# Parallel Concepts

**Feng Li**
feng.li@cufe.edu.cn

**School of Statistics and Mathematics**
**Central University of Finance and Economics**

April 7, 2014

Revision: April 7, 2014

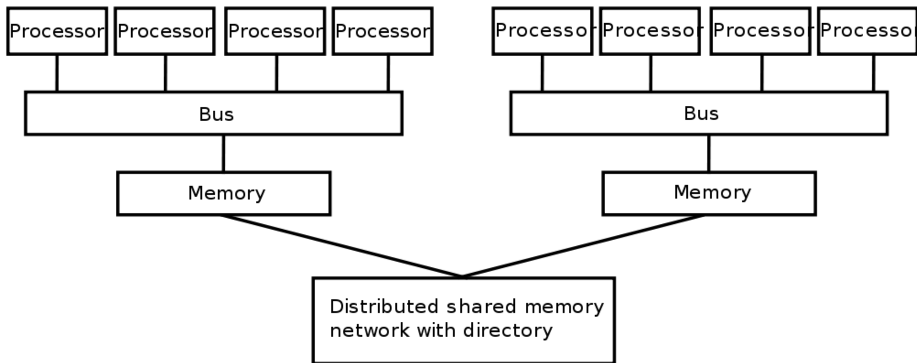
# Today we are going to learn...

1. **The basic concepts in parallelism**
2. **Your first (pseudo) parallel code**
3. **The R parallelism**

# The basic concepts

- **Parallel computing** is a form of computation in which many calculations are carried out simultaneously.
- Parallel computers can be roughly classified according to the level at which the hardware supports parallelism.
- With **multi-core** and **multi-processor** computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task.
- **Shared memory**: shared between all processing elements in a single address space.
- **Distributed memory**: in which each processing element has its own local address space.
- Main memory in a parallel computer is either shared memory, or distributed memory.
- Accesses to local memory are typically faster than accesses to non-local memory.

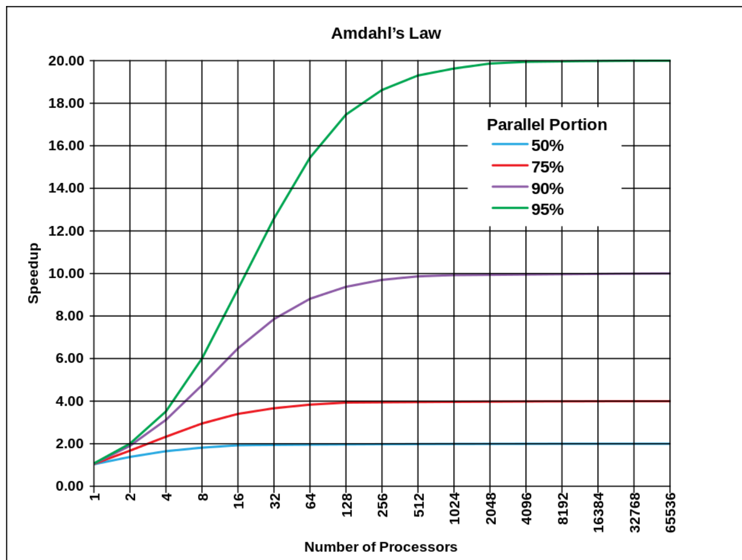
IBM
Blue Gene/P


Processor
Processor
Processor
Processor
Processor
Processor
Processor
Processor
Bus
Bus
Memory
Memory
Distributed shared memory
network with directory

# The landscape of parallelism

- The landscape of parallel computing has changed with the advent of shared-memory computers with multiple (and often many) CPU cores.
- Until the late 2000's parallel computing was mainly done on clusters of large numbers of single- or dual-CPU computers: nowadays even laptops have two or four cores, and servers with 8, 32 or more cores are commonplace.
- It is such hardware that package parallel is designed to exploit. It can also be used with several computers running the same version of R connected by (reasonable-speed) ethernet: the computers need not be running the same OS.

# Can we speed up as much as we like with enough computers? ↛ Amdahl's Law



## The way we code

- The sequential coding structure: The code is executed step by step.
- The parallel coding structure: Same tasks are executed simultaneously.
- Parallel computer programs are more difficult to write than sequential ones.
  - More difficult to find bugs.
  - Usual sequential code could not be directly parallelized.
  - This is even more difficult in statistical computing.

# Matrix multiplications

- Given two matrices $A_{m\times n}$ and $B_{n\times p}$. Carry out the calculation $AB$
- How do you split the task?
- What do you need to consider when you write the code?
- What is the best way to split, split as many as possible?

# Matrix inverse

- If you are given a diagonal matrix. That is easy.
- But, for a general squared matrix.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \\ -(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & (\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \end{bmatrix}$$

# Explicit parallelism and implicit parallelism

- **Explicit parallelism** is the representation of concurrent computations by means of primitives in the form of special-purpose directives or function calls.
- **Implicit parallelism** is a characteristic of a programming language that allows a compiler or interpreter to automatically exploit the parallelism inherent to the computations expressed by some of the language's constructs. A pure implicitly parallel language does not need special directives, operators or functions to enable parallel execution.
  - Example: The vectorization in R.
- Pros and cons
  - I/O
  - How much can you extend it?
  - How much effort do you need code it?

# Two more concepts

- **sockets**: communication between computers.
- In computing, particularly in the context of the Unix operating system and its workalikes, **fork** is an operation whereby a process creates a copy of itself

# How R parallels
## ↛ The basic computer model

**(a)** Start up M 'worker' processes, and do any initialization needed on the workers.

**(b)** Send any data required for each task to the workers.

**(c)** Split the task into M roughly equally-sized chunks, and send the chunks (including the R code needed) to the workers.

**(d)** Wait for all the workers to complete their tasks, and ask them for their results.

**(e)** Repeat steps (b-d) for any further tasks.

**(f)** Shut down the worker processes.

# Parallelism with R

- Default package `parallel`
  - `detectCores()`: Attempt to detect the number of CPU cores on the current host.
  - `makeCluster()`: Creates a set of copies of R running in parallel and communicating over sockets.
  - `stopCluster()`: It is good practice to shut down the workers when the tasks are done.
- Ways of parcallelism
  - `mclapply()`: mclapply is a parallelized version of lapply, it returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X. It relies on forking and hence is not available on Windows unless mc.cores = 1.
  - `mcmapply()`: is a parallelized version of mapply

# Apply Operations using Clusters

- `parLapply(cl = NULL, X, fun, ...)`
- `parSapply(cl = NULL, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`
- `parApply(cl = NULL, X, MARGIN, FUN, ...)`
- `parRapply(cl = NULL, x, FUN, ...)`
- `parCapply(cl = NULL, x, FUN, ...)`
- `parLapplyLB(cl = NULL, X, fun, ...)`
- `parSapplyLB(cl = NULL, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`

# Lab 1

Evaluating a double integral:

$$\int_{-1}^{2}\int_{-1}^{2}(x^3 - 3x + y^3 - 3y)dxdy$$

- serial implementation with a double loop
- vectorized implementation without loops
- implementation with `apply()`
- parallel implementation with `parLapply()`

Submit your code online at course homepage
`http://feng.li/teaching/stat-software/`.

# Serial Loop

```
integLoop <- function(func, xint, yint, n)
{
    local_sum <- 0
    xincr <- ( xint[2]-xint[1] ) / n
    yincr <- ( yint[2]-yint[1] ) / n
    for(xi in seq(xint[1], xint[2],length.out = n)){
        for(yi in seq(yint[1], yint[2],length.out = n)){
            box <- func(xi, yi) * xincr * yincr
            local_sum <- local_sum + box
        }
    }
    return(local_sum)
}
```

# Serial Vectorization

```
integVec <- function(func, xint, yint, n)
{
    xincr <- ( xint[2]-xint[1] ) / n
    yincr <- ( yint[2]-yint[1] ) / n
    local_sum <- sum(
        func( seq(xint[1], xint[2], length.out = n),
              seq(yint[1], yint[2], length.out = n) )
        ) * xincr * yincr * n
    return(local_sum)
}
```

# Serial Apply

```
integApply <- function (func, xint, yint, n)
{
    applyfunc <- function(xrange, xint, yint, n, func)
        {
            yrange <- seq(yint[1], yint[2], length.out = n)
            xincr <- ( xint[2]-xint[1] ) / n
            yincr <- ( yint[2]-yint[1] ) / n
            local_sum <- sum( sapply(xrange, function(x)
                                     sum( func(x, yrange)
                                         )) ) * xincr * yincr
            return(local_sum)
        }
    xrange <- seq(xint[1], xint[2], length.out = n)
    local_sum <- sapply(xrange, applyfunc, xint, yint, n, func)
    return( sum(local_sum) )
}
```

# Serial Apply

## ↛ A much simplified version: Thanks Shan

```
integApply <- function (func, xint, yint, n)
{
    xrange <- seq(xint[1], xint[2], length.out = n)
    yrange <- seq(yint[1], yint[2], length.out = n)
    xincr <- ( xint[2]-xint[1] ) / n
    yincr <- ( yint[2]-yint[1] ) / n
    local_sum <- sum( sapply(xrange,
                             function(x) sum(func(x, yrange))
                             ) ) * xincr * yincr
    return(local_sum)
}
```

# Parallel Apply

```
integClusterApplyLB <- function(cluster, func, xint, yint, n)
{
    nworkers <- length(cluster)
    local_sum <- clusterApplyLB(cluster, 1:nworkers,
                                workerfunc, nworkers,
                                xint, yint, n, func)
    return( sum(unlist(local_sum)) )
}
```

# Speed comparison

```
Serial runs
-------------------------
Type   n= 1000   n= 10000
-------------------------
Loop   6.44      678.23
Vec    0.001     0.003
Apply  0.269     16.90
-------------------------

Parallel runs, n= 10000
-------------------------
Processes Time
-------------------------
1         16.90
2          7.16
4          3.61
8          2.63
16         1.39
-------------------------
```