Parallelization

ANLY502 - Big Data and Cloud Computing

Vaisman & Dasgupta

Spring 2021

Look back

- Great use of **Slack**
- Any further issues for Windows users with `ssh-agent`/`ssh-add` ?
- You'll get much more practice in bash, which is actually an extremely fast and efficient environment
  - A cheatsheet, and Google will help you find others.

Agenda and Goals for Today

- Scaling up and scaling out
- Parallelization
- Map and Reduce functions
- Lab: Parallelization with Python
  - Use the `multiprocessing` module
  - Implement synchronous and asynchronous processing

Glossary

Term	Definition
Local	Your current workstation (laptop, desktop, etc.), wherever you start the terminal/console application.
Remote	Any machine you connect to via ssh or other means.

Typical real world scenarios

- You are a Data Scientist and you want to cross-validate your models. This involves running the model *1000 times* but each run takes over an hour.
- You are a genomics researcher and have been using small datasets of sequence data but soon you will receive a new type of sequencing data that is *10 times* as large. This means 10x more transcripts to process, but the processing for each transcript is similar.
- You are an engineer using a fluid dynamics package that has an option to run in parallel. So far, you haven't used this option on your workstation. When moving from 2D to 3D simulations, the simulation time has more than tripled so it may make sense to take advantage of the parallel feature

Parallel Programming

Linear vs. Parallel

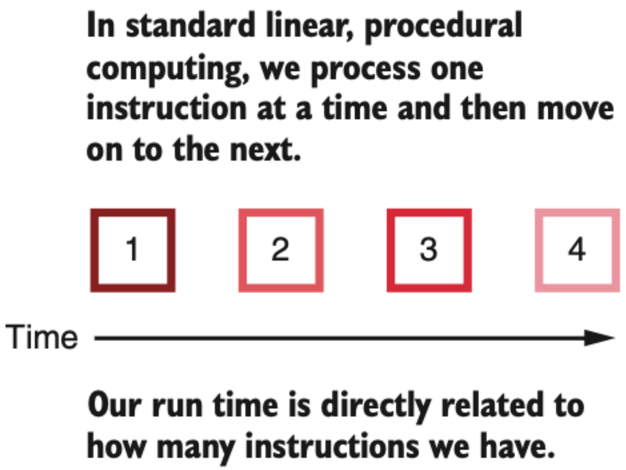
Linear

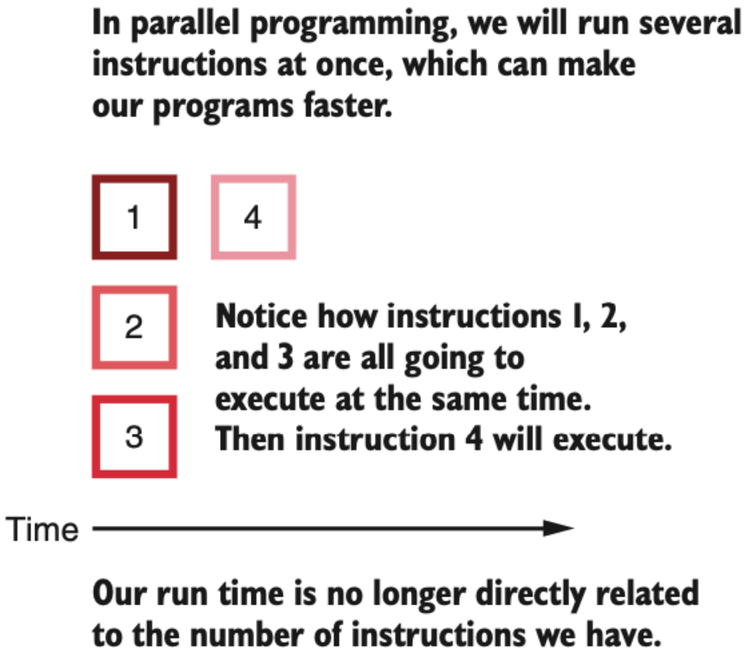
1. A program starts to run
2. The program issues an instruction
3. The instruction is executed
4. Steps 2 and 3 are repeated
5. The program finishes running

Parallel

1. A program starts to run
2. The program **divides up** the work into chunks of instructions and data
3. Each chunk of work is executed independently
4. The chunks of work are reassembled
5. The program finishes running

Linear vs. Parallel





Linear vs Parallel

From a data science perspective

Linear

- The data remains monolithic
- Procedures act on the data sequentially
    - Each procedure has to complete before the next procedure can start
- You can think of this as a single pipeline

Parallel

- The data can be split up into chunks
- The same procedures can be run on each chunk at the same time
- Or, independent procedures can run on different chunks at the same time
- Need to bring things back together at the end

Embarrasingly Parallel

It's **easy** to speed things up when:

- You need to calculate the same thing many times
- Calculations are **independent** of each other
- Each calculation takes a decent amount of time

Just run **multiple calculations at the same time**

Embarrasingly Parallel

The concept is based on the old middle/high school math problem:

> If 5 people can shovel a parking lot in 6 hours, how long will it take 100 people to shovel the same parking lot?

Basic idea is that many hands (cores/instances) make lighter (faster/more efficient) work of the same problem, as long as the effort can be split up appropriately into nearly equal parcels

The classical answer to the problem is 18 minutes

Is this Embarassingly Parallel?

Yes

- Group by analysis
- Simulations
- Resampling / Bootstrapping
- Optimization
- Cross-validation
- Training bagged models (like Random Forests)
- Multiple chains in a Bayesian MCMC
- Scoring (predicting) using trained models

No

- SQL Operations
- Inverting a matrix
- Training linear regression
- Training logistic regression
- Training trees
- Training neural nets
- Training boosted models (like gradient boosted trees)
- Each chain in a Bayesian MCMC
- Most things time series

Pros and cons of parallelization

Pros

- Higher efficiency
- Using modern infrastructure
- Scalable to larger data, more complex procedures
  - *proviso* procedures are embarassingly parallel

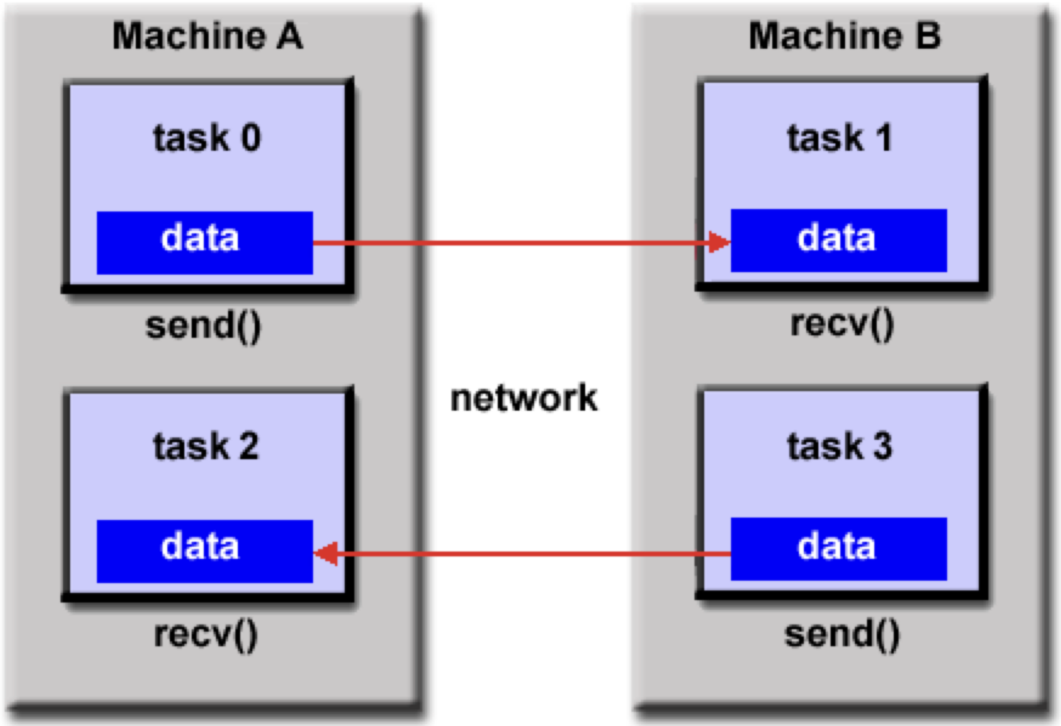
Cons

- Higher programming complexity
  - Need proper software infrastructure (MPI, Hadoop, etc)
  - Need to ensure right packages/modules are distributed across processors
- Need to account for a proportion of jobs failing, and recovering from them
  - Hence, Hadoop/Spark and other technologies
- Higher setup cost in terms of time/expertise/money

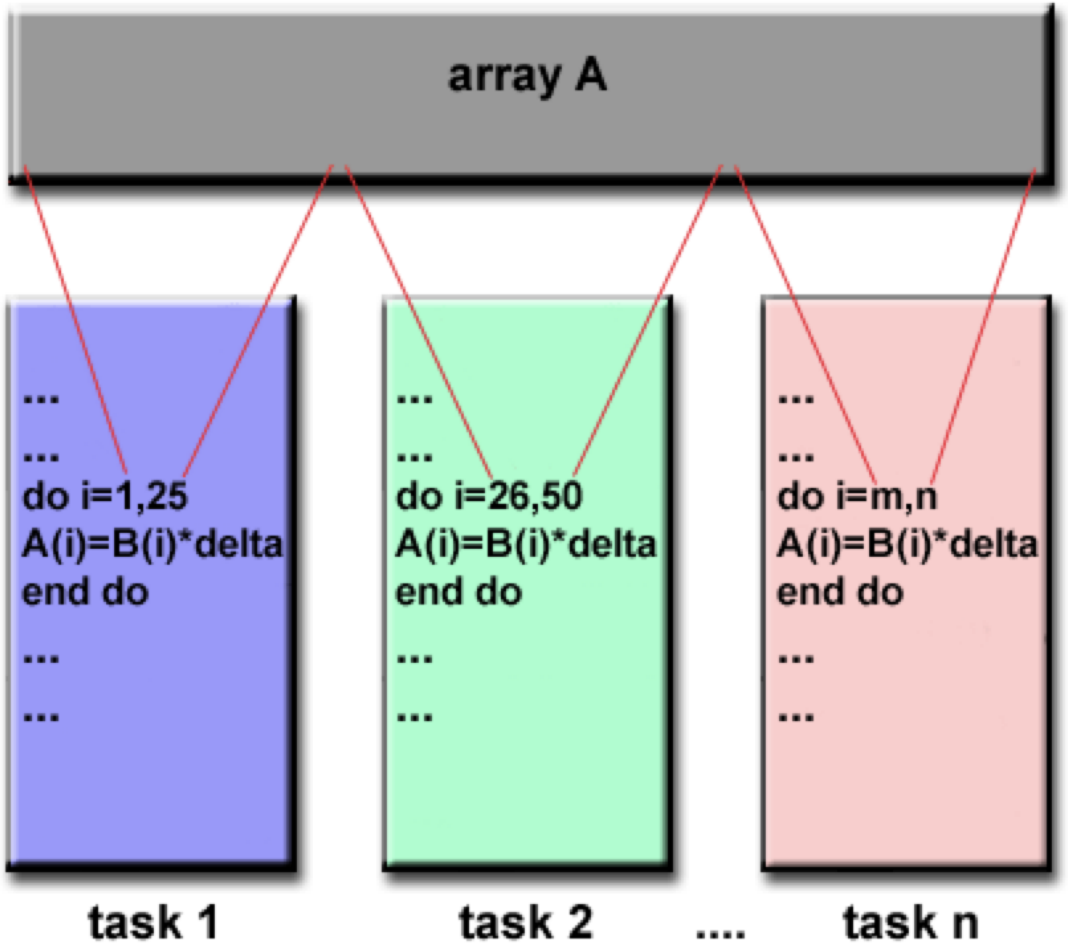
There are good solutions today for most of the cons, so the pros have it and so this paradigm is widely accepted and implemented

Parallel Programming Models

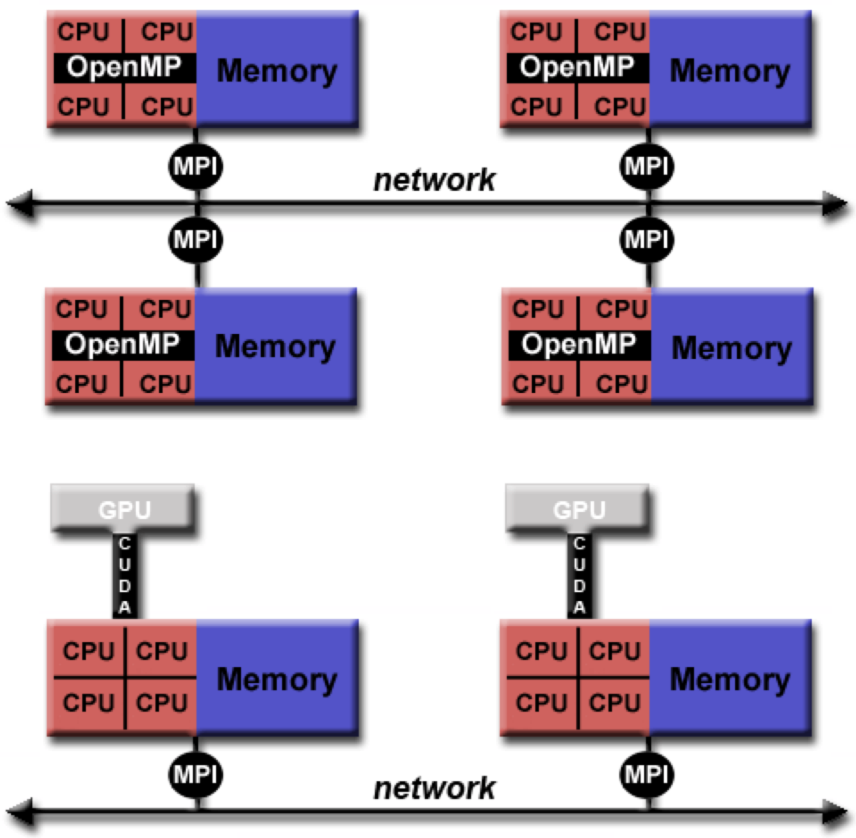
Distributed memory / Message Passing Model



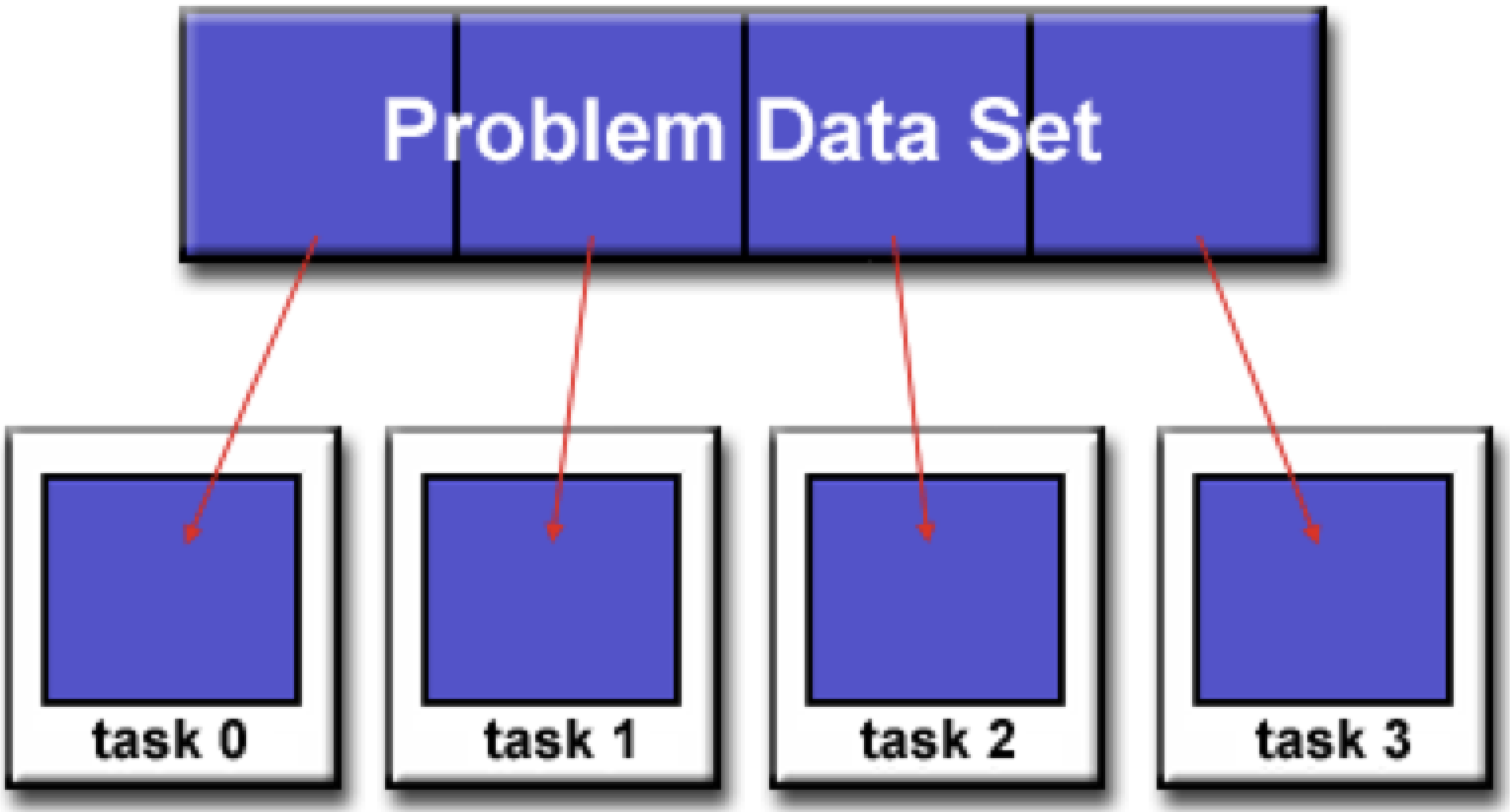
Data parallel model



Hybrid model



Partitioning data



Designing parallel programs

- Data partitioning
- Communication
- Synchronization / Orchestration
- Data dependencies
- Load balancing
- Input and Output (I/O)
- Debugging

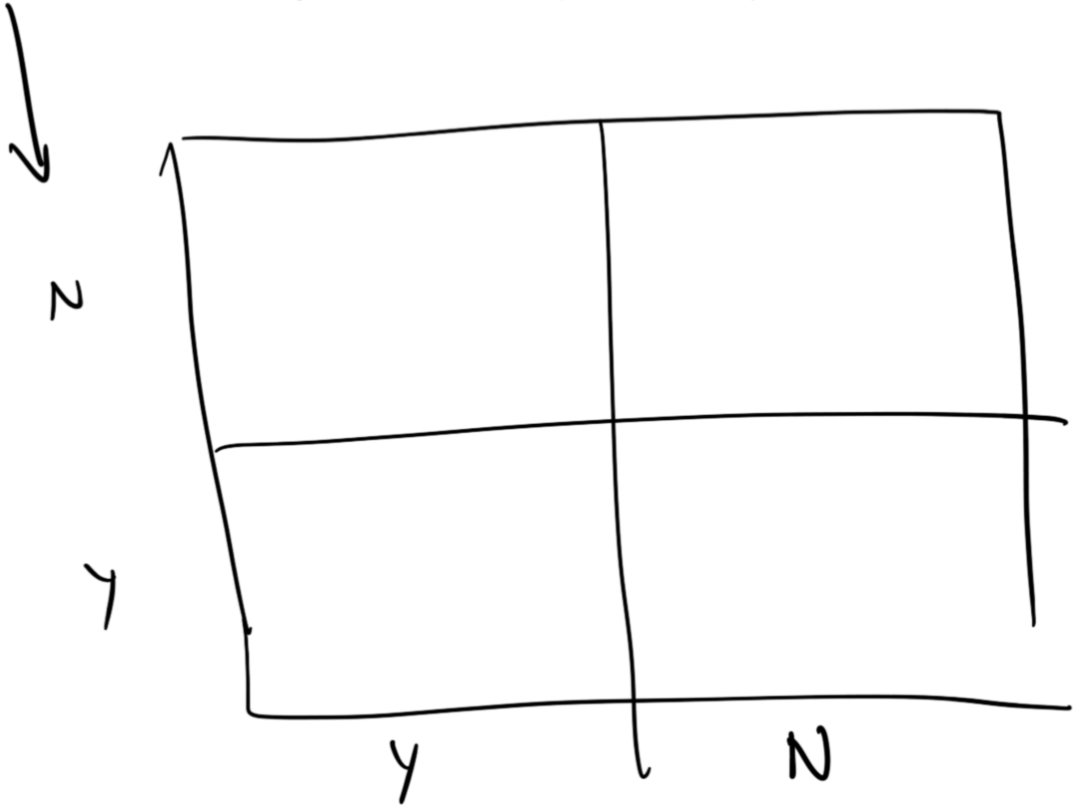
A lot of these components are data engineering and DevOps issues

Infrastructures have standardized many of these and have helped data scientists implement parallel programming much more easily

We'll see in the lab how the `multiprocessing` module in Python makes parallel processing on a machine quite easy to implement

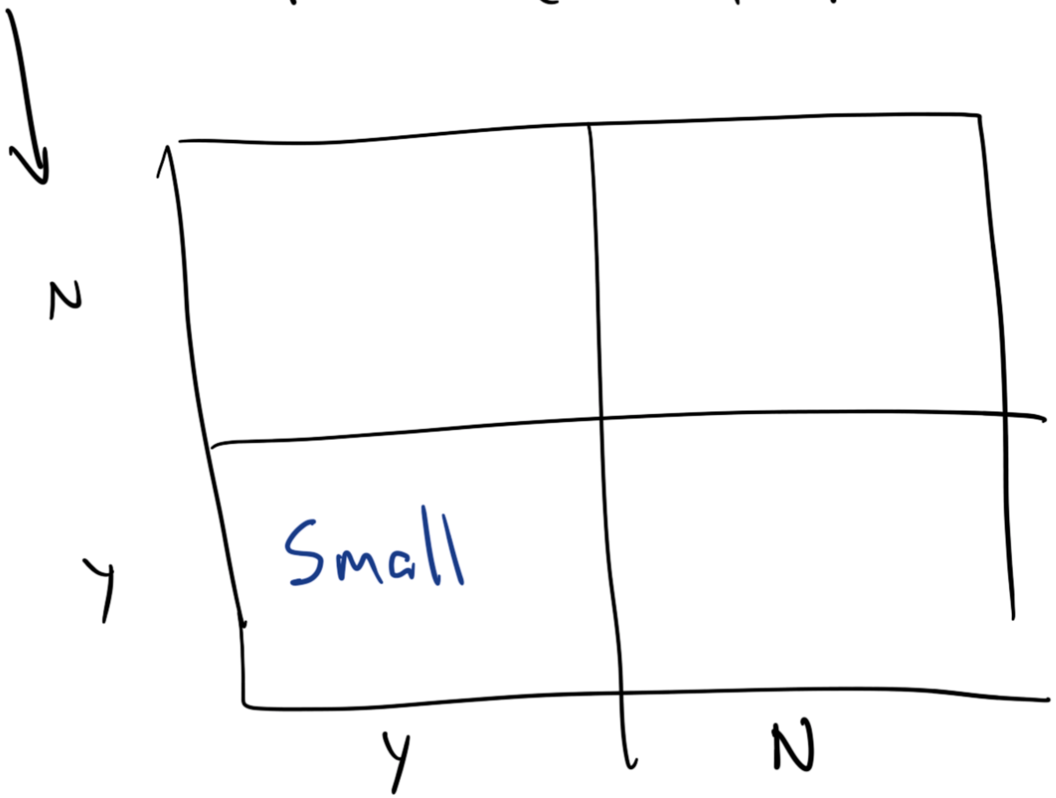
Parallel Computing

Data can be processed (memory/cpu) on a single machine?



Data fits in a single machine?

Data can be processed (memory/cpu) on a single machine?



Data fits in a single machine?

Data can be processed (memory/cpu) on a single machine?

	Y	N
N	Medium	
Y	Small	Medium

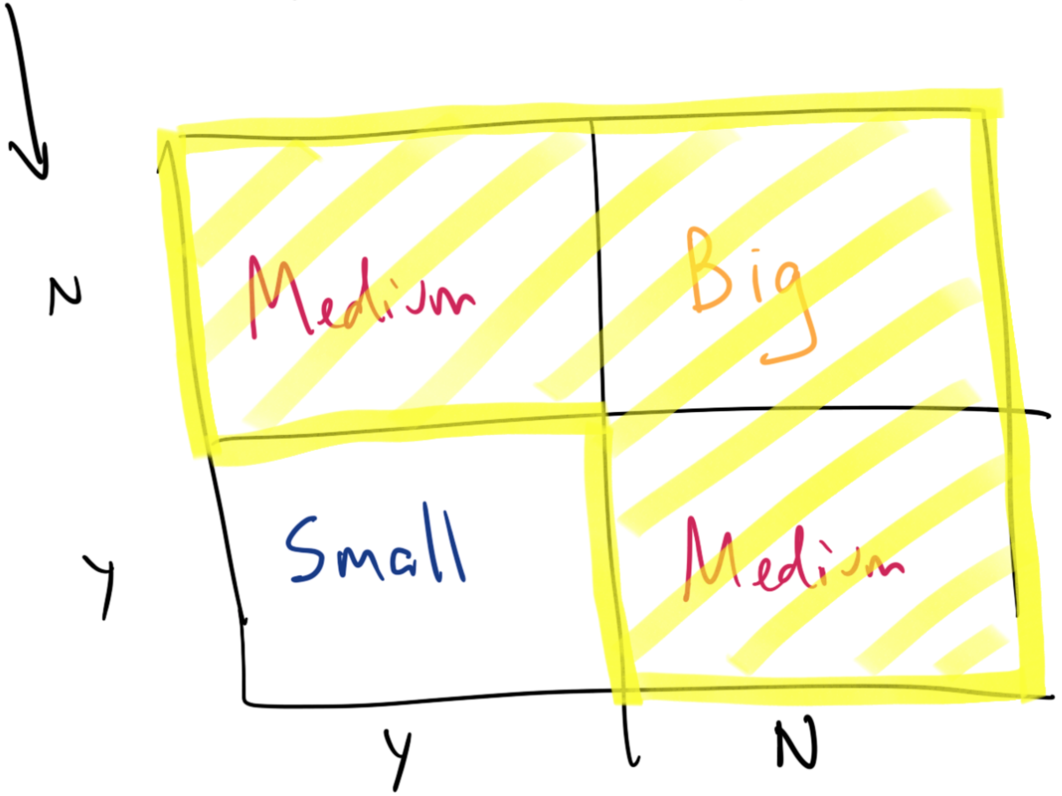
Data fits in a single machine?

Data can be processed (memory/cpu) on a single machine?

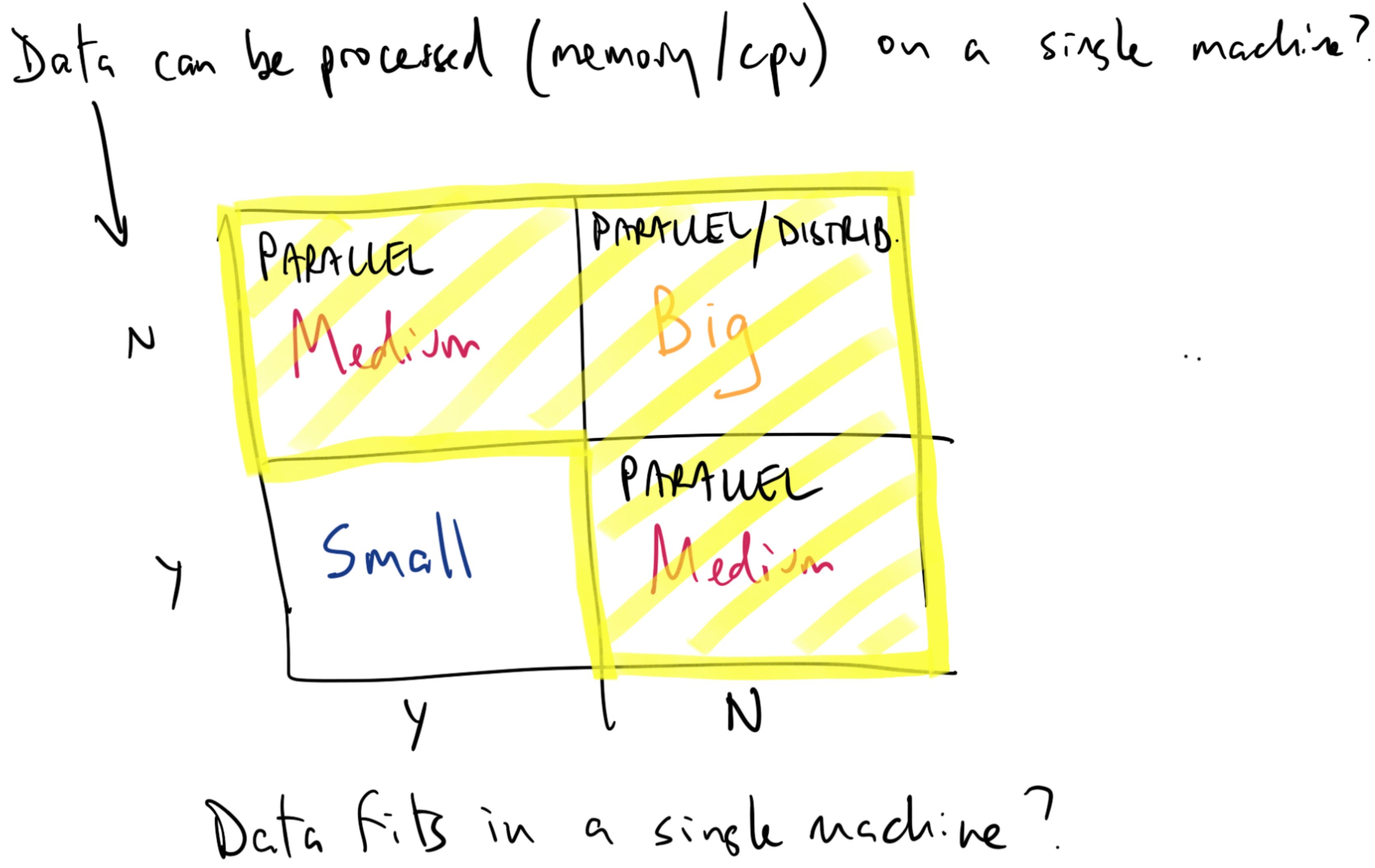
N	Medium	Big
Y	Small	Medium
	Y	N

Data fits in a single machine?

Data can be processed (memory/cpu) on a single machine?



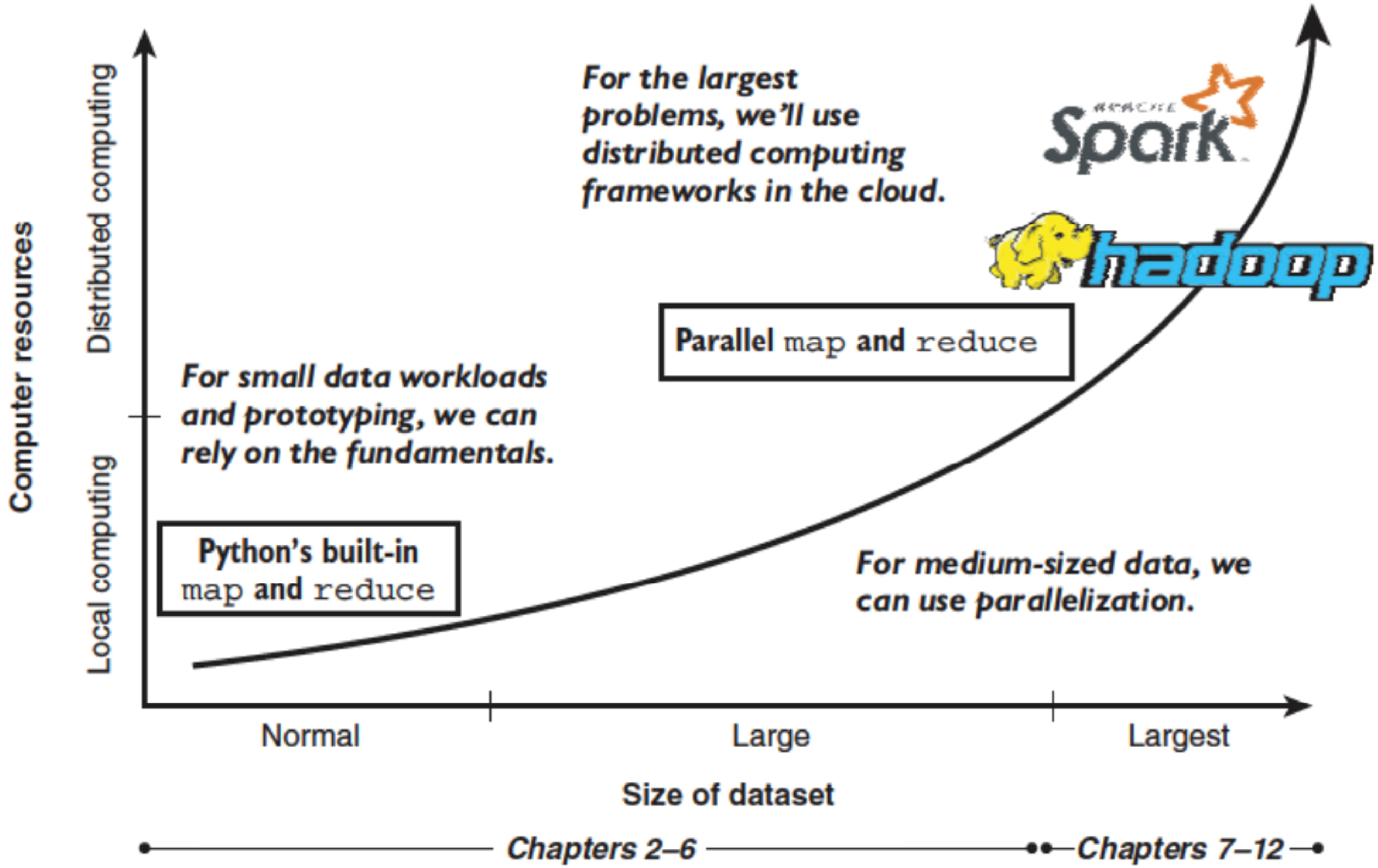
Data fits in a single machine?


Data can be processed (memory/cpu) on a single machine?
N
Y
PARALLEL
Medium
PARALLEL/DISTRIB.
Big
Small
PARALLEL
Medium
Y
N
Data fits in a single machine?

Functional Programming

Map and Reduce

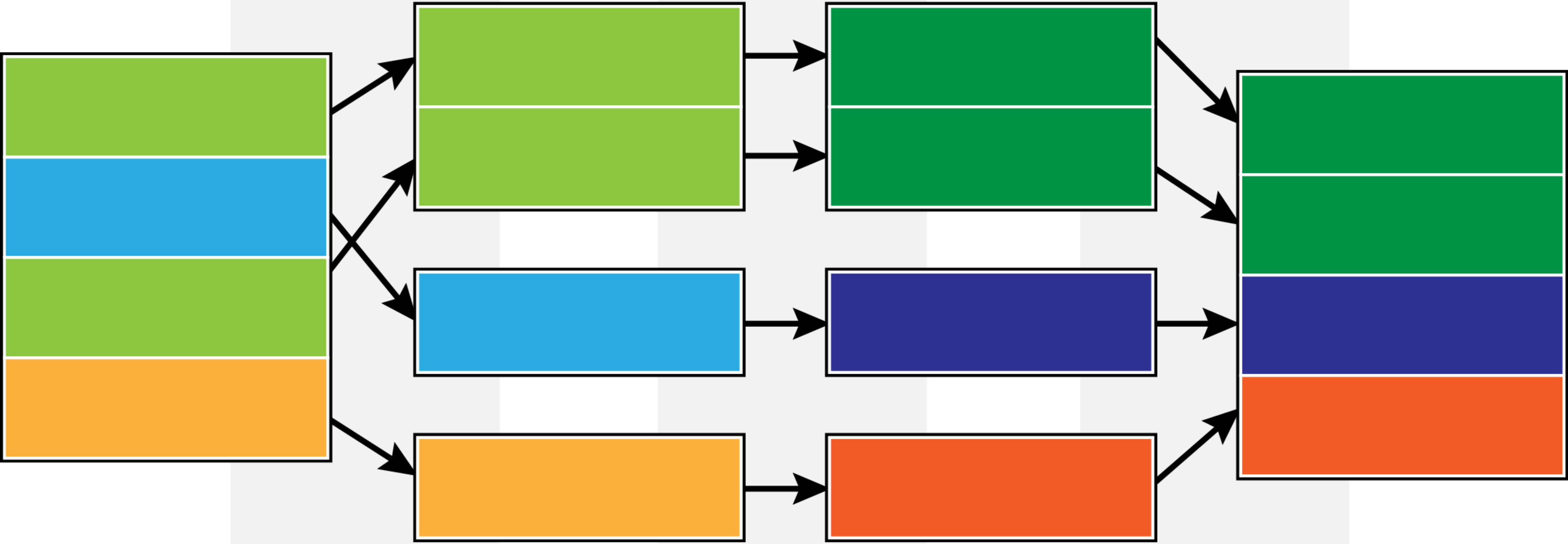
Tools used in a map and reduce style of programming, by dataset size and compute resources available



Components of a parallel programming workflow

1. Divide the work into chunks
2. Work on each chunk separately
3. Reassemble the work

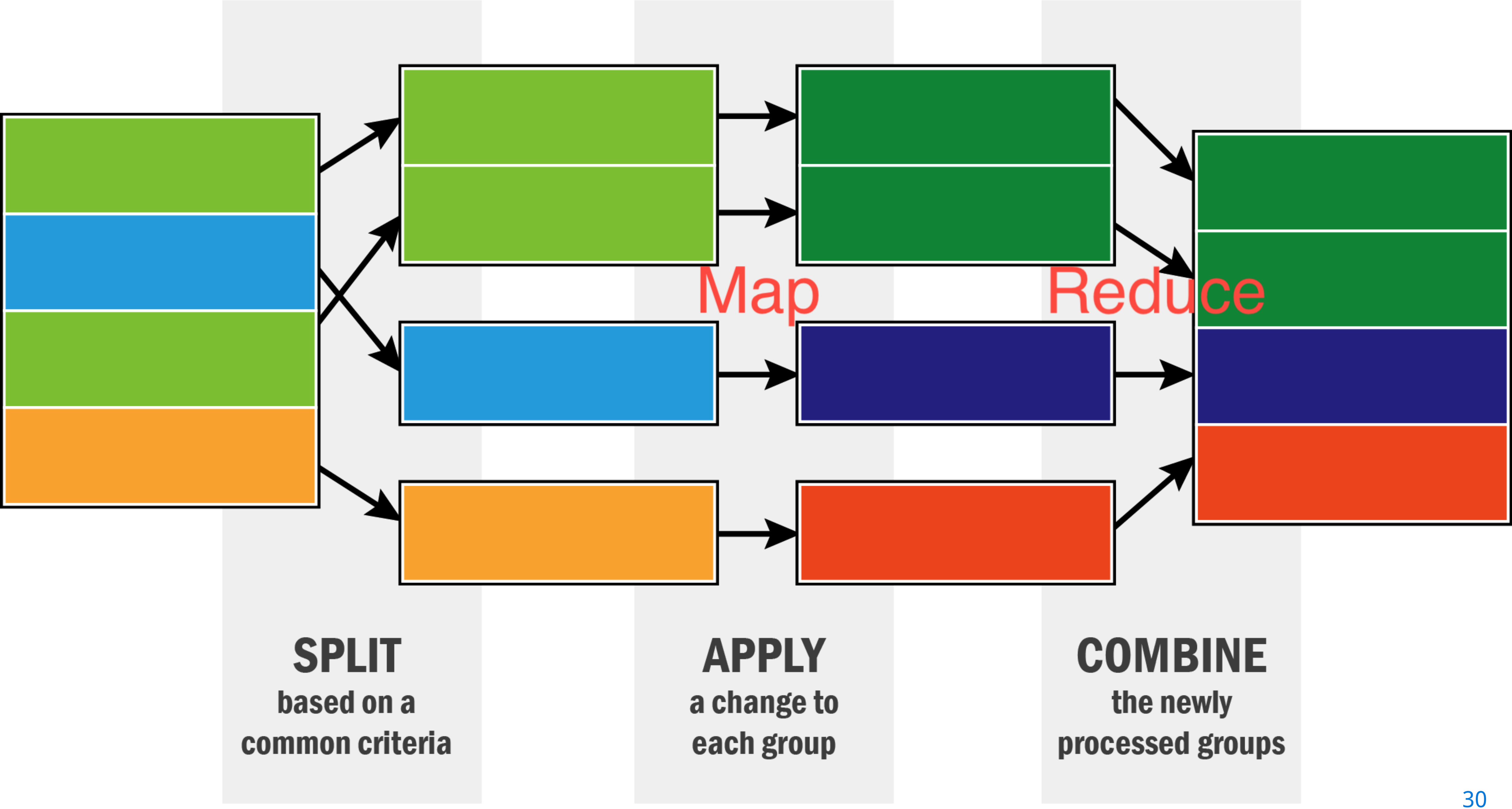
This paradigm is often referred to as a **map-reduce framework**, or, more descriptively, the **split-apply-combine** paradigm



SPLIT
based on a
common criteria

APPLY
a change to
each group

COMBINE
the newly
processed groups


Map
Reduce
SPLIT
based on a
common criteria
APPLY
a change to
each group
COMBINE
the newly
processed groups

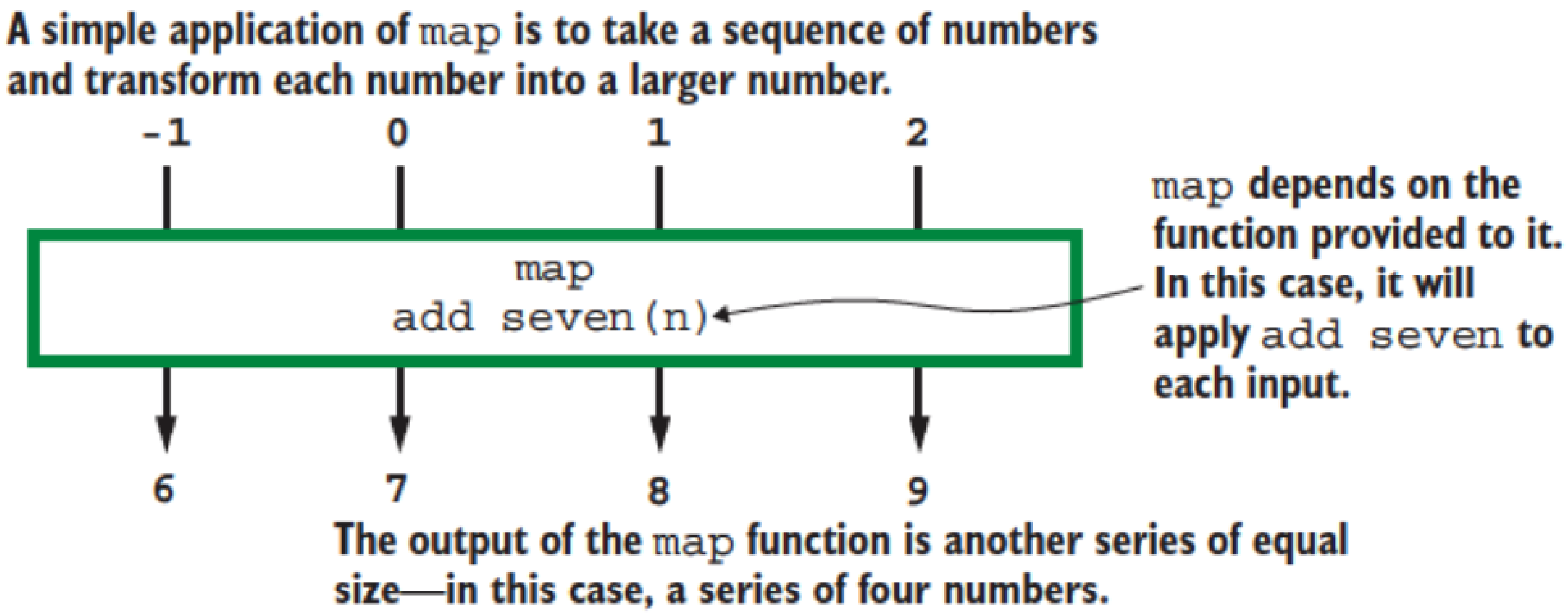
Map

Map

The **map** operation is a 1-1 operation that takes each split and processes it

The **map** operation keeps the same number of objects in its output that were present in its input

Map



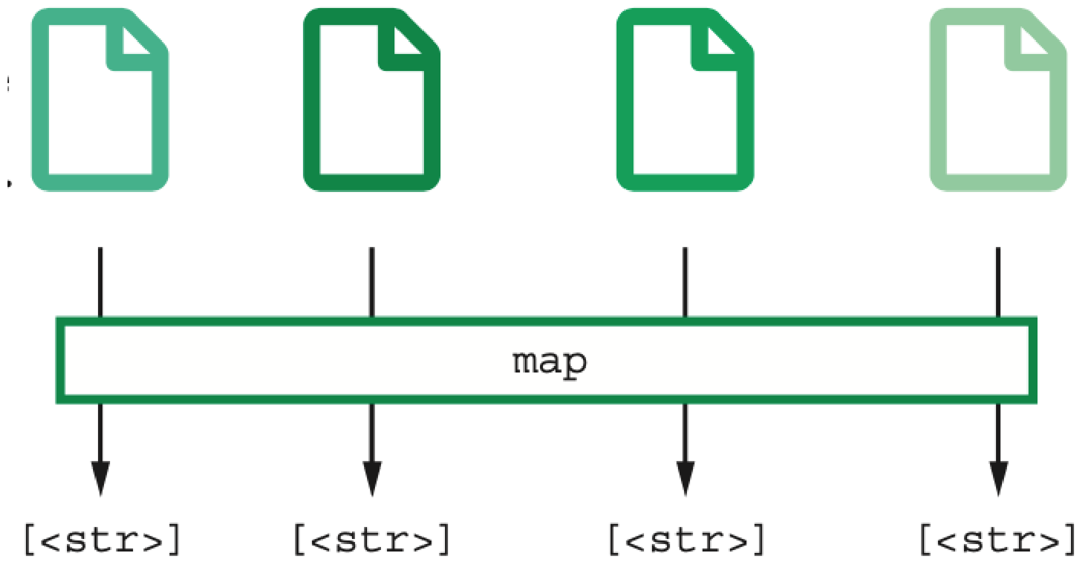
The operations included in a particular **map** can be quite complex, involving multiple steps. In fact, you can implement a *pipeline* of procedures within the **map** step to process each data object.

The main point is that the *same* operations will be run on each data object in the **map** implementation

Map

Some examples of a **map** operations are

1. Extracting a standard table from online reports from multiple years
2. Extracting particular records from multiple JSON objects
3. Transforming data (as opposed to summarizing it)
4. Run a normalization script on each transcript in a GWAS dataset
5. Standardizing demographic data for each of the last 20 years against the 2000 US population

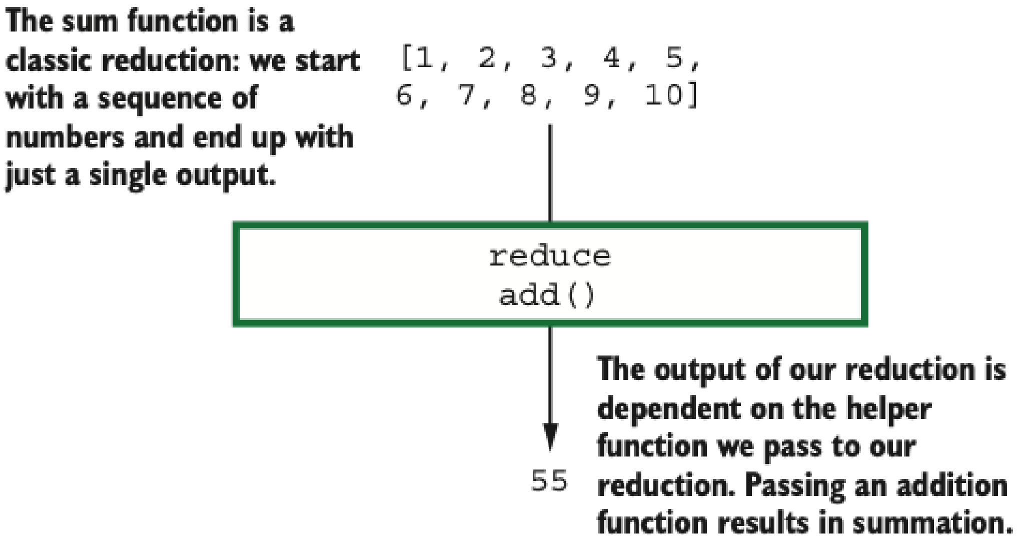


Reduce

The **reduce** operation takes multiple objects and *reduces* them to a (perhaps) smaller number of objects using transformations that aren't amenable to the **map** paradigm.

These transformations are often serial/linear in nature

The **reduce** transformation is usually the last, not-so-elegant transformation needed after most of the other transformations have been efficiently handled in a parallel fashion by **map**

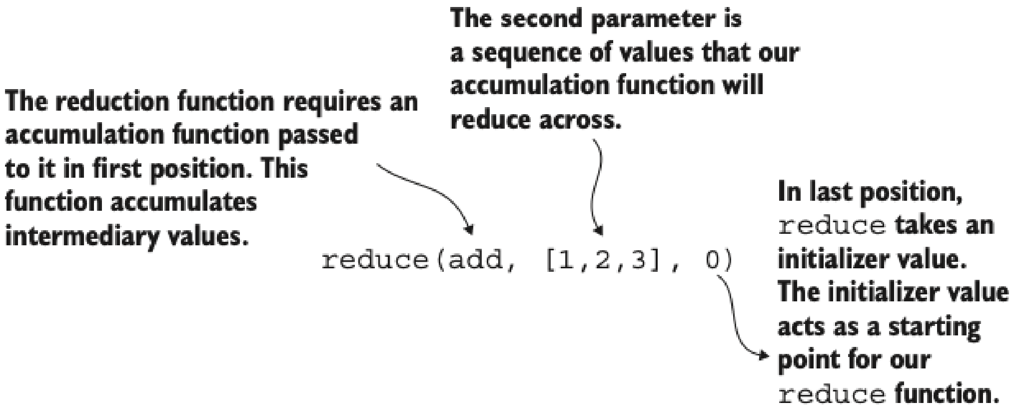


Reduce

The **reduce** operation requires

a. An *accumulator* function, that will update serially as new data is fed into it
b. A sequence of objects to run through the accumulator function
c. A starting value from which the accumulator function starts

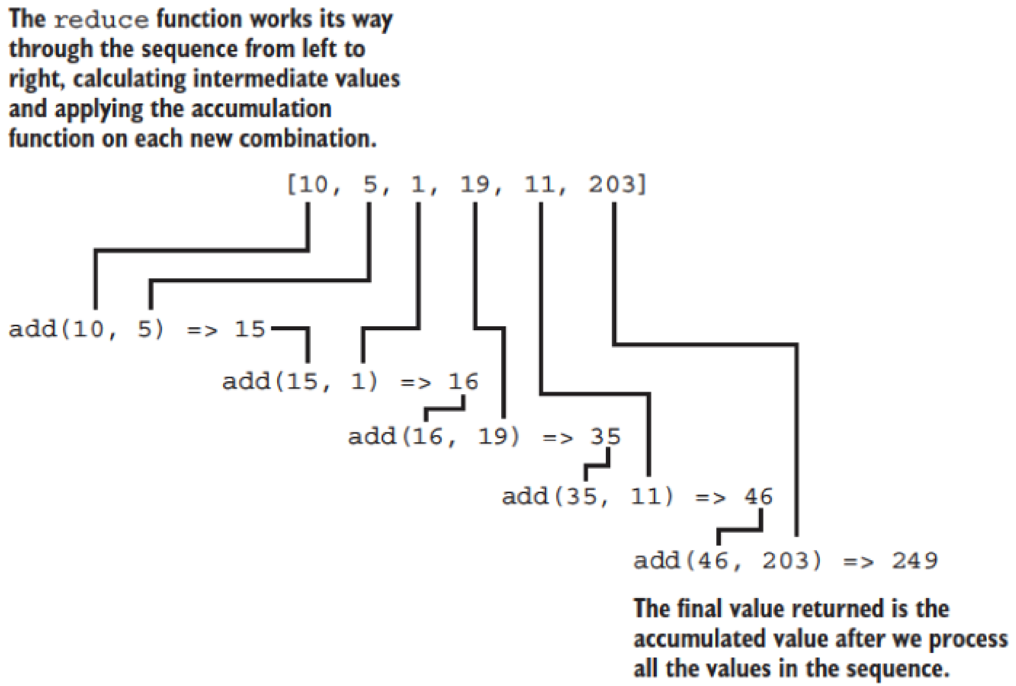
Programmatically, this can be written as



Reduce

The **reduce** operation works serially from "left" to "right", passing each object successively through the accumulator function.

For example, if we were to add successive numbers with a function called `add`...



Reduce

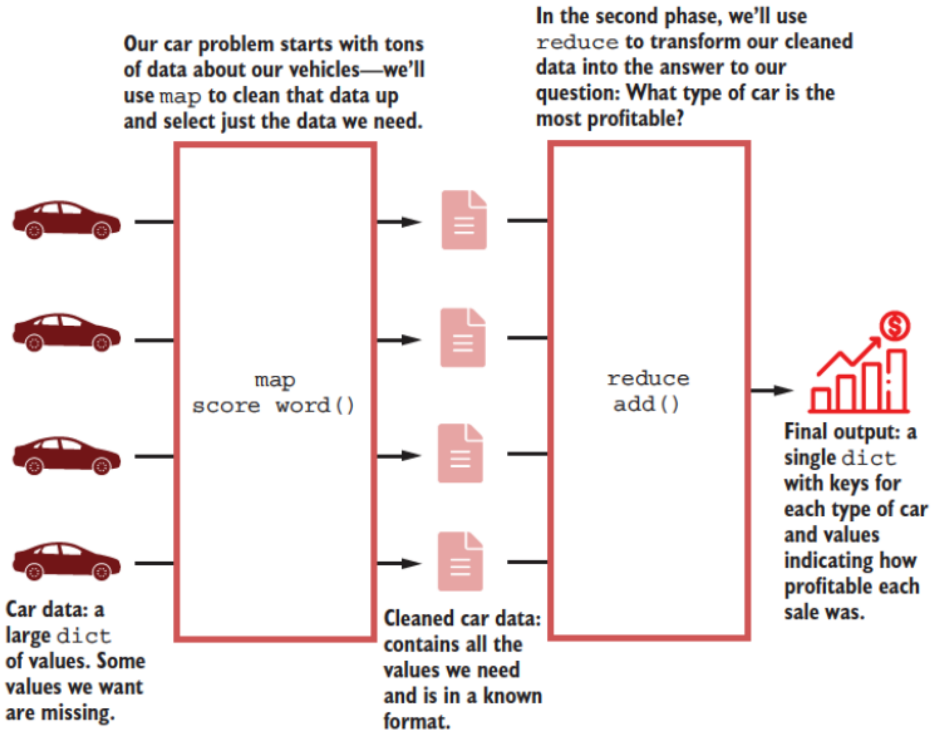
Some examples:

1. Finding the common elements (*intersection*) of a large number of sets
2. Computing a table of group-wise summaries
3. Filtering
4. Tabulating

Map & Reduce

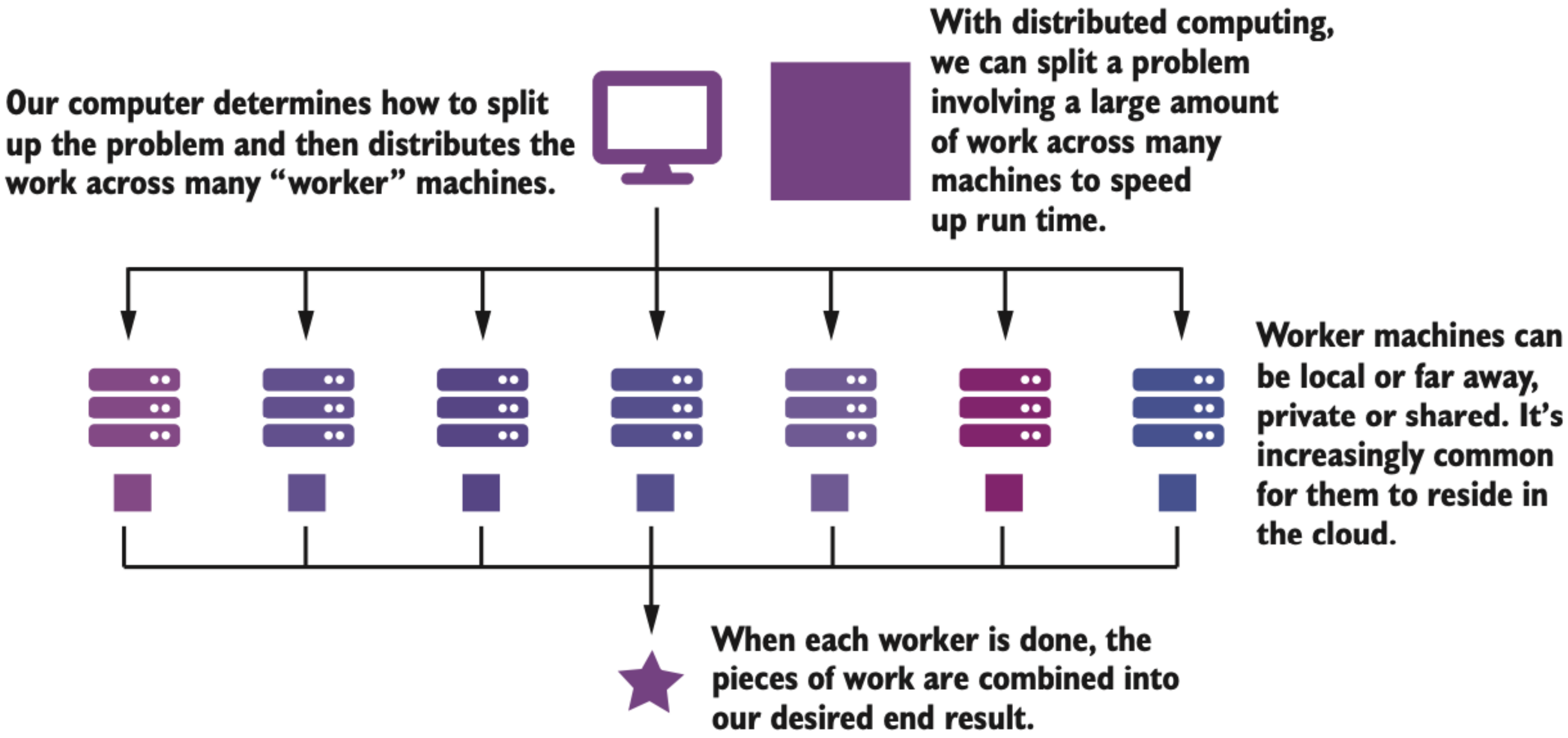
map-reduce

Combining the **map** and **reduce** operations creates a powerful pipeline that can handle a diverse range of problems in the Big Data context



Parallelization and map-reduce

Parallelization and map-reduce are bed-mates



One of the issues here is, how to split the data in a "good" manner so that the map-reduce framework works well