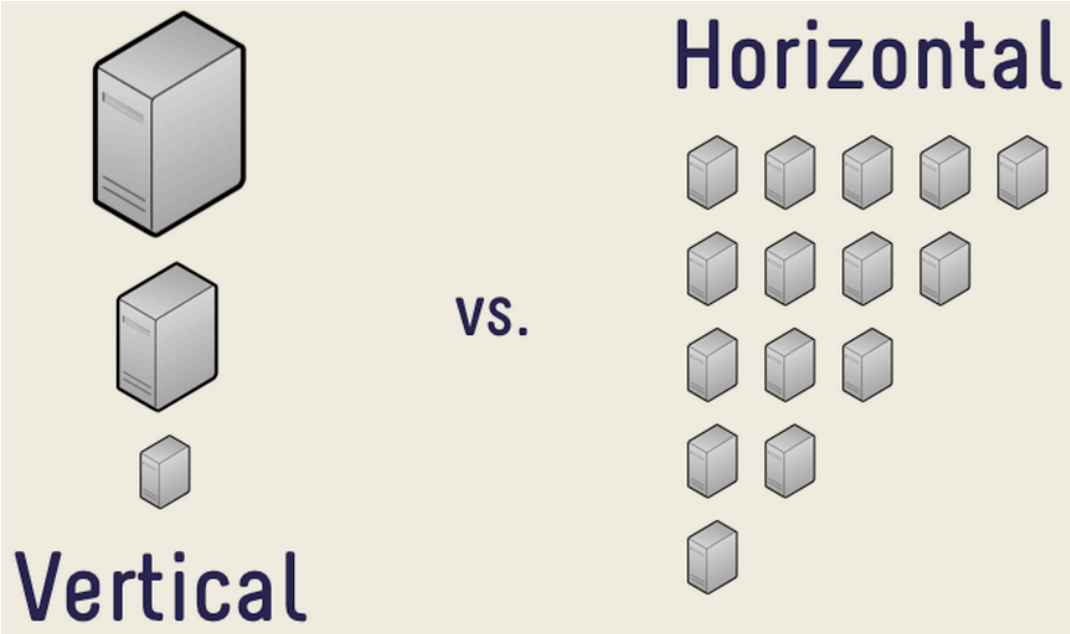
Storage and computing resources vs Big Data

- How to store and process ever increasing amounts of data in a sustainable manner?
  - Both the data storage and computing resources should be scalable.
    - More data → more storage and computing resources.

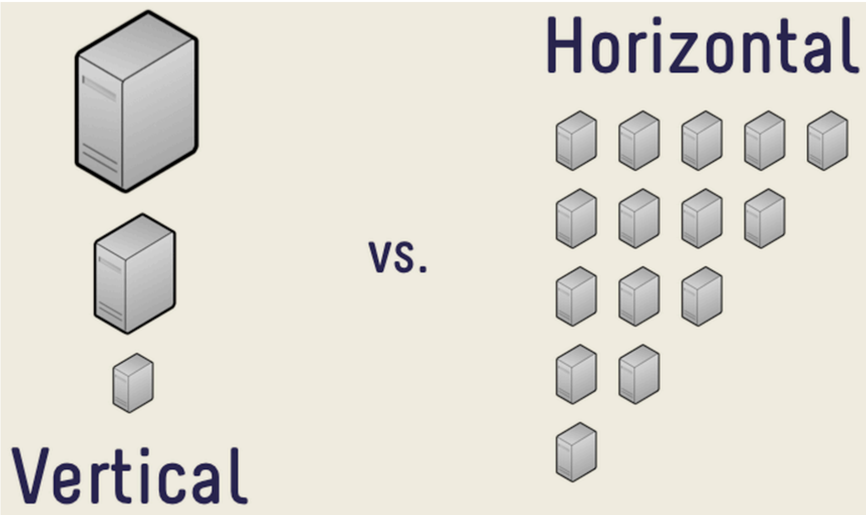


(Image source: Centric Consulting)

- Approaches to scaling:
  - Scale up (vertical scaling).
    - Upgrade the computing nodes.
      - More RAM, better CPU, larger disk drives, etc.
  - Scale out (horizontal scaling):
    - Add more computing nodes.

Storage and computing resources vs Big Data...

- Vertical scaling:
  - Advantage: improvement is universal.
  - Disadvantage: limits of speed and scale of hardware improvement.
- Horizontal scaling → parallel / distributed storage and computation.
  - Advantages: in principle infinite scaling, can tolerate hardware errors.
  - Disadvantage: improvement is confined to parallelizable tasks.
- Big data: emphasis on horizontal scaling.
  - The ability to scale vertically is not enough for handling truly big data.



Hadoop Distributed File System (HDFS)

- HDFS stores files in a cluster of name nodes and data nodes.
  - Data nodes: store the actual file data in distributed manner.
    - File is split into blocks, blocks stored on different data nodes.
    - Data blocks are also replicated in order to gain error tolerance.
  - Name nodes: file information (e.g. which data nodes store its blocks).

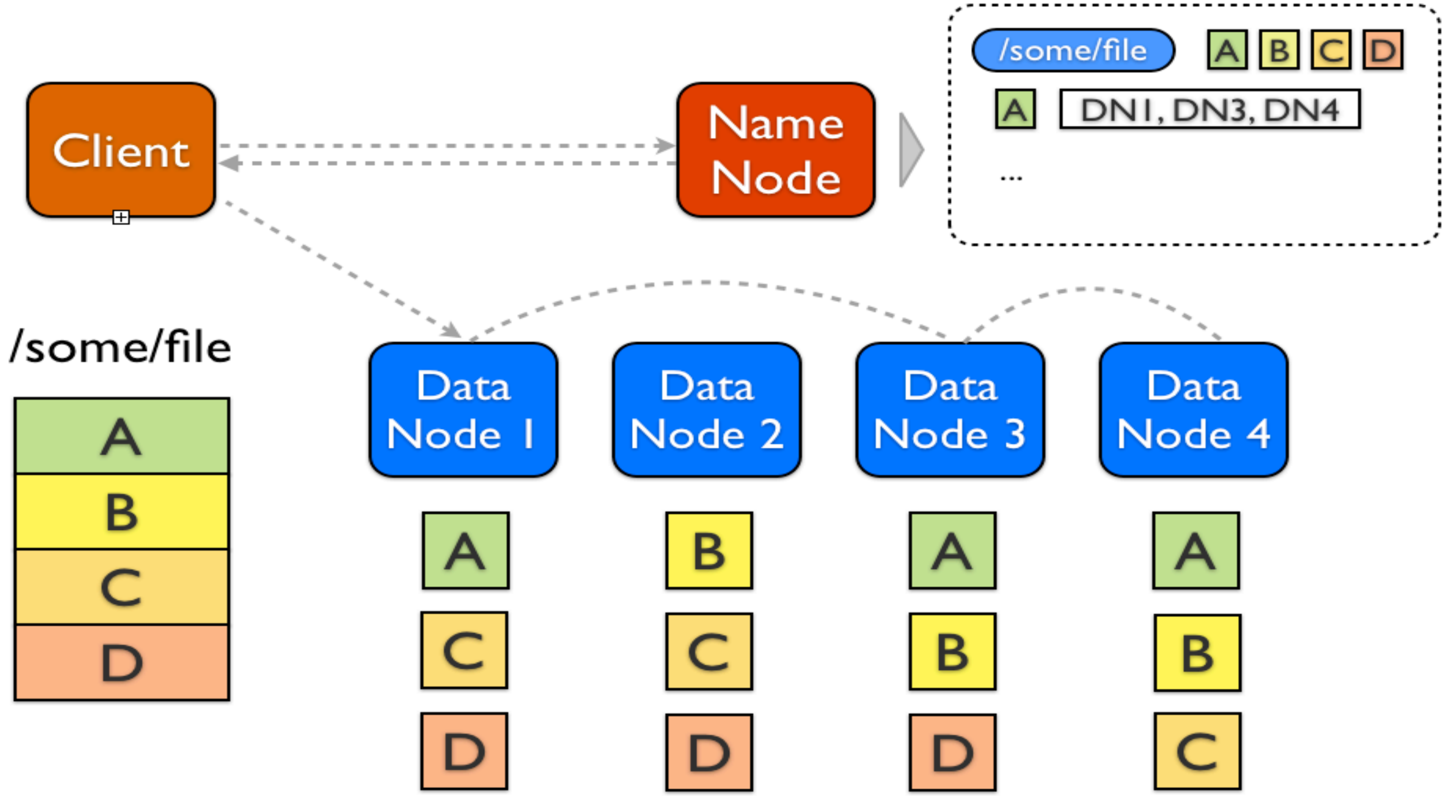


Figure source: Fernando Rodriguez Olivera / Nosqlessentials

Parallel computation

- Example 1: iterative computation of square root with *n*-decimal accuracy.
  - Note : the pseudocode is not truly in any real programming language...
  - The code does not parallelize well: strict dependence between steps.

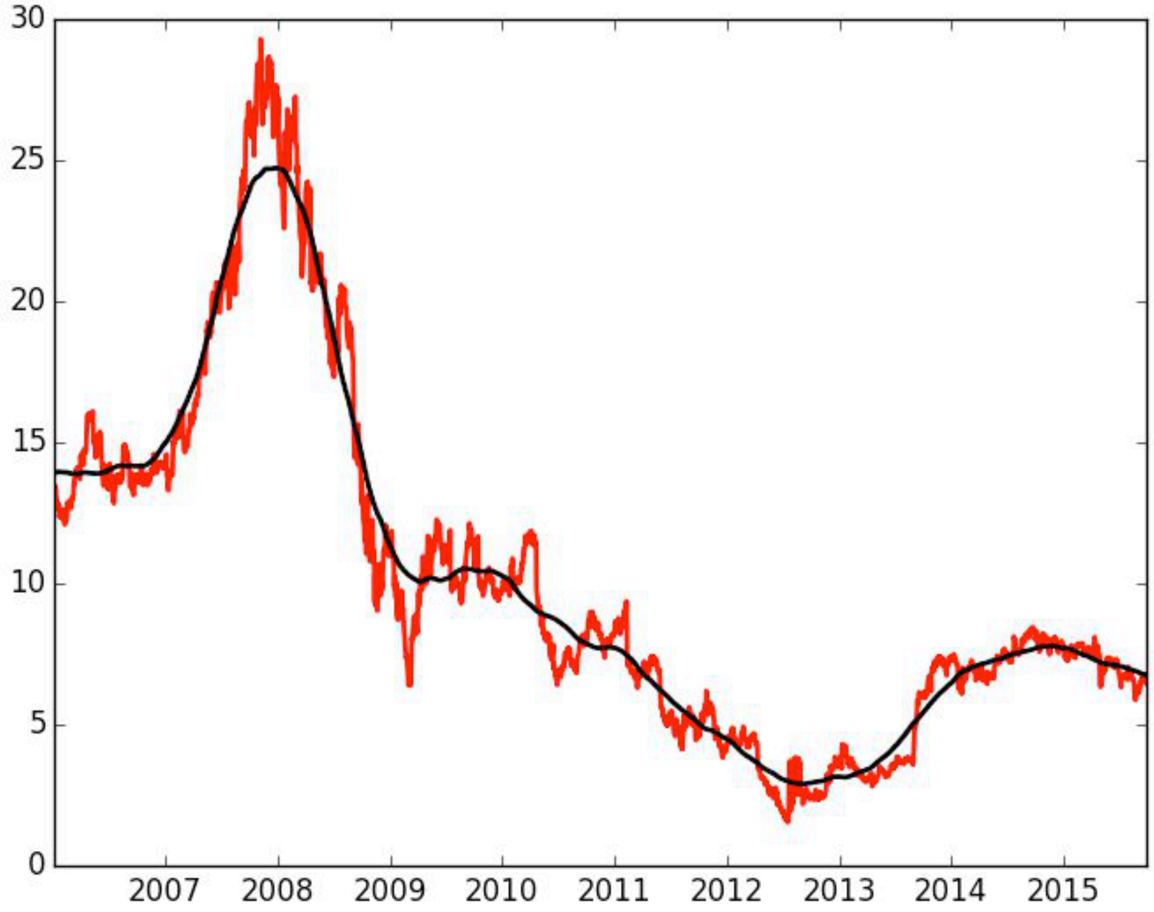
```
// Assume that BigFLoat can handle arbitrary-precision floats
BigFloat sqrt(BigFloat x, int n)
{
  low = 0
  high = max(1, x)
  // Assume that v.decimals(n) gives the
  // value of v rounded to n-decimal precision
  while(low.decimals(n) != high.decimals(n))
  { // Binary search until solution good enough
    mid = (low + high) / 2
    if(mid*mid < x)
      low = mid
    else
      high = mid
  }
  return low.decimals(n)
}
```

Sqrt(3, 2) ≈ 1.73:

Low: 0	High: 3
Low: 1.5	High: 3
Low: 1.5	High: 2.25
Low: 1.5	High: 1.875
Low: 1.688	High: 1.875
Low: 1.688	High: 1.782
Low: 1.688	High: 1.735
Low: 1.712	High: 1.735
Low: 1.724	High: 1.735
Low: 1.730	High: 1.735
Low: 1.730	High: 1.732

Parallel computation...

- Example 2: computing moving average (the black line in the plot).
  - ♦ Does parallelize well.

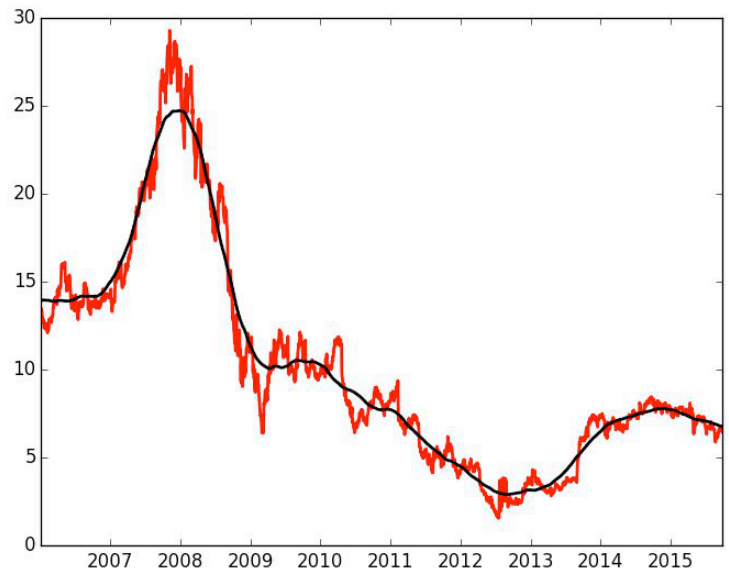



```
def m_avg(lst, i, m):
  s = max(0, i-m/2)
  e = min(len(lst), i + m/2)
  total = 0
  for i in range(s, e):
    total += lst[i]
  return total/(e-s)

def moving_average(data, m):
  result = []
  for i in range(len(data)):
    result.append(m_avg(data,i,m))
  return result
```

Parallel computation...

- Example 2: computing moving average.
  - Does parallelize well: one simple possibility is sketched below.




```
def m_avg(lst, i, m):
  s = max(0, i-m/2)
  e = min(len(lst), i + m/2)
  total = 0
  for i in range(s, e):
    total += lst[i]
  return total/(e-s)

def slice_ma(data, m, a, b):
  result = []
  for i in range(a, b):
    result.append(m_avg(data,i,m))
  return result

def sliced_moving_average(data, m, slice_size):
  result = []
  i = 0
  while i < len(data):
    result += slice_ma(data, m, i, min(i+slice_size, len(data)))
    i += slice_size
  return result
```

These function calls could be run in parallel: they are independent of each other.

Amdahl's law

- Amdahl's law:
  - Maximum parallel speedup using **n** computers: 1 / (**f** + (1-**f**)/**n**)
    - Here **f** is the fraction of code that can not be parallelized (is "serial").

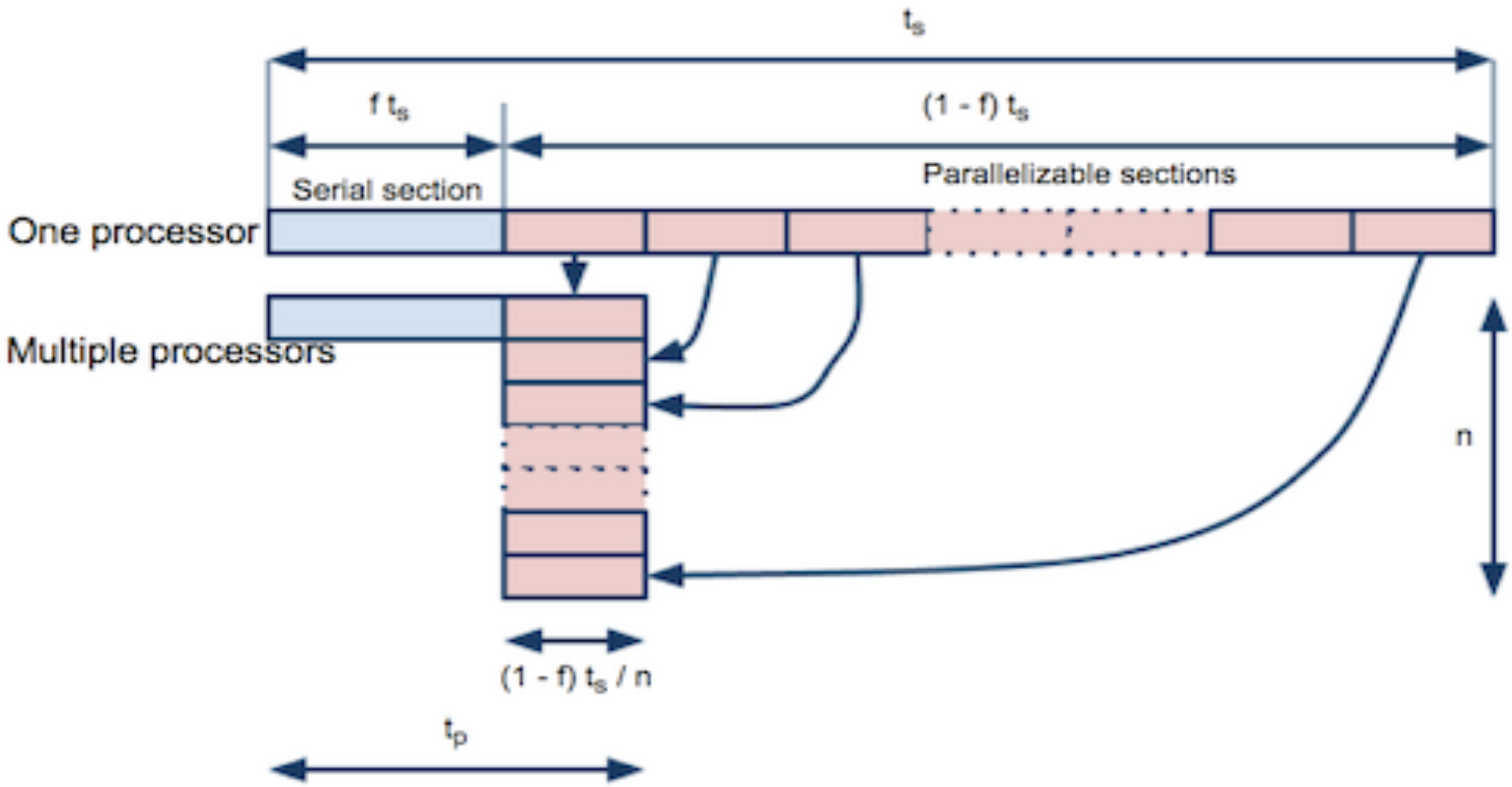


Figure source: Carlos P Sosa (IBM / Patton Fast Supercomputing Institute)

Amdahl's law...

- Amdahl's law:
  - Maximum parallel speedup using **n** computers: 1 / (**f** + (1-**f**)/**n**)
    - Here **f** is the fraction of code that can not be parallelized (is ”serial”).

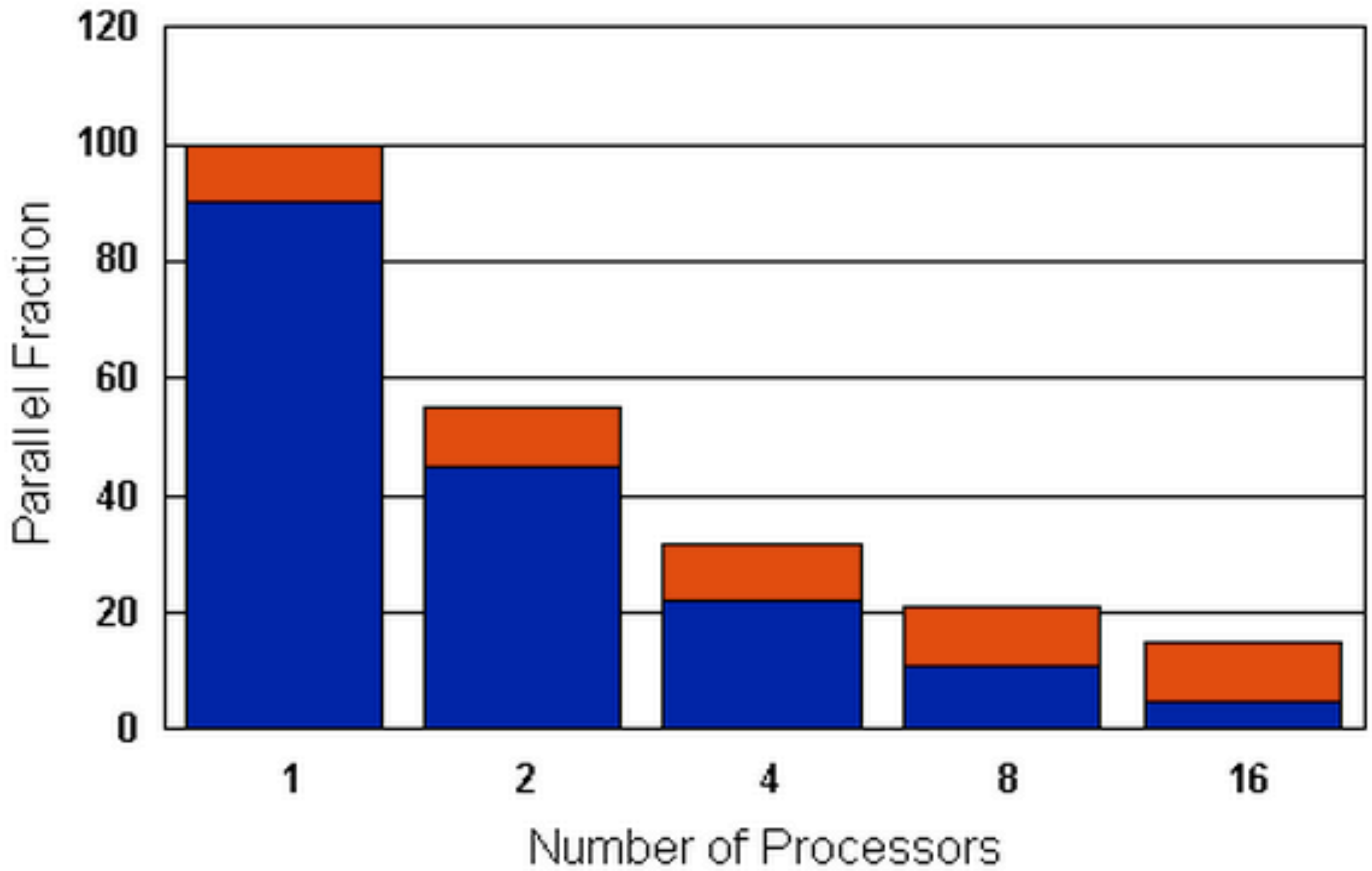


Figure source: Carlos P Sosa (IBM / Patton Fast Supercomputing Institute)

Hadoop MapReduce

- Parallel computation using two basic operations.
  - Map: group the data into chunks (that will be processed by same node).
  - Reduce: process chunks, output = the overall result (of one iteration).

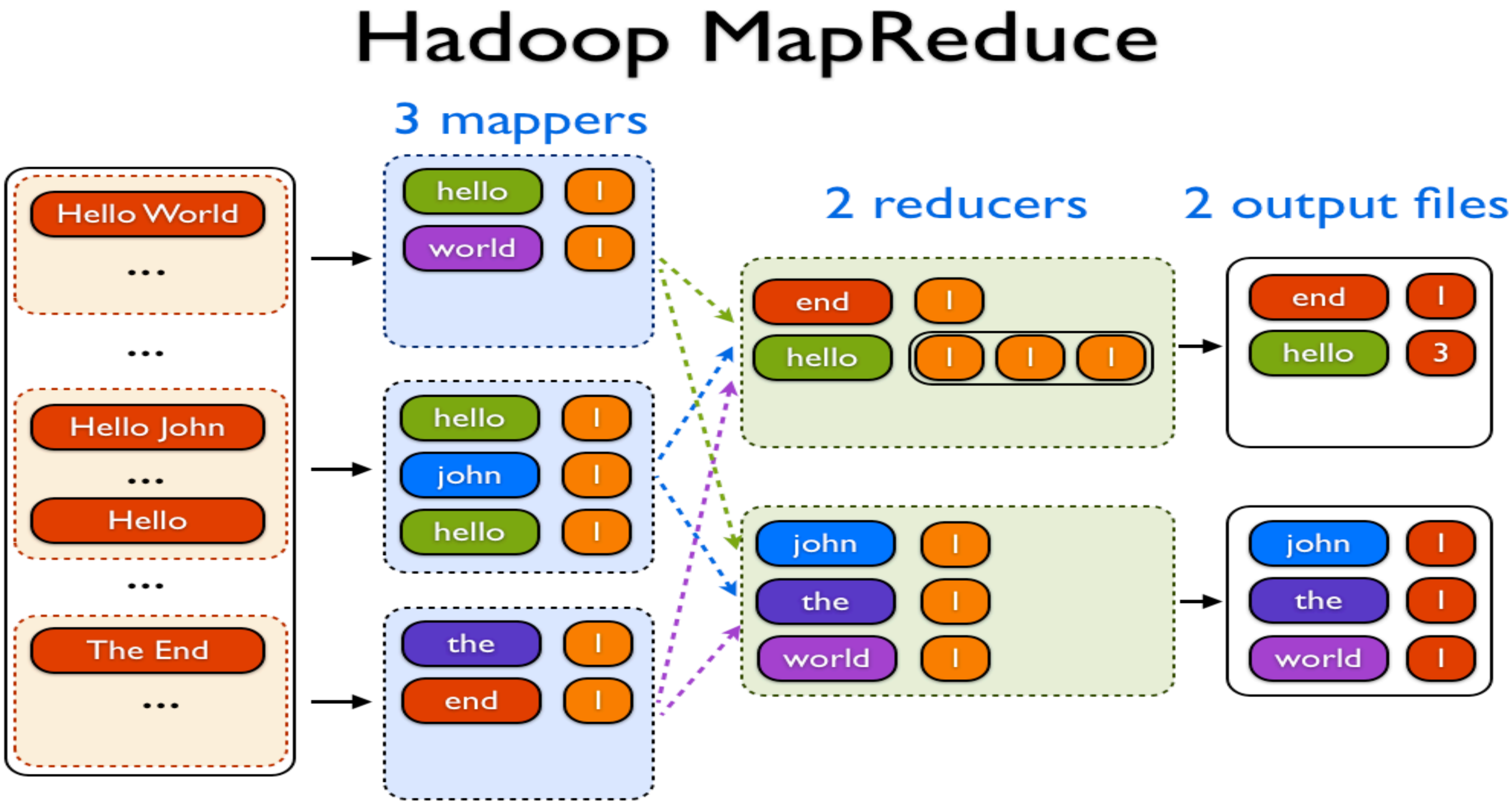


Figure source: Fernando Rodriguez Olivera / Nosqlessentials