Parallelization and R

Dennis Wylie, UT Bioinformatics Consulting Group

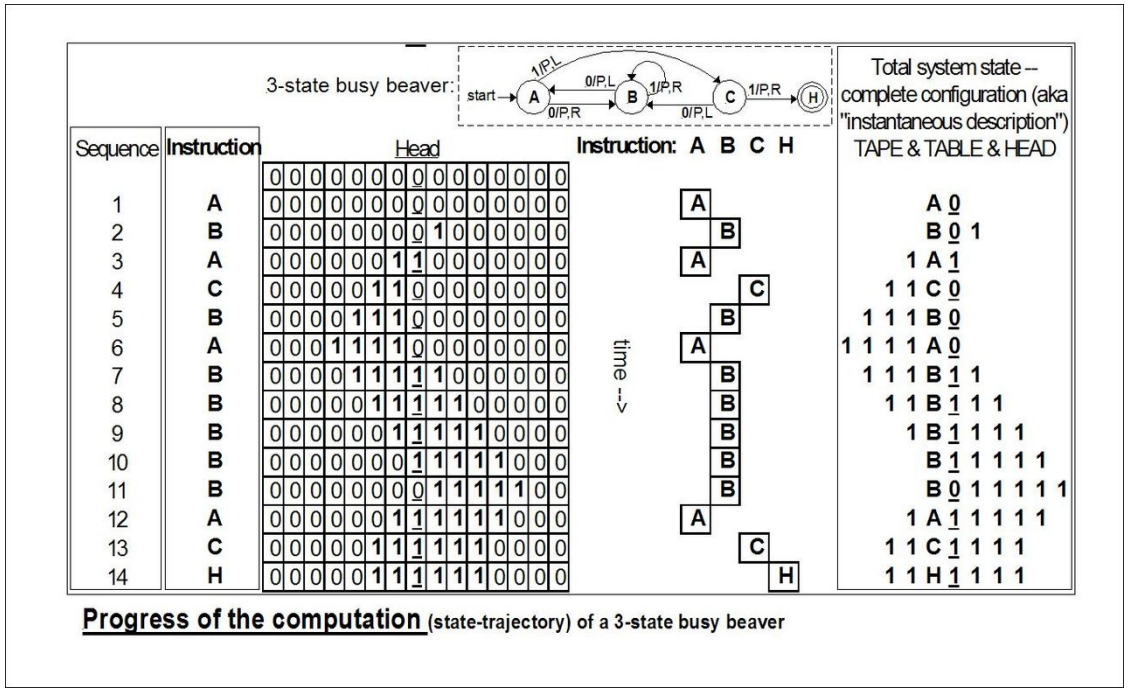
November 19, 2015

Outline

Serial Computing

What is computation?

- Many proposed definitions
- Perhaps most accepted relate to (in many ways equivalent) concepts of Turing machine and lambda calculus



- Inherent to the definition of a Turing machine is sequential operation
- Time assumed to progress in discrete, totally ordered steps

Parallel Computing

Multiple machines operating simultaneously

- For some problems, machines may work totally independently
- May share memory (i.e., same tape)
  - In practice, share some but not all memory
- Alternatively, may exchange information (message-passing)
  - Often one machine passes out work to other machines, waits for them to report back results, then aggregates for further processing

Types of Parallel Machines

- Single multicore machine
  - most modern desktops / laptops
  - easier to share memory
  - R: if linux or mac, can use `mclapply`
- Clusters of multiple machines networked together
  - harder to share memory
  - R: can use `clusterApply` family of functions
- Specialized parallel devices (ignored here)

How Much Faster?

Generally not all parts of a computation can be parallelized.

Amdahl's Law: if fraction p of the work to be done can be parallelized, the speedup with n processors is

$$\begin{aligned} S &= \frac{(1-p)+p}{(1-p)+\frac{p}{n}} \\ &= \frac{1}{1-p+\frac{p}{n}} \\ &\to \frac{1}{1-p} \text{ when } n \to \infty \end{aligned}$$

How Much Faster?

Generally not all parts of a computation can be parallelized.

Amdahl's Law: if fraction p of the work to be done can be parallelized, the speedup with n processors is

$$\begin{aligned} S &= \frac{(1-p)+p}{(1-p)+\frac{p}{n}} \\ &= \frac{1}{1-p+\frac{p}{n}} \\ &\to \frac{1}{1-p} \text{ when } n \to \infty \end{aligned}$$

But this doesn't take into account the extra overhead incurred if processors have to communicate with each other...

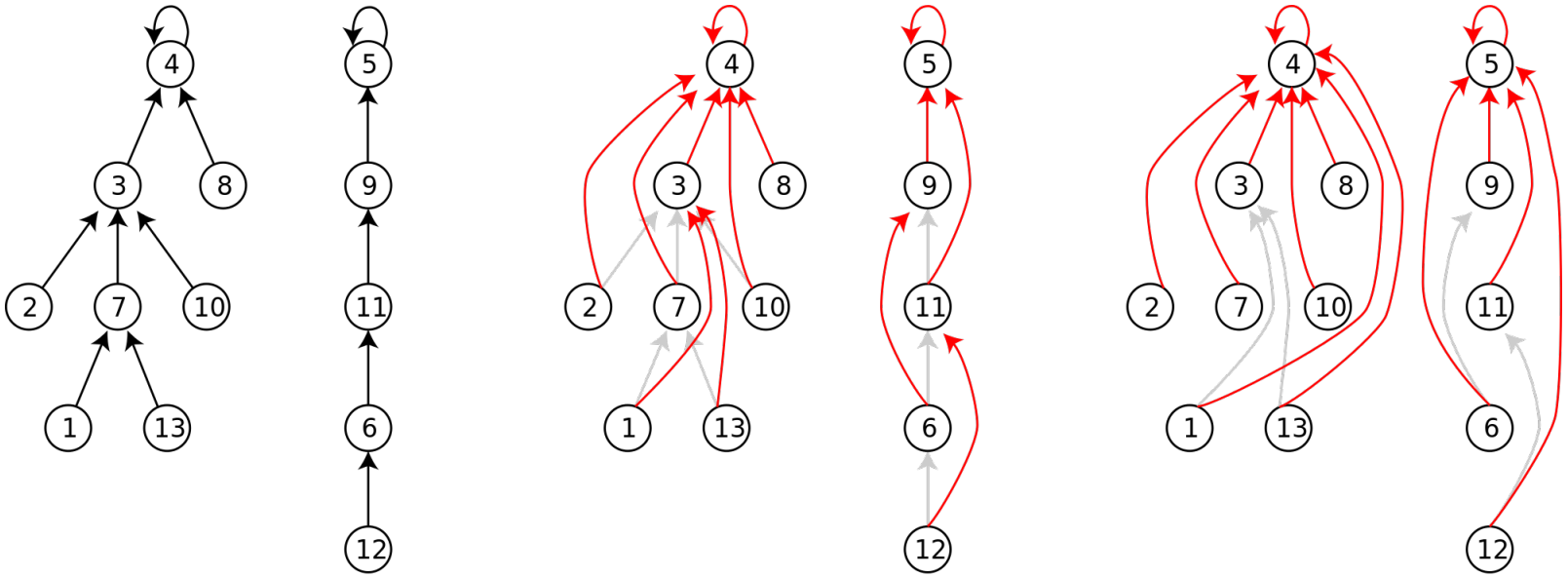
Hill & Marty (2008) provides some (relatively) recent commentary on Amdahl's law.

Techniques for Parallelization

From Blelloch & Maggs (2004):

- Divide and Conquer split problem into independent subproblems, solve, aggregate results (main approach we will discuss here)
- Randomization useful when optimal decomposition difficult to determine
  - **Sampling** e.g., to sort a list, select a few elements at random, sort these, use as interval boundaries
  - **Symmetry Breaking** to get started dividing up a problem
  - **Load Balancing** divide elements of list into random sublists instead of first n, second n, etc.
- Parallel Pointer Techniques useful for replacing traditional sequential algorithms on lists, trees, and graphs
  - **Pointer Jumping**
  - **Euler Tour** replace undirected tree with directed graph, construct linked structure representing Euler tour; good for tree traversal
  - **Graph Contraction**
  - **Ear Decomposition** partition of graph edges into ordered collection of paths; first is cycle, others "ears" with endpoints anchored on previous paths
- Others

Example: Pointer Jumping for Tree Root Finding



Initial Forest | First Iteration | Second Iteration

from https://en.wikipedia.org/wiki/Pointer_jumping

Minor Digression: Functional Programming

The lambda calculus of Alonzo Church emphasizes computation as the application of functions to map inputs into outputs.

Functional programming languages like Lisp (Sussman & Abelson (1983)) implement the ideas of the lambda calculus:

First class functions are treated just like other objects (numbers, strings, etc.); can be passed to other functions as arguments or returned as values

Immutability output value of a function depends only on the values of the arguments passed to the function; repeated function calls with same arguments always give same output.

Minor Digression: Functional Programming

The lambda calculus of Alonzo Church emphasizes computation as the application of functions to map inputs into outputs.

Functional programming languages like Lisp (Sussman & Abelson (1983)) implement the ideas of the lambda calculus:

First class functions are treated just like other objects (numbers, strings, etc.); can be passed to other functions as arguments or returned as values

Immutability output value of a function depends only on the values of the arguments passed to the function; repeated function calls with same arguments always give same output.

Immutability prevents many of the "race conditions" which can plague attempts at parallelization.

Minor Digression: Functional Programming

The lambda calculus of Alonzo Church emphasizes computation as the application of functions to map inputs into outputs.

Functional programming languages like Lisp (Sussman & Abelson (1983)) implement the ideas of the lambda calculus:

First class functions are treated just like other objects (numbers, strings, etc.); can be passed to other functions as arguments or returned as values

Immutability output value of a function depends only on the values of the arguments passed to the function; repeated function calls with same arguments always give same output.

Immutability prevents many of the "race conditions" which can plague attempts at parallelization.

R supports a functional programming style . . .

MapReduce

One of the main programming models for constructing parallel algorithms using functional programming techniques.

Map (more commonly known in `R` as `(l/m)apply`)

- Map(f, l) takes another function f and a list l with entries l_i and
- returns a new list with entries $f(l_i)$.

Reduce (known in `R` as `Reduce`)

- takes a function g and list l of length n,
- recursively defines the quantities $r_i = g(r_{i-1}, l_i)$, and
- returns the single value r_n.

MapReduce

This can be used to implement the "divide-and-conquer" approach to parallel programming:

- in the Map step, each computation $f(l_i)$ can be farmed out to a different processor.

Note that both Map and Reduce

- take functions (f and g) as arguments, and that
- immutability implies that it doesn't matter what order the processors finish their work in (since each computation depends only on f and l_i).

MapReduce

This can be used to implement the "divide-and-conquer" approach to parallel programming:

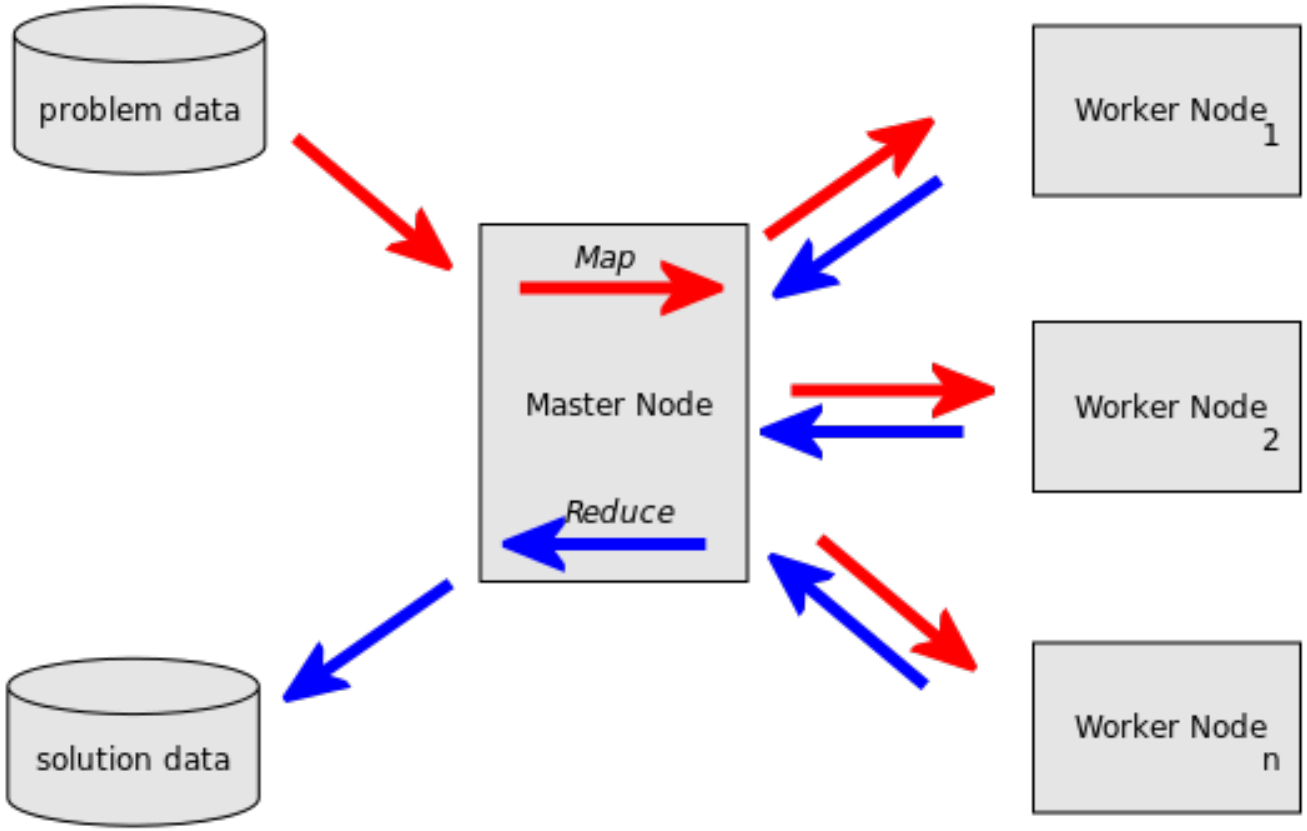
- in the Map step, each computation $f(l_i)$ can be farmed out to a different processor.

Note that both Map and Reduce

- take functions (f and g) as arguments, and that
- immutability implies that it doesn't matter what order the processors finish their work in (since each computation depends only on f and l_i).

Google has invested a lot of effort in further developing the MapReduce model (e.g., see Dean & Ghemawat (2008)) for parallel computing.

MapReduce



from https://en.wikipedia.org/wiki/File:Mapreduce_Overview.svg

Example: Matrix Multiplication

Matrix multiplication

$$w_i = \sum_{j=1}^{s} M_{ij} v_j,$$

where $i \in \{1, \ldots, r\}$, is ripe for MapReduce:

Split M into $M^{(b)}$ for $b \in \{1, \ldots, B\}$ with $B \leq r$ and each row index i assigned to exactly one block b.

Map step: for each block b (in parallel), calculate $w_i^{(b)} = \sum_{j=1}^{s} M_{ij}^{(b)} v_j$ for all i assigned to block b.

Reduce step: concatenate/interleave vectors $w^{(b)}$ into single vector w.

mclapply


```
## serial step to separate M into blocks
Mblocks = list()
for (i in 1:nPieces) {
    lower = (i-1) * rFrac + 1
    upper = ifelse(i==nPieces, r, i*rFrac)
    Mblocks[[i]] = M[lower:upper, ]
}

## parallel step to map the multiplication across blocks
wPieces = mclapply(
    X = Mblocks,
    FUN = function(Mb) {Mb %*slow% v},
    mc.cores = 3
)

## serial step to reduce the pieces into aggregated whole
wmc = Reduce(f=c, x=wPieces)
```

foreach

```
## parallel step to map the multiplication across blocks
wPieces = foreach (i=1:nPieces) %dopar% {
    ## note syntax:
    ## (1) (i=...) instead of (i in ...)
    ## (2) %dopar%
    lower = (i-1) * rFrac + 1
    upper = ifelse(i==nPieces, r, i*rFrac)
    return(M[lower:upper, ] %*slow% v)
}

## serial step to reduce the pieces into aggregated whole
wmc = Reduce(f=c, x=wPieces)
```

clusterApply


```
cl = makeCluster(3)  ## how many cores to use

## serial step to separate M into blocks
Mblocks = list()
for (i in 1:nPieces) {
    lower = (i-1) * rFrac + 1
    upper = ifelse(i==nPieces, r, i*rFrac)
    Mblocks[[i]] = M[lower:upper, ]
}

## cluster nodes needs to know about `%*slow%` and v
clusterExport(cl, "%*slow%")
clusterExport(cl, "v")

## parallel step to map the multiplication across blocks
wPieces = clusterApply(
    cl = cl,
    x = Mblocks,
    fun = function(x) {x %*slow% v}
)

## serial step to reduce the pieces into aggregated whole
wcl = Reduce(f=c, x=wPieces)
```

Multithreaded Programming

Threads are sequences of computation which can be executed concurrently but which can also share resources (memory, value of variables), in contrast with **processes**.

Multiple threads may be components of one process.

The package `Rdsm` by Norm Matloff builds on `parallel` paackage to support threaded programming for `R`:

`mgrmakevar`	allows creation of shared variables
`myinfo$id`	provides id of thread currently executing
`rdsmlock`	provides mutex lock
`barr`	provides barrier to keep threads in sync

Multithreaded Programming

Threads are sequences of computation which can be executed concurrently but which can also share resources (memory, value of variables), in contrast with **processes**.

Multiple threads may be components of one process.

The package `Rdsm` by Norm Matloff builds on `parallel` paackage to support threaded programming for `R`:

`mgrmakevar`	allows creation of shared variables
`myinfo$id`	provides id of thread currently executing
`rdsmlock`	provides mutex lock
`barr`	provides barrier to keep threads in sync

`Rdsm` (and parallel computing in "data science" in general) is treated well in Matloff (2013).

Race Conditions

When multiple threads concurrently execute while sharing mutable state variables, race conditions can occur:

```
## unreliable function
s = function(n) {
    for (i in 1:n) {
        tot[1, 1] = tot[1, 1] + 1
    }
}

c2 = makeCluster(2)
clusterExport(c2, "s")
mgrinit(c2)
mgrmakevar(c2, "tot", 1, 1)
tot[1, 1] = 0
clusterEvalQ(c2, s(1000))
tot[1, 1]  ## should be 2000, but likely far from it
```

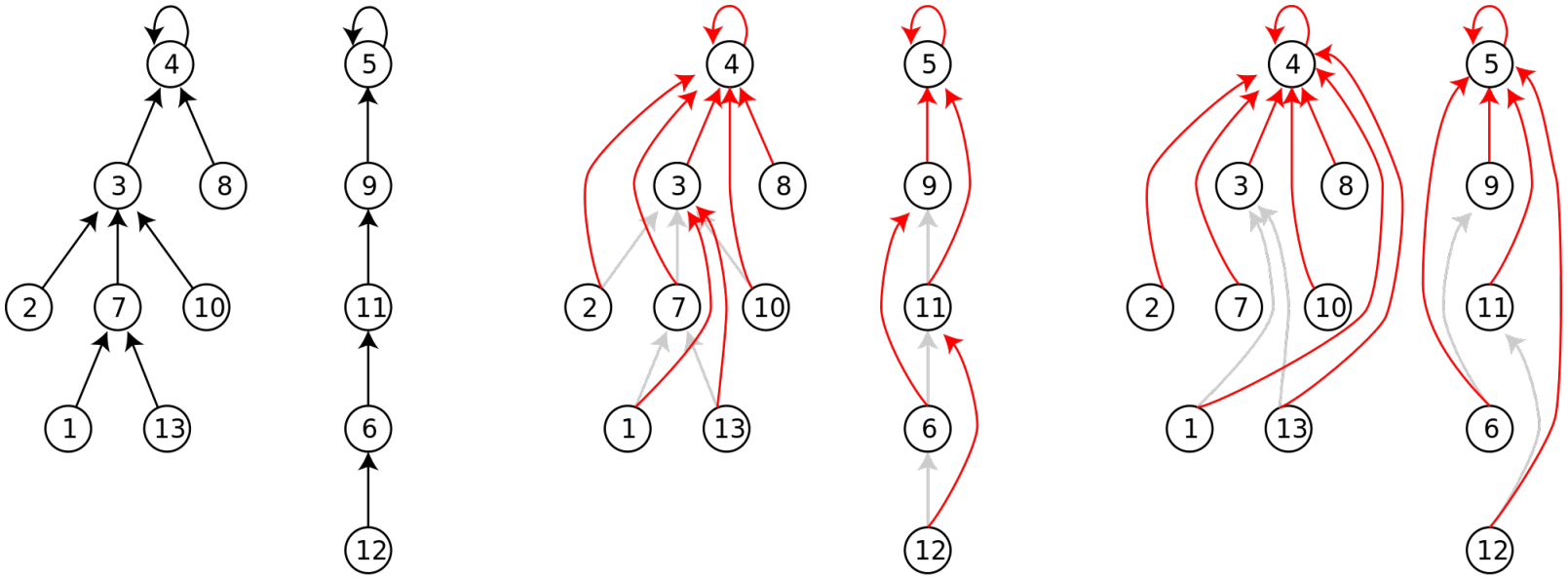
Mutual Exclusion (Mutex/Lock)


```
s1 = function(n) {
    require(Rdsm)
    for (i in 1:n) {
        rdsmlock("totlock")
        tot[1, 1] = tot[1, 1] + 1
        rdsmunlock("totlock")
    }
}

c2 = makeCluster(2)
clusterExport(c2, "s1")
mgrinit(c2)
mgrmakevar(c2, "tot", 1, 1)
tot[1, 1] = 0
mgrmakelock(c2, "totlock")
clusterEvalQ(c2, s1(1000))
```

Matrix Multiplication: Rdsm


```
slowMultThread = function(w, M, v) {
    require(parallel)

    ## determine which rows this thread will handle
    idx = splitIndices(nrow(M), myinfo$nwrkrs)[[myinfo$id]]
    w[idx, ] = M[idx, ] %*slow% v[ , ]

    return(0)  ## don't do expensive return of result
}

cl = makeCluster(3);  mgrinit(cl)

## cluster needs to define its (shared) versions of w, M, v
mgrmakevar(cl, "Mcl", nrow(M), ncol(M))
Mcl[ , ] = M
mgrmakevar(cl, "vcl", length(v), 1)
vcl[ , ] = v
mgrmakevar(cl, "wcl", nrow(M), 1)

## cluster needs to know about slowMult and friends
clusterExport(cl, c("slowMult", "%*slow%", "slowMultThread"))

## parallel step to do the multiplication
clusterEvalQ(cl, slowMultThread(wcl, Mcl, vcl))
```

Pointer Jumping Redux

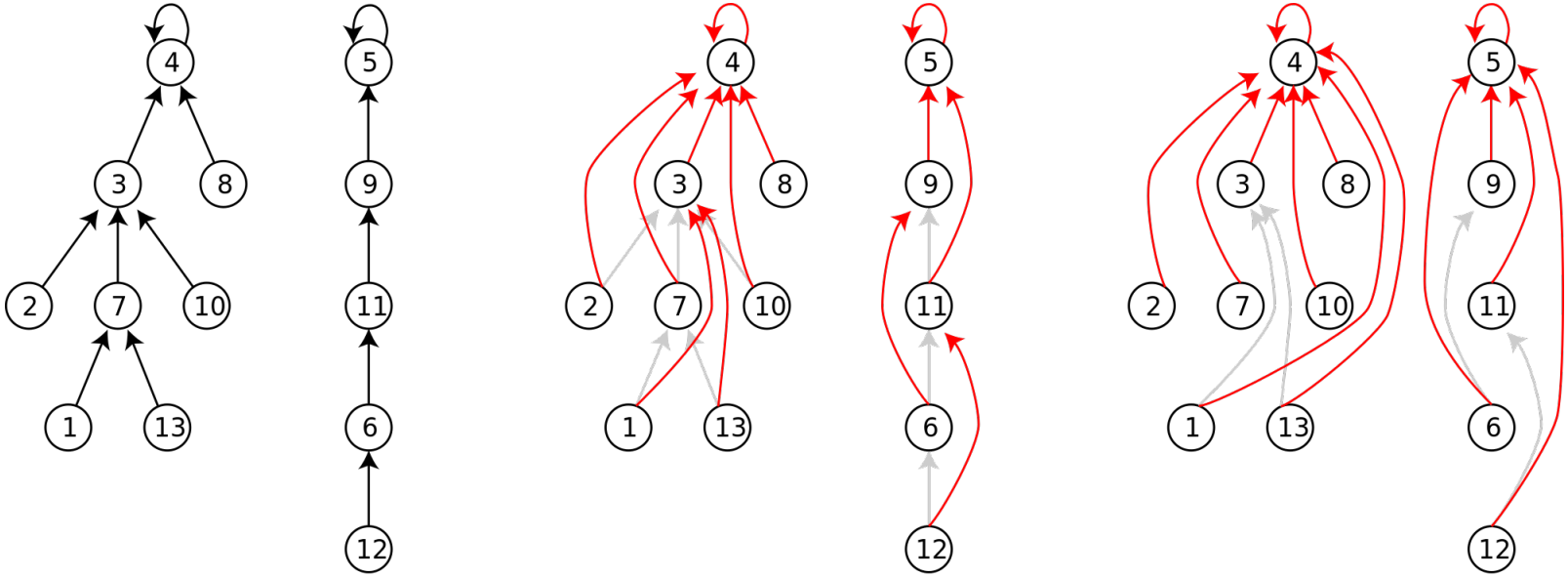


Initial Forest | First Iteration | Second Iteration

Important during pointer jumping to keep jump steps in sync.

Why? Imagine result if processor for node 11 didn't finish first iteration until after all others were done with both...

Pointer Jumping Redux

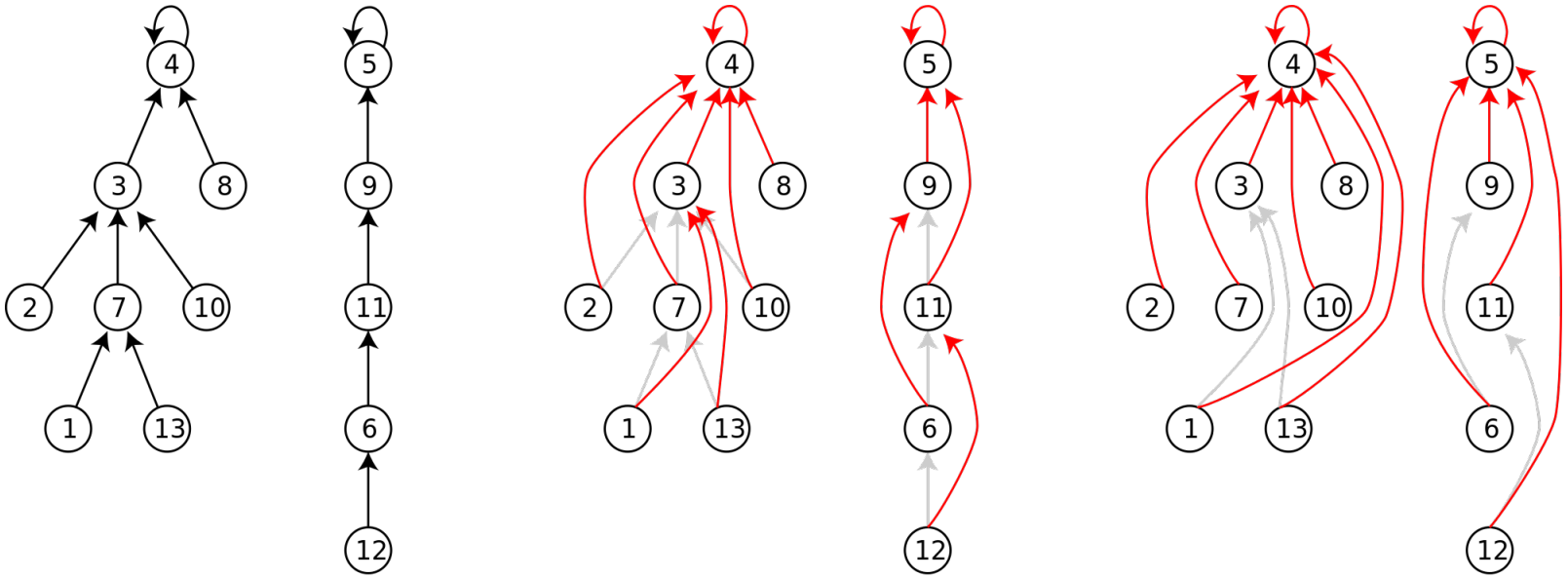


Initial Forest | First Iteration | Second Iteration

Important during pointer jumping to keep jump steps in sync.

Why? Imagine result if processor for node 11 didn't finish first iteration until after all others were done with both. . . then 12 would end up pointing to 9, not to 5.

Pointer Jumping Redux



Initial Forest | First Iteration | Second Iteration

Important during pointer jumping to keep jump steps in sync.

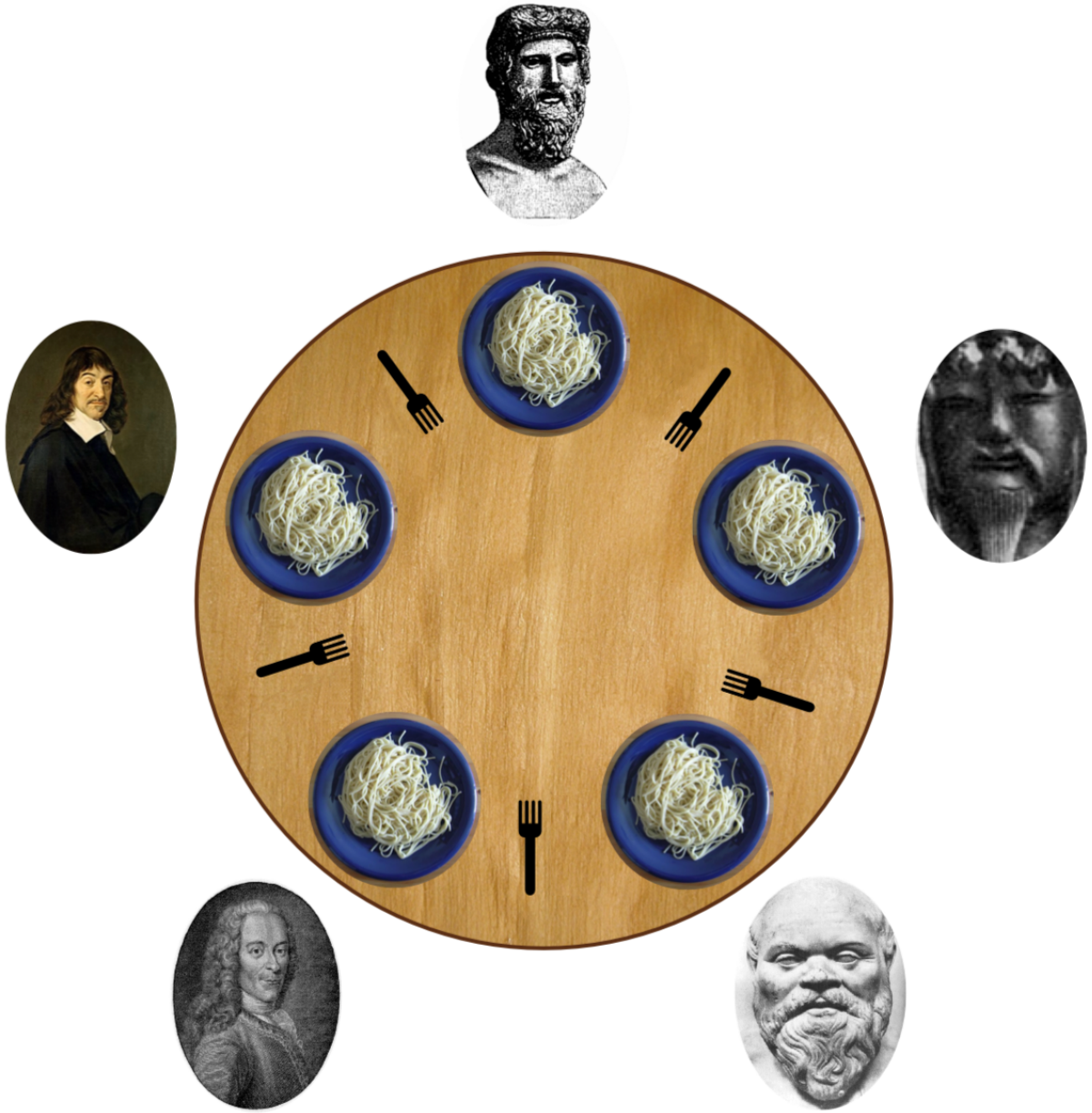
Why? Imagine result if processor for node 11 didn't finish first iteration until after all others were done with both. . . then 12 would end up pointing to 9, not to 5.

One method for enforcing synchronization is to use **barrier**.

Pointer Jumping and Barriers


```
iterParBar = function(nwpntcl, pntcl, ncl) {
    require(parallel)

    ## determine which rows this thread will handle
    idx = splitIndices(length(pntcl), myinfo$nwrkrs)[[myinfo$id]]

    for (i in 1:ncl[1, 1]) {
        ## do the jumping, store results in new variable
        nwpntcl[idx] = pntcl[ pntcl[idx] ]

        ## wait for all threads to finish jumping
        barr()

        ## reset old pointer to new, jumped values
        pntcl[idx] = nwpntcl[idx]

        ## wait for all threads to finish resetting
        barr()
    }

    ## don't do expensive return of result
    return(0)
}
```

Dining Philosophers



R on TACC: Interactive

TACC provides a bunch of multicore machines clustered together ready for you to use these techniques on...

As for how to do it, you've got a few options:

- Interactive Session:
  - Use RStudio on the TACC Visualization Portal (`https://vis.tacc.utexas.edu/`)
    - if you have your own version of `R` installed on TACC, this may not work!
  - `ssh` in, start and `idev` session and run `R` on the cores you've got available there

R on TACC: Batch Processing

TACC provides a bunch of multicore machines clustered together ready for you to use these techniques on...

As for how to do it, you've got a few options:

- Batch Processing (using `slurm`):
  - Write:
    - `R` script file(s) you want to run
    - job file containing calls to `Rscript`
  - Generate `.slurm` file (e.g., using `launcher_creator.py` from BioITeam bin)
  - Submit job: `sbatch <job>.slurm`

References I

Blelloch, Guy E, & Maggs, Bruce M. 2004. *Parallel Algorithms*.

Dean, Jeffrey, & Ghemawat, Sanjay. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, **51**(1), 107–113.

Hill, Mark D, & Marty, Michael R. 2008. Amdahl's Law in the Multicore Era. *Computer*, 33–38.

Matloff, Norman. 2013. *Parallel Computing for Data Science*. University of California, Davis.

Sussman, Gerry, & Abelson, Harold. 1983. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass.