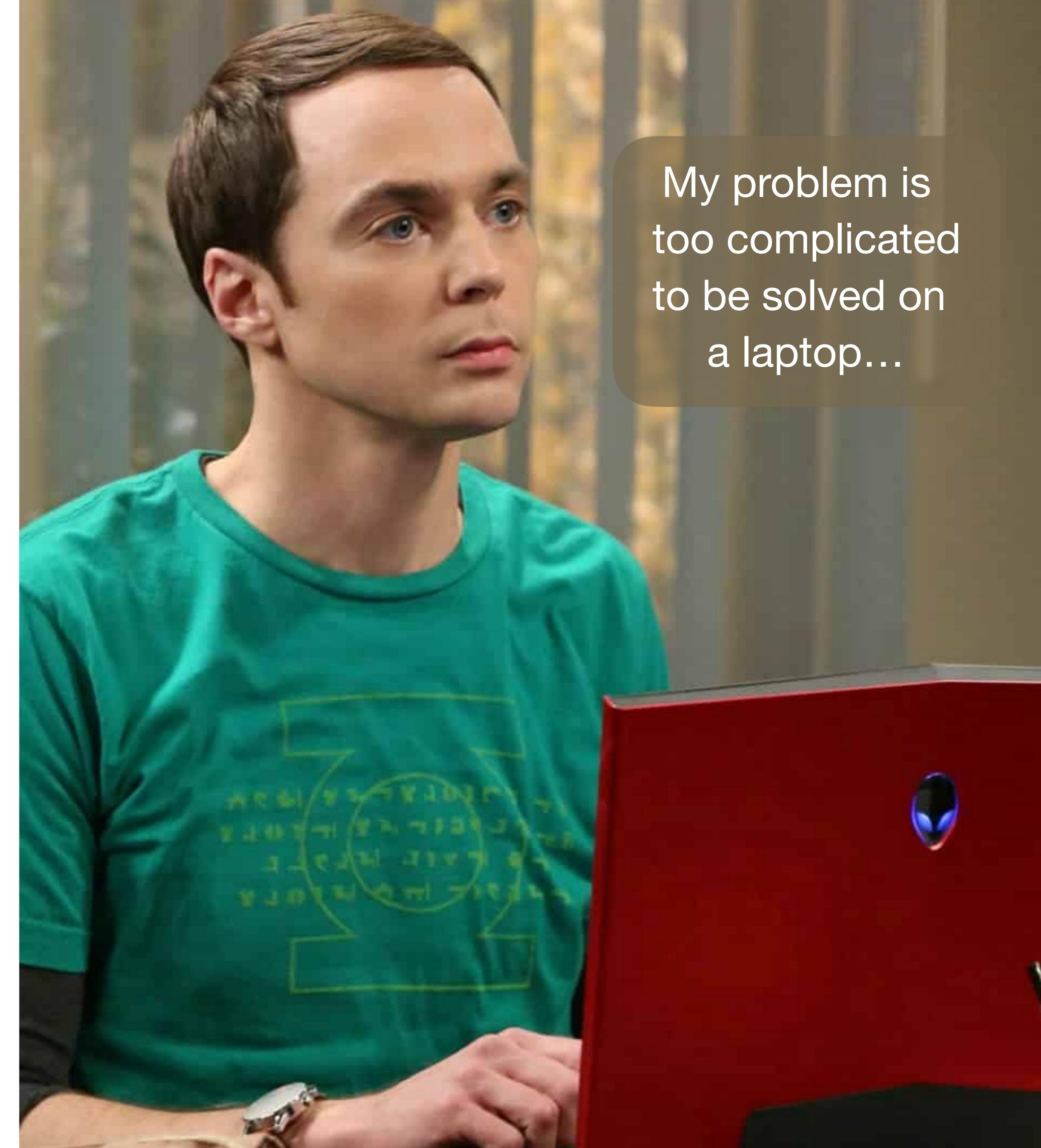


Distributed Computing beyond the Grid paradigm

How to deal with complex problems

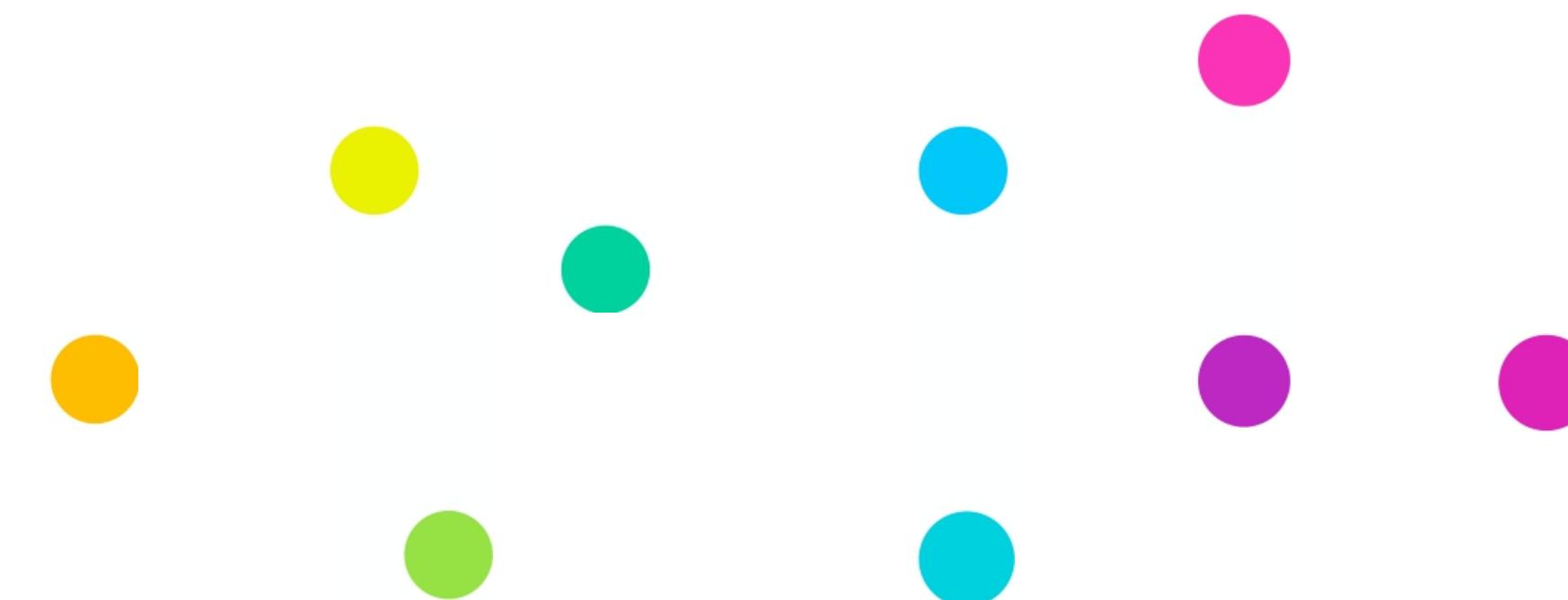
Distributed computing

- The problem is really computing intensive
- **Choose the right software stack:**
I need do distribute the calculation (vs. distributing the data, although often you need a combination of the two)
- **Choose the right infrastructure:**
should I use a High Performance Computing (HPC) platform or a High Throughput Computing (HTC) one?



Split the task in many smaller pieces

Embarrassingly parallel



Sub-tasks are
independent

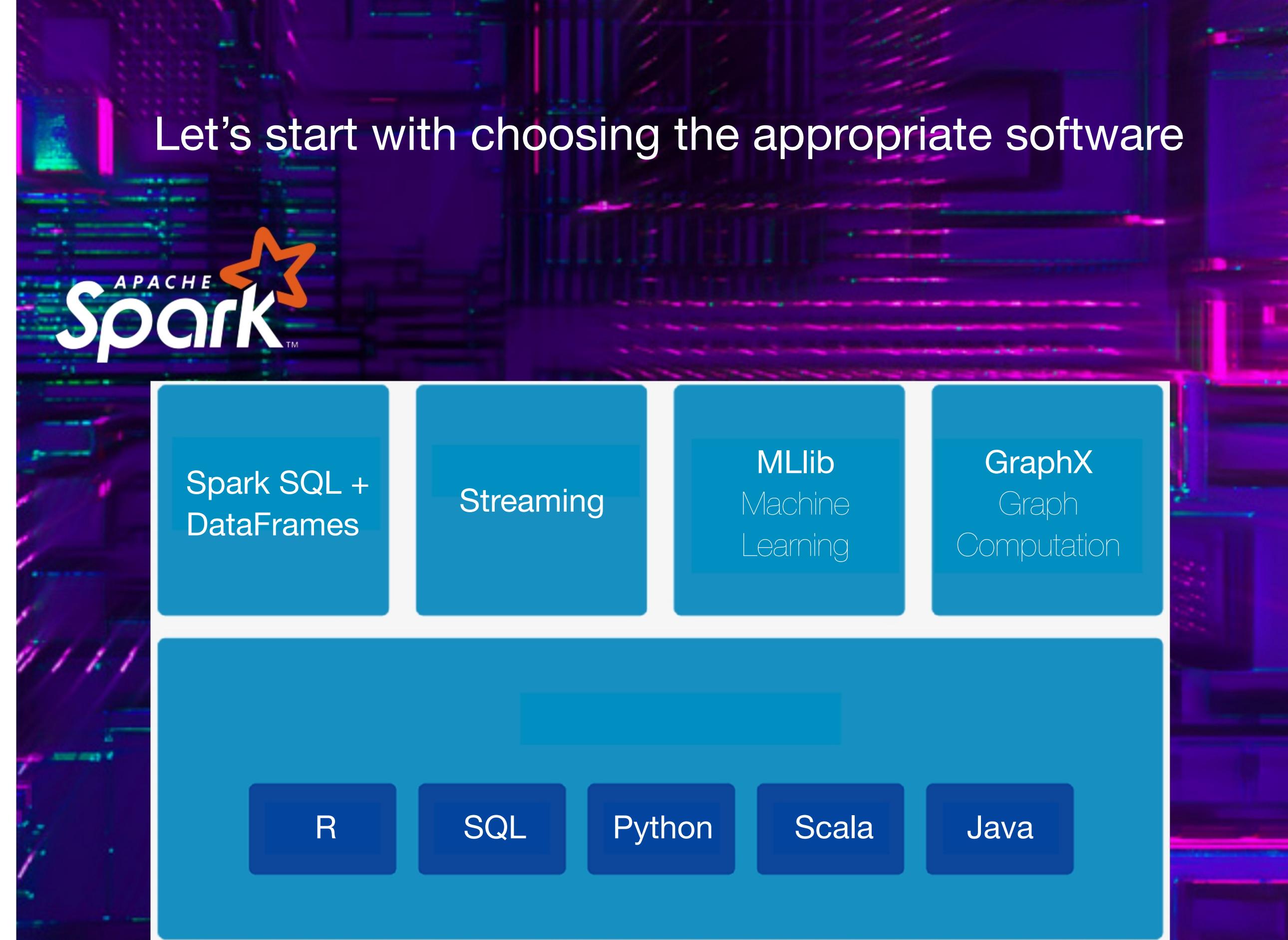
Challenging



Sub-tasks need to
communicate and
synchronise

Apache Spark

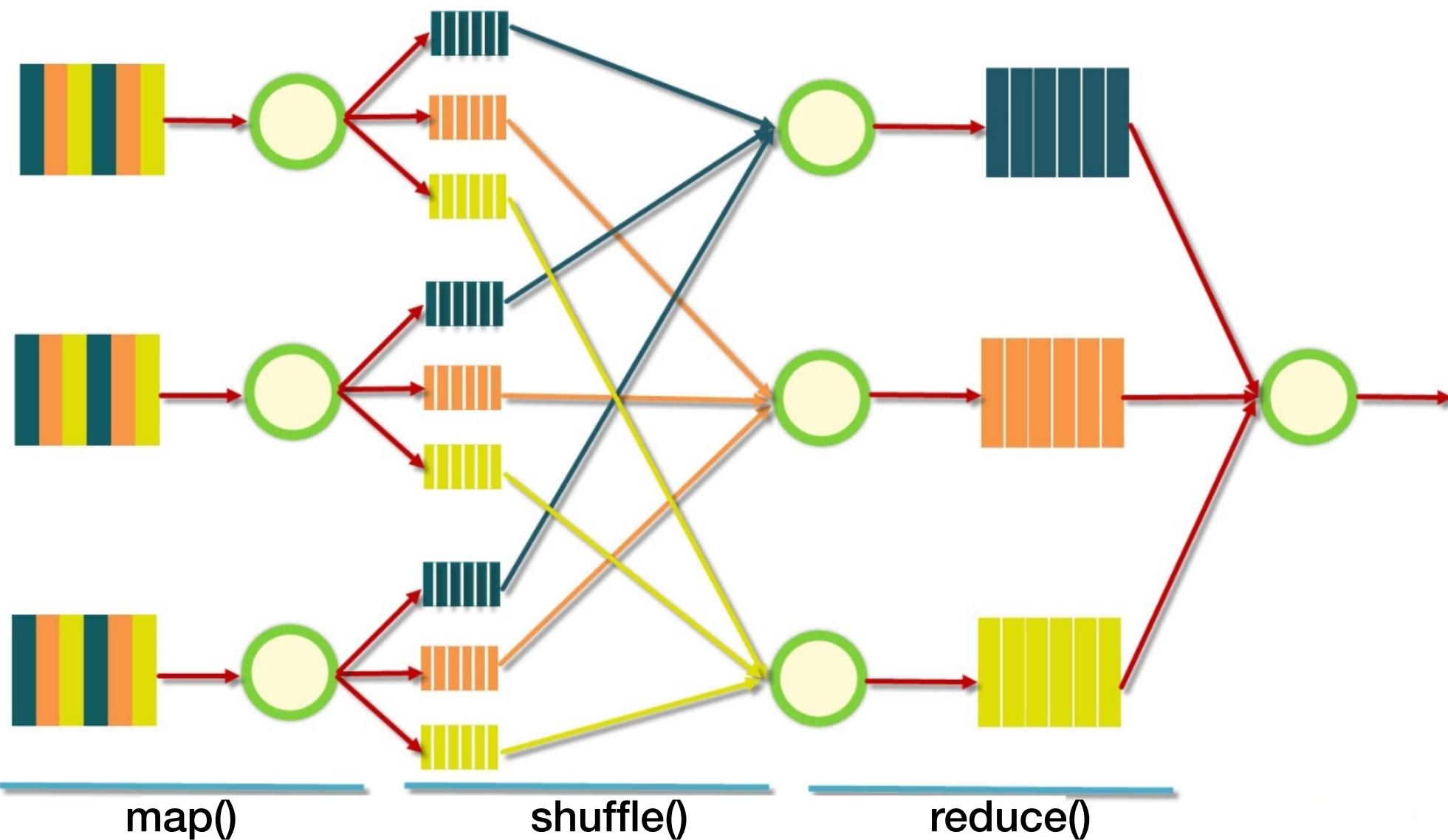
- A large-scale distributed **general-purpose** cluster-computing framework
- It provides an interface for programming entire clusters with **implicit data parallelism** and fault tolerance
- Core data processing engine + specific libraries
- These libraries can be combined in modern data **pipelines** (i.e. analytics and machine learning workloads)
- Jobs perform multiple operations consecutively, **in memory**, only spilling to disk when required
- The Spark project was born (Berkley 2009) as an improvement of the *Hadoop MapReduce* framework



PIPELINE: combine very different techniques and processes into a single, coherent whole

SQL: domain-specific language used in programming and designed for managing data held in a Relational Database Management System (RDBMS), or for stream processing in a Relational Data Stream Management System (RDSMS)

The MapReduce paradigm



See also: https://www.reddit.com/r/ProgrammerHumor/comments/55ompo/map_filter_reduce_explained_with_emojis/

map: each node applies the map function to the local data

shuffle: nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same node

reduce: nodes process each group of output data, per key, in parallel

```
function map(String name, String block):
    // name: block name
    // block: block contents
    for each color c in block:
        emit (c, 1)

function reduce(String color, Iterator partialCounts):
    // color: a color
    // partialCounts: a list of aggregated partial counts
    sum = 0
    for each pc in partialCounts:
        sum += pc
    emit (color, sum)
```

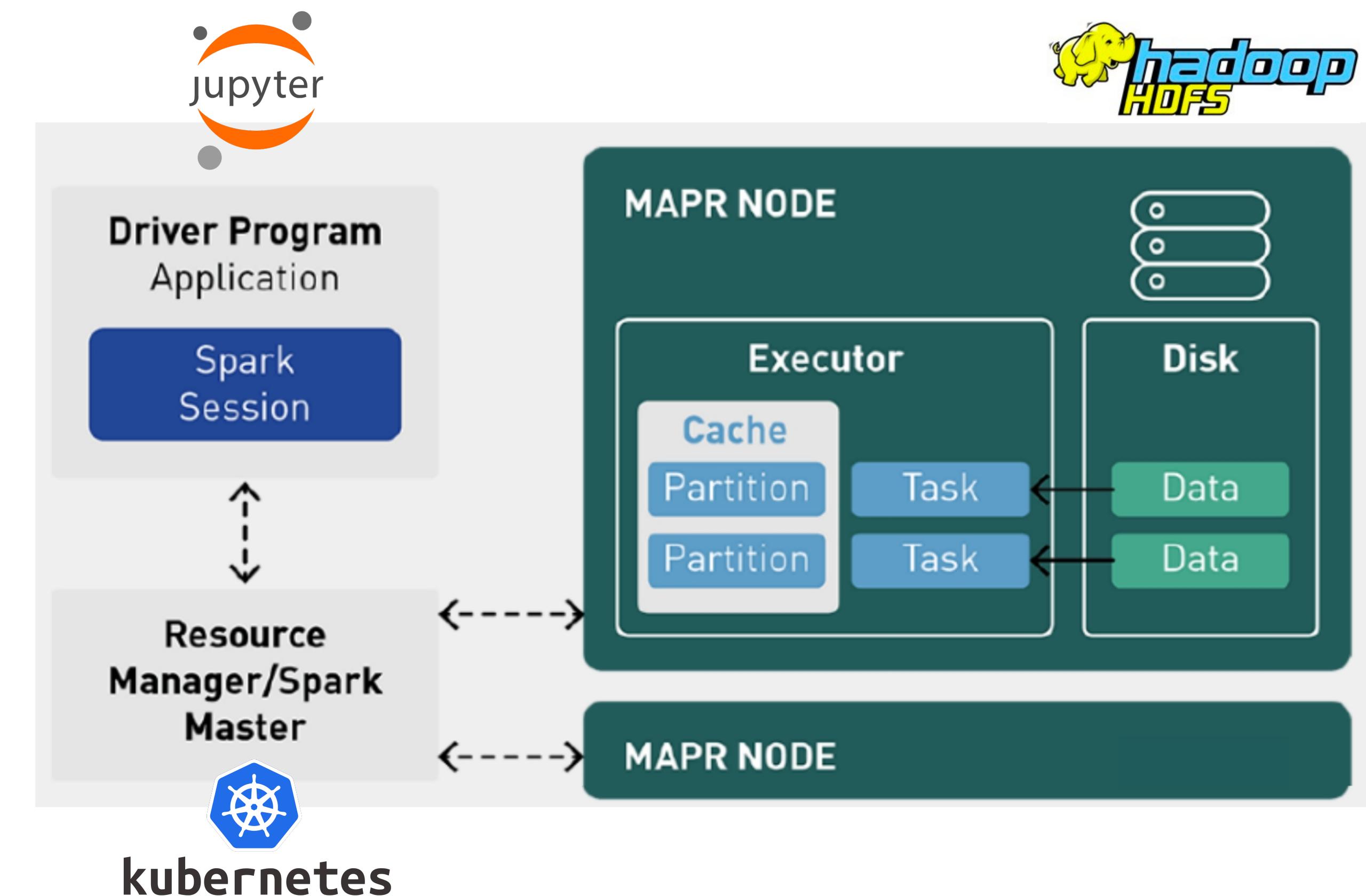
Caveats:

- Map Reduce ok for Big Data analytics
- Not ok for iterative processes (Machine Learning)
- Data not in memory

Spark vs. MapReduce

Spark:

- runs multi-threaded lightweight tasks inside of JVM (Java Virtual Machine) processes, providing fast job startup and **parallel multi-core CPU utilisation**
- **caches data in memory** across multiple parallel operations, making it especially fast for parallel processing of distributed data with iterative algorithms



Data structures

DataFrame:

- a **2-dimensional labeled data structure** with columns of potentially different types (like a spreadsheet or a table)
- a very convenient Python implementation is *Pandas*
- it's a **local** data structure

The diagram illustrates a DataFrame structure with 5 columns and 5 rows. The columns are labeled 0, 1, 2, 3, and 4, with 'Name' at index 0, 'Age' at index 1, 'Marks' at index 2, 'Grade' at index 3, and 'Hobby' at index 4. The rows are indexed 0 through 4, with row 0 labeled 'S1', row 1 'S2', row 2 'S3', row 3 'S4', and row 4 'S5'. Arrows point from labels to specific parts of the table:

- 'Column Label/Header' points to the column index 0.
- 'Index Label' points to the row index 0.
- 'Row Index' points to the bottom of the row 4 index.
- 'Column Index' points to the right of the column 4 header.
- 'Element/Value/Entry' points to the value '85.10' in the 'Marks' cell of row 0.
- 'Row' points to the bottom of the row 3 index.
- 'Column' points to the bottom of the column 2 index.

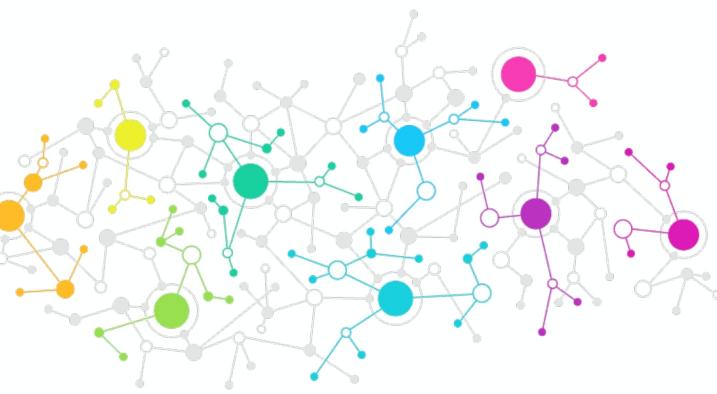
	0	1	2	3	4	
	Name	Age	Marks	Grade	Hobby	Column Index
0	S1	Joe	20	85.10	A	Swimming
1	S2	Nat	21	77.80	B	Reading
2	S3	Harry	19	91.54	A	Music
3	S4	Sam	20	88.78	A	Painting
4	S5	Monica	22	60.55	B	Dancing

Spark Resilient Distributed Dataset (RDD):

- a read-only collection of objects (data) **partitioned across a set of machines** that can be re-built if a partition is lost
- it's a **memory abstraction**
- can be built from:
 - parallelising an existing collection in your driver program
 - referencing a dataset in an external storage system (i.e. HDFS)

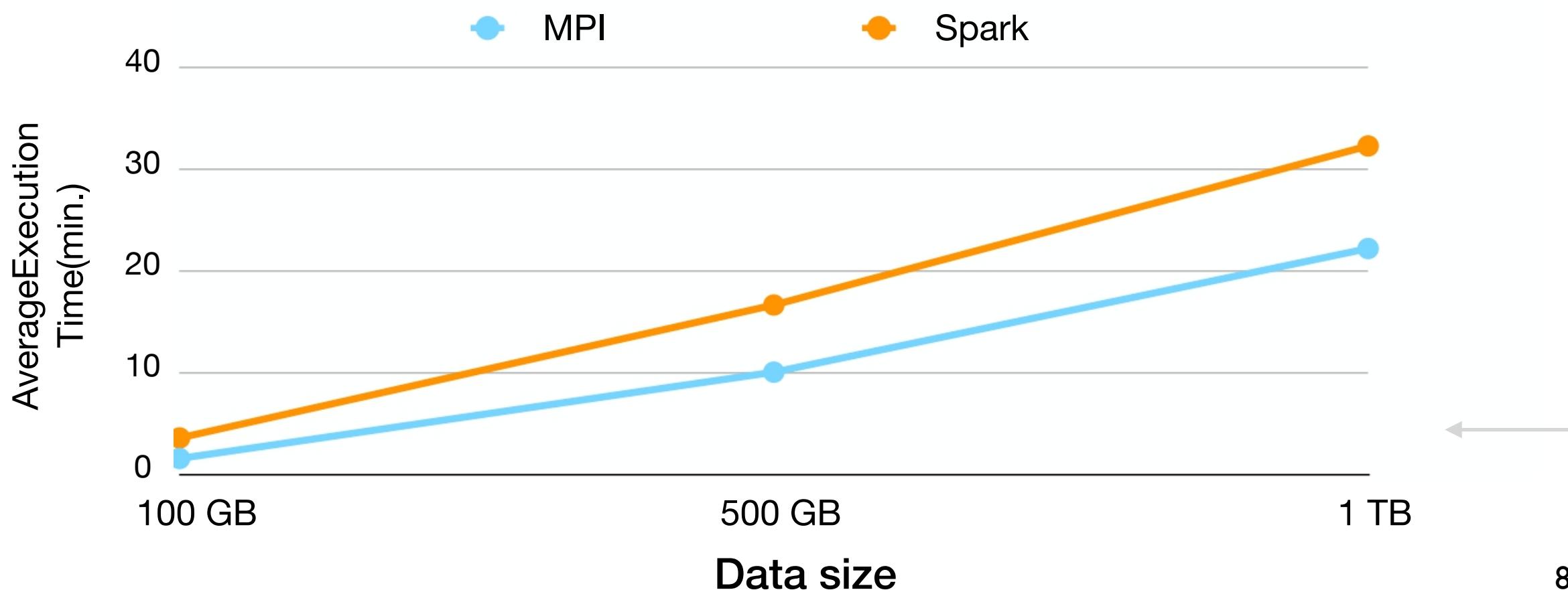
Spark vs. MPI

What is the best way to solve this?



Definitely MPI (for computing intensive workloads):

- Message Passing Interface (MPI) is a specification
- It defines a message passing model to facilitate the transport of data between distributed processing elements
- Specific implementations, i.e. OpenMPI or MPICH
- Very low latency and injection rate for short messages and high bandwidth (HPC's favorite)



DATA INTENSIVE
WORKLOADS

Why Spark then...

- Because it provides a common runtime for big data processing
- Availability and fault tolerance
- To be preferred $O(1\text{TB}) \rightarrow$ I/O access delays start to affect MPI
- Supports real-time data streaming

Kumar, Deepa & Rahman, M. (2017). Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis. *Journal of Computer Science*. 13. 781-794. 10.3844/jcssp.2017.781.794.

What about scalability?

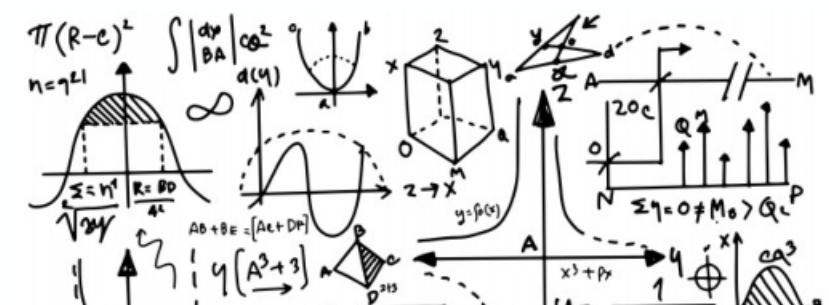
Scalability is the property of a system to deliver greater computational power when the amount of resources is increased.

Follow-up: <https://www.researchgate.net/publication/228367369> Reevaluating Amdahl's law and Gustafson's law

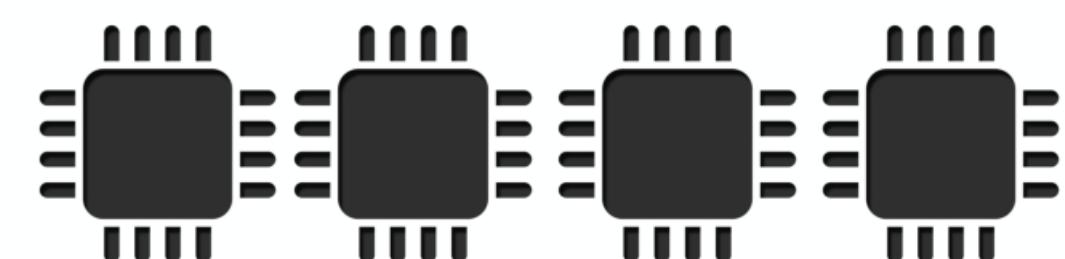
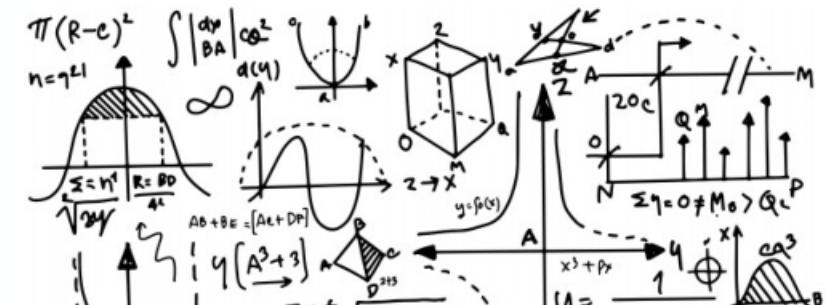
Strong scaling

Fixed problem size

t_1



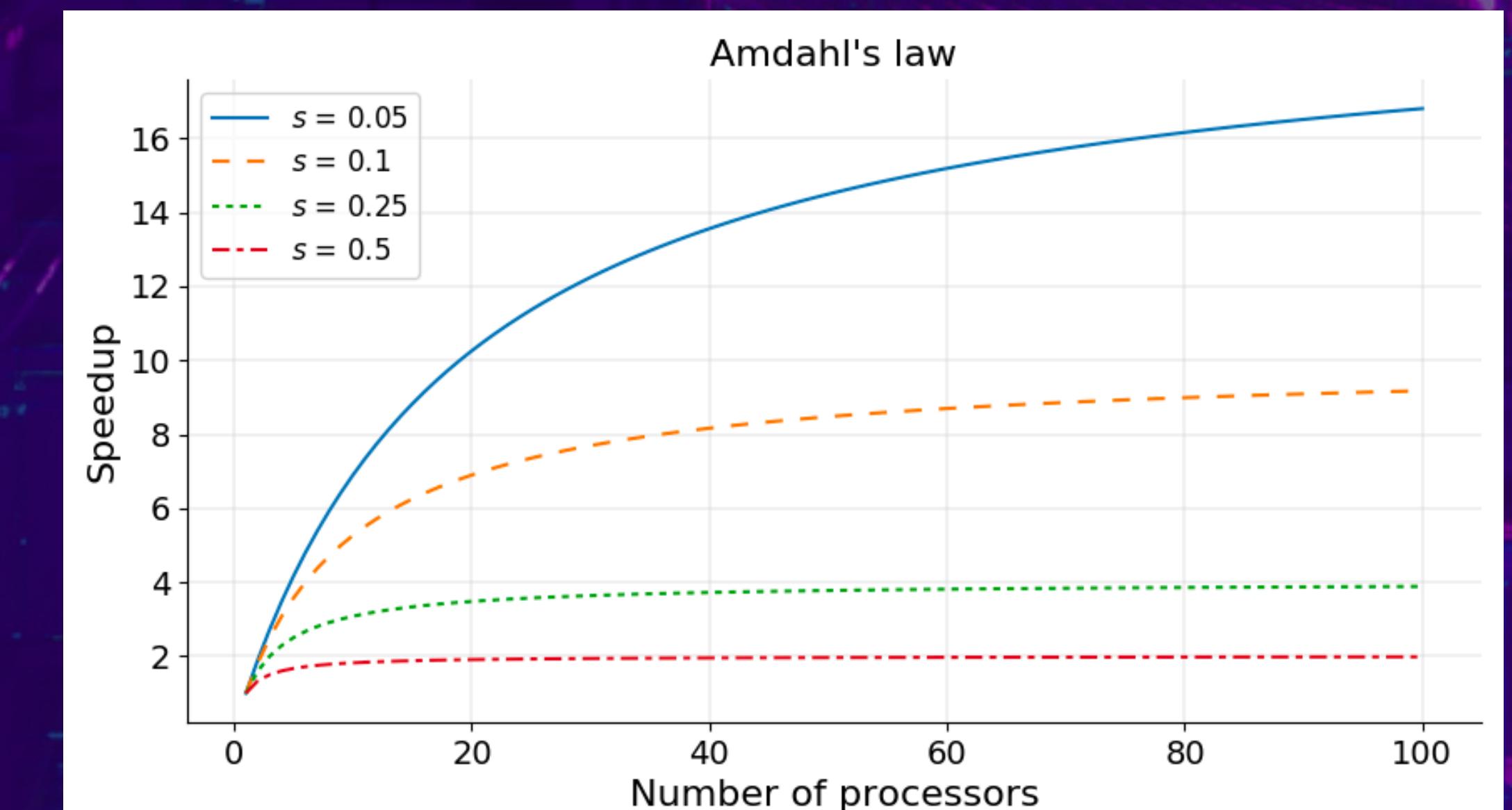
t_N



$$\text{Speedup (ideal)} = t_1 / t_N = N$$

Amdahl's law

$$\text{Speedup} = t_1 / t_N = 1/(s+p/N)$$



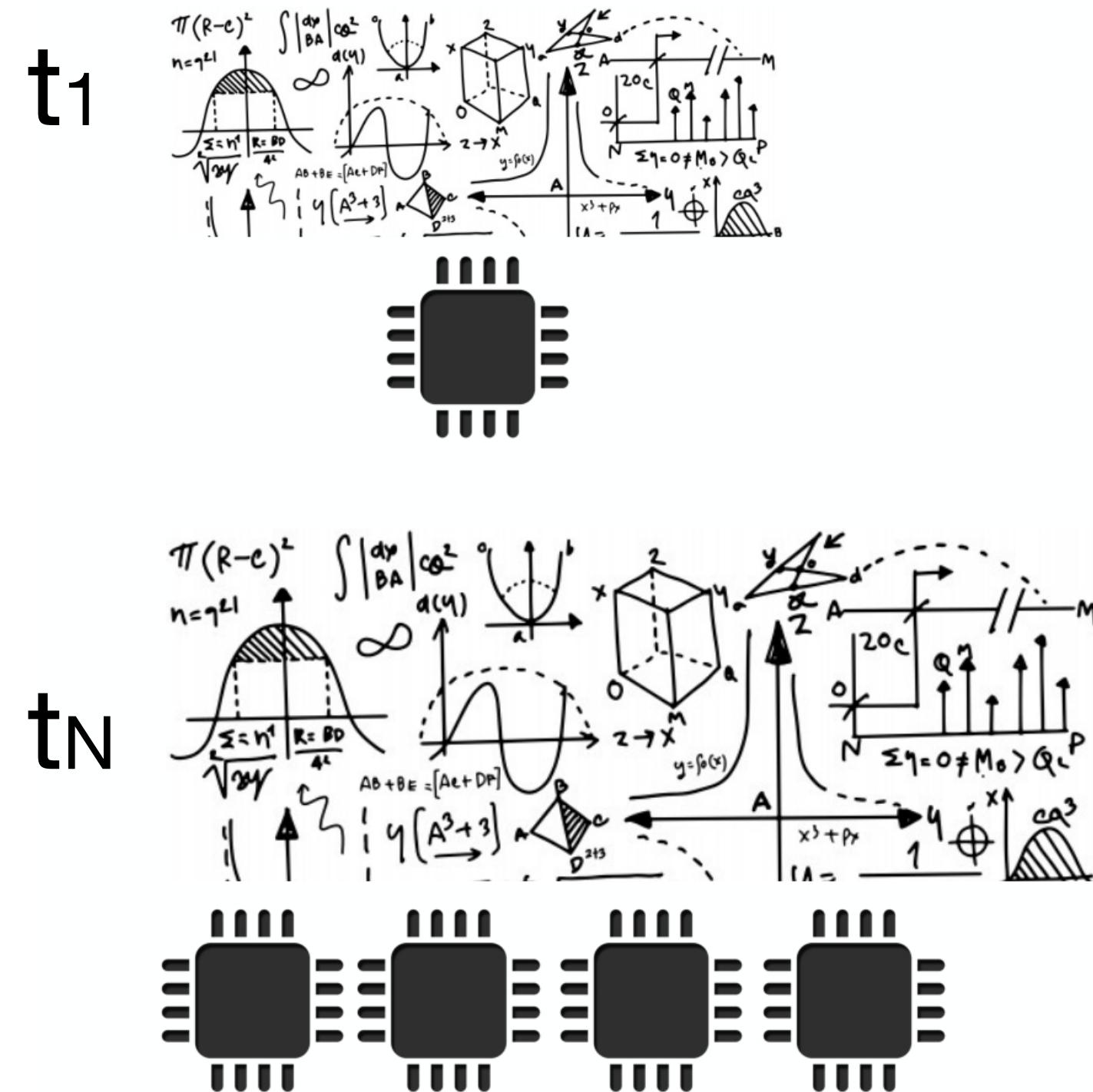
s = fraction of execution time spent in serial part

p = fraction of execution time spent in parallel part

N = number of processors

Weak scaling

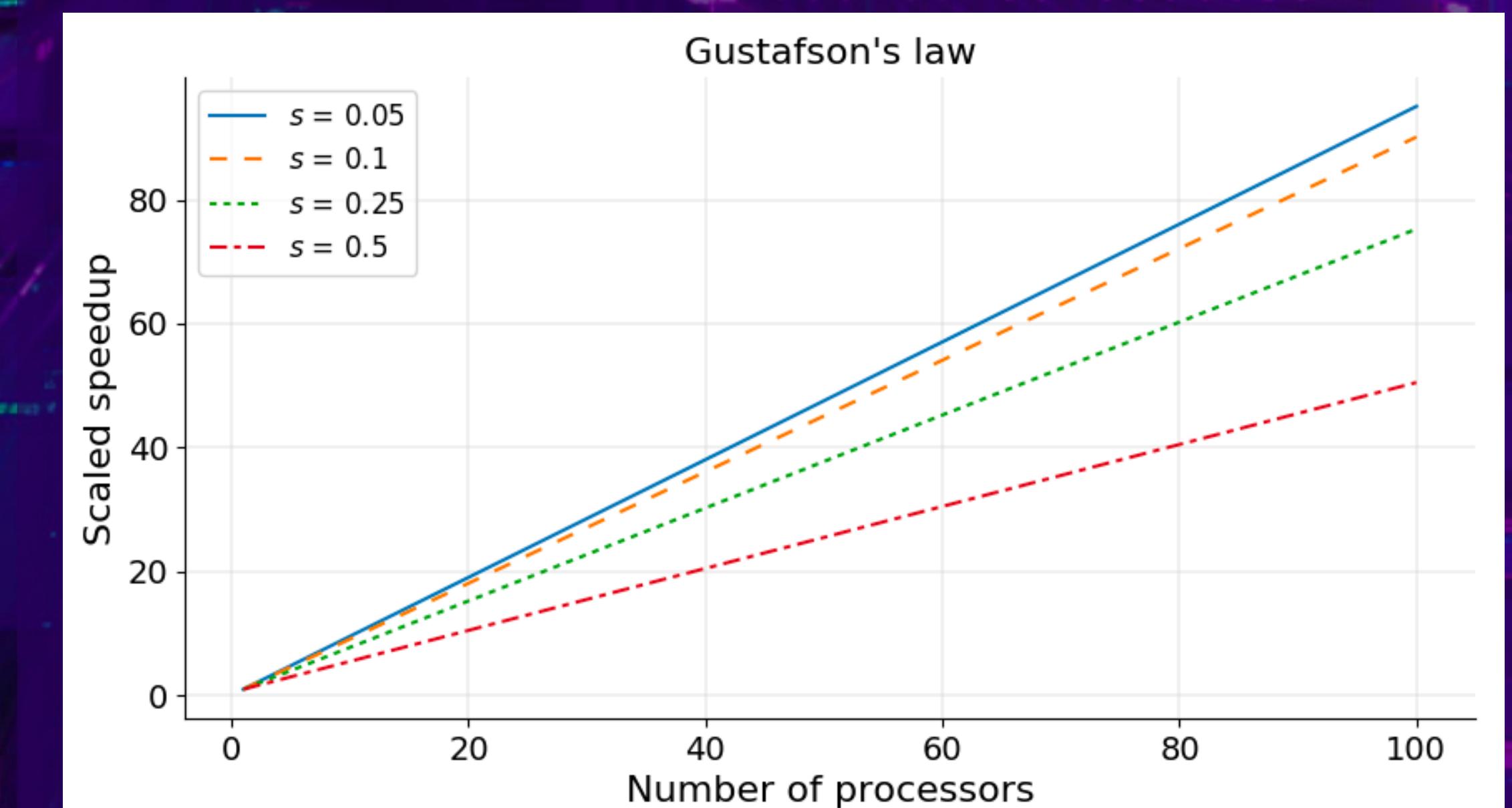
Increasing problem size



$$\text{Scaled Speedup (ideal)} = t_1 / t_N = N$$

Gustafson's law

$$\text{Scaled Speedup} = t_1 / t_N = s + pN$$



s = fraction of execution time spent in serial part
 p = fraction of execution time spent in parallel part
 N = number of processors

To take home

- Understand your problem's features:
is it embarrassingly parallel or not?
- Choose the right cluster framework:
 - MPI for computing intensive tasks
 - Spark for data intensive tasks
- Understand the scaling behaviour:
more cores does not always mean
shorter execution times

What about the Grid?

Geographically distributed facility for
High Throughput Computing.

Can accomodate embarrassingly
parallel jobs.

Image from Gartner (March 2016)