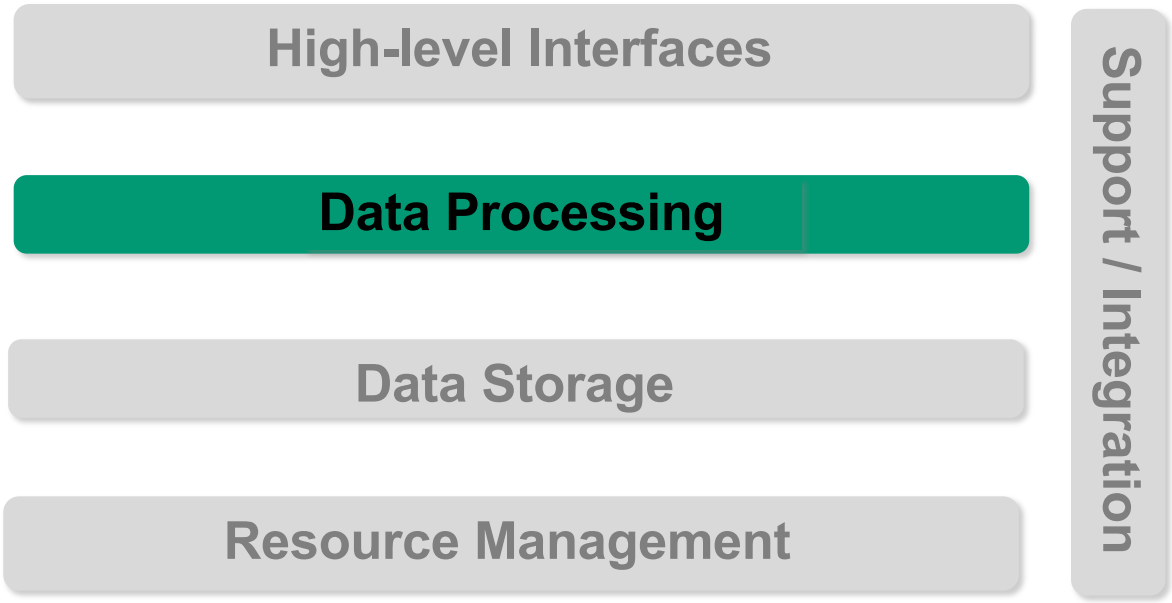
TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Data Processing Frameworks

Corso di Sistemi e Architetture per Big Data
A.A. 2025/26
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack

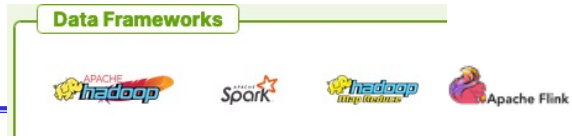


ML, AI and Data landscape (2024)

- Some open-source frameworks

https://mattturck.com/mad2024

Processing Big Data

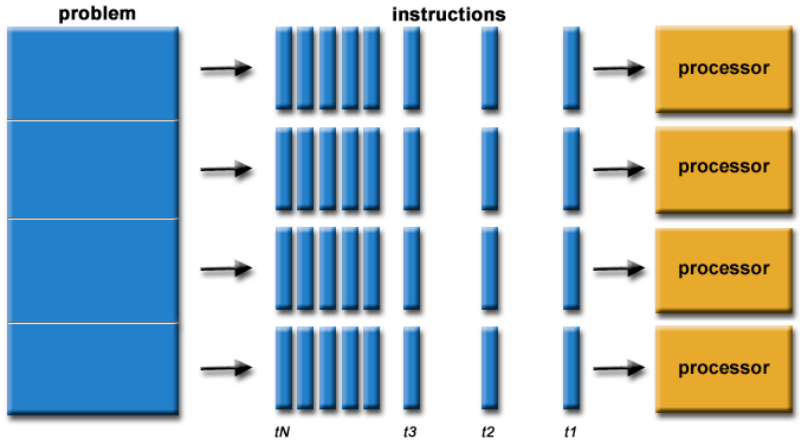


- Distributed processing frameworks to perform computation on data at scale
  - **Batch processing**: store and process datasets at massive scale
    - Handle **Volume** + **Variety**
    - Focus on throughput, not latency
    - MapReduce & Hadoop, Spark
  - **Data stream processing**: process data in real-time as it is generated, without storing first
    - Handle **Velocity**
    - Focus on latency

Focus of upcoming lessons

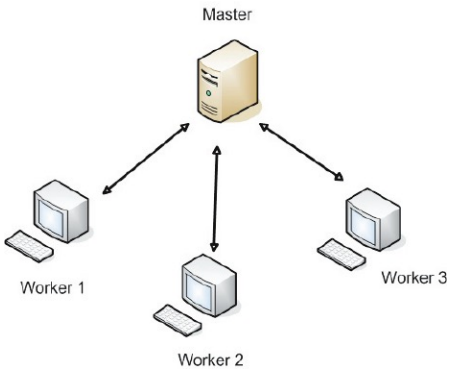
Parallel programming: background

- Parallel programming
  - Simultaneous use of multiple computing resources (e.g., nodes, cores) to solve a problem
  - How? Break processing into independent parts that are executed concurrently on multiple computing resources



Parallel programming: background

- Simplest environment for parallel programming: **master/worker** architecture
  - Master
    - Receives input data and splits it into chunks based on the number of workers
    - Distributes chunks evenly to workers
    - Collects results from all workers
    - Combines results into the final output
  - Workers:
    - Receive assigned data chunks from master
    - Perform processing
    - Send results back to master



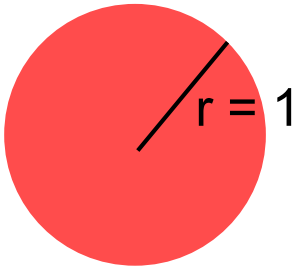
Parallel programming: background

- Several styles of parallel programming
- The most widely used is Single Program, Multiple Data (**SPMD**)
  - *Single Program*: all computing resources execute the same program simultaneously
  - *Multiple Data*: each computing resource works on different portions of data

Example: Pi estimation

- Estimating π with Monte Carlo method
  - Monte Carlo methods: class of algorithms that use repeated random sampling
  - Used to approximate numerical results
  - Useful when exact computation is difficult
- We first consider the sequential approach
- Then a parallel and faster version

Example: Pi estimation



- π is the area of a circle with radius equal to 1
- How to estimate π?

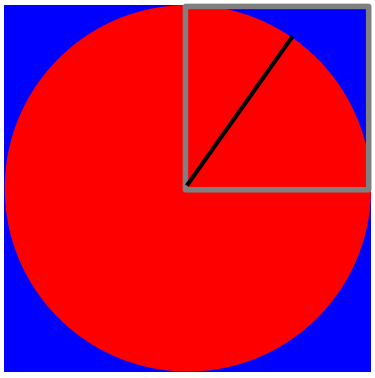
1. Randomly generate a large number of points inside the circumscribed unit square
2. Count how many points fall inside the inscribed circle
3. Use the ratio to approximate π

Example: Pi estimation

- Using formula:

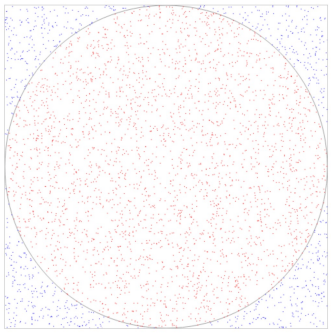
$$\frac{\pi}{4} \approx \frac{N_{inner}}{N_{total}}$$

$$\pi \approx 4\frac{N_{inner}}{N_{total}}$$

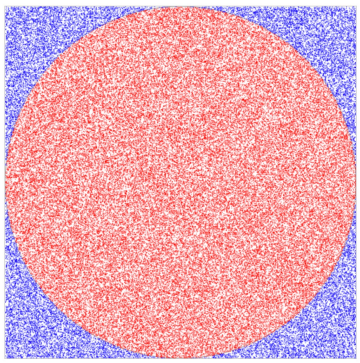


- The more points generated, the higher the accuracy of estimation

Example: Pi estimation



Total Number of points: 4163
Points within circle: 3259
Pi estimation: 3.13140



Total Number of points: 95770
Points within circle: 75212
Pi estimation: 3.14136

See animation at https://academo.org/demos/estimating-pi-monte-carlo/

Example: Pi estimation

- How to get an accurate and faster estimation of π?
- From sequential to parallel computation
- Use a master/worker architecture
  - Each worker generates a subset of random points, classifies points as inside or outside the circle, and counts how many end up inside the circle
  - The master aggregates (sums) the total points and the points inside the circle from all workers

Example: Pi estimation

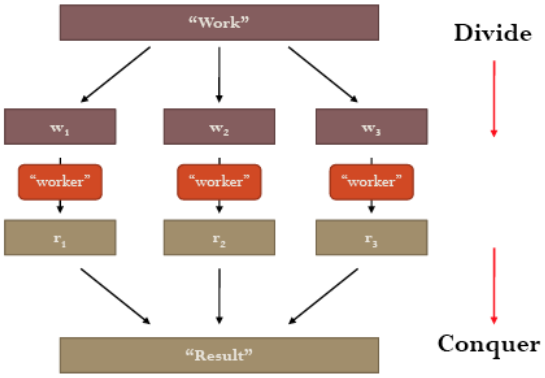
- Turning it into Spark code (details in upcoming lesson)
  - `samples` and `within_circle` are distributed collections processed in parallel
  - `samples`: all generated points
  - `within_circle`: only the points inside the circle

```
def inside(p):
  x, y = random.random(), random.random()
  return x*x + y*y < 1

samples = sc.parallelize(range(0, NUM_SAMPLES))
within_circle = samples.filter(inside)
count = within_circle.count()
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```


Key ideas behind MapReduce and Spark: Divide and conquer

- Feasible approach to tackle large-data problems
  - Partition a large problem into smaller sub-problems
  - Solve sub-problems independently and in parallel
  - Combine intermediate results from each worker node



Implementation details are complex

Divide and conquer: how?

- Decompose the original problem in smaller, parallel tasks
- Schedule tasks on workers distributed in a cluster, keeping into account:
  - Data locality
  - Resource availability
- Ensure workers get input data
- Coordinate synchronization among workers
- Share partial results
- Handle failures

Key ideas behind MapReduce and Spark: Scale out, not up

- For data-intensive workloads, use many commodity servers instead of a few high-end servers
  - Cost of super-computers is not linear
  - Data center efficiency
- Processing data is quick, I/O is slow
- Shared-nothing is preferred
  - Each node is independent, no shared memory or storage
    - ✓ Scalability and fault tolerance
  - Shared-state: nodes share a common/global state
    - ✗ Requires synchronization between nodes
    - ✗ Risks of deadlocks, bottlenecks (e.g., bandwidth to access stored data)

Data exchange patterns

- How nodes exchange data and coordinate work in Big Data frameworks
- Shuffle
  - Redistributes data across nodes based on keys
  - Use case: grouping, joins, sorting
  - Crucial step in Spark (`groupBy`, `reduceByKey`, joins) and Flink
- Reduce/aggregation
  - Combines partial results into a final result
  - Used in: MapReduce, aggregations in Spark/Flink

Data exchange patterns

- How nodes exchange data and coordinate work in Big Data frameworks
- Broadcast (one-to-all)
  - Send data to all workers
  - Used in: broadcast joins (Spark)
- Scatter (data partitioning)
  - Splits data across nodes
  - Used in:
    - initial data loading (HDFS blocks in Hadoop)
    - partitioned datasets in Spark

Data exchange patterns

- How nodes exchange data and coordinate work in Big Data frameworks
- Gather
  - Collect results back to a single node
  - Used in:
    - Final result collection (`collect`, `count` in Spark)
- Other common patterns
  - Point-to-point: direct data transfer between two nodes
  - Pipeline: data stays within the same worker, transformations are applied to data partitions locally
  - Publish-subscribe: primarily used for data ingestion

Data exchange patterns

- How nodes exchange data and coordinate work in Big Data frameworks
- All-reduce
  - Each node starts with its own value
  - Global operation (e.g., sum, average, max) is performed across all nodes
  - Every node receives the same final result
  - Common in distributed ML (e.g., Spark ML) and DL frameworks (e.g., TensorFlow)
  - Efficient implementations avoid a central bottleneck:
    - Tree-based (hierarchical aggregation)
    - Ring all-reduce (very common in ML systems)