Parallel Computing in R using the snowfall Package

Timothy Hayes

University of Southern California

Spring 2014

Structure of This Talk

1. Overview of Parallel Computing and The snowfall Package in These Slides

2. Demonstration of R Code
   A. Basic Examples
      - Simple example code designed to introduce the programming techniques.

   B. Advanced Examples
      - Code from real (and realistic) examples of how you might *actually* use the techniques.
        - Two examples of how to parallelize simulation studies.
        - An example of how snowfall interfaces with OpenMx.

Philosophy of Materials Included in this Presentation

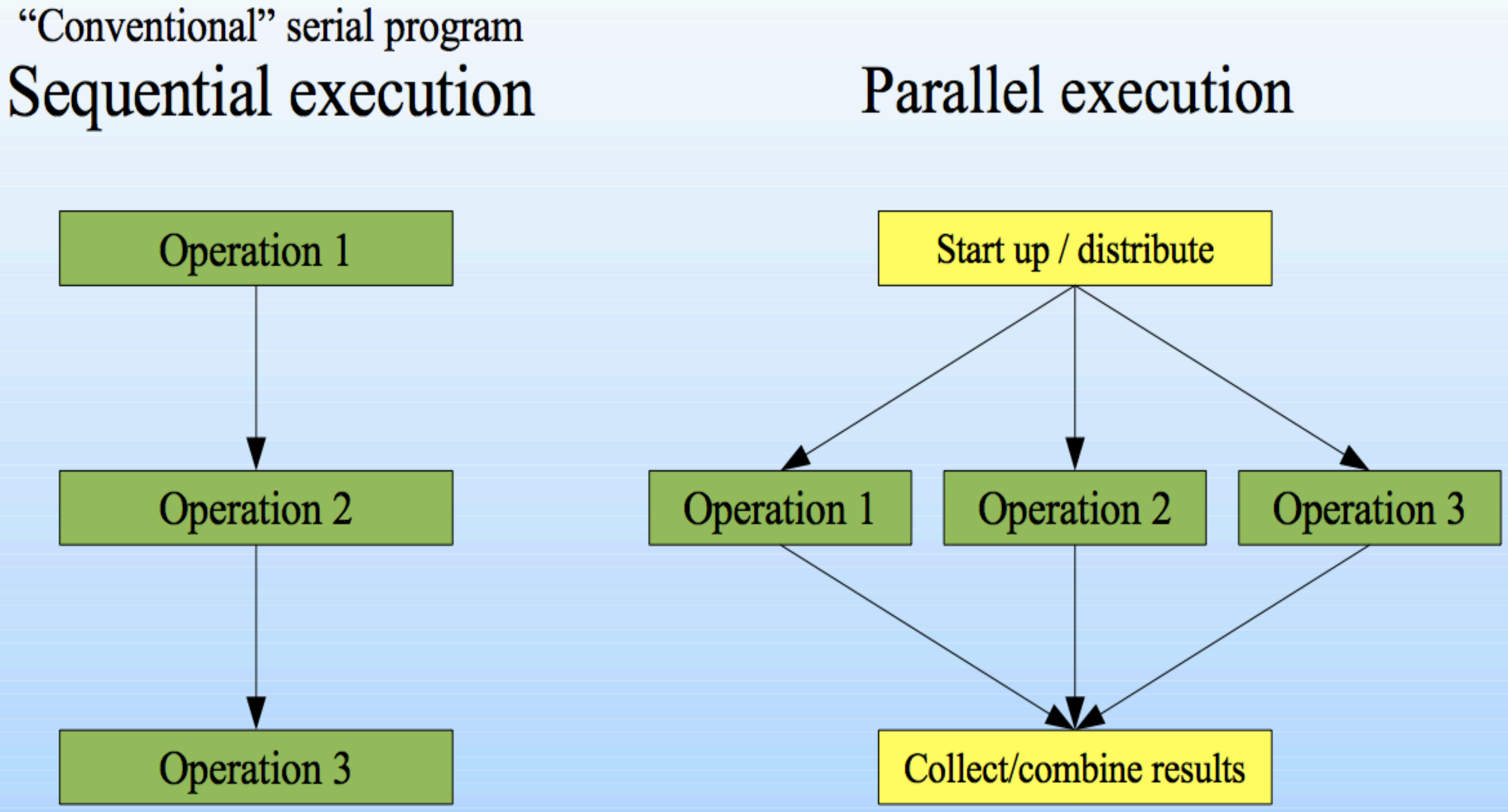
- These materials (including this slide show) are designed to be ***your future references***.

- Thus, they are a bit more text heavy than "good presentation etiquette" might dictate … but hopefully they will be more useful later.

Obligatory Introductory Section:

THE PROMISE OF PARALLEL COMPUTING

Parallel programs



From Knaus, 2008

Amdahl's Law: Theoretical maximum speedup with multiple processors

$n \in N$	the number of CPU cores
$B \in [0,\ 1]$	the serial portion of the algorithm
$T(n)$	the time to solution

$$T(n) = T(1)\left(B + \underbrace{\frac{1}{n}(1 - B)}\right)$$

Total time to solution if executed in parallel.

Total time to solution when executed serially.

This rescales proportions in units of time.

Proportion of algorithm that is **serial**.

Proportion of algorithm that is **not serial** divided by the number of cores.

http://en.wikipedia.org/wiki/Amdahl%27s_law

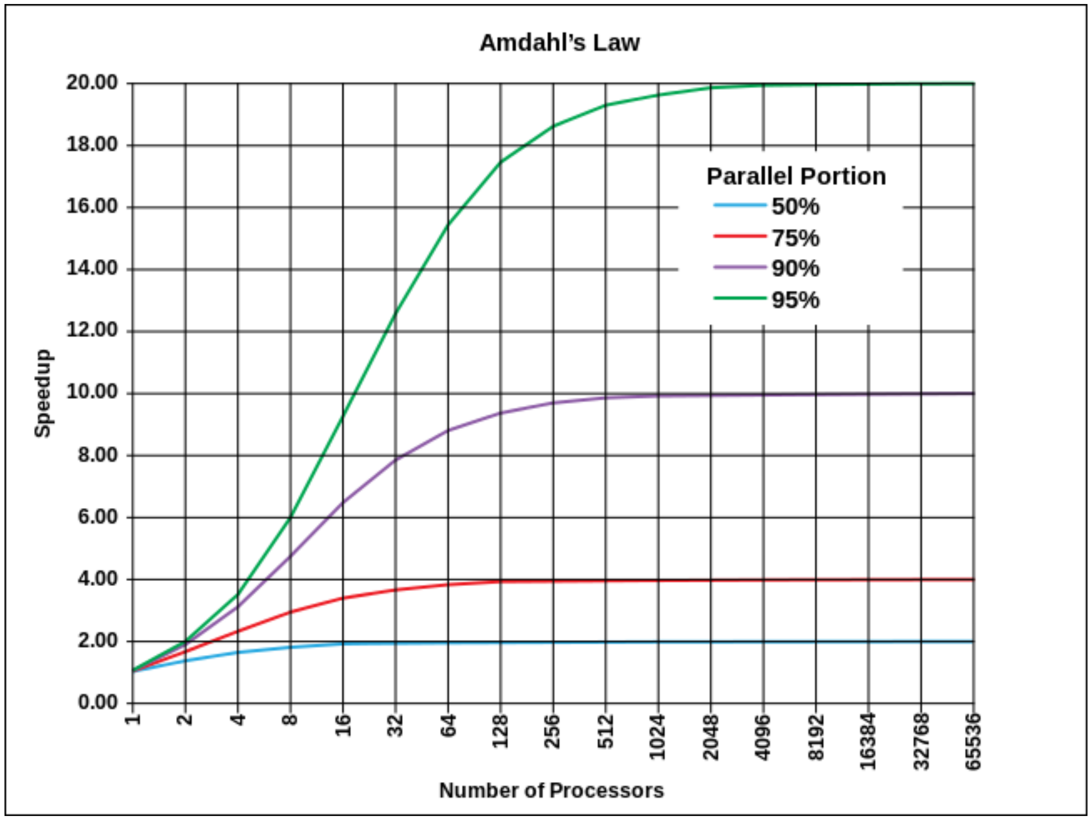
Amdahl's Law: Theoretical maximum speedup with multiple processors

Theoretical speedup gained by executing algorithm on a system capable of executing *n* threads in parallel:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)}$$

Serial execution time (without parallel) divided by execution time with parallel

http://en.wikipedia.org/wiki/Amdahl%27s_law



"The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used."

http://en.wikipedia.org/wiki/Amdahl%27s_law

How Much Speedup Can You Get From Parallel Programming with the HPC?

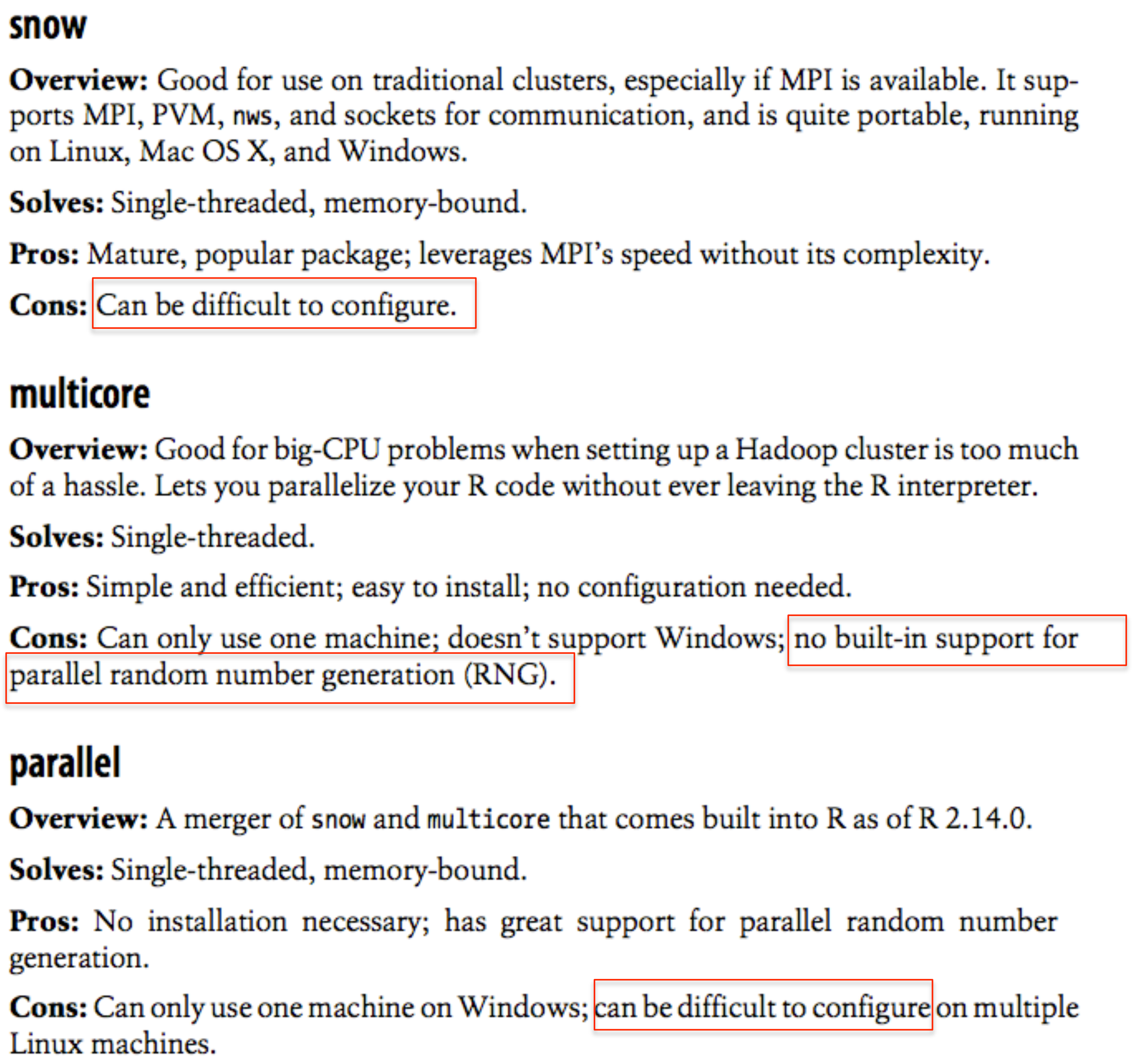
- Many HPCC nodes have as many as 16 cores.
- From experience, you can comfortably request ~6 nodes at a time.
- If you request 6 nodes with 16 processors per node, that is 6*16 = **96 operations running in parallel at any one time … <u>if you write your script in parallel.</u>**

Section 2

PARALLEL COMPUTING USING R

Parallel Computing in R

- Several packages are currently available to facilitate parallel computing in R:
  - `Multicore` (Urbanek, 2009)
  - `Parallel` (R-Core/Eddelbuettel, 2014)
  - `Snow` (<u>S</u>imple <u>N</u>etwork <u>o</u>f <u>W</u>orkstations in R; Tierney, Rossini, Sevcikova, 2003)



snow

Overview: Good for use on traditional clusters, especially if MPI is available. It supports MPI, PVM, `nws`, and sockets for communication, and is quite portable, running on Linux, Mac OS X, and Windows.

Solves: Single-threaded, memory-bound.

Pros: Mature, popular package; leverages MPI's speed without its complexity.

Cons: Can be difficult to configure.

multicore

Overview: Good for big-CPU problems when setting up a Hadoop cluster is too much of a hassle. Lets you parallelize your R code without ever leaving the R interpreter.

Solves: Single-threaded.

Pros: Simple and efficient; easy to install; no configuration needed.

Cons: Can only use one machine; doesn't support Windows; no built-in support for parallel random number generation (RNG).

parallel

Overview: A merger of `snow` and `multicore` that comes built into R as of R 2.14.0.

Solves: Single-threaded, memory-bound.

Pros: No installation necessary; has great support for parallel random number generation.

Cons: Can only use one machine on Windows; can be difficult to configure on multiple Linux machines.

McCallum & Weston, 2012

Snowfall (Knaus, 2010)

- Usability wrapper for the snow package.
- Interfaces with OpenMx.
- Relies on similar programming conventions to virtually all other parallel R packages, so easy to learn other packages after learning.
- Relatively thorough/helpful documentation available at:
  - https://www.imbi.uni-freiburg.de/parallel/

Some Highly Unfortunate Parallel Computing Terminology (Knaus, 2008)

- **Master:** Calculation is started on this node and the control program (start/init) is running here.
- **Slave (Worker):** CPU on which a raw parallelized calculation part is running.
- **CPU/Core:** Single calculation unit.
- **Node:** Single computer accessible in a cluster.
- **Universe:** All machines which can possibly be used as a node in a concrete cluster.
- **Cluster:** subset from the universe for specific use.

How does parallel computing in snowfall work? It’s simple…

In a bit more detail:

- Parallelization generally follows the following steps (Knaus, Porzelius, Binder, Schwarzer, 2009):
  1. Initialize the cluster using sfInit().
  2. Wrap parallel code in a wrapper function callable by an R list function.
  3. Export objects, packages, and/or source code to cluster nodes.
  4. Start a network random number generator if need be.
  5. Distribute the calculation to the cluster by using a parallel list function.
  6. Stop the cluster via sfStop().

snowfall functions corresponding to these steps:

- Cluster initialization
  - **sfInit**: initializes cluster with # of cpus set by user
- Exporting Objects, Libraries, and Packages
  - **sfLibrary**: load library on the slaves.
  - **sfSource**: source file on the slaves.
  - **sfExport**: export variables (objects) to slaves.
  - **sfExportAll**: export all variables (objects) in workspace.
  - **sfRemove**: remove specific objects from slaves
  - **sfRemoveAll**: remove anything from slave memory.
- Generating Parallel Random Numbers
  - **sfClusterSetupRNGstream:** set up parallel random number generation.
- Cluster Apply Functions for Actual Computations
  - **sfClusterLapply, sfClusterSapply, sfClusterApply**: Parallel apply functions that work on lists (first two) and arrays (last one) respectively.
  - **sfClusterApplyLB:** load balanced version of clustered Lapply (roughly speaking). Documentation: "If a node finished it's list segment it immediately starts with the next segment. Use this function in infrastructures with machines with different speed."
  - **parMM:** parallel matrix multiplication (I have not used this, but it seems extremely useful).

A Practical Perspective on Setting up Parallel Programs

- Researchers often "think in serial mode" and have a plan for how they would normally execute a script – e.g., a big, nested for loop.
- If so, these serial scripts will need to be rewritten to make them suitable for parallel computation (read: put a list on it).
- Because people don't typically "think in parallel list mode" the challenge, in practice, becomes figuring out where best to intervene in existing sequential programs to ***parallelize*** those programs effectively.

Where to Parallelize

- Where in your script should you intervene to parallelize your code?
  - Search for two requirements:
    1. Independent operations that are
    2. Computationally intensive and time consuming.

Where to Parallelize

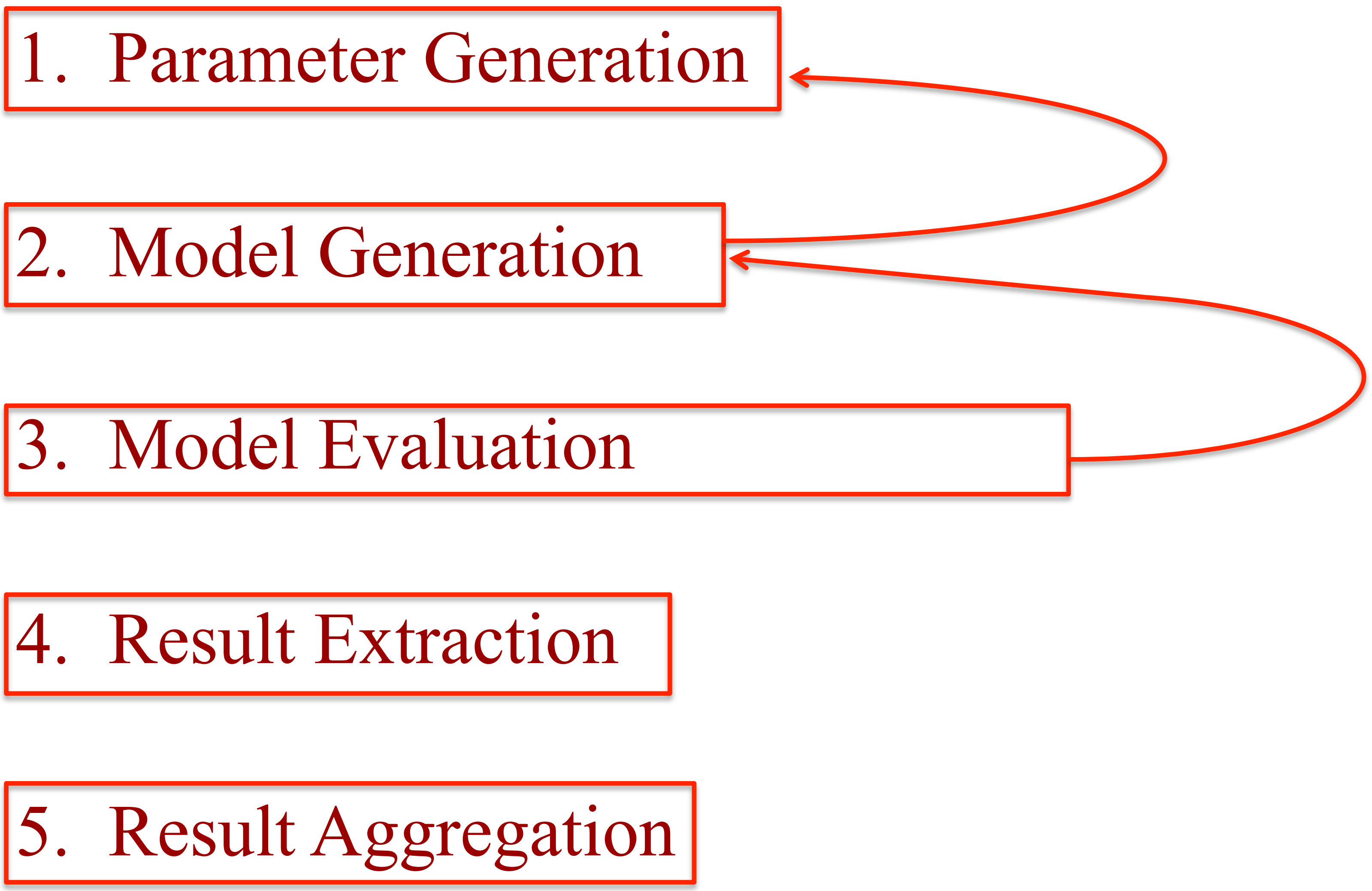
- Some blocks of code may involve independent operations and these blocks of code may even *look* computationally intensive (i.e., the code could look "big" in the script), but in practice they may only take fractions of a second to compute in serial mode.

- Thus, don't parallelize for parrallelization's sake. Intervene where it really matters.

- With that said, sometimes you will be faced with multiple ways to parallelize computationally time-intensive blocks of code, and this is where creativity and programming savvy come in.

Where to Parallelize

- Simulation Steps:



1. Parameter Generation
2. Model Generation
3. Model Evaluation
4. Result Extraction
5. Result Aggregation