Queen Mary University of London

PARALLEL SYSTEMS PERFORMANCE

BIG DATA PROCESSING

Félix Cuadrado

felix.cuadrado@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Queen Mary
University of London

Contents

- **Parallel speedup**
- Hadoop Performance
- Iterative MapReduce

Speedup concept

- speedup of a parallel processing system is a function of n, the number of processors:

$$s(n) = \frac{time\_taken\_with\_1\_processor}{time\_taken\_with\_n\_processors}$$

- speedup is problem-dependent as well as architecture-dependent
- A 100% embarrasingly parallel job can be fully parallelized.

$$s_{emb}(n) = n$$

Amdahl's Law

- In many jobs, some parts of the computation can only be executed on one processor.
  - if single processor parts take a fraction *f* of the total work, then the maximum speedup is *s(infinite)* = 1/*f*
  - Amdahl's law: if the remaining 1 - *f* of the work can be perfectly parallelized, then the speedup with n processors is

$$s(n) = \frac{n}{1+(n-1)f}$$

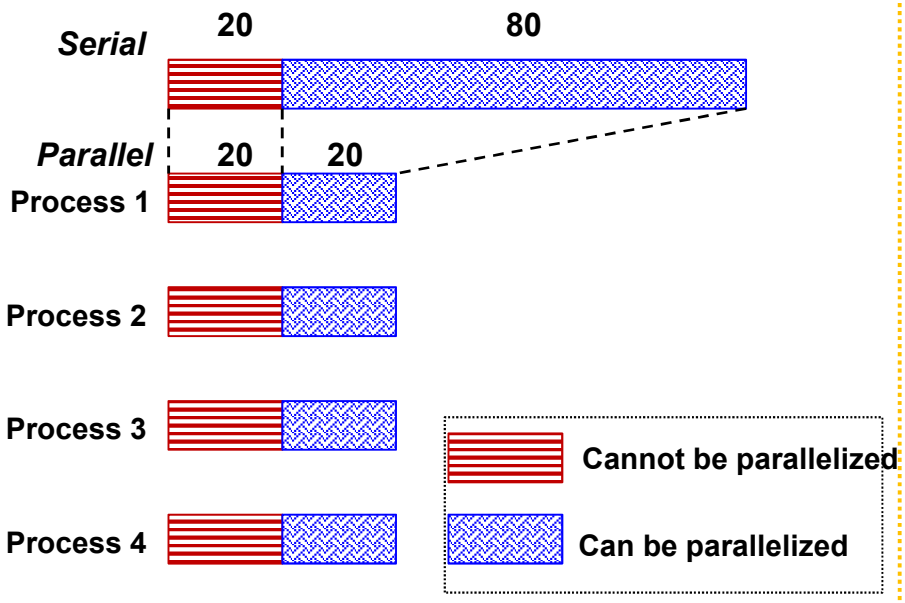
  - *s(n)* grows with n, never gets larger than 1/*f*

Amdahl's law exercises

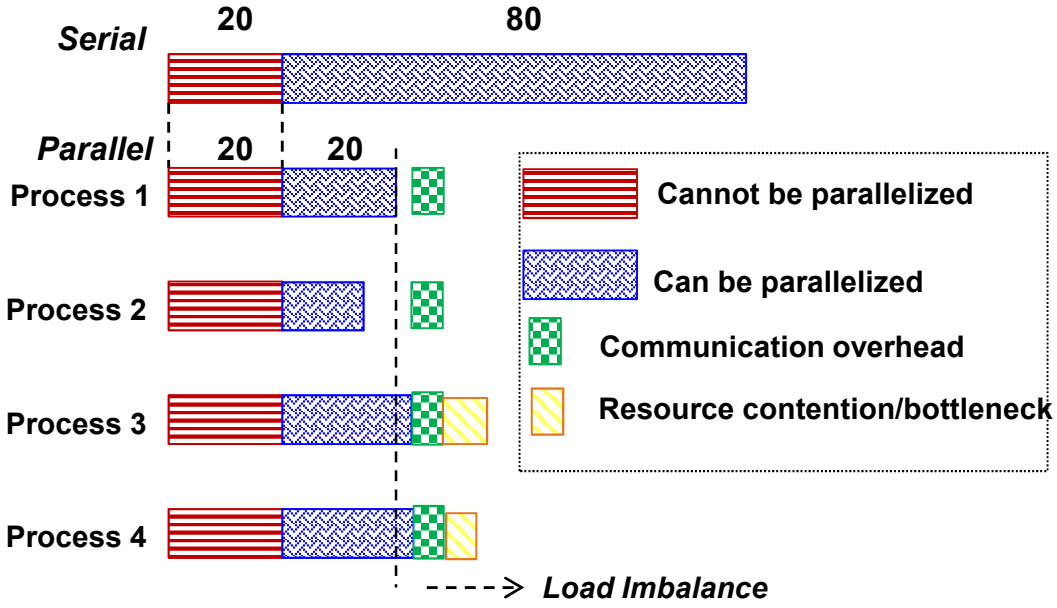
- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?
- 5% of a parallel program's execution time is spent within inherently sequential code. What is the maximum speedup achievable by this program, regardless of how many parallel cores are used?

Speedup: Real Vs. Actual Cases

- Amdahl's argument is too simplified to be applied to real cases
- When we run a parallel program, there are a communication overhead and a workload imbalance among processes in general



1. Speedup: Amdahl's Law Ideal Case



2. Speed-up: An Actual Case

Queen Mary
University of London

Contents

- Parallel speedup
- **Hadoop Performance**
- Iterative MapReduce

Amdahls Law on Map/Reduce jobs

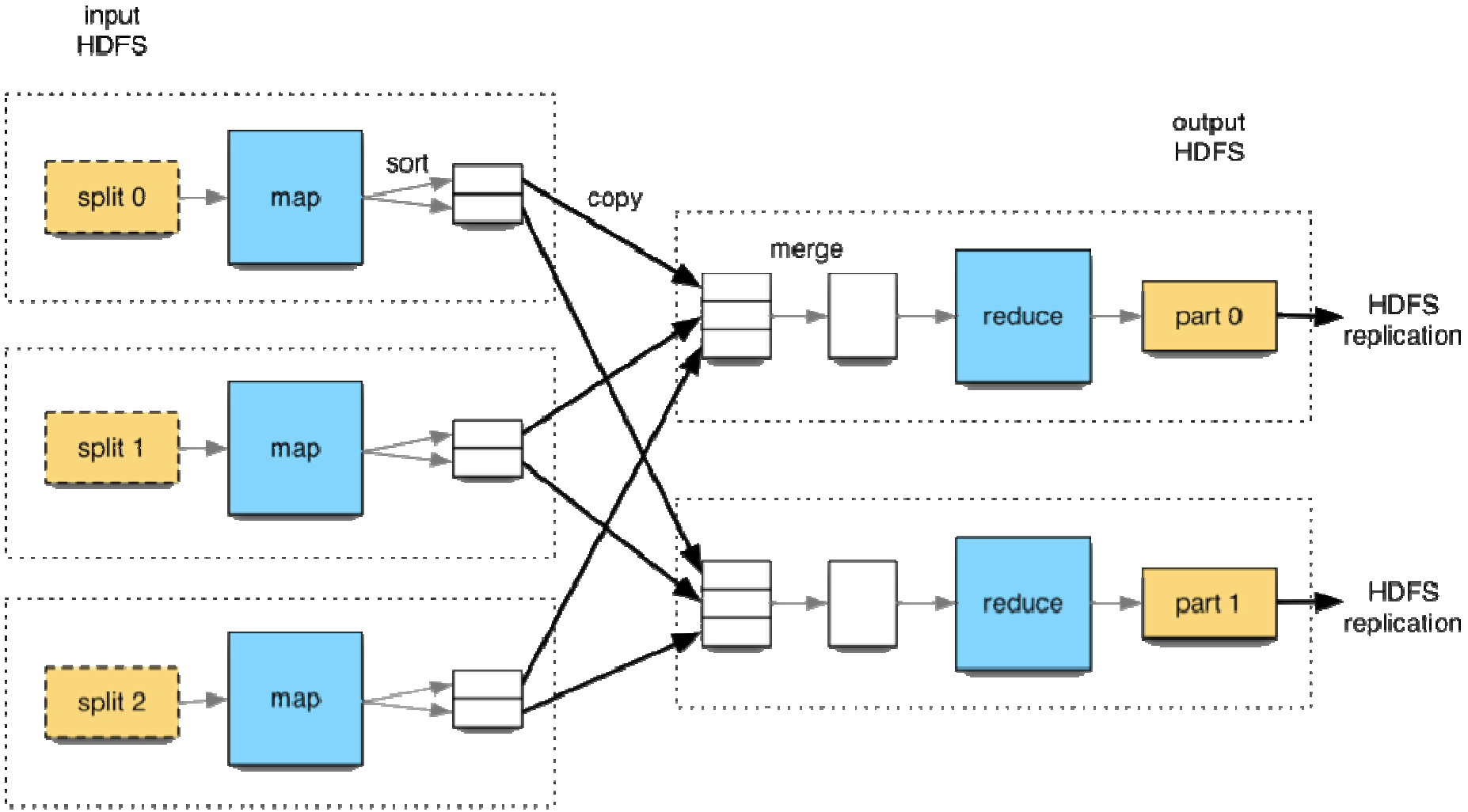


Image source: Hadoop: the definitive guide, Tom White

Amdahls Law on Map/Reduce jobs

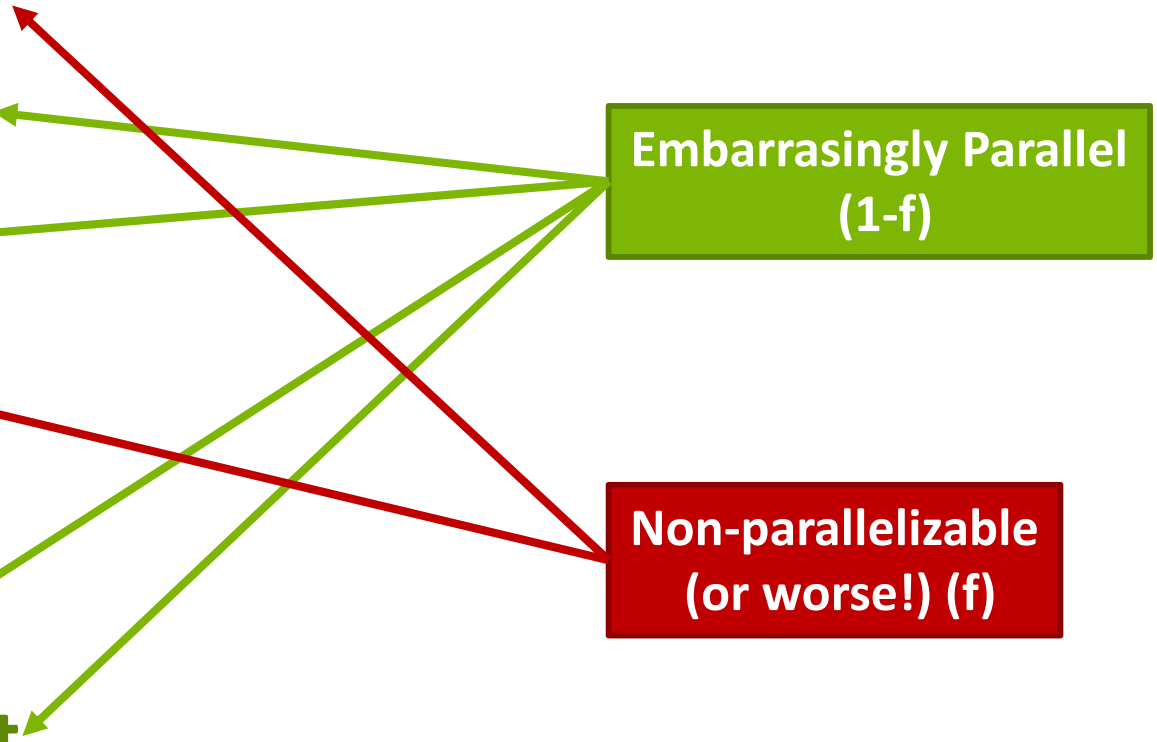
1. Job setup
2. Load Split
3. Map
4. Copy
5. Merge
6. Reduce
7. Write part

Embarrasingly Parallel (1-f)

Non-parallelizable (or worse!) (f)

Amdahls Law on Map/Reduce jobs

1. **Job setup**
2. **Load split**
3. **Map**
4. **Copy**
5. **Merge**
6. **Reduce**
7. **Write part**



Indicators for Hadoop job performance

- **Latency** is the time between the start of a job and when it starts delivering output
  - In Hadoop: total job execution time
- **Throughput** measured in bytes/second
- high latency is entirely compatible with high throughput…
- …especially in something like Hadoop where we have a lot of coordination to do at the beginning

Hadoop performance overheads

- Job setup is costly, becomes more complex the bigger the dataset is
- Reading from HDFS takes up some CPU cycles
  - seconds per gigabyte
- HDFS has some latency (microseconds per block read)
- concurrent threads give lock contention
- disks finite throughput (MB/sec)
- Hadoop is I/O or network bound, often not CPU bound
- more at: http://www.slideshare.net/cloudera/hadoop-world-2011-hadoop-and-performance-todd-lipcon-yanpei-chen-cloudera

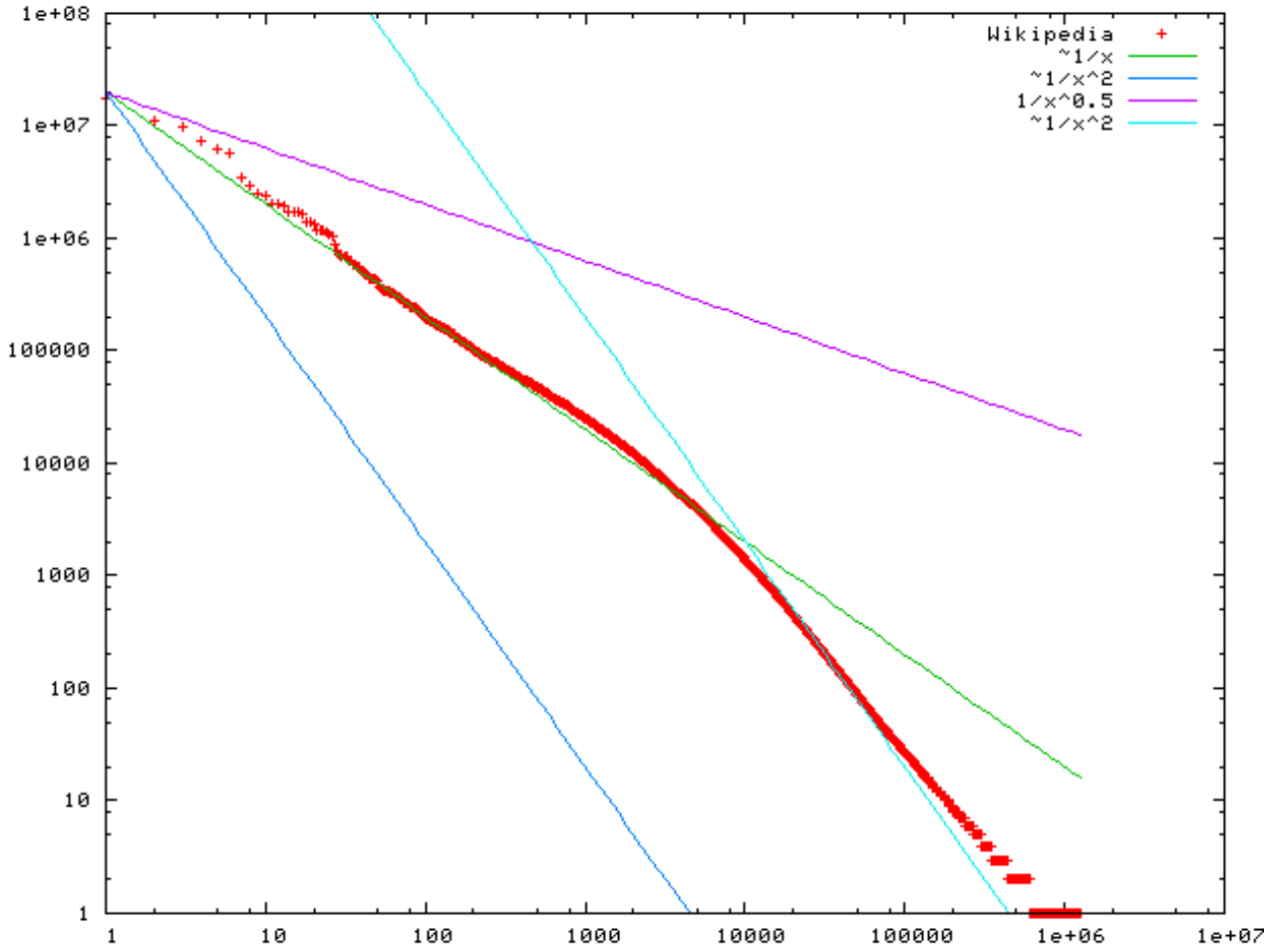
Optimization points

- For improving MapReduce job performance focus must be on bottlenecks:
  - number of keys emitted by Mappers (Combiner)
    - reduces Map-> Reduce network transfer
    - simplifies shuffle and sort activities at Reducer
  - sorting in the shuffle and sort stage
    - comparison of keys (typically with WritableComparable)
  - avoid unnecessary Java object creation
    - Reuse Writable objects

Load balancing problems: Data skew

- Problem: Not every task processes the same amount of data
  - Mappers: in general splits are balanced
  - **Reducers**: number of keys, number of values per key
    - e.g. think of Word Count, partitioning by starting letter. Some are much more common than others
- The Partitioning of Keys to Reducers can be trained to provide a balanced spread regardless of the skew
  - Initial sampling of data

Wikipedia word frequency distribution



Performance analysis of MapReduce jobs

- Input dataset: size, number of records
- Average number of records generated per Mapper
  - How much information is being sent over the network
  - Effect of a Combiner?
- Number of keys/records sent to each Reducer
  - Data skew?
  - Key with too many records?

Queen Mary
University of London

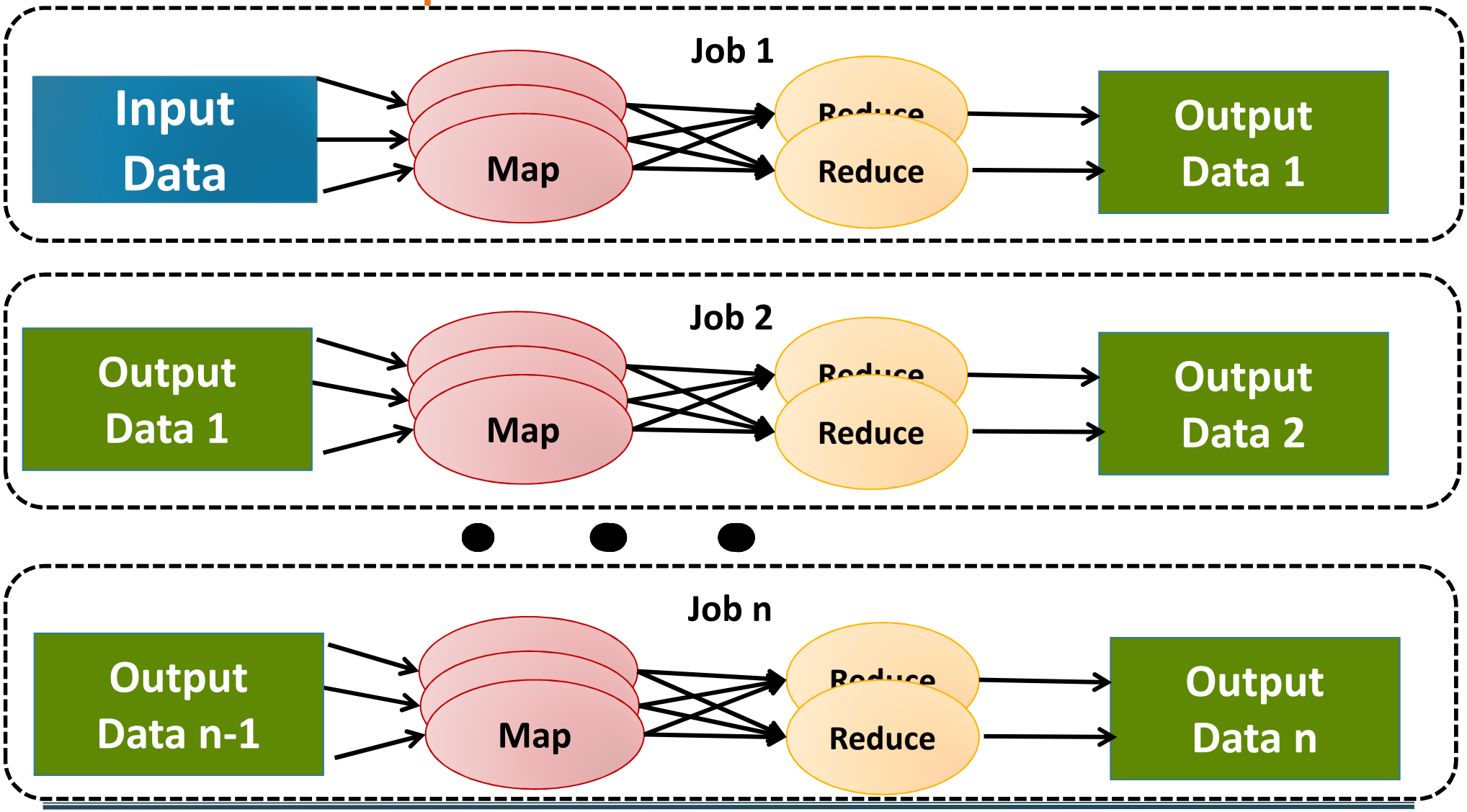
Contents

- Parallel speedup
- Hadoop Performance
- **Iterative MapReduce**

Expressing Complex algorithms in MapReduce

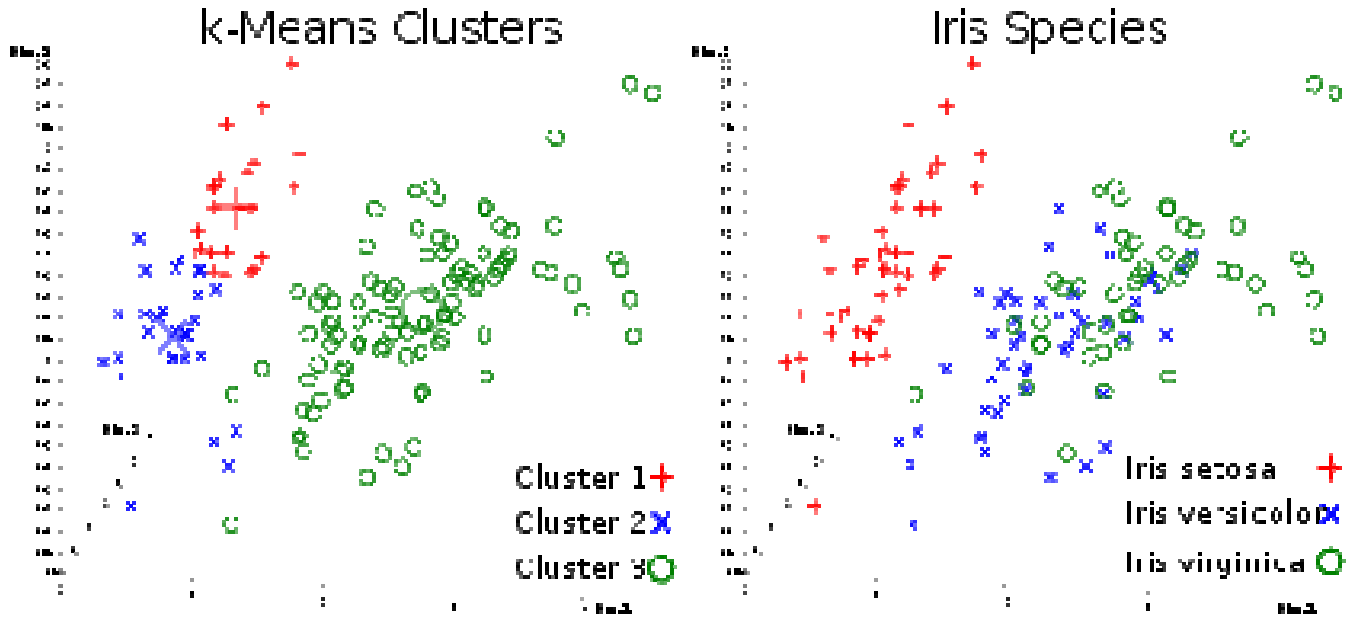
- A MapReduce job can only have up to one Map, one Reduce task
- Many complex algorithms have dependencies on previous results
  - Cannot parallelise these, only data parallelism
- Approach: chain sequentially MapReduce jobs
- Output of a job becomes input of the next one
- Either programmatically, or manually run one after another

Iterative MapReduce



Example: K-means clustering

- Classic clustering algorithm
- Clusters all data points across x centroids
- Iterative refinement until convergence



https://en.wikipedia.org/wiki/K-means_clustering

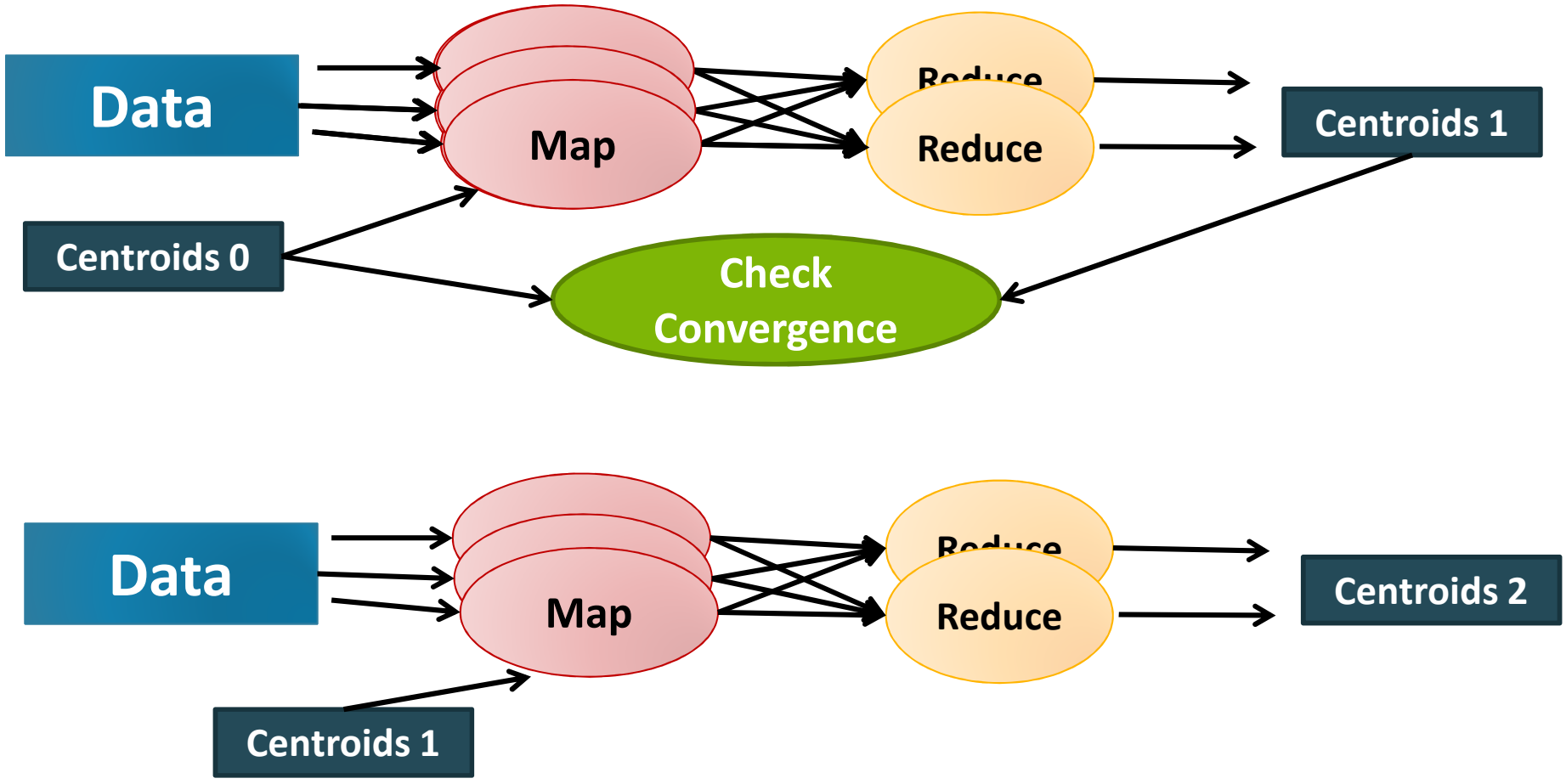
K-Means using MapReduce

Step1: Initially randomly select 3(k=3) centroids from data.
Step2: The Input file contains initial centroid and data.
Step3: ***Mapper setup***: read the centroids file and store in the data structure(we used ArrayList)
Step4: *Run Map:* emit [nearest centroid, data point].
Step5: ***Run Reducer***: Calculate the new centroids and emit.
Step6: In the ***job configuration***, checking **if** difference between old and new centroid is less than 0.1

then convergence is reached
else repeat **Step 2** with new centroids.

Algorithm from https://github.com/himank/K-Means

MapReduce k-means computation



Iterative Map-Reduce performance

- Every MapReduce job is independent
  - No shared state
- Data has to be transferred from Mappers to Reducers
- Data has to be loaded from disk every iteration
- Results are saved to HDFS, with multiple replication
- Performance verdict?