4

Fundamentals of Scaling ML

Topics

- Introduction to Scalability in ML
- Challenges in Scaling ML Algorithms
- ML Libraries and its parallel Capabilities
- NUMBA and Dask
- Summary

“Introduction to Scalability in ML

Introduction to Scalability in ML

Horizontal

Parallel

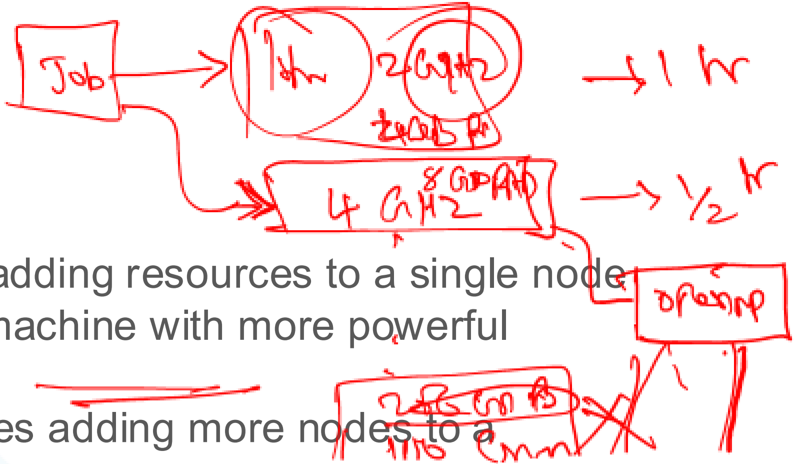
Scalability

- **Hardware Scalability**
  - Ability of the computing hardware, such as CPUs, GPUs, memory, and storage systems, to scale up or scale out to support larger computations.
  - Scaling up involves adding more resources to a single node (e.g., more CPUs, memory)
  - Scaling out involves adding more nodes to a distributed system.

Vertical

- **Software Scalability**
  - Software's ability to efficiently utilize the available hardware resources as those resources are added or expanded.
  - This includes algorithms and software architectures that can take advantage of parallel and distributed computing environments.
- **Performance Scalability**
  - Ability of a system to maintain or improve its performance efficiency as the workload increases.
  - Ideally, a scalable system should show a linear increase in performance with a linear increase in resources, though this is challenging to achieve in practice

Introduction to Scalability in ML



Scalability

- **Vertical vs. Horizontal Scalability**
  - Vertical scalability (scaling up) refers to adding resources to a single node in a system, typically involving a single machine with more powerful capabilities.
  - Horizontal scalability (scaling out) involves adding more nodes to a system, expanding the system's capacity by including more machines or instances.
- **Load Scalability**
  - The capability of a system to manage and balance the load as the demand or workload increases.
  - This often involves techniques and mechanisms for load balancing and distributing computational tasks across a system's resources to prevent any single node from becoming a bottleneck.

Introduction to Scalability in ML

Scalability in ML

data to

- The capability of a machine learning system to efficiently process increasing volumes of data or to accommodate more complex models, without a proportional increase in resource consumption or degradation in performance.
- Ensures that as data grows or model complexity increases, the system can adapt and manage the additional workload effectively, maintaining or improving performance metrics.

size of ML model

Introduction to Scalability in ML

Why Scalability Matters?

- **Handling Larger Datasets**
  - In the era of big data, being able to scale means making feasible the analysis of vast datasets that can lead to more accurate and insightful models.
- **Complex Models Requirement**
  - Advanced models, especially deep learning networks, require significant computational power and data, making scalability a necessity for cutting-edge ML research and applications.
- **Operational Efficiency**
  - Scalable systems can optimize resource utilization, reducing costs and improving the efficiency of machine learning operations.

“Scalability Metrics

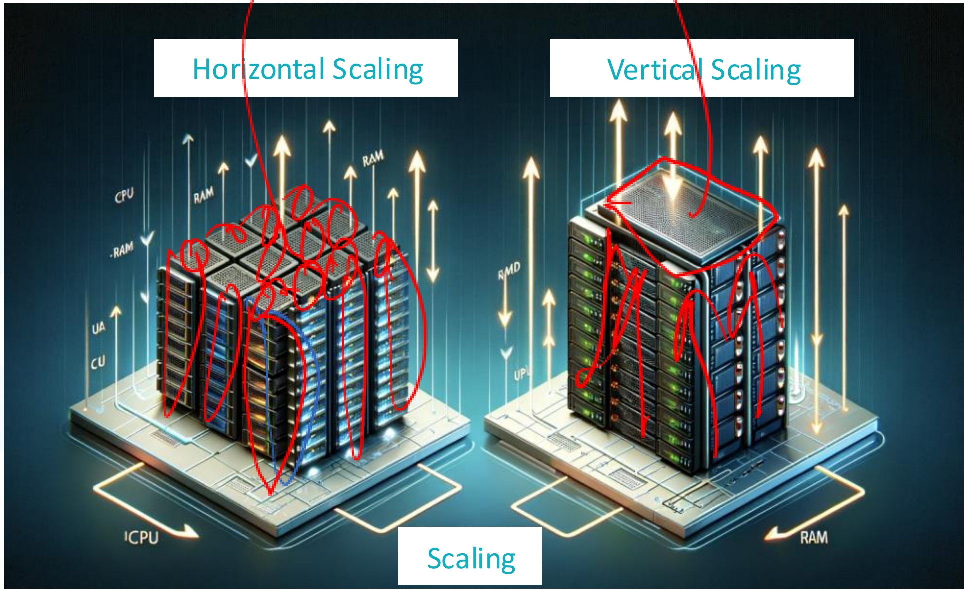
Introduction to Scalability in ML

Methods of Scaling

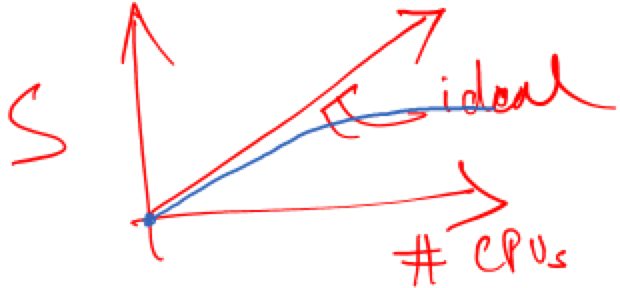
- **Horizontal Scaling (Scaling Out/In)**
  - Involves adding more machines or nodes to a network to distribute the processing and memory load.
  - Especially suited for cloud computing environments, it allows for flexibility in managing fluctuating workloads.
- **Vertical Scaling (Scaling Up/Down)**
  - Entails upgrading the existing hardware's capacity (CPU, RAM, storage) of a system to handle more tasks simultaneously.
  - It's often quicker to implement but has physical and cost-related limitations.

MPI + CUDA (HIP)

OpenMP + CUDA/HIP



Introduction to Scalability in ML



Key Metrics for Scalability

- **Speedup**
  - Speedup refers to the ratio of the time taken to complete a task using a single processing element (or the baseline system) to the time taken using multiple processing elements in parallel.
  - It's a measure of how much faster a parallel system performs compared to a serial/sequential one.
  - Mathematically, speedup S is defined as:

$T_S = 1\ hr$

1 cpu → 1 hr

2 cpu

ideal $S = 2$

1 hr

$$S = \frac{T\_Serial}{T\_Parallel} = \frac{60}{30} = 2$$

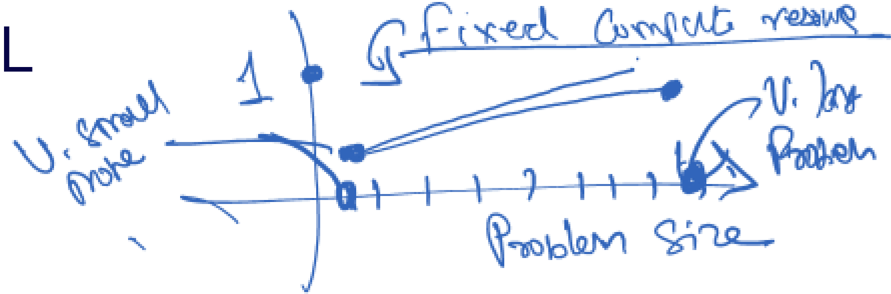
1 Processor | core |

where T_Serial is the execution time of the best serial algorithm for solving a problem, and T_Parallel is the time taken by the parallel algorithm using N processors.

P/cores

T_ Paralel = 30 m

Introduction to Scalability in ML



Key Metrics for Scalability

- **Efficiency** Parallel
  - Efficiency is the measure of the effective utilization of the computing resources in a parallel system.
  - It's defined as the speedup divided by the number of processors used in the parallel computation.
  - Mathematically, efficiency E is defined as:

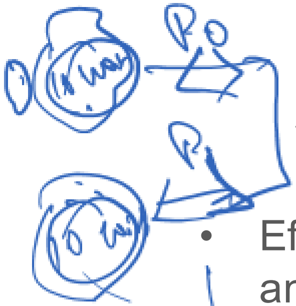
$$\boxed{E} = \frac{S}{N}$$

Speed up

1 cpu → 1 hr

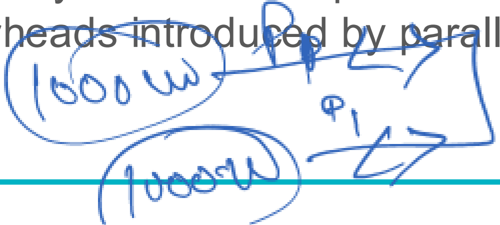
$\frac{2}{2} = 1 \times 100\%$ 2 cpu

ideally s=2



where S is the speedup and N is the number of processors.

- Efficiency can help identify how well the parallel system is being utilized and highlights the overheads introduced by parallelization.



Introduction to Scalability in ML

I/O.

P >> S

P

S

5%.

5%

1-P

Key Metrics for Scalability

- **Amdahl's Law**
  - Amdahl's Law provides a theoretical limit on the possible speedup (S_max) for a program based on the proportion of the program that can be parallelized.
  - It's expressed as:

$$S\_max = \frac{1}{(1-P)+P/N}$$

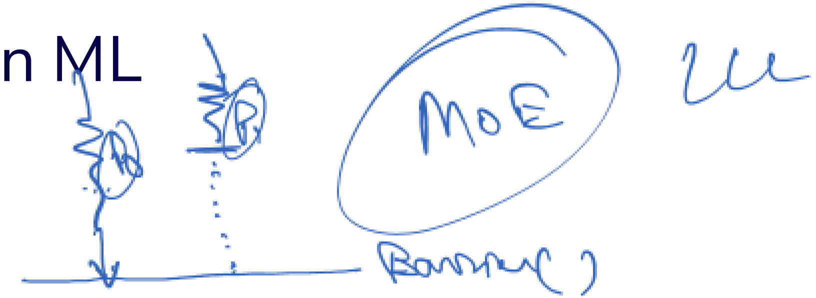
LLM

30 ~ 40% parallel effort

where P is the parallelizable portion of the task, and N is the number of processors.

  - Amdahl's Law highlights the importance of the parallelizable portion of a task in achieving significant speedup.
  - **Gustafson's Law** addresses a limitation of Amdahl's Law by suggesting that as the problem size scales, the parallel portion of the task also increases, allowing for greater scalability and speedup.

Introduction to Scalability in ML



Key Metrics for Scalability

- **Load Balancing**
  - Refers to the distribution of work evenly across the processors in a parallel system
  - Effective load balancing is crucial for achieving high performance in parallel computing as it ensures that no single processor becomes a bottleneck.
- **Communication Overhead**
  - In parallel systems, especially distributed ones, processors need to exchange data, which introduces communication overhead.
  - This overhead can significantly affect the overall performance and scalability of parallel applications.
- **Granularity Granularity**
  - Refers to the size of the tasks into which a computational problem is divided for parallel execution.
  - Fine-grained parallelism means tasks are small, requiring frequent communication.

Introduction to Scalability in ML

Key Metrics for Scalability

- **Throughput**
  - The number of tasks or amount of data processed in a given time frame.
  - A higher throughput rate is indicative of a more scalable system.
- **Latency**
  - The time it takes for a system to respond to a request.
  - Reduced latency is critical for real-time processing needs in scalable ML applications.
- **Resource Utilization**
  - Efficient use of available computational resources (CPU, memory, bandwidth) suggests a system is scalable, balancing cost with computational needs.

code

MPI – Communication

“Challenges in Large-Scale ML Workflows

Challenges in Large-Scale ML Workflows

- **Data Management and Storage**
  - **Issue:** Handling, storing, and processing large volumes of data efficiently.
  - **Impact:** Increased demand on storage capacity and data retrieval speed.
    - **Example:** Managing terabytes of image data for a deep learning model.
  - More discussion in Data Engineering Module

Data Engg. Challe

- **Extended Training Times**
  - **Issue:** Training complex models on large datasets can take hours or even days.
  - **Impact**: Delays in model development and deployment.
  - **Example**: Training a neural network on high-resolution video footage.

Challenges in Large-Scale ML Workflows

- **Resource Allocation and Cost**
  - **Issue**: Requirement for more computational resources, leading to increased costs.
  - **Impact**: Budget constraints can limit the scope of ML projects.
  - **Example**: The high cost of using cloud-based GPU instances for model training.
- **Model Scalability and Deployment**
  - **Issue**: Ensuring that ML models scale effectively as data and user base grow.
  - **Impact**: Potential degradation in performance and user experience.
  - **Example**: A recommendation system that struggles to keep up as the number of users increases.

ChatGPT

API Scaling

Deploy

users >> 1

size of input per user

cache/memory

Parallelization in ML: Challenges

- **Data Synchronization**
  - **Challenge:** Ensuring that all processors have access to the data they need, and that updates to the data are properly synchronized.
  - **Solution: I**mplementing robust data management and communication protocols
- **Load Balancing**
  - **Challenge:** Ensuring that all processors are utilized efficiently, without some being overburdened while others remain idle.
  - **Solution**: Dynamic allocation of tasks and resources based on real-time performance monitoring.
- **Scalability**
  - **Challenge:** Ensuring that the parallelized system can handle growth in data and complexity without performance degradation.
  - **Solution**: Designing the system for horizontal and vertical scalability and utilizing parallel ML libraries that are optimized for scalability

Benefits of Parallelization

Benefits of Parallelization

- **Speed and Efficiency**
  - **Benefit**: Parallelization allows for the simultaneous processing of data and tasks, significantly reducing training and processing times.
  - **Example**: Using multi-threading to preprocess data while simultaneously training a model.
- **Resource Utilization**
  - **Benefit**: Efficient use of available computational resources, leading to improved performance. **Example**: Distributing ML tasks across a cluster of machines to maximize resource utilization.

Benefits of Parallelization

- **Scalability**
  - **Benefit**: Parallel ML libraries and frameworks are designed to scale horizontally, providing the ability to handle increased workloads without a significant drop in performance.
  - **Example**: Adding more nodes to a distributed computing cluster to handle larger datasets.
- **Cost-Effectiveness**
  - **Benefit:** By optimizing resource utilization and reducing processing times, parallelization can lead to cost savings, especially in cloud-based environments where resources are billed per usage.
  - **Example**: Completing ML tasks in less time and with fewer resources reduces cloud computing costs.

“ML Libraries & its parallelization capabilities

ML Libraries & its parallelization capabilities

Types of Parallelism in ML

P_0 P_1

- There are generally two ways to distribute computation across multiple devices:

$x\left[x_1 \cdots x_{n/2} \,\middle|\, x_{n/2+1} \cdots x_n\right]$

- **Data parallelism**
  - A single model gets replicated on multiple devices or multiple machines.
  - Each of them processes different batches of data, then they merge their results.
  - There exist many variants of this setup, that differ in how the different model replicas merge results, in whether they stay in sync at every batch or whether they are more loosely coupled, etc.
- **Model parallelism**
  - Different parts of a single model run on different devices, processing a single batch of data together.
  - This works best with models that have a naturally-parallel architecture, such as models that feature multiple branches.

P_0 for$(i = 1$ to $n/2)$ for$(i = n/2+1$ to $n)$ for$(i = 1$ to $100)$

ML Libraries & its parallelization capabilities

NumPy and Parallel Computing

- NumPy is a foundational library for numerical computing in Python, widely used for array manipulation and mathematical operations.

- **Parallel Capabilities:**
  - Utilizes optimized, low-level C APIs to perform efficient parallel operations on arrays.
  - Though inherently not parallel, it benefits from vectorized operations that leverage SIMD (Single Instruction, Multiple Data) architecture under the hood.

Vectorized / Parallel for loop unrolling

- **Integration with Parallel Tools:**
  - Can be combined with libraries like Dask or Joblib to achieve explicit parallel computatior atasets.

NumPy

ML Libraries & its parallelization capabilities

Pandas for Concurrent Data Processing

- Pandas is a high-level data manipulation tool, perfect for data cleaning, exploration, and analysis.

= Only sequential

- **Parallel Capabilities**
  - By default, pandas operates in a single-threaded manner.
  - However, its operation can be parallelized using external libraries such as Modin or Dask, which mimic pandas' API and distribute data processing across multiple cores or nodes.

```
In [1]: import dask.dataframe as dd

In [2]: ddf = dd.read_parquet("data/timeseries/ts*.parquet",
engine="pyarrow")

In [3]: ddf
Out[4]:
Dask DataFrame Structure:
                 id    name      x       y
npartitions=12
              int64  string  float64  float64
                ...    ...     ...     ...
...             ...    ...     ...     ...
                ...    ...     ...     ...
                ...    ...     ...     ...
Dask Name: read-parquet, 1 graph layer
```

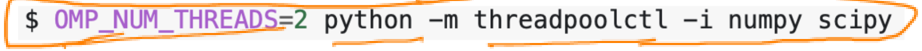
pandas

ML Libraries & its parallelization capabilities

Scikit-learn's Approach to Parallelism

- Scikit-learn is a primary library for machine learning in Python, providing simple and efficient tools for data mining and data analysis.

- **Parallel Capabilities**
  - Offers built-in support for parallel processing in algorithms that can be parallelized, such as random forests, using Joblib library.
  - scikit-learn parallelize costly operations using multiple CPU cores.
  - This is either done:
    - with higher-level parallelism via joblib.
    - with lower-level parallelism via OpenMP, used in C or Cython code.



```
$ OMP_NUM_THREADS=2 python -m threadpoolctl -i numpy scipy
```

    - with lower-level parallelism via BLAS, used by NumPy and SciPy for generic operations on arrays.
    - *Refer: https://scikit-learn.org/stable/computing/parallelism.html*

DASK

Numpy + Pandas → Scikit-learn + Keras / CUDA / TF / PyTorch

ML Libraries & its parallelization capabilities

Scikit-learn's Approach to Parallelism

- **Higher-level parallelism with joblib**
  - scikit-learn generally relies on the **loky** backend, which is joblib’s default backend
  - the number of workers (threads or processes) that are spawned in parallel can be controlled via the n_jobs parameter
  - Joblib is able to support both multi-processing and multi-threading.
  - Whether joblib chooses to spawn a thread or a process depends on the backend that it’s using

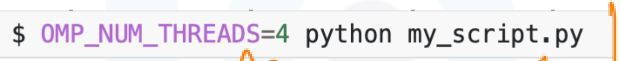
```
from joblib import parallel_backend

with parallel_backend('threading', n_jobs=2):
    # Your scikit-learn code here
```

ML Libraries & its parallelization capabilities

Scikit-learn's Approach to Parallelism

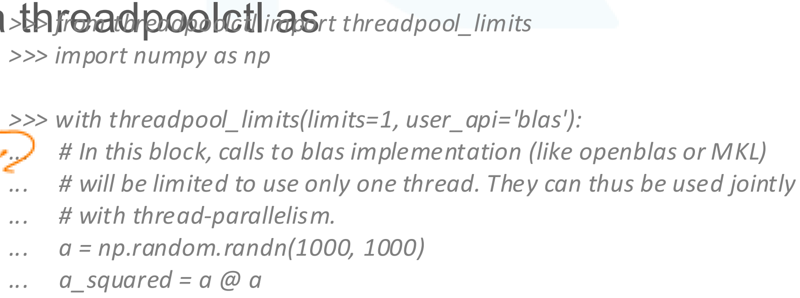
- **Lower-level parallelism with OpenMP**
  - OpenMP is used to parallelize code written in Cython or C, relying on multi-threading exclusively
  - We can control the exact number of threads that are used either
    - via the OMP_NUM_THREADS environment variable, for instanc

Open-MP.



```
$ OMP_NUM_THREADS=4 python my_script.py
```

    - or via threadpoolctl as



```
>>> from threadpoolctl import threadpool_limits
>>> import numpy as np

>>> with threadpool_limits(limits=1, user_api='blas'):
...     # In this block, calls to blas implementation (like openblas or MKL)
...     # will be limited to use only one thread. They can thus be used jointly
...     # with thread-parallelism.
...     a = np.random.randn(1000, 1000)
...     a_squared = a @ a
```

ML Libraries & its parallelization capabilities

Keras for Distributed Deep Learning

DNN

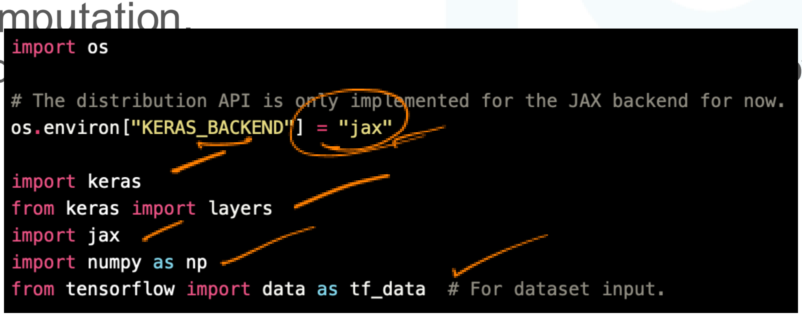
- Keras is a high-level neural networks API, capable of running on top of TensorFlow, Theano, or CNTK, designed for human beings, not machines.

- **Parallel Capabilities:**
  - Multi-GPU distributed training with JAX
  - Multi-GPU distributed training with TensorFlow
  - Multi-GPU distributed training with PyTorch
  - Distributed training with Keras 3

ML Libraries & its parallelization capabilities

Distributed training with Keras 3

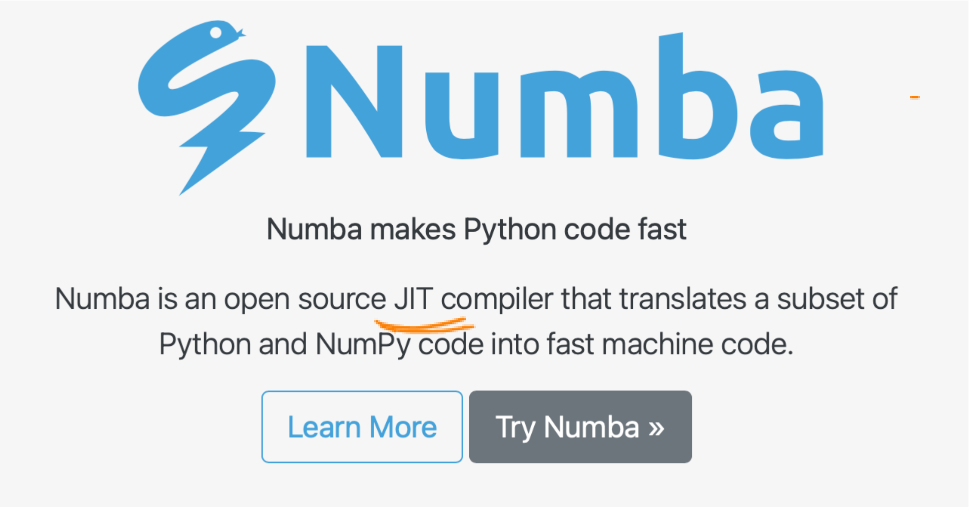
- Introduces Keras distribution API for distributed deep learning.
- Supports various backends including JAX, TensorFlow, and PyTorch.
- Offers tools for data and model parallelism.
- Enables efficient scaling on multiple accelerators and hosts (GPUs, TPUs).
- Streamlines distributed environment initialization, device mesh definition, and tensor layout orchestration.
- Includes classes like DataParallel and ModelParallel for simplified parallel computation.
- Aic ws by abstracting complexity.



```
import os

# The distribution API is only implemented for the JAX backend for now.
os.environ["KERAS_BACKEND"] = "jax"

import keras
from keras import layers
import jax
import numpy as np
from tensorflow import data as tf_data  # For dataset input.
```

https://keras.io/guides/distribution/

“Parallel ML Libraries



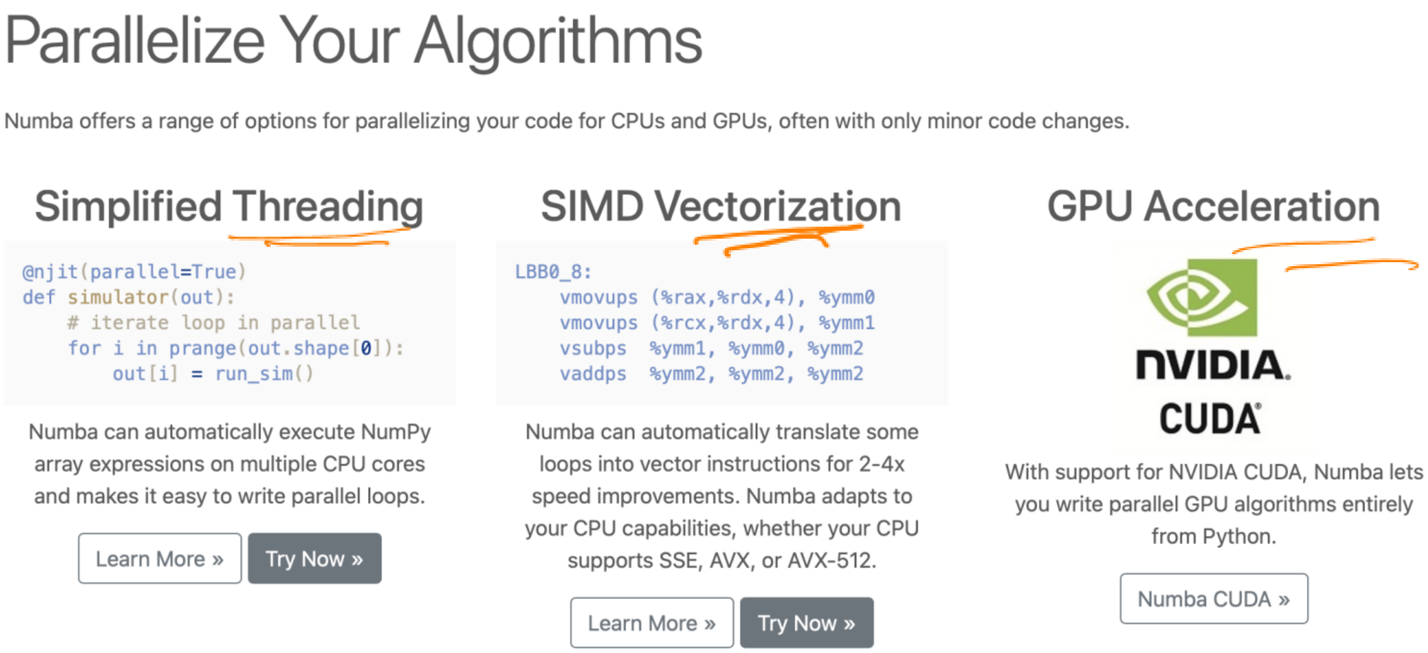
Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

```
$ conda install numba
```

```
$ pip install numba
```

https://numba.pydata.org



Parallelize Your Algorithms

Numba offers a range of options for parallelizing your code for CPUs and GPUs, often with only minor code changes.

Simplified Threading

```
@njit(parallel=True)
def simulator(out):
    # iterate loop in parallel
    for i in prange(out.shape[0]):
        out[i] = run_sim()
```

Numba can automatically execute NumPy array expressions on multiple CPU cores and makes it easy to write parallel loops.

Learn More » Try Now »

SIMD Vectorization

```
LBB0_8:
    vmovups (%rax,%rdx,4), %ymm0
    vmovups (%rcx,%rdx,4), %ymm1
    vsubps  %ymm1, %ymm0, %ymm2
    vaddps  %ymm2, %ymm2, %ymm2
```

Numba can automatically translate some loops into vector instructions for 2-4x speed improvements. Numba adapts to your CPU capabilities, whether your CPU supports SSE, AVX, or AVX-512.

Learn More » Try Now »

GPU Acceleration

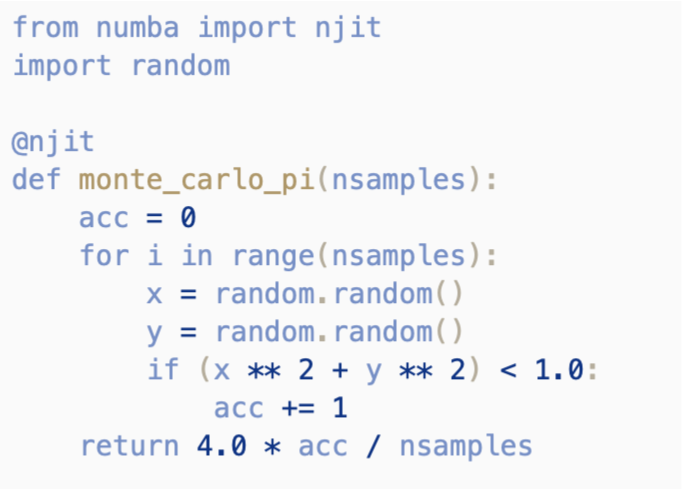
With support for NVIDIA CUDA, Numba lets you write parallel GPU algorithms entirely from Python.

Numba CUDA »

https://numba.pydata.org

NUMBA

- Numba converts Python functions into optimized machine code at runtime via LLVM.
- Python algorithms compiled with Numba can rival C or FORTRAN speeds.
- No need to swap Python interpreter, compile separately, or install C/C++ compiler.
- Simply use a Numba decorator on your Python function for automatic conversion.

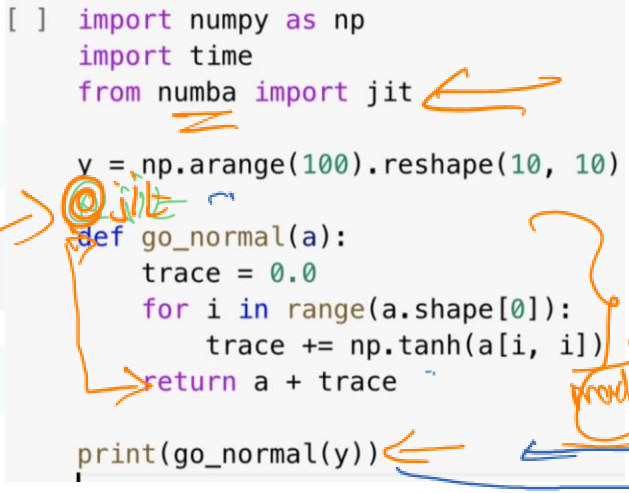


```python
from numba import njit
import random

@njit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Just-in-time Compiler

How to use NUMBA?

- Through its collection of decorates that can be applied to functions to instruct Numba to compile
- Decorated function is compiled to machine code when a call is made to Numba



```
[ ]  import numpy as np
     import time
     from numba import jit

     y = np.arange(100).reshape(10, 10)
     @jit
     def go_normal(a):
         trace = 0.0
         for i in range(a.shape[0]):
             trace += np.tanh(a[i, i])
         return a + trace

     print(go_normal(y))
```

Numba

machine code

jit

C++

Python

main.cpp

→ Compile

main.exe

main.py

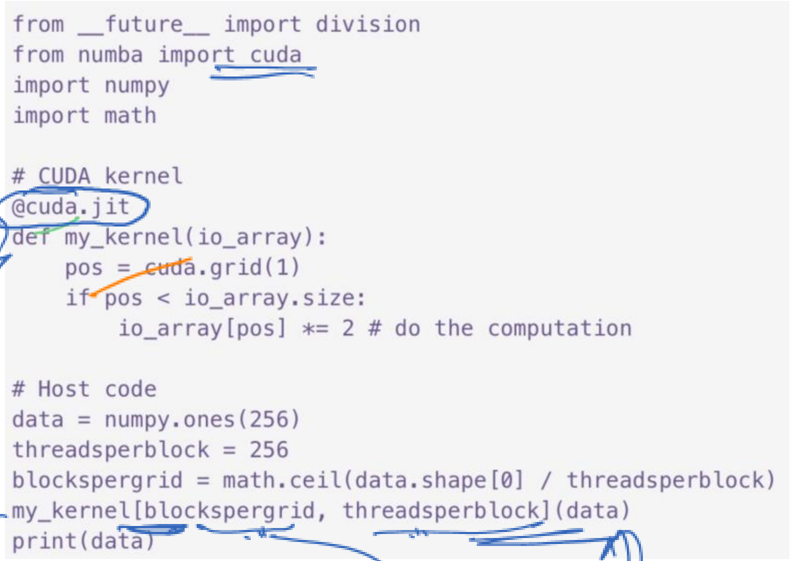
~~Compile~~

execute line-by-line

human readable to machine readable * 0,1s.

NUMBA with GPU

- Through its decorates
- User handles the index



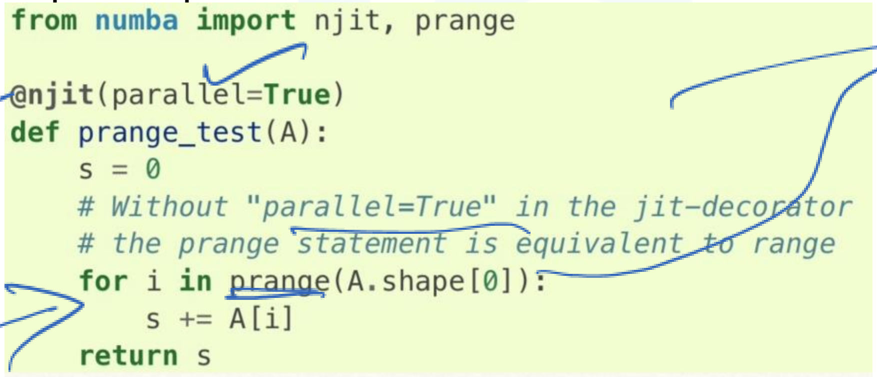
```
from __future__ import division
from numba import cuda
import numpy
import math

# CUDA kernel
@cuda.jit
def my_kernel(io_array):
    pos = cuda.grid(1)
    if pos < io_array.size:
        io_array[pos] *= 2 # do the computation

# Host code
data = numpy.ones(256)
threadsperblock = 256
blockspergrid = math.ceil(data.shape[0] / threadsperblock)
my_kernel[blockspergrid, threadsperblock](data)
print(data)
```

https://numba.readthedocs.io/en/stable/cuda/index.html#

<<< m, # >>>

NUMBA with Multicore

- Numba supports automatic parallelism (works only on CPUs at the moment), eg., Numpy reduction functions: sum, prod, min, max, mean, var, std etc
- Explicit parallel loops are supported. Use prange instead of range to specify the loop to be parallelised



```
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

OpenMP - type Parallelism

https://numba.readthedocs.io/en/stable/user/parallel.html

JAX: Overview

JAX

- JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning.
- JAX is NumPy on the CPU, GPU, and TPU, with great automatic differentiation for high-performance machine learning research.
- JAX is much more than just a GPU-backed NumPy.
  - It also comes with a few program transformations that are useful when writing numerical code.
    - jit(), for speeding up the code
    - grad(), for taking derivatives
    - vmap(), for automatic vectorization or batching.
- Visit the official website for more information: JAX Official Site.

Advanced Parallel ML Libraries

Popular Parallel ML Libraries

- **Dask**
  - Python library for parallel and distributed computing, integrates seamlessly with existing Python libraries (https://www.dask.org)
- **Apache Spark Mllib**
  - Library for machine learning on distributed systems, providing a wide range of algorithms (https://spark.apache.org/mllib/)
- **Horovod**
  - Framework to efficiently train machine learning models on distributed systems, popular in deep learning scenarios (https://horovod.ai)
- **RAY**
  - An open-source unified compute framework that makes it easy to scale AI and Python workloads from reinforcement learning to deep learning to tuning, and model serving (https://www.ray.io)

“Exploring DASK for Parallel Computing

Exploring DASK for Parallel Computing

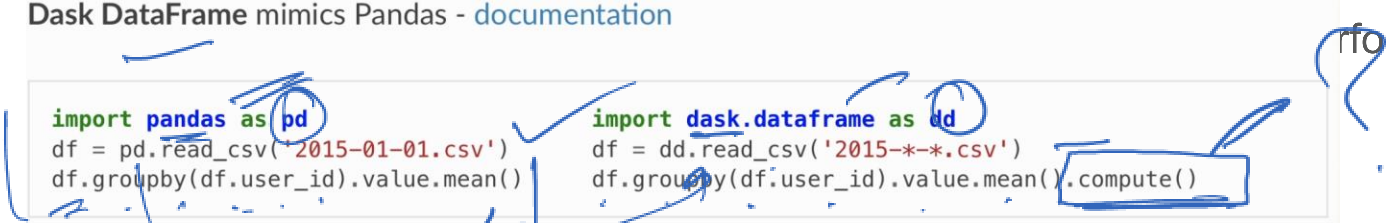
- **What is Dask?**
  - Dask is a parallel computing library that seamlessly integrates with popular Python libraries like NumPy, Pandas, and Scikit-Learn
  - Designed to handle larger-than-memory computing on top of Pandas dataframes, using familiar APIs
  - Allows for dynamic task scheduling and works well with distributed systems
- Why Dask?
  - Addressing Limitations of Single-Node Computation
  - Traditional libraries like NumPy and Pandas are limited to single-node computation.
  - Dask enables distributed computing, allowing for scalability and faster computation times.
  - Ideal for working with big data, complex algorithms, or any task that requires additional computing power.

Exploring DASK for Parallel Computing

- **Key Features for ML**
  - **Parallelism**: Leverages all the cores on your machine or all the machines in your cluster.
  - **Lazy Evaluation:** Builds up a task to execute, and nothing is computed until you ask for the final result.
  - **Distributed Computing:** Works on a single machine or on a cluster, without changing the code.
  - **Integration:** Works seamlessly with existing ML libraries and tools.
  - **Scalability:** Can handle larger-than-memory datasets.

Exploring DASK for Parallel Computing

- **Dask Arrays** numPy
  - Dask arrays are like NumPy arrays, but they operate on larger-than-memory datasets in parallel.
  - We can perform operations on Dask arrays as we would with NumPy arrays, but with the added benefit of parallel computation.
  - Example: Creating a large random Dask array and computing its mean value.
- **Dask DataFrames** Pandas
  - Dask DataFrames are large parallel DataFrames composed of many smaller Pandas DataFrames.
  - They support a large portion of the Pandas DataFrame API.



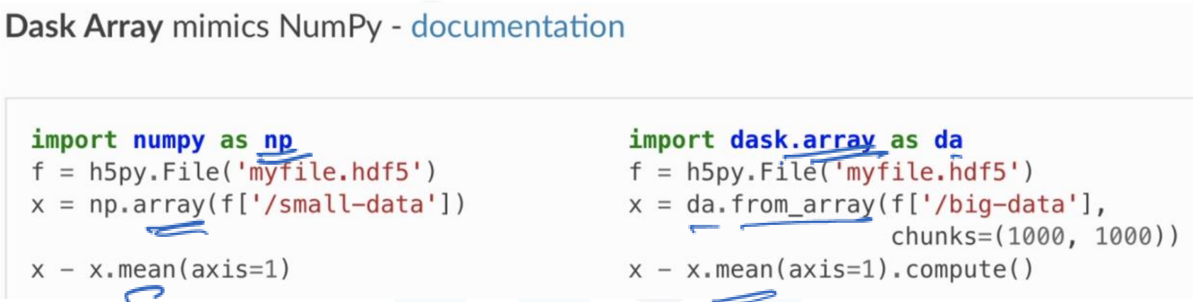
Dask DataFrame mimics Pandas - documentation

```
import pandas as pd
df = pd.read_csv('2015-01-01.csv')
df.groupby(df.user_id).value.mean()
```

```
import dask.dataframe as dd
df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean().compute()
```

rforming a

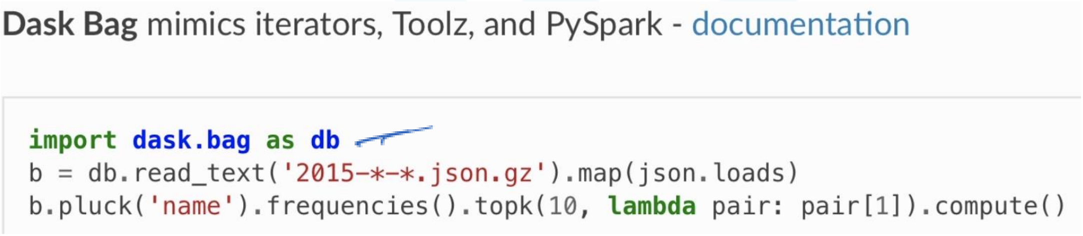
Exploring DASK for Parallel Computing



Dask Array mimics NumPy - documentation

```
import numpy as np
f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])

x - x.mean(axis=1)
```

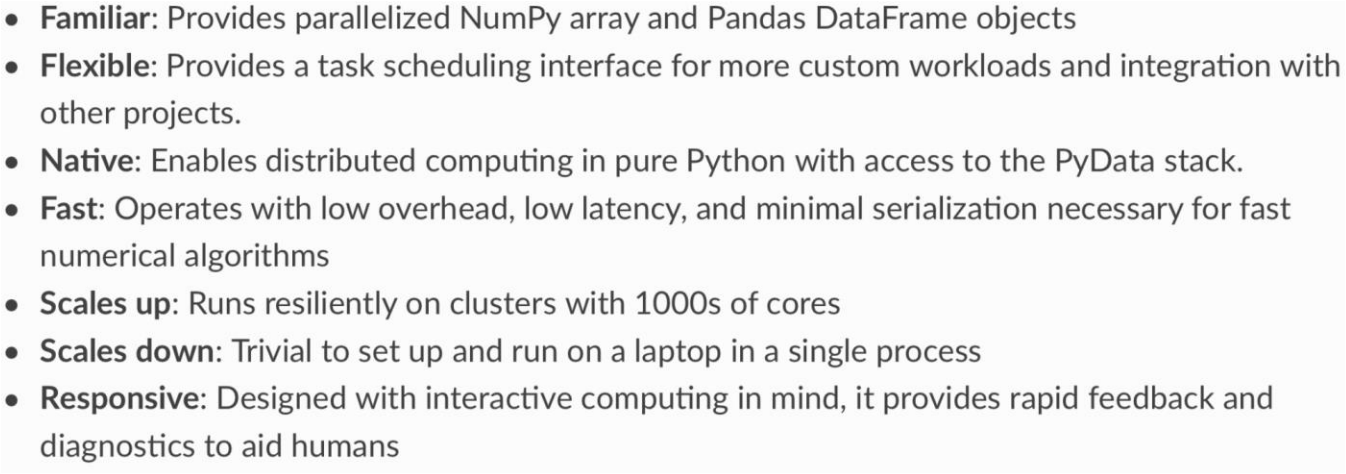
```
import dask.array as da
f = h5py.File('myfile.hdf5')
x = da.from_array(f['/big-data'],
                  chunks=(1000, 1000))
x - x.mean(axis=1).compute()
```



Dask Bag mimics iterators, Toolz, and PySpark - documentation

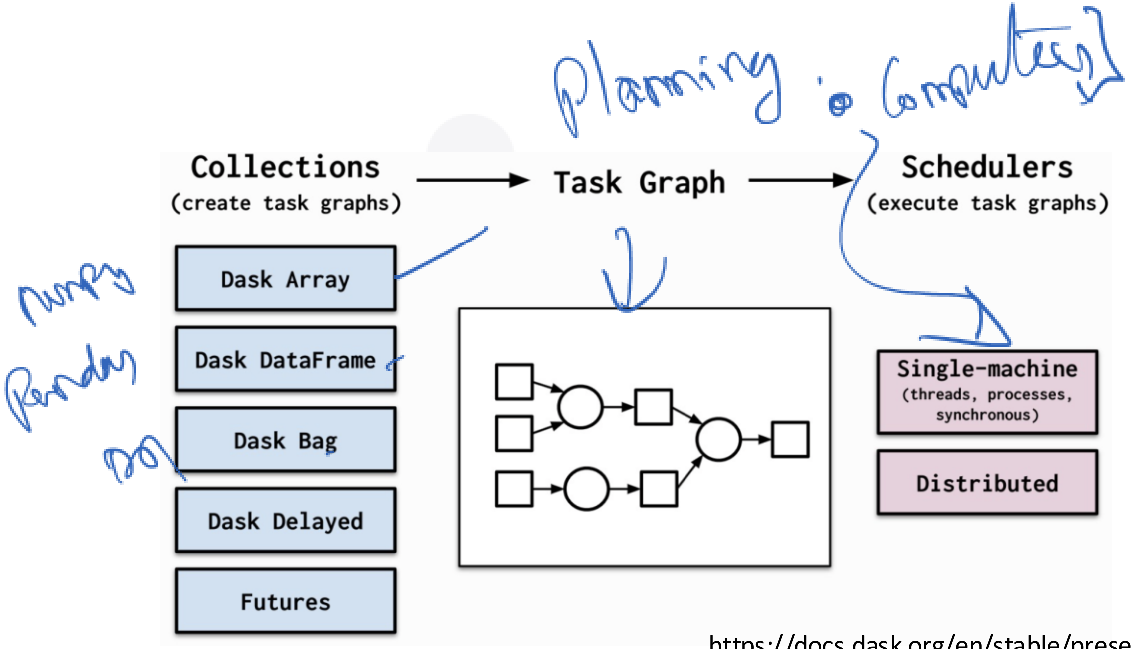
```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(10, lambda pair: pair[1]).compute()
```

Exploring DASK for Parallel Computing



- **Familiar**: Provides parallelized NumPy array and Pandas DataFrame objects
- **Flexible**: Provides a task scheduling interface for more custom workloads and integration with other projects.
- **Native**: Enables distributed computing in pure Python with access to the PyData stack.
- **Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms
- **Scales up**: Runs resiliently on clusters with 1000s of cores
- **Scales down**: Trivial to set up and run on a laptop in a single process
- **Responsive**: Designed with interactive computing in mind, it provides rapid feedback and diagnostics to aid humans

Exploring DASK for Parallel Computing



https://docs.dask.org/en/stable/presentations.html

Exploring Dask for Parallel Computing

- **Getting Started with Dask**
  - **Installation:** *pip install dask*
  - **Creating a Dask Array:** Similar to a NumPy array but works on larger-than-memory datasets.
  - **Creating a Dask DataFrame:** Similar to a Pandas dataframe
- **More from**

  https://tutorial.dask.org

Summary

meet @ 11 am

Scalability in ML

- Scalability in ML refers to the ability to efficiently process increasing amounts of data and more complex algorithms without significant performance loss.

Challenges in Scaling ML Algorithms

- Major challenges include handling high-dimensional data, algorithm complexity, computational resource limitations, and maintaining performance efficiency with larger datasets

ML Libraries and its Parallel Capabilities

- ML libraries like TensorFlow and PyTorch are designed with parallel processing in mind to harness the full power of modern multi-core CPUs and GPUs

NUMBA and Dask

- Numba is used to optimize Python code for performance, while Dask provides advanced parallelization capabilities, particularly for scaling NumPy, pandas, and scikit-learn.

Effective scalability in ML is crucial for advancing the field, and despite challenges, progress is being facilitated by libraries that offer sophisticated parallel processing and optimization techniques.