Parallel Processing in R

James Balamuta

Departments of Informatics and Statistics
University of Illinois at Urbana-Champaign

December 07, 2018

On the Agenda

- Parallel Processing
  - Terminology
  - Serial vs. Parallel
- Parallel in *R*
  - `parallel`

On the Agenda

1. Parallelization
   - Motivation
   - Parallelization Lingo
   - Processing
   - Serial Processing
   - Parallel Processing
   - Job Distribution
   - Dependency Types
   - Achieving Parallelization
   - Laws of Parallelization
2. Parallelization in R
   - Explicit
   - Structure
   - Apply Functions

Parallel Computing

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."
— Gene Amdahl in 1967.

NCSA Video: A world without supercomputers

Figure 1: NCSA Video: A world without supercomputers

Parallel Computing

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."
— Tim Mattson, principal engineer at Intel

Revisiting Oxen vs. Chicken

Recall:

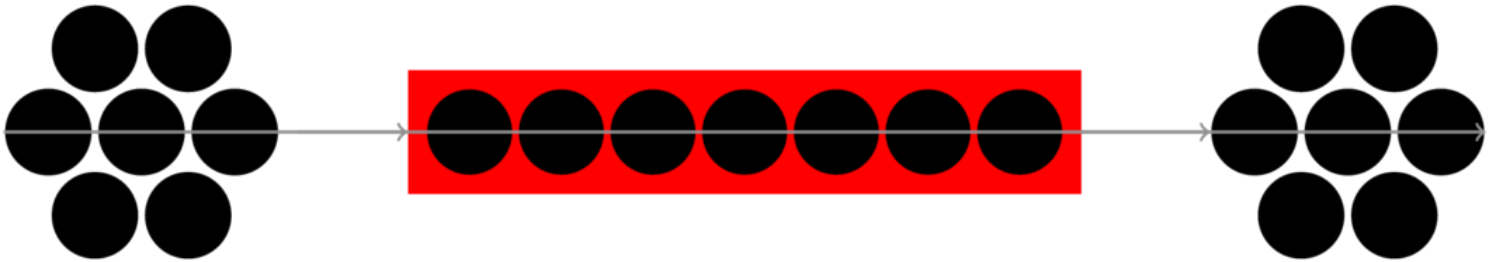
> *"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"*
> — *Seymore Cray*

- Before the chickens were equally as powerful just not **united** well.
- How can we **unite** the power of that many chickens?

What is Serial Processing?

Definition:

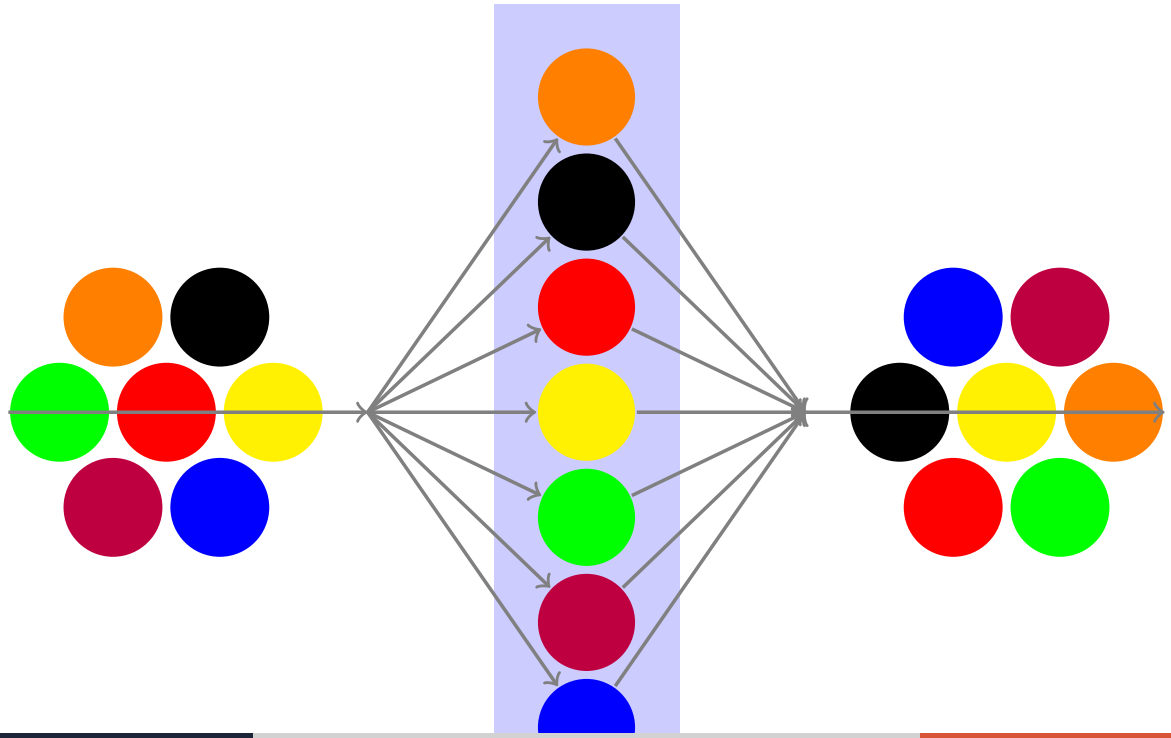
- *Serial Processing* is a method of sequentially working on tasks.
- Only one task may be worked upon at a time.
- After that task is finished, the next task is then worked upon.



What is Parallel Processing?

Definition:

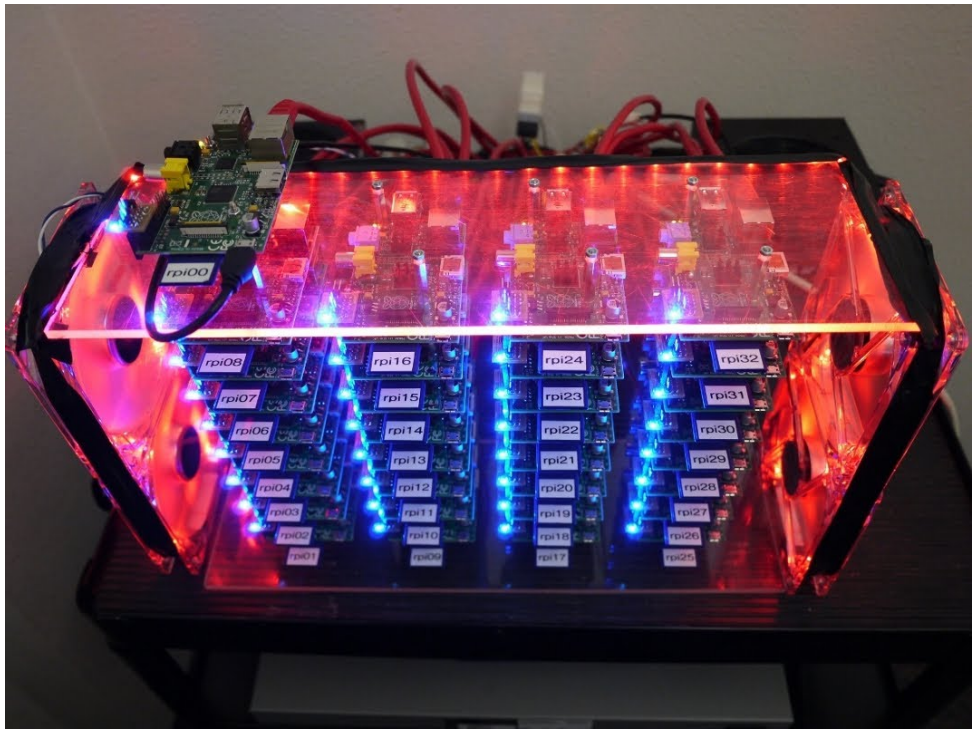
- *Parallel Processing* is the act of carrying out multiple tasks simultaneously to solve a problem.
- This is accomplished by dividing the problem into independent subparts, which are then solved concurrently.



Parallel Lingo

Cluster:

A cluster is a set of computers that are connected together and share resources as if they were one gigantic computer.



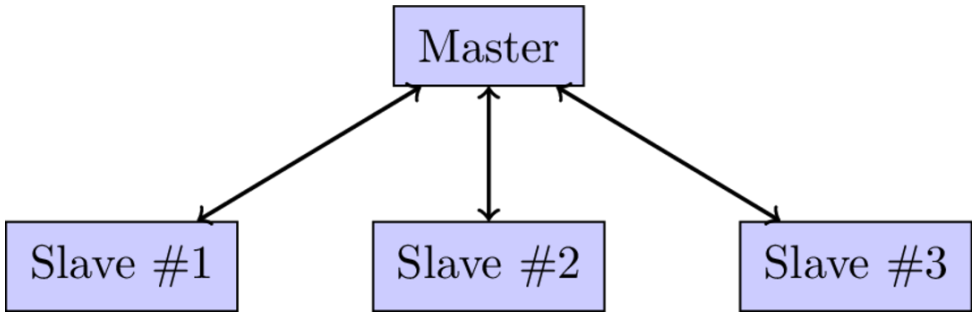
Parallel Lingo Cont.

Head (Master):

The primary computer that controls operations on the cluster.

Compute (Slave / Worker):

Computers that perform computations given to them by the master.



Parallel Paradigm - Industry

Parallelism is **everywhere** in the real world.

- CEOs distribute work to their employees, leave for a week, and then expect a report on their desks.
- Architects distributing blueprints to construction workers who individually work on sections of the building.

Parallel Paradigm - Academia

- Professors distribute ideas to graduate students, wait for them to solve the ideas, then aggregate the results into a paper.

Source: The Simpson S18E6 - Moe'N'A Lisa

Types of Processing

There are three types of processing:

1. Serial
2. Parallel
3. Hybrid (both approaches)

Serial Example - Loop

Here is a typical serial processing with a `for` loop.

```
n = 10000                                   # Number of obs
x = numeric(n)                              # Set up storage
system.time({
  for(i in seq_along(x)){                   # For loop
    x[i] = pf(i, df1 = 4, df2 = 5)
  }
})
```

```
##    user  system elapsed
##   0.017   0.001   0.021
```

Serial Example - Vectorization

Here is a typical serial processing with for

```
n = 10000                                  # Number of obs
system.time({
  x = pf(1:n, df1 = 4, df2 = 5)   # Pre-vectorized
})
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

Timing As Expected...

- Given the work previously done, the `for` loop is a bit slower than the vectorized component.
- The reason for this is due to *R* evaluating the vectorization at a low *C* level vs. high level interpretation.
- Though, there is a common misconception that vectorization evaluates items in a non-sequential manner... **That simply isn't true.**

Vectorization is sequential - Waiting for Godot...

Vectorization will still lead to waiting for the items to be sequentially evaluated. e.g.

```
# Wrapper function for use in *apply
wait = function(i) {
  function(x){ Sys.sleep(i) }
}

# 10 iterations * .25 seconds = 2.5 seconds elapsed
system.time({ sapply(1:10, wait(0.25)) })
```

```
##    user  system elapsed
##   0.000   0.000   2.523
```

Before an example...

- To work with parallel features in *R* you need to subscribe to either `doParallel` or `parallel` paradigm.
- Generally, `doParallel` approach is used by novices and `parallel` is used by more advanced users.

```
# Load doParallel, which will load parallel
require(doParallel, quiet = TRUE)

# How many cores do you have?
( cores = parallel::detectCores() )
```

```
## [1] 8
```

Before an example... quick parallelization.

To spawn our *workers*, we need to first create a cluster.

```
# Start a cluster with that many cores
cl = makeCluster(cores)
```

After the cluster is created, some computation normally occurs and then the cluster is stopped.

```
# Stop the cluster
stopCluster(cl)
```

>> **Always remember to stop the cluster!** <<

Parallel Example - Loop via foreach

Note the `for` in this case is `foreach`, which is a special looping function introduced by `doParallel`.

```
cl = makeCluster(3)      # Create snow Cluster
registerDoParallel(cl) # Register it with foreach
n = 10000                # Number of obs

# Use foreach in place of for to parallelize loop
system.time({
  out = foreach(i = 1:n, .combine=rbind) %dopar% {
    pf(i, df1 = 4, df2 = 5) # Note the different loop syntax
  }
})
```

```
##    user  system elapsed
##   2.643   0.266   2.973
```

Parallel Example - Vectorization

Parallelized version of the vectorized apply function

```
cl = makeCluster(3) # Create snow Cluster
n = 10000           # Number of obs

# Use parApply to parallelize apply.
system.time({
  out = parSapply(cl = cl,    # Cluster
                  X = 1:n,    # Data
                  FUN = pf,   # Function
                  df1 = 4,    # Function params
                  df2 = 5)
})
```

```
##    user  system elapsed
##    0.01    0.00    0.02
```

Parallel Example - Speed

- There is a considerable **speed** difference between the two parallelization operations.
- Part of the reason for this difference is the infrastructure setup of the `foreach` loop within the `doParallel` package.
- The `foreach` loop really is a wrapper around the `parallel` components.
  - `https://github.com/cran/doParallel/blob/master/R/doParallel.R#L469-L494`
- Hence, why more advanced users opt for pure `parallel` and beginning users typically use `doParallel`.

Parallel vs. Sequential Example - Speed

- Note that the speed of the *sequential* apply in this case was about ~8.5 times FASTER than that of the *parallel* operation.
- The **operation being performed is not computationally sufficient to warrant parallelization** due to the time associated with setting up the components.

Waiting for Godot... Redux

Using the same `wait()` function as before, we opt to see what happens under parallelization.

```
# Create snow Cluster
cl = makeCluster(4)

system.time({
  parLapply(cl = cl, X = 1:10, fun = wait(0.25))
})
```

```
##    user  system elapsed
##   0.003   0.000   0.763
```

```
# Stop the cluster
stopCluster(cl)
```

Distribution of Parallel Jobs

- Per the last example, note that the total amount of time spent was slightly over 0.75 of a second.
- The decrease in time was directly related to how the jobs were assigned.

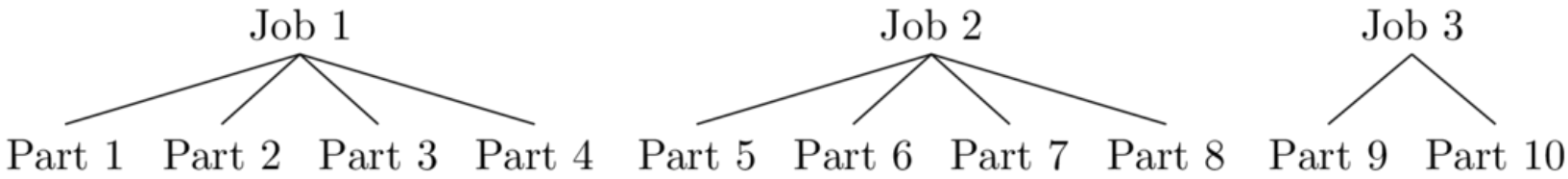


Figure 2: Job Creation with 4 Cores in Cluster

Distribution of Parallel Jobs

- What would happen if we added a 5th worker to the cluster?
  - What would be the runtime of the job?
  - How does the job structure change?

Distribution of Parallel Jobs

- Adding a 5th worker would yield only two jobs and have a total runtime of 0.5 seconds instead of 0.75.

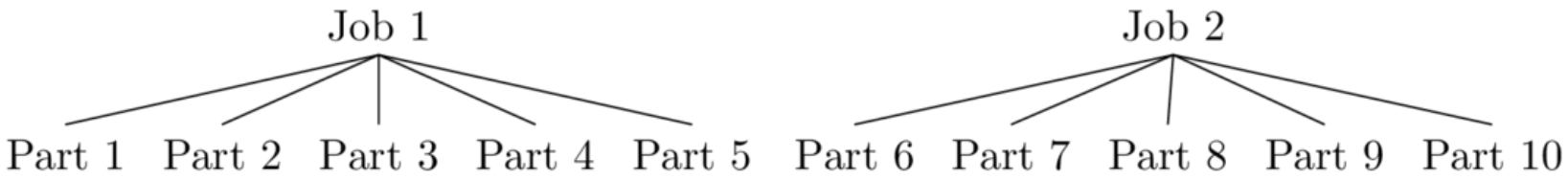


Figure 3: Job Creation with 5 Cores in Cluster

Parallelization Problem Types

- Loosely coupled (Embarrassingly Parallel)
  - There is lots of *INDEPENDENCE* between computations.
  - Very easy to parallelize
  - e.g. Tree growth step of random forest algorithm
- Tightly Coupled
  - Considerable amounts of *DEPENDENCE* exist among the computations.
  - Very hard if not impossible to parallelize.
  - e.g. Gibbs Sampler

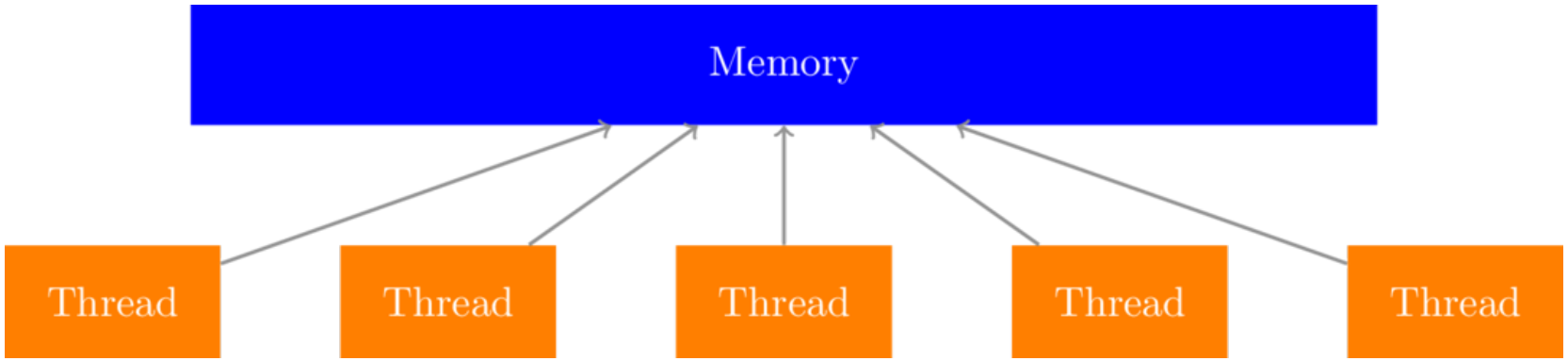
General Parallelization Ideas

There are two ways to achieve parallelization:

- Explicit Parallelism
  - The user creates and controls the parallelization
  - a.k.a The user writes code that activates the parallelization.
  - Example: OpenMP and MPI
- Implicit Parallelism
  - The system handles it.
  - a.k.a The user is clueless that parallelization is happening on their behalf!
  - Example: Arithmetic and Matrix related operations

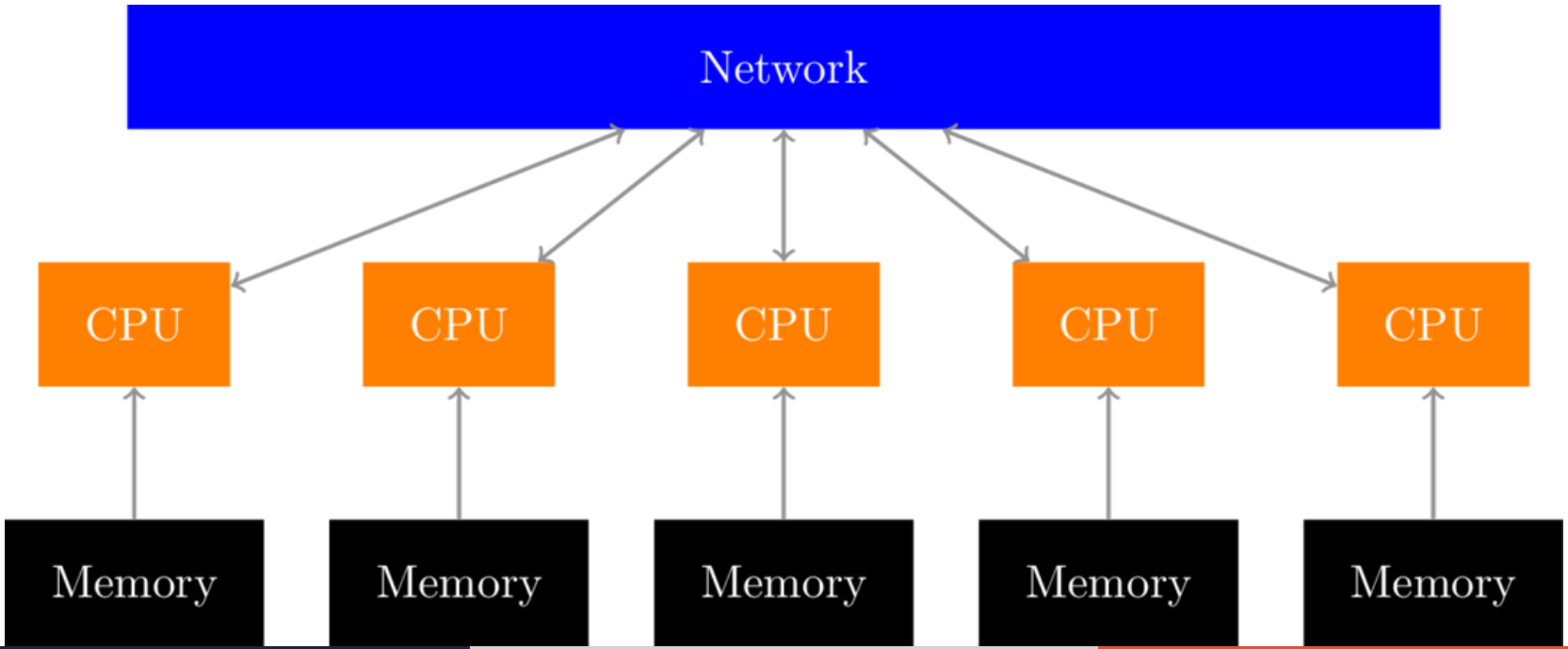
Explicit Parallelism: Shared Memory (OpenMP)

- **Shared memory** programming languages communicate by manipulating shared memory variables.
- **OpenMP** provides an implementation of this shared memory variables framework.
  - The **master** thread forks a specified number of **slave** threads, with tasks divided among threads.
  - Each thread executes parallelized sections of code independently.



Explicit Parallelism: Distributed Memory (MPI)

- **Distributed Memory** uses message-passing APIs to coordinate how a job should be completed.
- **Message Passing Interface** (MPI) uses a master-slave communication model over a network to communicate with separate machines
  - e.g the master sends instructions to the slave and the slave acts.



Comparison of Explicit Parallelism

Shared Memory	Distributed Memory
Open MP	MPI (send & receive)
Multi-core Processors	Distributed Network Required
Directive Based	Message Based
Easy to program & debug	Flexible and Expressive
Requires memory	Requires good communication

Amdahl's law (Fixed Problem Size)

Given:

- $N \in \mathbb{N}$, the number of processors being used
- $P \in [0, 1]$, the proportion of a program that can be made parallel
- The problem is of fixed size so that the total amount of work to be done in parallel is also independent of the number of processors

Then the maximum speedup obtainable via parallelization is:

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

Amdahl's law (Fixed Problem Size)

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

Note, as $N \to \infty$, then:

$$\lim_{N \to \infty} S(N) = \frac{1}{1-P}$$

- This means that the maximum speedup with infinite amounts of processors is restricted to $(1-P)$.
- Or the proportion that cannot be parallelized and, hence, is serial.

Gustafson's law (Scaled Speedup)

Given:

- $N \in \mathbb{N}$, the number of processors being used
- $\alpha \in [0, 1]$, the proportion of a program that can be made parallel
- The total amount of work to be done in parallel varies linearly with the number of processors

Then the maximum speedup obtainable via parallelization is:

$$S(N) = N - \alpha \cdot (N - 1)$$

Gustafson's law (Scaled Speedup)

- The fraction of time spent in sequential sections of the program sometimes depends on the problem size.
- That is, by scaling the problem size, you may improve the chances of speedup.
- In application, this may yield more accurate "speed up" results.

On the Agenda

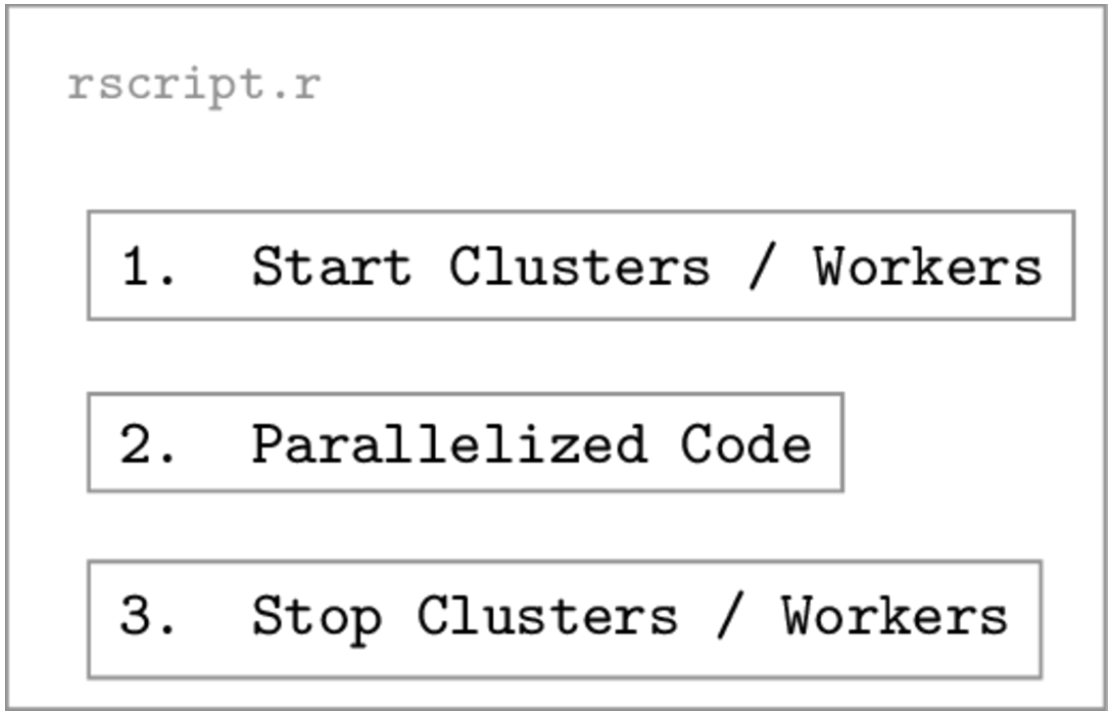
1. Parallelization
   - Motivation
   - Parallelization Lingo
   - Processing
   - Serial Processing
   - Parallel Processing
   - Job Distribution
   - Dependency Types
   - Achieving Parallelization
   - Laws of Parallelization
2. Parallelization in R
   - Explicit
   - Structure
   - Apply Functions

Explicit Parallelization in R

- As stated earlier, packages we will be focusing on for converting R code are: `doParallel` (in turn `foreach`), and `parallel`.
- These packages provide parallel scaffolding for Windows machines as well as *nix.
- Note: The `parallel` package combines both `multicore` and `snow` packages.
  - **Unfortunately, `multicore` doesn't play nicely with Windows.**

Parallelization Structure

Within the R script that is able to be parallelized, the structure that must be followed is:



Poking around at `parallel`'s `snow` backend

Important functions for initializing a `parallel` computing environment with *R*.

Function	Description
`detectCores()`	Detect the number of CPU cores
`makeCluster()`	Sets up a cluster
`stopCluster()`	Stop the cluster from running
`clusterSetRNGStream()`	Set seeds for reproducibility

parallel examples

```
# Load doParallel package
library(doParallel, quiet = TRUE)

( cores = detectCores() ) # Number of Cores
```

```
## [1] 8
```

```
cl = makeCluster(cores)    # Start Cluster

stopCluster(cl)            # Stop the cluster
```

Poking around at `parallel`'s `snow` backend - Cont.

Important functions for manipulating data with *parallel*:

Function	Description
`clusterCall()`	Calls a function each node in the cluster
`clusterApply()`	Distribute function by vectorized args
`clusterApplyLB()`	Identical to above but uses load balancing

- We're not going to worry about *load balancing*. However, I've added in these slides at the end for those interested.

Hello Parallel World

```
hello.world = function(){ print("Hello Parallel World!") }

cl = makeCluster(3)  # Create snow Cluster

# Issue function to all threads
clusterCall(cl = cl, fun = hello.world)
```

```
## [[1]]
## [1] "Hello Parallel World!"
##
## [[2]]
## [1] "Hello Parallel World!"
##
## [[3]]
## [1] "Hello Parallel World!"
```

Low Level Functions

Subtle functions that are helpful for sending data to clusters:

Function	Description
`clusterEvalQ()`	Evaluating an expression on each node
`clusterExport()`	Export variables and functions to each node.

Loading a Package on Multiple Nodes

```
cl = makeCluster(2)                    # Create cluster for snow

clusterEvalQ(cl, library(cIRT)) # Load package on all nodes
```

```
## [[1]]
## [1] "cIRT"      "stats"     "graphics"  "grDevices"
## [5] "utils"     "datasets"  "methods"   "base"
##
## [[2]]
## [1] "cIRT"      "stats"     "graphics"  "grDevices"
## [5] "utils"     "datasets"  "methods"   "base"
```

```
stopCluster(cl)                        # Stop cluster
```

Exporting Values to All Nodes

```
cl = makeCluster(2)      # Create cluster for snow

x = 1:5                  # Create variable on master

clusterExport(cl = cl, # Send variable to slaves
              varlist = c("x"))

stopCluster(cl)          # Stop cluster
```

Note: `varlist` requires a `character` vector of expression names.

Parallel *apply() functions

Within `parallel`, there are parallelized replacements of `lapply()`, `sapply()`, `apply()` and related functions.

- **`snow`:**
  - `parLapply()`
  - `parSapply()`
  - `parApply()`
  - `clusterMap` (e.g. `mapply()`)
- **`multicore`:**
  - `mclapply()`
  - `mcmapply()` (e.g. `mapply()`)

Parallel *apply functions

- `snow` also provides load balance versions of `parLapplyLB()` and `parSapplyLB()`
- There also column and row apply functions `parCapply` and `parRapply` to avoid having to remember `MARGIN = 1` or `MARGIN = 2` in `parApply()`.

Sample *apply

```
# regular apply
mat = matrix(1:100000, ncol=5)

system.time({
  apply(X = mat, MARGIN = 2, FUN = sum)
})
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

Note: There was a memory issue when it tried to mimic the parallelized version.

Sample Parallelized *apply

```
x = 1:10000000                # Data
mat = matrix(x, ncol=5) # Matrix form

cl = makeCluster(4)           # Create snow Cluster

system.time({                 # Parallel Apply
  parApply(cl = cl, X = mat,
           MARGIN = 2, FUN = sum)
})
```

```
##    user  system elapsed
##   0.268   0.044   0.373
```

```
stopCluster(cl)               # Stop cluster
```

for loops to vectorization

Embarrassly simple parallelization is akin to being able to convert from a for loop to a vectorized statement. Speed ups may vary.

```
square_x = function(x){ # Squaring function
  for(i in seq_along(x)){
    x[i] = x[i]^2
  }
  return(x)
}

x = 1:1e7                    # Data

system.time({
  out = square_x(x)
})
```

for loops to vectorization

The conversion of the prior `for` to a standard `*apply` statement would be:

```
system.time({
  y = sapply(x, function(x) x^2) # 1-line call to apply
})
```

```
##    user  system elapsed
##  11.186   0.441  11.693
```

Vectorization to Parallel Vectorization

To go from the **standard** vectorization to a **parallelized** vectorization procedure only involves slight tweaks in the code.

```
cl = makeCluster(4)

squared = function(x) { x^2 }

system.time({
  parSapply(cl = cl, X = x,
            FUN = squared)
})
```

```
##    user  system elapsed
##   6.458   1.049   9.627
```

```
# End cluster for snow
stopCluster(cl)
```

Vectorized Parallelization

The bottom line:

If a loop can be vectorized, then it is able to be parallelized.

Summary of `parallel`

- Combination of `snow` (all platforms) and `multicore` (unix only) packages
- Part of `Base R` since v2.14.0
- Small overhead vs. foreach loops (next)

Extra Material: Load Balancing

Load balancing is an optimization of how to be best distribute jobs.

- If the length p of the sequence is not greater than the number of nodes n, then a job is sent to p nodes.
- Otherwise:
  - The first n jobs are placed in order on the n nodes.
  - So, there are $p - n$ jobs that need to be assigned.
  - When the first job completes, the node receives the next job from the $p - n$ job pool.
  - This continues until all jobs are complete.
- **You gain speed but lose reproducibility this way!**