col sums and means

For sums and means of matrix dimensions, we have some shortcuts.

- rowSums = apply(x, 1, sum)
- rowMeans = apply(x, 1, mean)
- colSums = apply(x, 2, sum)
- colMeans = apply(x, 2, mean)

The shortcut functions are much faster, but you won't notice unless you're using a large matrix. As they are written for speed, they blur over some of the subtleties of NaN and NA.

# tapply

`tapply` is used to apply a function over subsets of a vector.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

# tapply

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
[21] 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
         1          2          3
-0.1365905  0.5965098  1.3423046
```

# mapply

`mapply` is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
    USE.NAMES = TRUE)
```

- `FUN` is a function to apply
- `...` contains arguments to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

# mapply

The following is tedious to type

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

# mapply

```
> mapply(rnorm,n=1:5,mean=1:5)
[[1]]
[1] 0.352385

[[2]]
[1] 1.933362 2.011596

[[3]]
[1] 2.616066 4.542486 3.254903

[[4]]
[1] 4.061980 3.641198 5.091221 3.167971

[[5]]
[1] 4.331122 4.452907 4.624645 5.711368 6.276642
```

# Recursive function/factorial

```
> fact1 <- function(x) {
+ if (x==1) return(1) else return(x*fact1(x-1)) }
> fact2 <- function(x) { prod(1:x) }
> fact3 <- function(x) { gamma(x+1) }
>
> system.time(for (i in 1:1e4) fact1(100))
   user  system elapsed
   4.86    0.00    5.41
> system.time(for (i in 1:1e4) fact2(100))
   user  system elapsed
   0.03    0.00    0.03
> system.time(for (i in 1:1e4) fact3(100))
   user  system elapsed
   0.01    0.00    0.01
```

# Recursive function/Fibonacci

```
> fibR <- function(n) {
+     if (n == 0) return(0)
+     if (n == 1) return(1)
+     return (fibR(n - 1) + fibR(n - 2)) }
> fibonacci <- local({
+     memo <- c(1, 1, rep(NA, 100))
+     f <- function(x) {
+         if(x == 0) return(0)
+         if(x < 0) return(NA)
+         if(x > length(memo))
+         stop("'x' too big for implementation")
+         if(!is.na(memo[x])) return(memo[x])
+         ans <- f(x-2) + f(x-1)
+         memo[x] <<- ans
+         ans }
+ })
```

# Volume under the ROC surface

$$VUS = P(Y_1 < Y_2 < Y_3)$$
$$\widehat{VUS} = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \sum_{k=1}^{n_3} I(Y_{1i} < Y_{2j} < Y_{3k})$$

```
> vus.simple <- function(new.1,new.2,new.3) {
+ ksum <- 0
+ n1=length(new.1)
+ n2=length(new.2)
+ n3=length(new.3)
+ for (i in 1:n1) {
+  for (j in 1:n2) {
+   for (k in 1:n3) {
+ ksum = ksum + 1*((new.1[i]<new.2[j])&(new.2[j]<new.3[k]))
+                  }}}
+ return(ksum/n1/n2/n3)
+ }
```

# Using sapply

Here is a revised version of the above function.

```
> vus.MWU <- function(new.1,new.2,new.3) {
+ n1=length(new.1)
+ n2=length(new.2)
+ n3=length(new.3)
+ usum <- function(u) {
+ is.small = new.2[u<new.2]
+ if (length(is.small)>0) return(
+       sum(sapply(is.small, function(v) sum(v<new.3)))
+        ) else return(0) }
+ return(sum(sapply(new.1,usum))/n1/n2/n3)
+ }
```

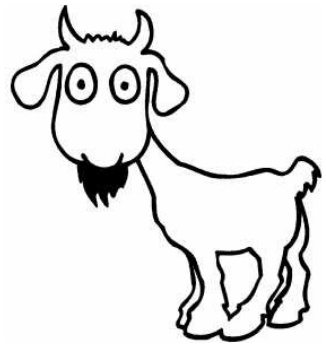
This function may seem to be awful.... But, the improvement is obvious! Try it.

# Internal parallelization

- *apply() functions
- **Parallel** package
  - Released with base beginning with R 2.14.0.
  - Authors: Brian Ripley, Luke Tierney and Simon Urbanek
  - Functionality derived from **snow** and **multicore**
  - Includes support for multiple random number generation streams
  - Works better with Linux than Windows

# Demonstration with the Monty Hall Problem

First described by Selvin in 1975 in the *American Statistician*. Appeared in Marilyn vos Savant column in Parade Magazine in 1990.

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1 [but the door is not opened], and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat  . He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

# Monty Hall Problem


Note that if you switch, you would lose only if your original choice (door 1) had the car.

# Results

With `parLapply()`, which is Windows compatible

| avg | runtime | cores |
|---|---|---|
| 0.666313 | 9.639 | 2 |
| 0.665941 | 5.011 | 4 |
| 0.666065 | 3.152 | 8 |

nsim = 1000000

# Computing cluster/parallel architectures

Example: UB CCR/U2, FDA/BETSY, Argonne/Blue Gene...

- Shared memory
- Threads
- Message Passing Interface (MPI), e.g., high-dimension matrix multiplication with partitioning.
- Data Parallel

# UB Center for computational research

Tutorials and Presentations

http://ccr.buffalo.edu/support/training-resources/tutorials/tutorials-date.html User

Guide

http://ccr.buffalo.edu/support/UserGuide.html

# Running jobs on cluster with PBS

## <pbs_script>

```
#PBS -S /bin/bash
##PBS -q debug
#PBS -l walltime=00:10:00
#PBS -l nodes=1:ppn=1
#PBS -M xxxxxx@buffalo.edu
#PBS -m e
#PBS -N test
#PBS -o test.out
#PBS -j oe

# The above directives can be commented out using an
# additional "#" (as in the debug queue line above)

. ${MODULESHOME}/init/bash
module load r

# cd to directory from which job was submitted
#
cd $PBS_O_WORKDIR

NPROCS=‘cat $PBS_NODEFILE | wc -l‘
which R
R CMD BATCH --no-save --no-restore test.r test.log
```

```
[lekang@u2 ~]$ qsub pbs_script
```

# Running different jobs using Perl scripts

```
#!/usr/bin/perl -w
use strict;
my $TEMPLATE = "power.r";

for (my $dist_choice=1; $dist_choice<=3; $dist_choice++) {
  my $BASENAME = "sim_dist" . "$dist_choice" ;
  my $SUBNAME  = "sub_dist" . "$dist_choice" ;
  open(BASEFILE,"<$TEMPLATE") or die "Unable to open template file: $TEMPLATE\n";

  my $newRfile = $BASENAME . ".R";
  open(RFILE,">$newRfile") or die "Unable to open new R input file: $newRfile\n";
  if ($dist_choice==1) {  printf RFILE "dist='Nor'  \n";               }
  if ($dist_choice==2) {  printf RFILE "dist='logNor' \n";             }
  if ($dist_choice==3) {  printf RFILE "dist='Exp'\n";                 }
  while (<BASEFILE>) {  chomp;    printf RFILE "$_\n"; }
  close(RFILE);

  my $sge_name = $SUBNAME;

  open(SGEFILE,">$sge_name") or die "Unable to open/create SGE input file: $sge_name\n";
  printf SGEFILE "#!/bin/sh\n";
  ...
  printf SGEFILE "R CMD BATCH --no-save --no-restore $newRfile $newRfile.txt\n";
  close(SGEFILE);

  system("qsub $sge_name");
}
```

Submit jobs with different simulation configrations/parameters!

# Array jobs – alternative to Perl scripts

- You only have to write one shell script
- You don't have to worry about deleting thousands of shell scripts, etc.
- If you submit an array job, and realize you've made a mistake, you only have one job id to `qdel`, instead of figuring out how to remove 1000s of them.
- You put a lot less of burden on the head node/cluster front end machine.

# Array jobs – implementation on PBS/SGE

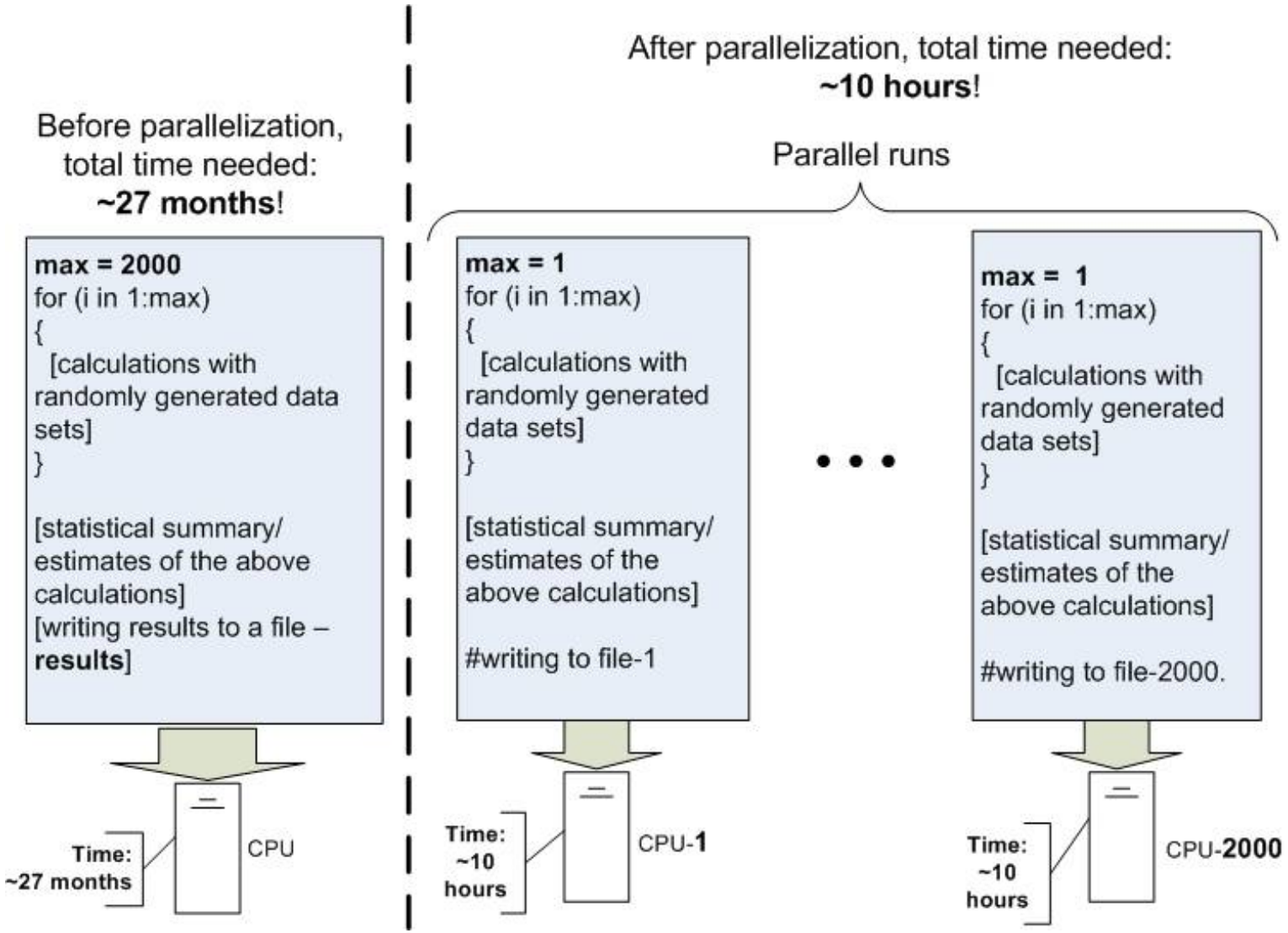
```
#PBS -t 1-100 ## on Portable Batch System

#$ -t 1-100   ## on Sun Grid Engine, number of times to run
```

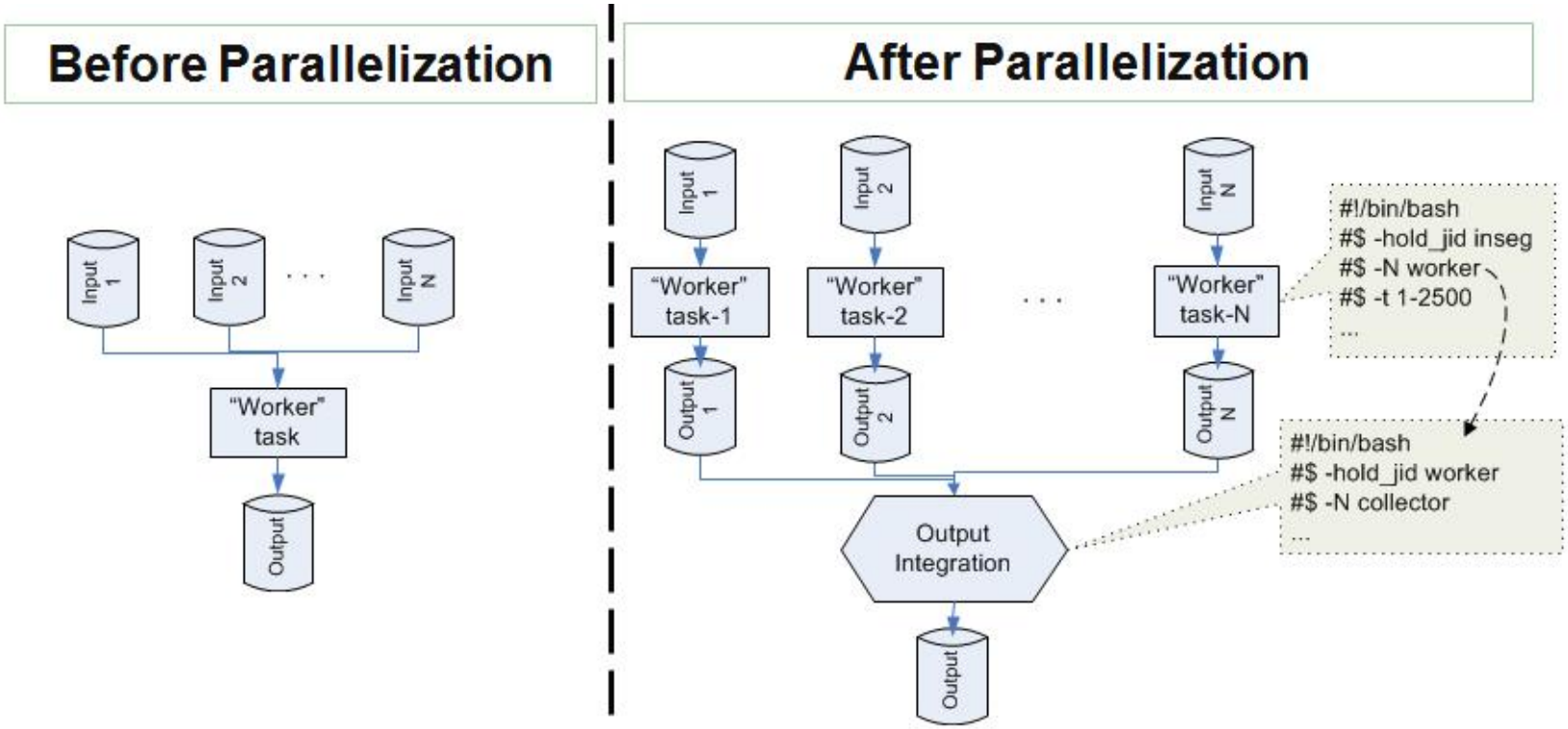
In R code, modify correspondingly with array ID
**Sys.getenv("PBS_ARRAYID")** or
**Sys.getenv("SGE_TASK_ID")**

*Bonus:* this array job feature has not yet documented on UB CCR Support Page!

# Idea underlying array jobs



# Data analysis parallel



# Random numbers in parallel runs

During the parallelization process, multiple seeds are used to generate sub-sets of pseudo random number sequence (PRNS) on different computing nodes, requiring

- A systematic way to provide unique seeds on the nodes;
- Minimization or avoidance of correlated/ overlapping PRNSs caused by use of multiple seeds.

Using current time within application instances mixed with unique ranks or task ids of the instances to form unique seeds cannot guarantee the **uniqueness** because the instances may not start in any specific order.

# One solution

Before submitting the parallel job for execution:

- The current time is taken and kept in an environment variable (e.g., RAND_INIT);
- It is passed to the application instances as a parameter.

A unique random seed is formed from within a code segment, where an independent stream of random numbers is required:

$$\text{seed} = \text{RAND\_INIT} + \text{task\_id}$$

# R integration with compilers

It is well known that low-level Assembly language or mid-level one like C/Fortran is highly efficient in dealing with loops, iterations and etc.

For this reason, the R development core group discussed a way to get all the speed advantages of compiled code with most of the convenience of R.

# Writing R function calling compiled C code

A typical procedure to call C from R:

- compile the C/C++ codes into a shared library (.dll on Windows platform and .so (.o) on Unix/Linux). note that Unix/Linus has free GNU gcc/g++ compiler.
- load the shared library by dyn.load("LIBNAME").
- write a wrapped function in R and call the compiled C function by .C("FUNNAME",...).

Here LIBNAME is the name of the shared library file and FUNNAME is the name of the C function.

# Obtain returned values from C

The C functions should have no return value. To return some values from a C function to R, one should set an argument as the value passer.

In the C functions to be called by R, all the arguments should be pointers. Therefore, the C functions should assign the to-be-returned value(s) to the value(s) pointed by this value-passer pointer.

# VUS estimation revisited

```
###### filename: empAUC_VUS.c
#include<stdio.h>
#include<R.h>
#include<Rmath.h>

void empVUS(double *x, double *y, double *z,
       int *n1,   int *n2,   int *n3,  double *result) {
    int i, j, k;
    double ksum = 0;

for (i=0; i<*n1; i++) {
for (j=0; j<*n2; j++) {
for (k=0; k<*n3; k++) {
   if ( (x[i]<y[j]) && (y[j]<z[k]) ) ksum++; }}}
   *result = ksum/(*n1 * *n2 * *n3);
}

###### nonp_VUS.r
dyn.load("empAUC_VUS.dll")
nonp.vus <- function(x,y,z) {
.C("empVUS",as.double(x),as.double(y),as.double(z),
            as.integer(length(x)),
            as.integer(length(y)),
            as.integer(length(z)),
            result=double(1))[["result"]]
}
```

Wrapped function `nonp.vus` calls the compiled C code to calculate VUS.

# Disadvantages

C does not have many data types as R does! e.g., matrix/array.

A lot of coding details (float?integer?...) that can be confusing!

Also, need to know C besides R! Nightmare....I just want to go sleeping....

But! Think about the time and effort you could save in hundreds thousands of simulations....

May help you find a job! More and more positions are saying C/C++ is a plus.

It is worthwhile to start learning right now!

# References

*An Introduction to R*

*Writing R Extensions*

*R Data Import/Export*

*An Introduction to the .C Interface to R*

*High-Performance and Parallel Computing with R*

*Reproducible Research with R, LATEX, & Sweave*

*UB CCR: PBS User's Guide*

*Beginner's Guide to SUN GRID ENGINE 6.2*