# APSTA-GE 2352

Statistical Computing: Lecture 12

Klint Kanopka

New York University

# Table of Contents

# Announcements

- PS6 is posted
  - I think there is one tricky coding part
  - The rest is really interpretation—use historical and geographical resources to help you!
- Lab next week is still cancelled
- After today:
  - 2 more lectures
  - 2 more labs
  - 1 final exam
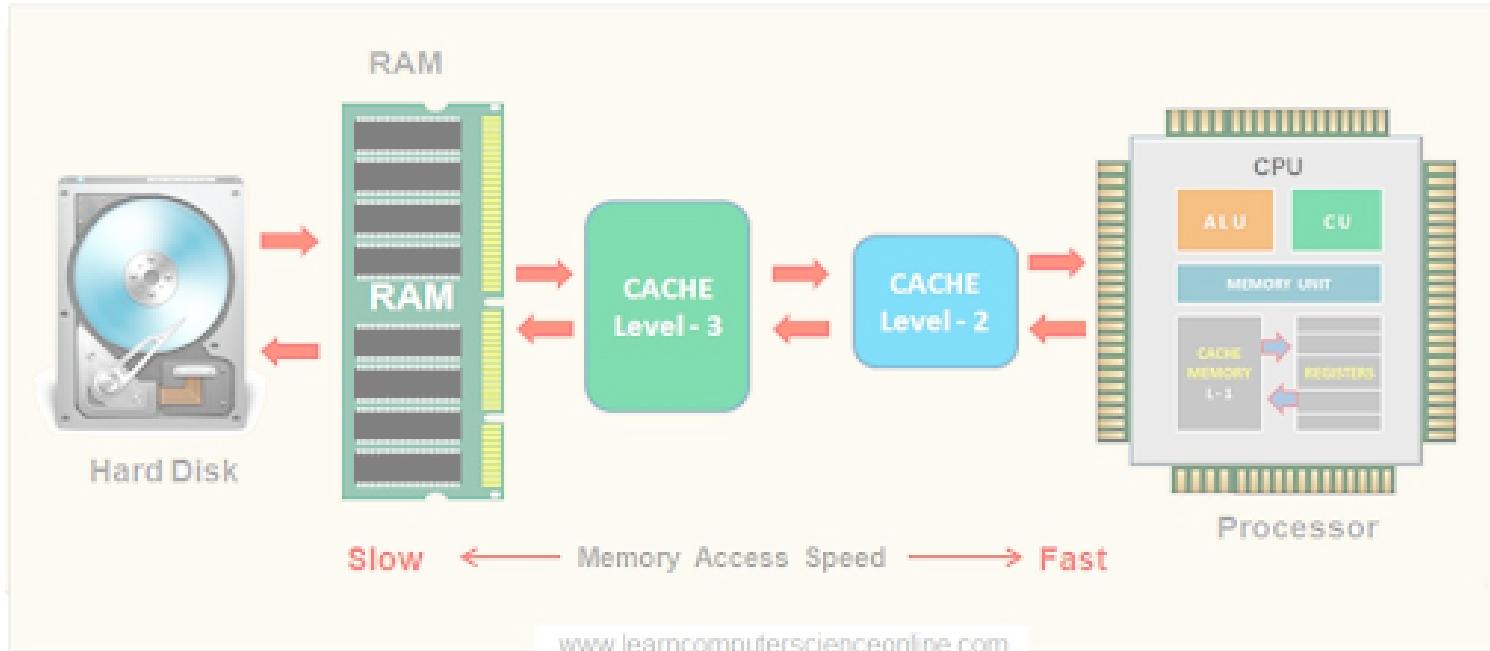
# Course-related Announcements

- You should be registered for my Generalized Linear Models and Extensions course (GLMs)
  - You will learn a lot about GLMs
    - Treatment is practical and mathematical
    - Course has a lot less programming (but it's still me, so you know)
  - What are the extensions?
    - Multilevel Models (MLMs)
    - Generalized Additive Models (GAMs)
- You might be interested in Modern Approaches in Measurement
  - Unsupervised machine learning approaches to latent variable measurement
  - We deal with continuous and categorical latent variables
  - Real data: item response/survey data, behavioral data, and text
  - People generally like it!
  - It's also generally a pre-req for summer research opportunities with me
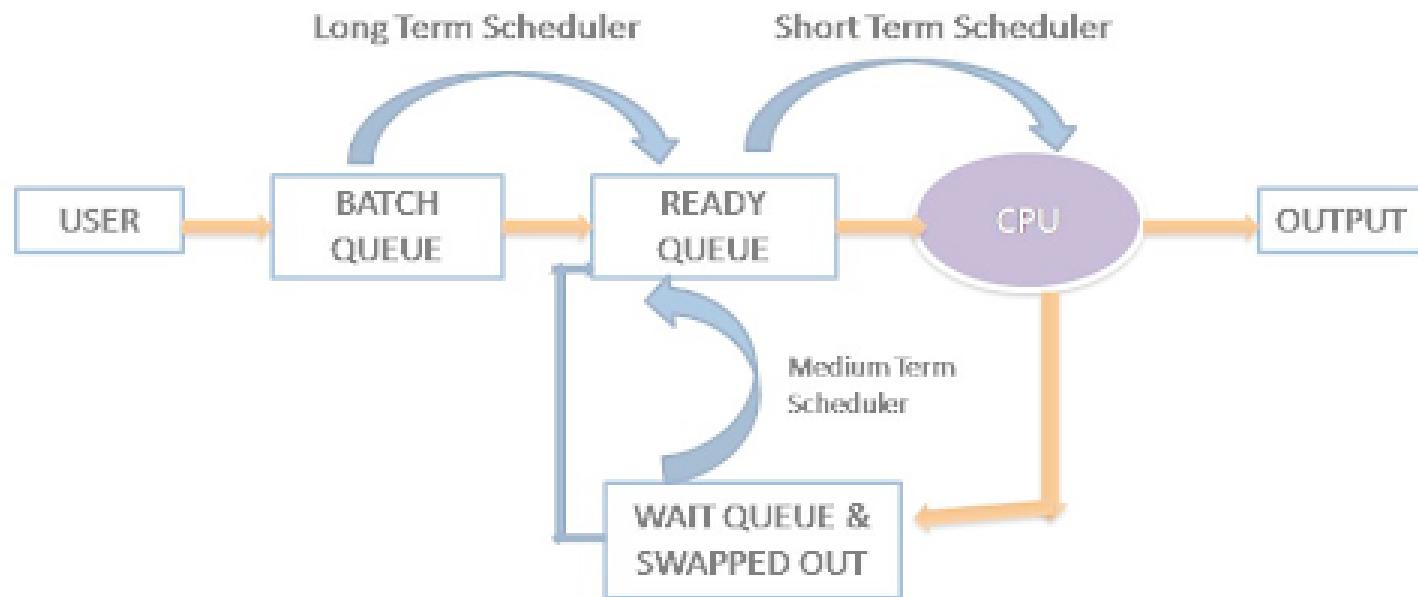
# Computer Architecture

# Memory



Computer System Memory Hierarchy
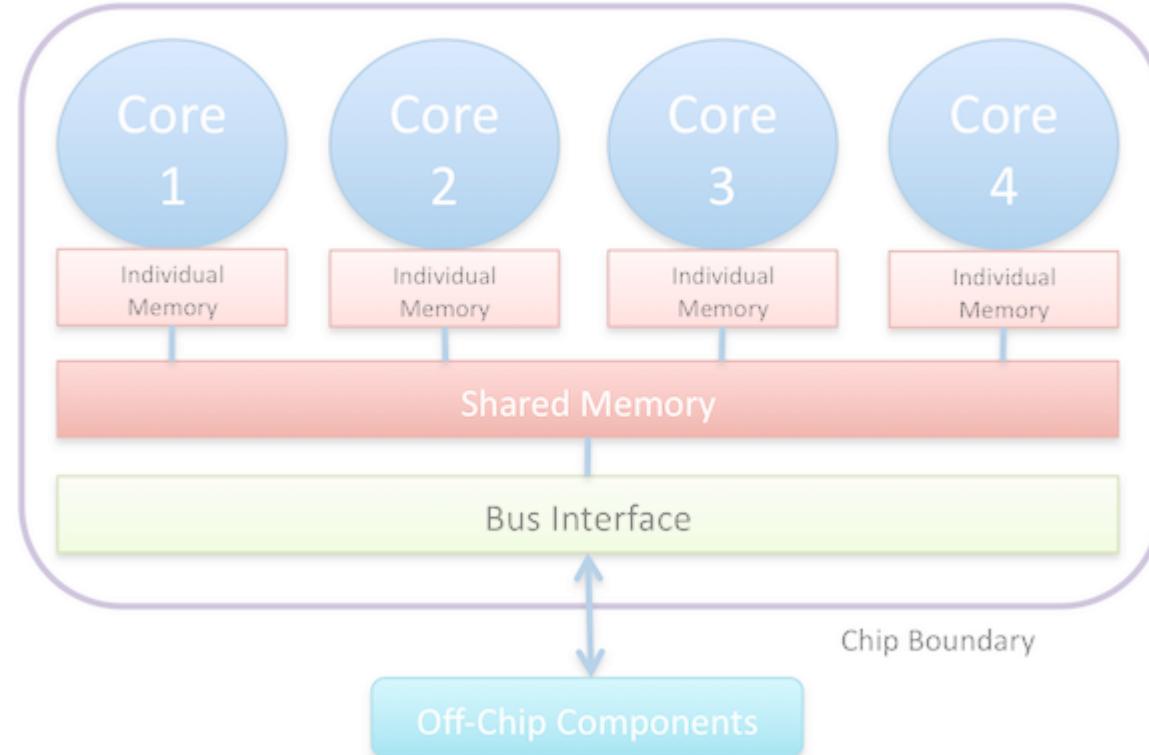
# Scheduling

# Multi-core Processing



Multi-core Processor

Core 1 · Core 2 · Core 3 · Core 4

Individual Memory · Individual Memory · Individual Memory · Individual Memory

Shared Memory

Bus Interface

Chip Boundary

Off-Chip Components

# CPUs vs GPUs

# Parallel Computing

# Parallelization

- Some computations and tasks depend on previous results
  - We call these sequential computations
  - MCMC is a great example of this!
- Some computations don't!
  - Bootstrap replications are a great example of this!
  - If they're really easy to separate, we call them *embarrassingly parallel*
- Some computations can be parallelized (or partially parallelized) with some work
  - Map-Reduce is a great example of this!

# What is Parallel Computing?

- Parallel computing refers to engaging multiple compute cores in a task simultaneously
- Computers with multiple cores and multiple processors can do multiple things at once
- Modern computers often have high core counts (my Macbook Air has eight, my desktop at home has 24!)
- Graphical processing units (GPUs) are purpose built for parallel matrix multiplication operations
- Parallel computing engages more resources in a task, so computation happens simultaneously
- Parallel computing also allocates memory resources to tasks, so there are fewer costs with moving things around and accessing slower memory

# What about Vectorization?

- Vectorized code takes advantage of *implicit parallelization*
- An example of single instruction, multiple data (SIMD) computation
- At the compiler level, the computer knows how to interact with the scheduler to divide up this work across multiple cores and/or threads
- This is why vectorized computation is so much faster: It's parallel by default!

# Implementation in `R`

- We'll use the `parallel` package
  - Built into `R` and a combination of two old packages: `snow` and `multicore`
  - The core idea is this: if you can modify your code to use an `apply()` function to solve your problem, you can parallelize for almost no additional cost
- Other options for parallelization in `R` exist
  - The `doParallel` and `foreach` package work together in ways I find to be really silly
  - The `future` package is another option for parallelization

# Workflows

## Option 1: `mc` versions of `apply()` functions

- Replace your `apply()` function with an `mc` prefixed version:

    - `lapply()` becomes `mclapply()`

    - `mapply()` becomes `mcmapply()`

- Specify the number of cores you want to use in the `mc.cores =` argument

- Note that this *forks* processes and can cause huge problems in any GUI-based programs

- May also behave badly on some operating systems

# Workflows

## Option 2: Create a socket cluster

- Create a local cluster using `makeCluster()`

- Export data and functions to the cluster using `clusterExport()`

- Replace your `apply()` function with a parallelized version

  - `parApply()`, `parLapply()`, `parSapply()` are drop-in replacements

  - I usually use `clusterMap()`, which is like `mapply()` or `Map()`

- Stop the cluster with `stopCluster()` (This is important!)

- This may behave badly on other operating systems

# Load-Balancing

- There are also load-balanced versions like `parLapplyLB()` and `clusterMap()` has a dynamic scheduling option
- Normal versions use *static* scheduling, where work is divided among cores before starting
- Load-balanced versions use *dynamic* scheduling, assigning tasks to cores as other tasks finish
- In most small to medium sized tasks, normal versions are more efficient
- Load-balancing is most effective when individual jobs have wildly different runtimes

# Wrap Up

# Recap

- If your code is slow, try to parallelize it
- If you wrote your code well to begin with, it'll be easy to do!