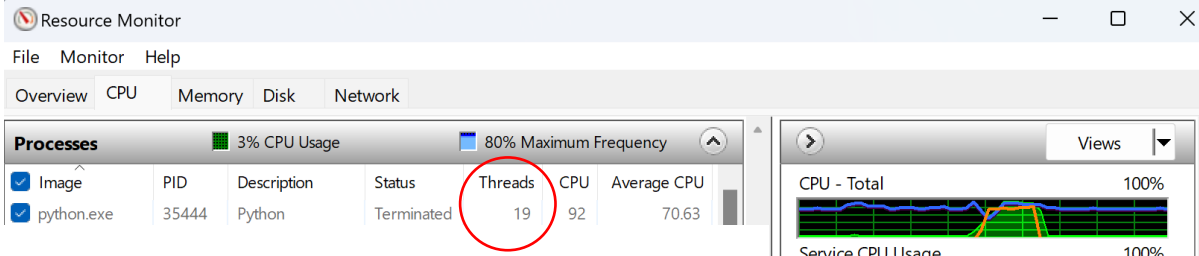
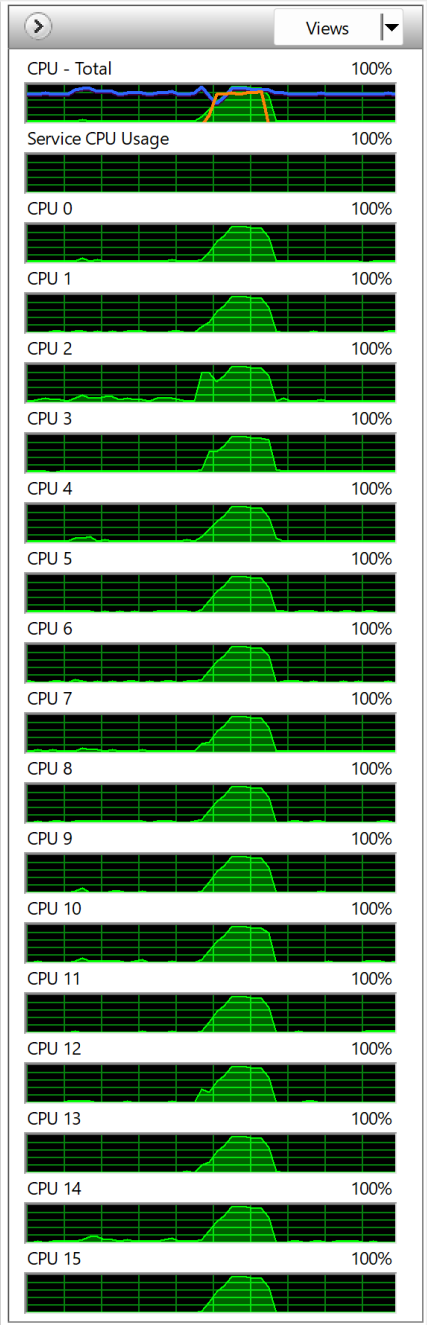
Large-scale Parallel Computing

Outline

- Parallel program evaluation example
- Scaling-out methods
- High-performance cluster
- Big data cluster

Performance Evaluation of Matrix Multiplication






```
import numpy as np

import datetime
start = datetime.datetime.now()

size = 10000
a = np.random.random_sample((size, size))
b = np.random.random_sample((size, size))
n = np.dot(a,b)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")
```

```
PS C:\Users\veera\Documents\2 Teaching\625
0:00:09.250277 h:mm:ss
```

Numpy dot-product function is very fast because it is multithreaded.

Getting CPU information in Python

```
!pip install -q psutil
!pip install -q py-cpuinfo
import platform
import psutil
import cpuinfo

print('platform       :', platform.platform())
print('cpu model      :', cpuinfo.get_cpu_info()['brand_raw'])
print('physical cpu   :', psutil.cpu_count(logical=False))
print('logical cpu    :', psutil.cpu_count(logical=True))
print('virtual memory :', psutil.virtual_memory())
```

```
platform       : Windows-10-10.0.22621-SP0
cpu model      : AMD Ryzen 7 6800HS Creator Edition
physical cpu   : 8
logical cpu    : 16
virtual memory : svmem(total=14702026752, available=5296963584, percent=64.0, used=9405063168, free=5296963584)
```

Parallel Computing in Numpy

Numpy uses several libraries that are multithreaded and uses SIMD (vector) instructions.

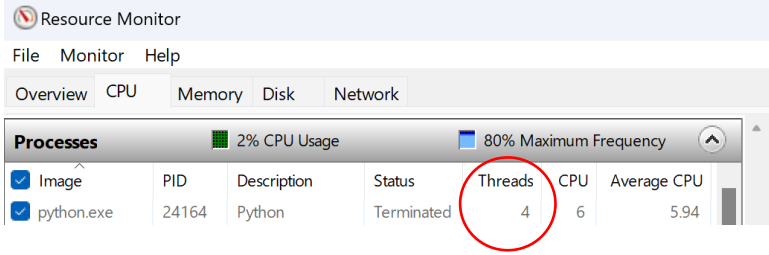
```
import numpy as np
np.show_config()
```

```
openblas64__info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__info']
    libraries = ['openblas64__info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None)]
blas_ilp64_opt_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__info']
    libraries = ['openblas64__info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None)]
openblas64__lapack_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__lapack_info']
    libraries = ['openblas64__lapack_info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
lapack_ilp64_opt_info:
    library_dirs = ['D:\\a\\numpy\\numpy\\build\\openblas64__lapack_info']
    libraries = ['openblas64__lapack_info']
    language = f77
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'), ('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3
    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
    not found = AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX,AVX512_CNL,AVX512_ICL
```

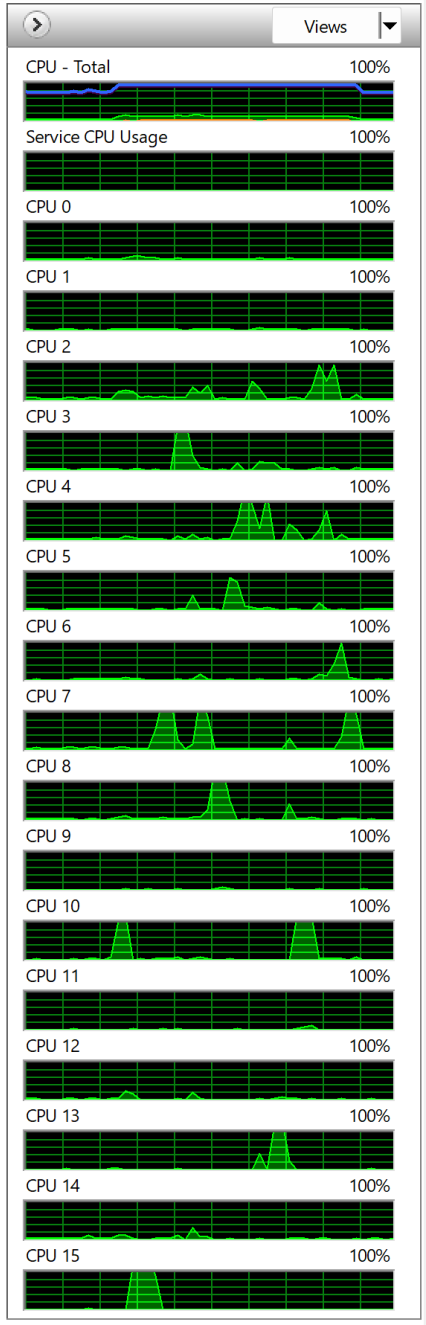
```
!pip install threadpoolctl

from threadpoolctl import threadpool_info
from pprint import pprint
import numpy
pprint(threadpool_info())
```

```
Requirement already satisfied: threadpoolctl in c:\users\veera\appdata\local\programs\python\python39\lib\site-packages (3.1.0)
[{'architecture': 'Zen',
  'filepath': 'C:\\Users\\veera\\AppData\\Local\\Programs\\Python\\Python39\\Lib\\site-packages\\numpy\\.libs\\libopenblas.FB5A
E2TYXYH2IJRDKGDGQ3XBKLKTF43H.gfortran-win_amd64.dll',
  'internal_api': 'openblas',
  'num_threads': 16,
  'prefix': 'libopenblas',
  'threading_layer': 'pthreads',
  'user_api': 'blas',
  'version': '0.3.20'}]
```

Performance Evaluation of Matrix Multiplication



Specified number of threads + 3

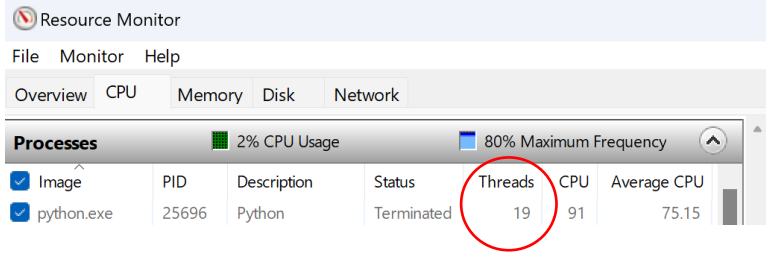


Setting the number of threads before running Numpy code.

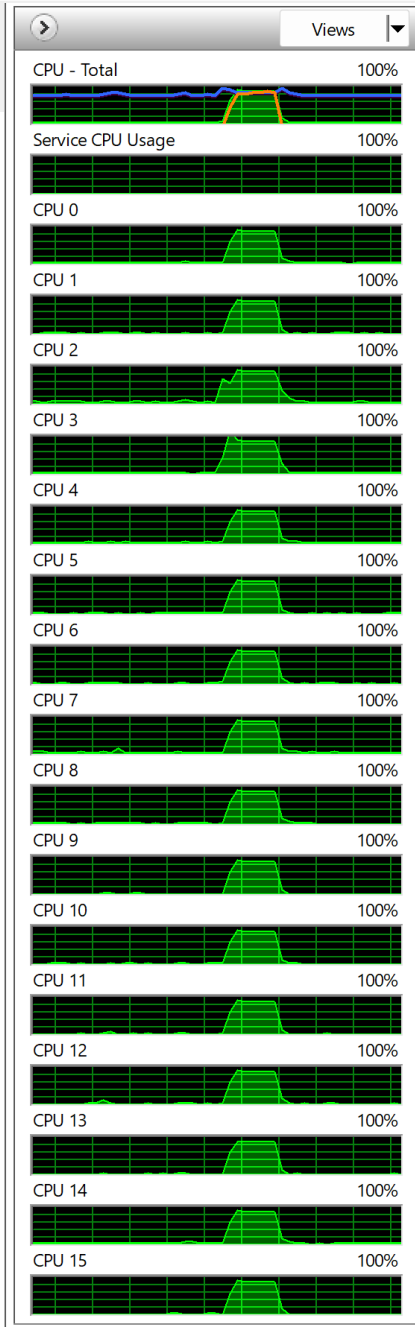
```
import os
os.environ["OMP_NUM_THREADS"] = "1"                 # export OMP_NUM_THREADS=1
os.environ["OPENBLAS_NUM_THREADS"] = "1"            # export OPENBLAS_NUM_THREADS=1
os.environ["MKL_NUM_THREADS"] = "1"                 # export MKL_NUM_THREADS=1
os.environ["VECLIB_MAXIMUM_THREADS"] = "1"          # export VECLIB_MAXIMUM_THREADS=1
os.environ["NUMEXPR_NUM_THREADS"] = "1"             # export NUMEXPR_NUM_THREADS=1

import numpy as np

import datetime
start = datetime.datetime.now()

size = 10000
a = np.random.random_sample((size, size))
b = np.random.random_sample((size, size))
n = np.dot(a,b)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")
```

```
PS C:\Users\veera\Documents\2 Teaching\625 Data Sci Arch\programs> python .\dotproduct.py
0:00:32.347729 h:mm:ss
```

Performance Evaluation of Matrix Multiplication



Specified number of threads + 3



Setting the number of threads before running Numpy code.

```
import os
os.environ["OMP_NUM_THREADS"] = "1"              # export OMP_NUM_THREADS=1
os.environ["OPENBLAS_NUM_THREADS"] = "1"         # export OPENBLAS_NUM_THREADS=1
os.environ["MKL_NUM_THREADS"] = "1"              # export MKL_NUM_THREADS=1
os.environ["VECLIB_MAXIMUM_THREADS"] = "1"       # export VECLIB_MAXIMUM_THREADS=1
os.environ["NUMEXPR_NUM_THREADS"] = "1"          # export NUMEXPR_NUM_THREADS=1

import numpy as np

import datetime
start = datetime.datetime.now()

size = 10000
a = np.random.random_sample((size, size))
b = np.random.random_sample((size, size))
n = np.dot(a,b)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")
```

```
PS C:\Users\veera\Documents\2 Teaching\625 Data Sci Arch\programs> python .\dotproduct.py
0:00:32.347729 h:mm:ss
```

Change “1” to 2, 4, 8, 12, 16 and rerun the program

Testing on Jupyter Notebook needs restarting kernel.

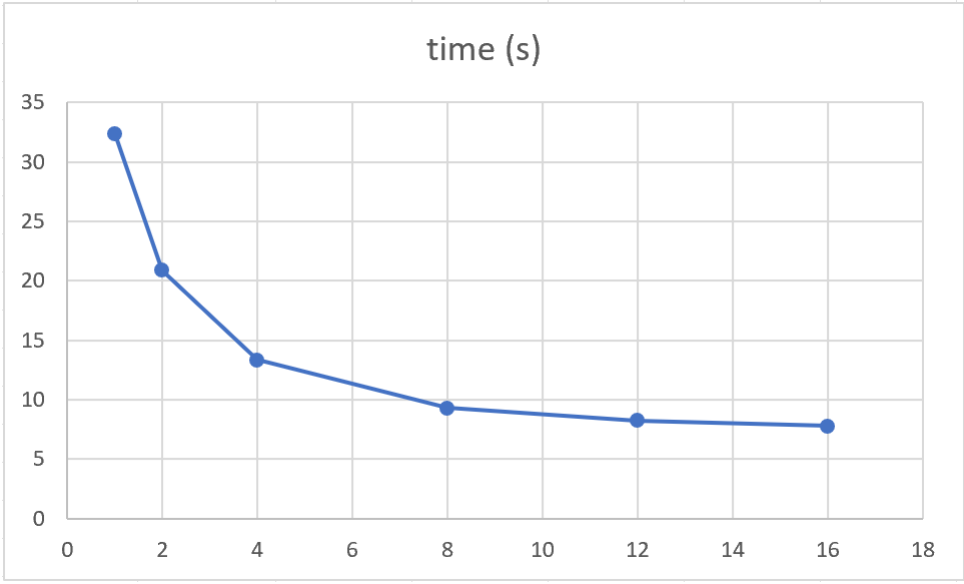
```
import os
os.environ["OMP_NUM_THREADS"] = "16"             # export OMP_NUM_THREADS=16
os.environ["OPENBLAS_NUM_THREADS"] = "16"        # export OPENBLAS_NUM_THREADS=16
os.environ["MKL_NUM_THREADS"] = "16"             # export MKL_NUM_THREADS=16
os.environ["VECLIB_MAXIMUM_THREADS"] = "16"      # export VECLIB_MAXIMUM_THREADS=16
os.environ["NUMEXPR_NUM_THREADS"] = "16"         # export NUMEXPR_NUM_THREADS=16
```

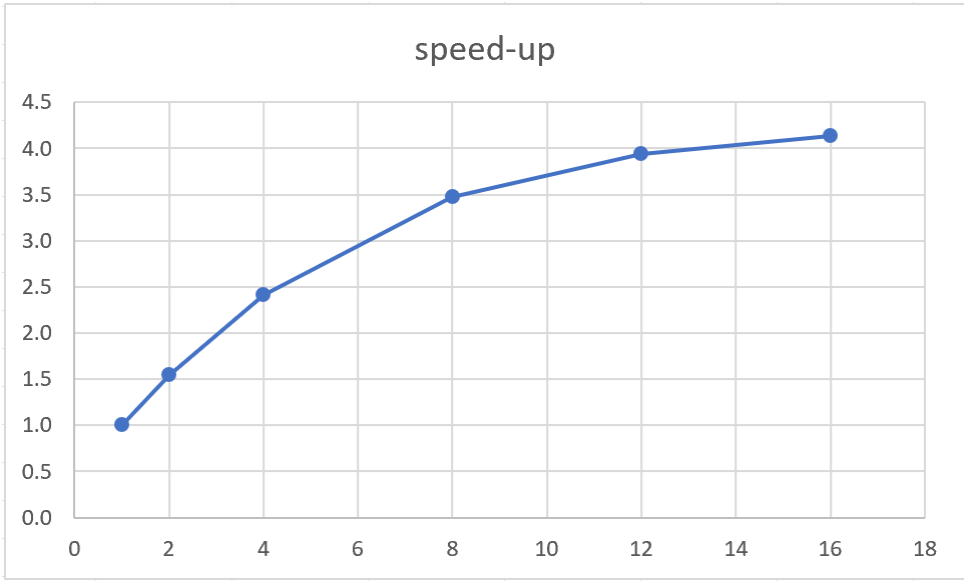
```
PS C:\Users\veera\Documents\2 Teaching\625 Data Sci Arch\programs> python .\dotproduct.py
0:00:07.793126 h:mm:ss
```

Performance Evaluation of Matrix Multiplication

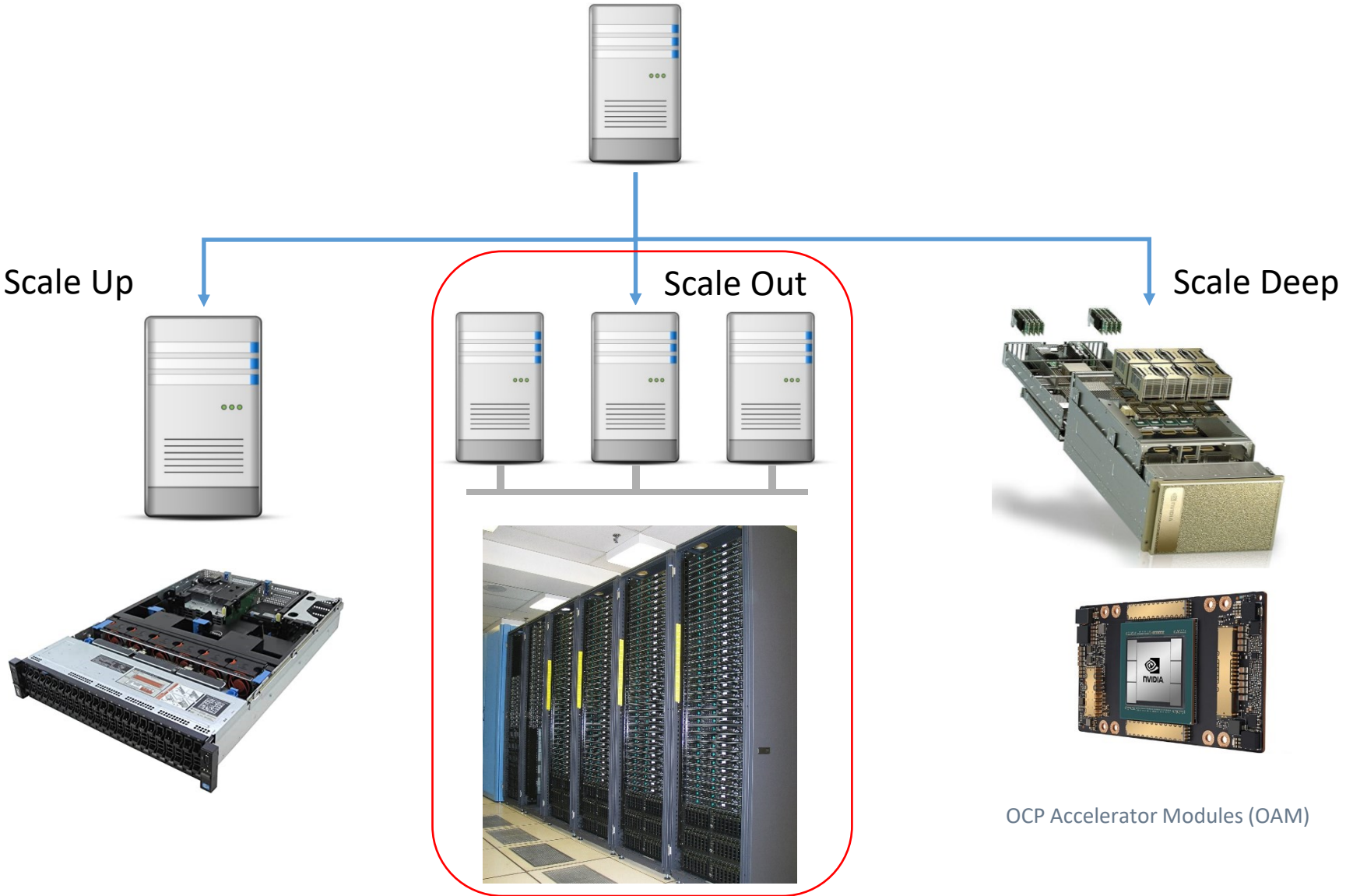
threads	time (s)	speed-up
1	32.3	1.0
2	20.9	1.5
4	13.4	2.4
8	9.3	3.5
12	8.2	3.9
16	7.8	4.1

Max speed-up is about 4X





Scaling Out provides more scalability



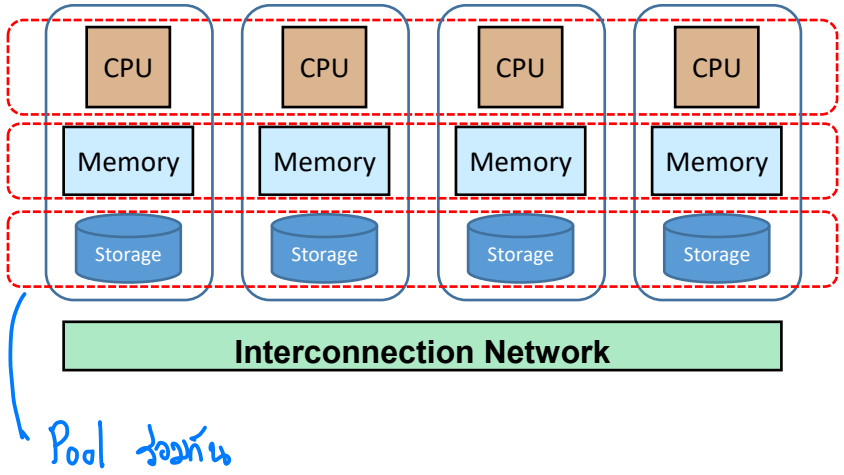
OCP Accelerator Modules (OAM)

Scale-out: Distributed Computing

Distributed computing is about making distributed components work together as one.

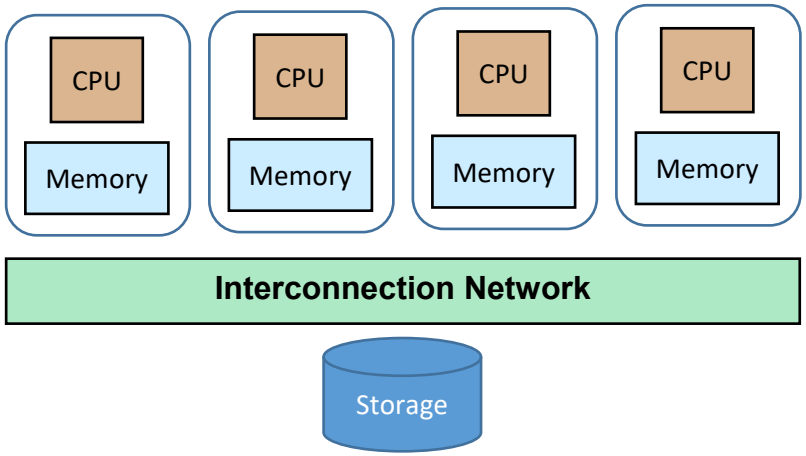
- Distributed Processors CPU มองเป็น Pool CPU เพื่อทำงานร่วมกัน
  - A collection of stand-alone computers connected over a local network and working together as a single computer system
- Distributed Shared Memory (Spark)
  - Providing a shared memory programming model over distributed memory computers.
- Distributed File System (HADOOP)
  - A shared file system consisting of storage devices that are physically distributed across machines.

ทำให้ Storage ที่แยกกันมองเป็นตัวเดียวกัน
อาจใช้เพื่อเก็บข้อมูลอย่างเดียวก็ได้
เช่น Hadoop ทำเรื่อง Parallel ด้วย

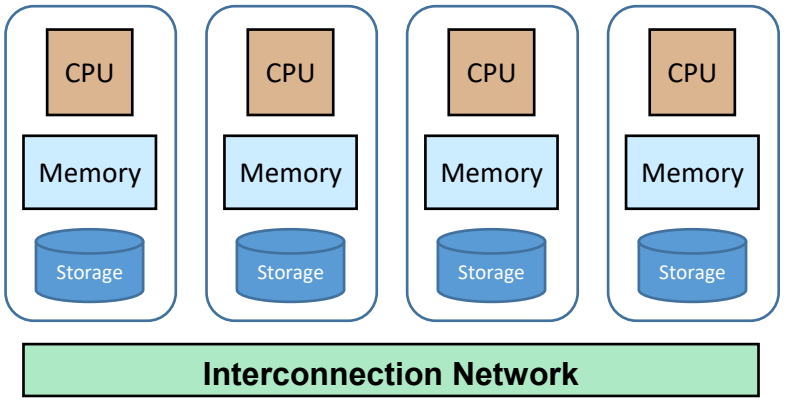


Clusters

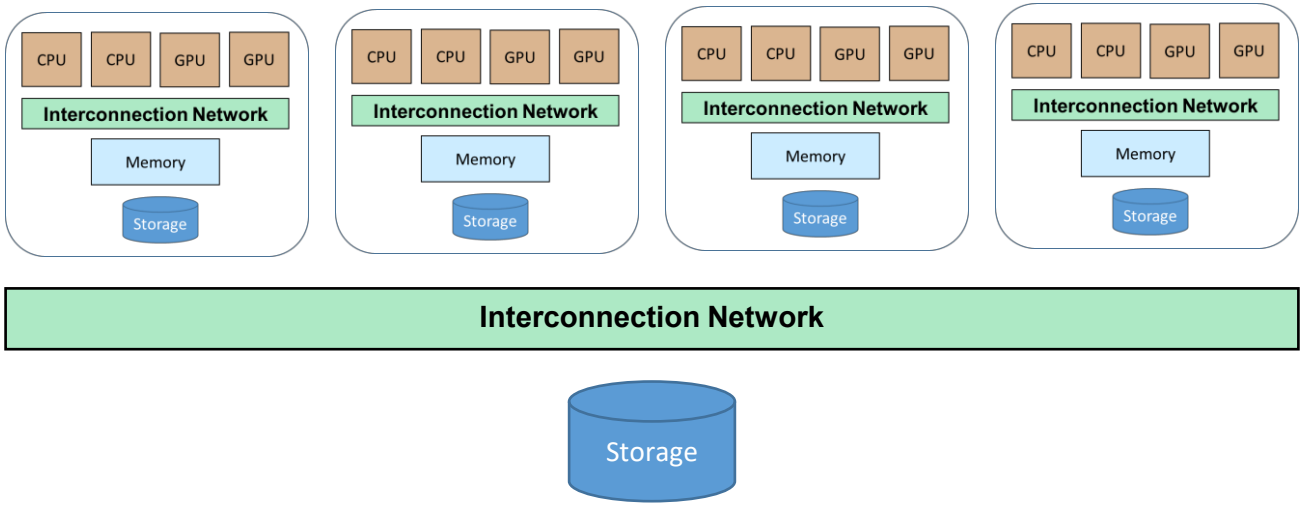
Cluster with centralized file system



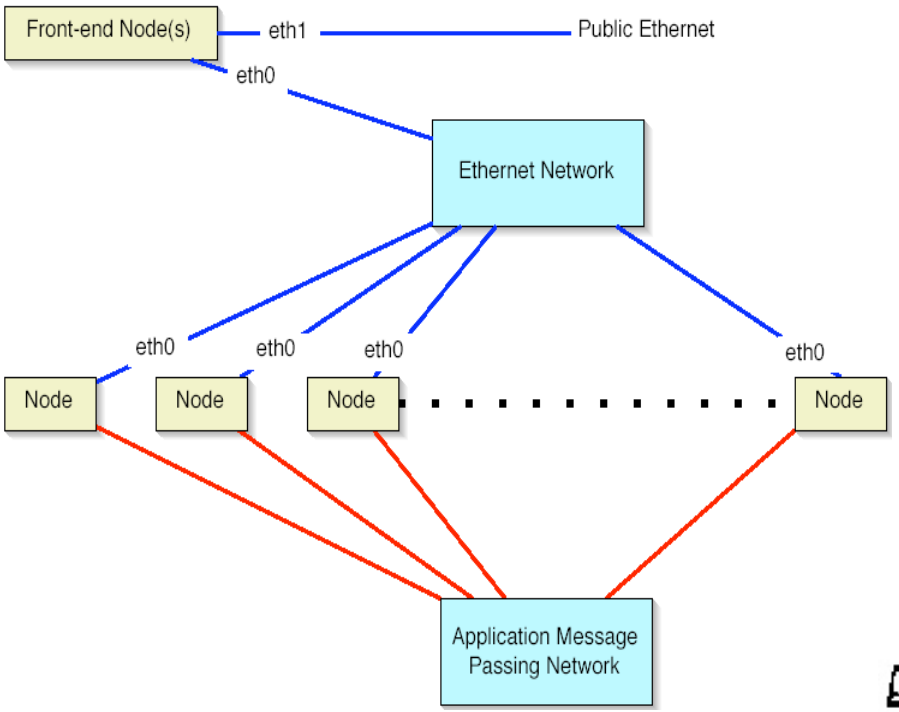
Cluster with distributed file system



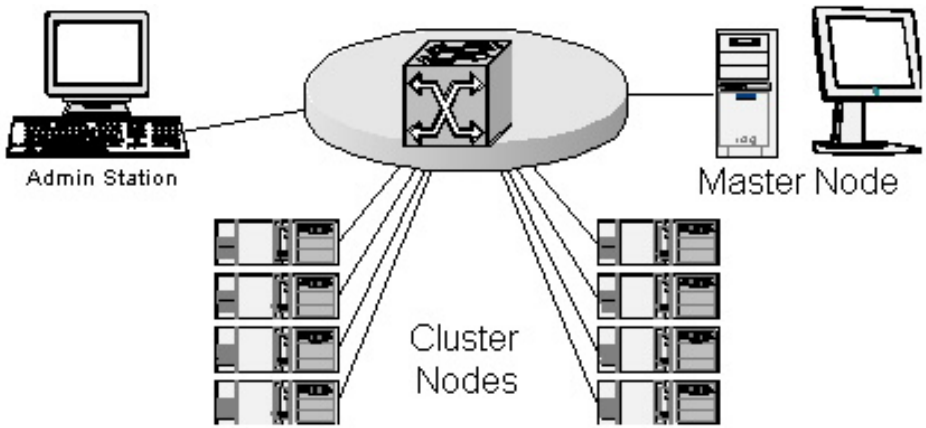
All-in-One Cluster



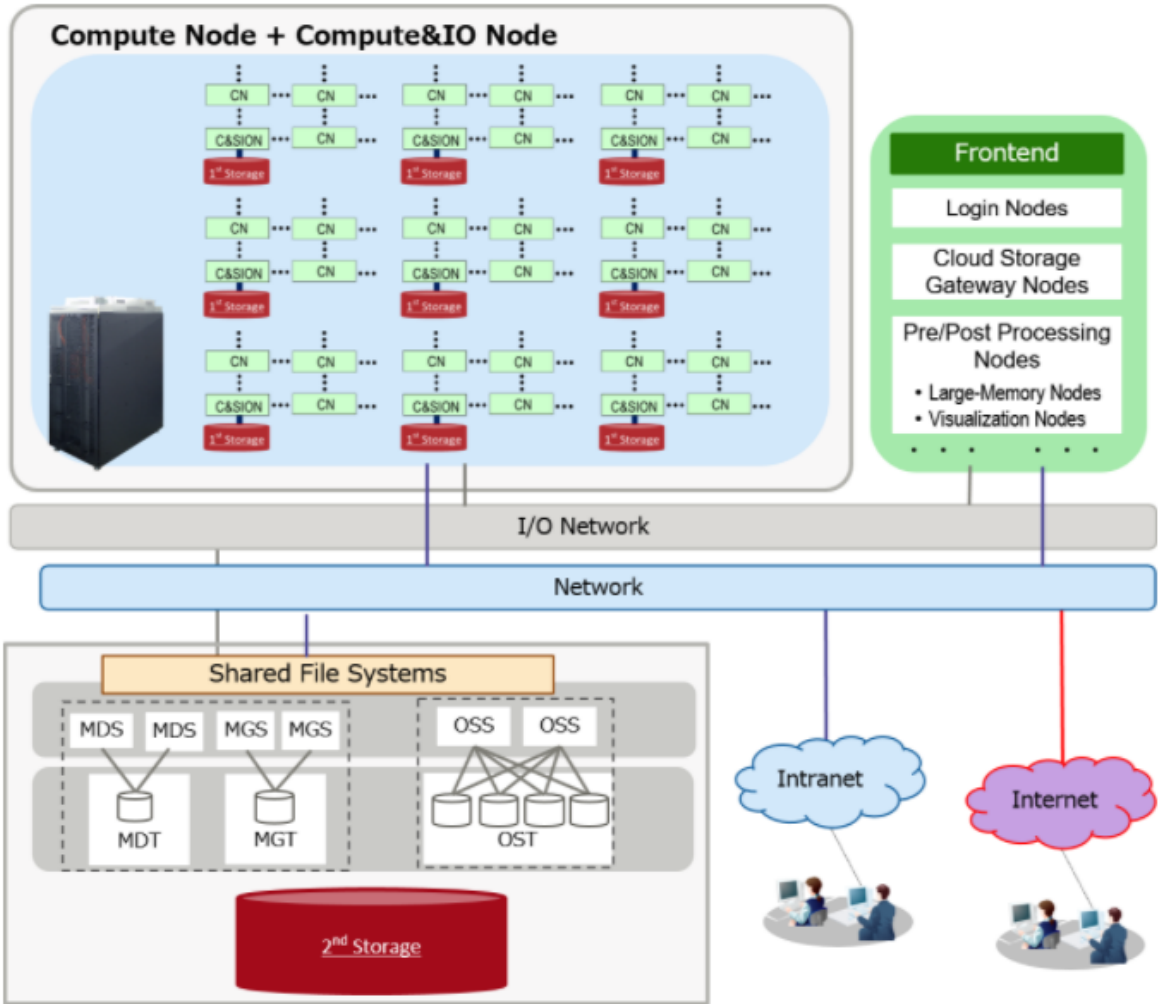
High-Performance Cluster

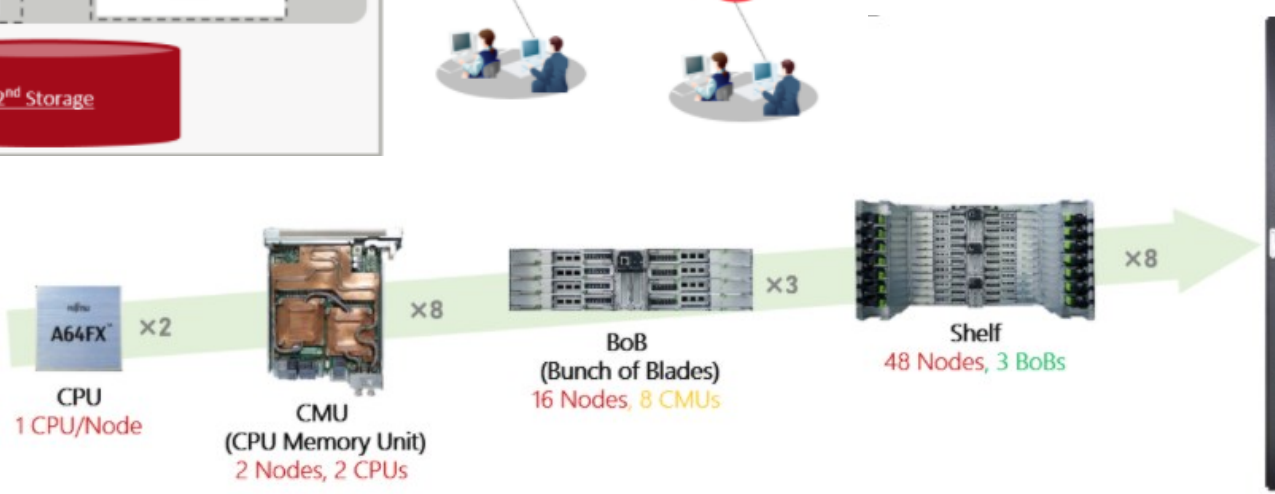


- CPU-intensive tasks
- Parallel computing
- Processes are distributed to low-cpu-utilization nodes
- Processes communicate via messages
- Move data to computing
- Programming Model: Message Passing



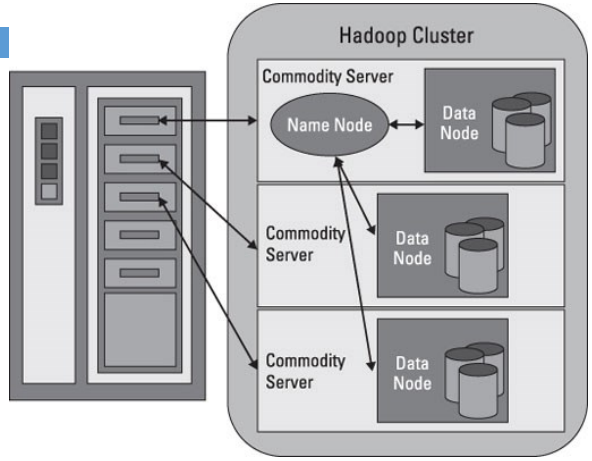
Fugaku

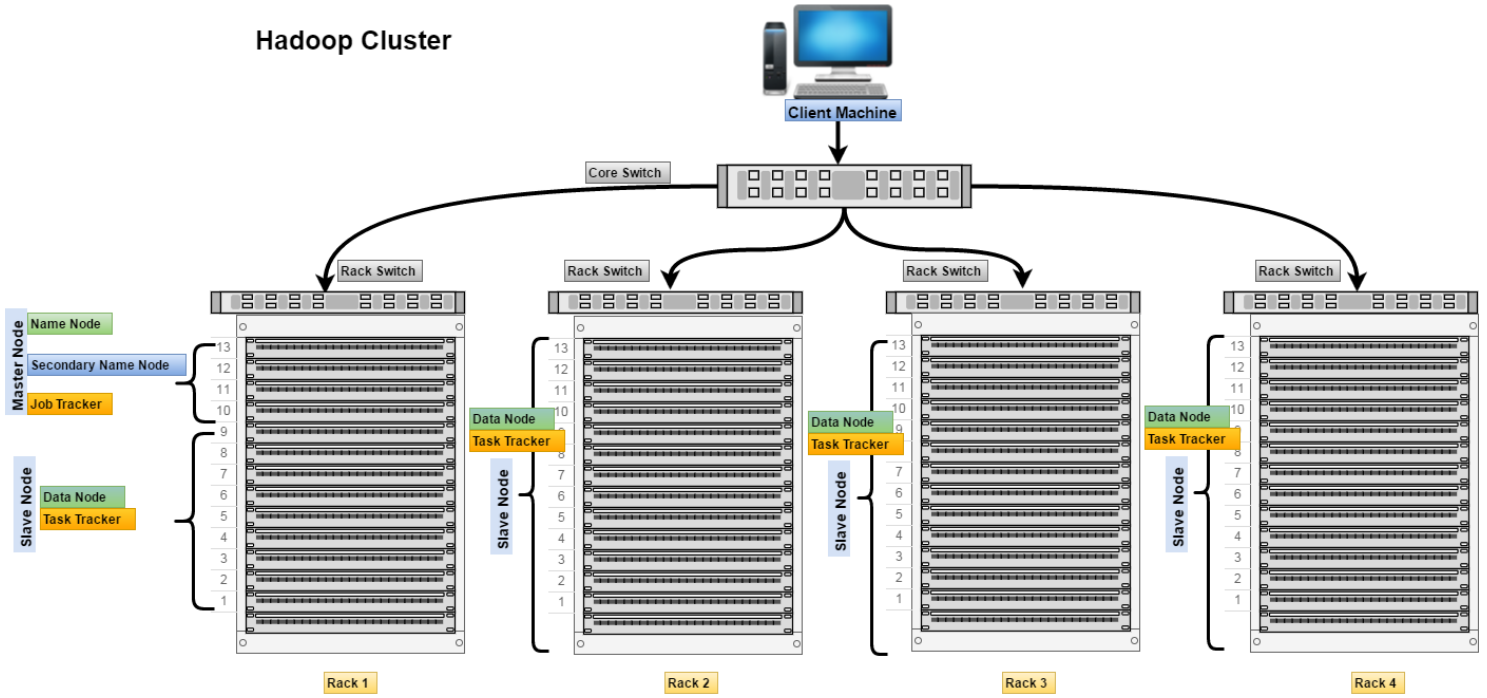

Compute Node + Compute&IO Node
CN
C&SION
1st Storage
Frontend
Login Nodes
Cloud Storage Gateway Nodes
Pre/Post Processing Nodes
• Large-Memory Nodes
• Visualization Nodes
I/O Network
Network
Shared File Systems
MDS
MDS
MGS
MGS
OSS
OSS
MDT
MGT
OST
2nd Storage
Intranet
Internet


A64FX
×2
CPU
1 CPU/Node
×8
CMU
(CPU Memory Unit)
2 Nodes, 2 CPUs
×3
BoB
(Bunch of Blades)
16 Nodes, 8 CMUs
Shelf
48 Nodes, 3 BoBs
×8
Rack
384 Nodes, 8 Shelves

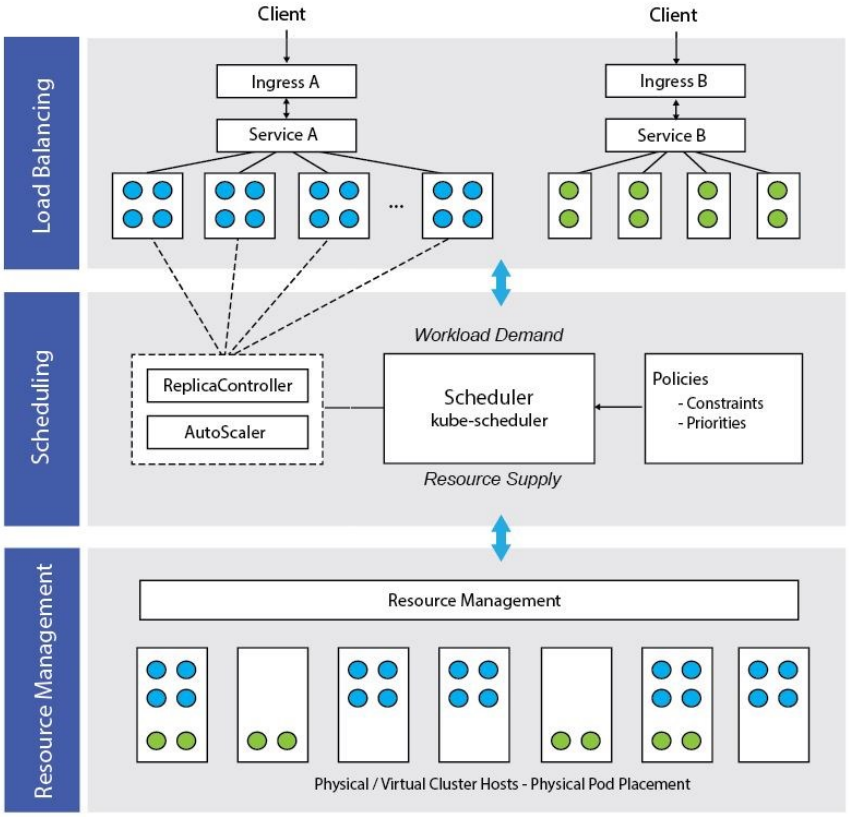
Big Data Cluster

- Data-intensive computation
- High-availability distributed file system
  - Each node is a part of a distributed file system
- Jobs are distributed to nodes that contain required data
- Move computing to data
- Programming models: MapReduce, Resilient Distributed Data
- E.g. Hadoop, Spark





Container Cluster



- Service-oriented applications
- Container-based
  - Process-level virtualization
- Focus on
  - Availability
  - Scaling
  - Load balancing
- Multiple virtual clusters on a single physical cluster
- E.g. Kubernetes cluster

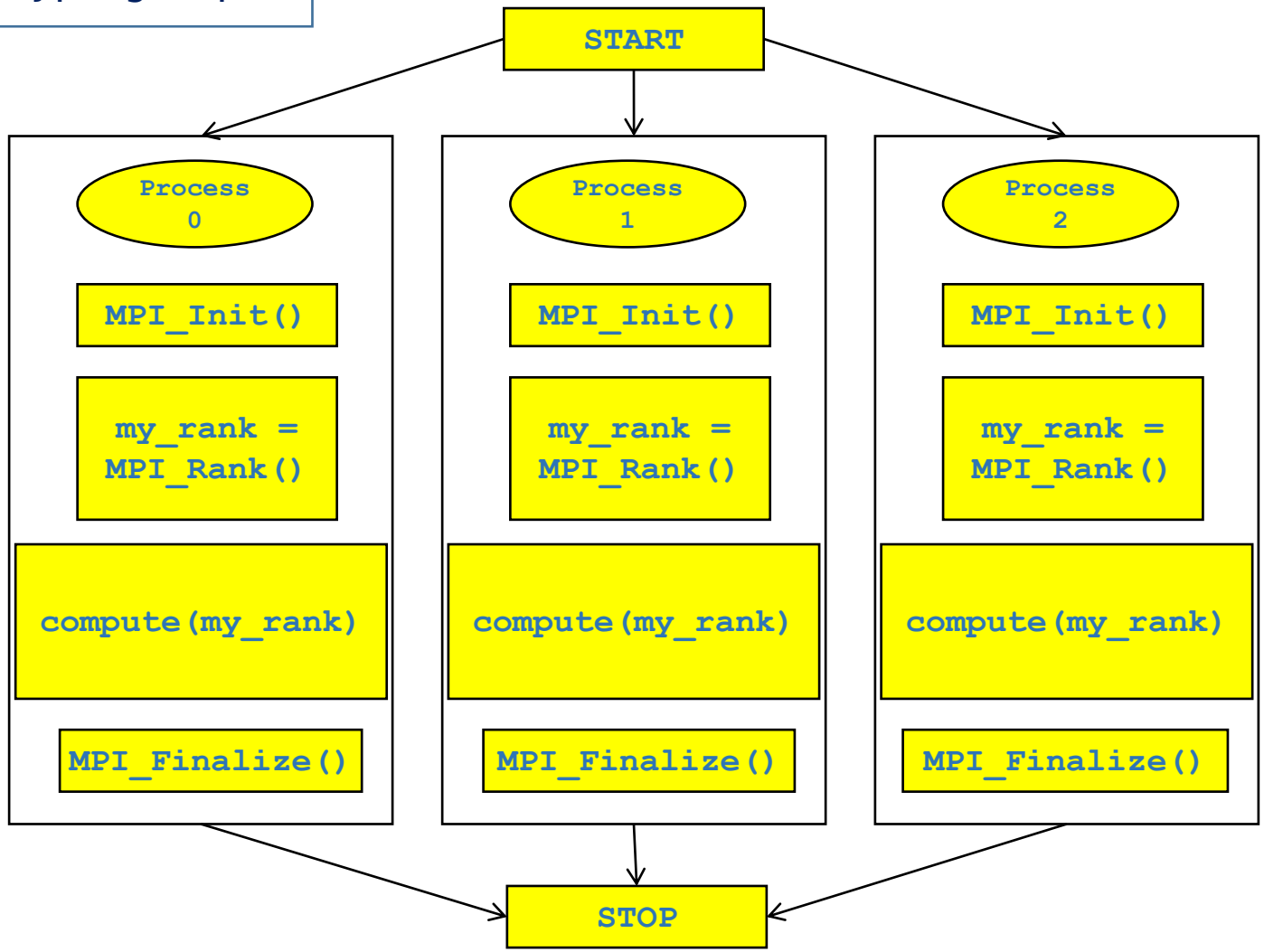
source: https://rancher.com/three-pillars-kubernetes-container-orchestration

Message-Passing Programming Model

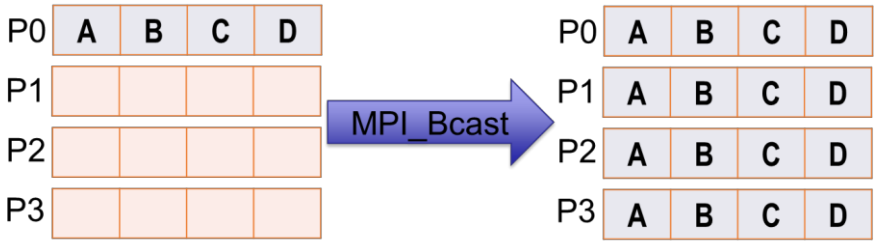
- A group of processes on one or more computers
- All processes execute the same program.
- Processes communicate and synchronize with each other by passing messages containing data.
- One-to-one and collective communication.
- MPI (Message Passing Interface) is a de facto standard.

MPI Processes

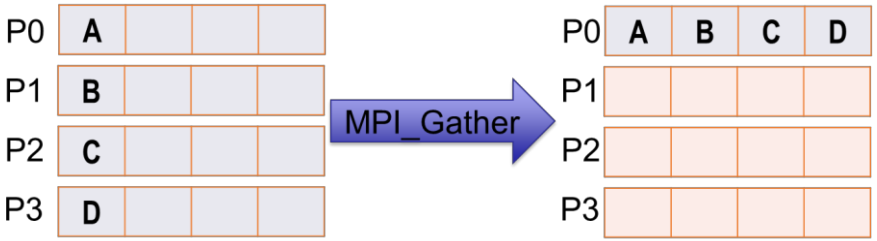
```
mpirun myprog -np 3
```



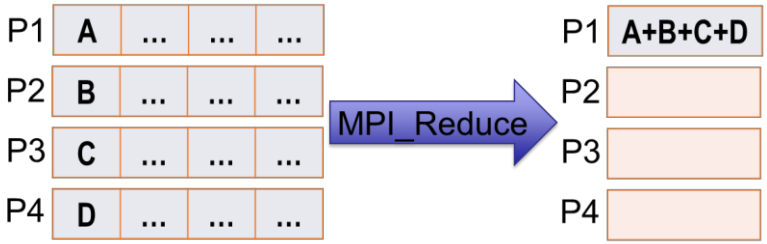
MPI Collective Communication



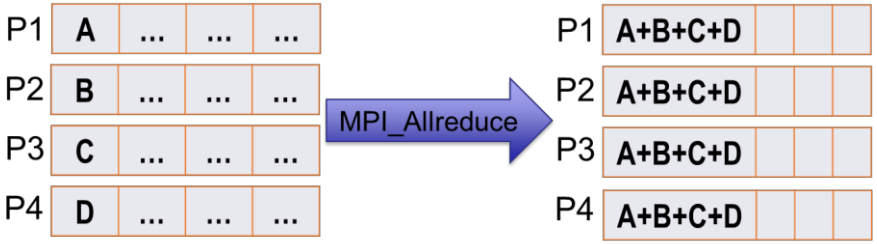
Broadcast



Gather



Reduce



All-Reduce

Example: Dot Product in MPI (in C language)

```
#include <stdio.h>
#include <mpi.h>

int main(int argc,char *argv[])
{
    double sum, sum_local, a[256], b[256];
    int n=256, i, numprocs, id, first, last;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    first = id * n/numprocs;
    last = (id + 1) * n/numprocs;
    for (i = 0; i < n; i++) {
        a [i] = i * 0.5;
        b [i] = i * 2.0;
    }
    sum_local = 0;
    for (i = first; i < last; i++) {
        sum_local = sum_local + a[i]*b[i];
    }
    MPI_Reduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (id == 0) printf ("sum = %f\n", sum);
}
```

Example: Parallel Matrix Multiplication with MPI

matmul_mpi.py

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Dimensions of the matrices A, B
n = 200

# Generate matrices A, B on root process
if rank == 0:
    A = np.random.rand(n, n)
    B = np.random.rand(n, n)
else:
    A = None
    B = None

# Broadcast matrix B to all processes
B = comm.bcast(B, root=0)

# Scatter rows of matrix A to all processes
rows = np.empty((n // size, n), dtype='float64') # Prepare a buffer for received rows
comm.Scatter(A, rows, root=0)

# Perform local computation of matrix multiplication
local_result = np.dot(rows, B)

# Gather the local results back to the root process
result = None
if rank == 0:
    result = np.empty((n, n), dtype='float64') # Prepare a buffer to receive the full result matrix
comm.Gather(local_result, result, root=0)

# Print the result in the root process
if rank == 0:
    print("Result of matrix multiplication:")
    print(result)}
```

Dask

- Dask is a Python library that allows parallel and distributed computing.
- It enables scaling from a single machine to a cluster of machines, making it suitable for processing large datasets and complex computations.

Matrix Multiplication in Dask

Multithread version

```
import dask.array as da
from dask.distributed import Client, LocalCluster

def parallel_matrix_multiplication(n, client):
    # Create random matrices
    A = da.random.random((n, n), chunks=(n // 10, n // 10))
    B = da.random.random((n, n), chunks=(n // 10, n // 10))

    # Perform matrix multiplication
    result = da.matmul(A, B)

    result_computed = result.compute()

    return result_computed

if __name__ == "__main__":
    # Setup a LocalCluster to use threads instead of processes
    with LocalCluster(processes=False, n_workers=4, threads_per_worker=2) as cluster:
        with Client(cluster) as client:
            n = 2000  # Size of the matrix
            result = parallel_matrix_multiplication(n, client)
            print(f"Result of the matrix multiplication: \n{result}")
```

Matrix Multiplication in Dask

Distributed computing version

```
import dask.array as da
from dask.distributed import Client, performance_report

def parallel_matrix_multiplication(n, client):
    # Create random matrices
    A = da.random.random((n, n), chunks=(n // 10, n // 10))
    B = da.random.random((n, n), chunks=(n // 10, n // 10))

    # Perform matrix multiplication
    result = da.matmul(A, B)

    # Compute the result with a performance report
    with performance_report(filename="dask_report.html"):
        result_computed = result.compute()

    return result_computed

if __name__ == "__main__":
    # Connect to your Dask cluster
    client = Client("tcp://scheduler:8786")  # Replace with your scheduler address

    n = 2000  # Size of the matrix
    result = parallel_matrix_multiplication(n, client)
    print(f"Result of the matrix multiplication: \n{result}")
```

Dask Cluster on Docker

docker-compose.yml

```
version: '3'

services:
  scheduler:
    image: daskdev/dask:latest
    command: dask-scheduler
    ports:
      - "8786:8786" # Dask scheduler dashboard
      - "8787:8787" # Dask diagnostics dashboard
  volumes:
      - ./data:/data

  worker:
    image: daskdev/dask:latest
    command: dask-worker scheduler:8786
  depends_on:
      - scheduler
  deploy:
    resources:
      limits:
        cpus: "0.25"
        memory: 512M

  jupyter:
    image: daskdev/dask-notebook:latest
    ports:
      - "8888:8888"
    command: start-notebook.sh --NotebookApp.token='mytoken'
    environment:
      - DASK_SCHEDULER_ADDRESS=tcp://scheduler:8786
      - JUPYTER_TOKEN=mytoken
    volumes:
      - ./notebooks:/home/jovyan
    depends_on:
      - scheduler
```

When running on a single machine, setting limits of resources for each worker will better simulate cluster scaling (adding workers → more resources).
Without this setting, total resources are divided among workers.
When running on a real cluster with multiple machines, this can be removed.

This configuration sets up:

- A Dask scheduler service, exposing ports 8786 (Dask dashboard) and 8787 (Diagnostics dashboard).
- A Dask worker service, which connects to the scheduler.
- An optional Jupyter notebook server with Dask integration, exposing port 8888 for web access. The notebook server automatically connects to the Dask scheduler.

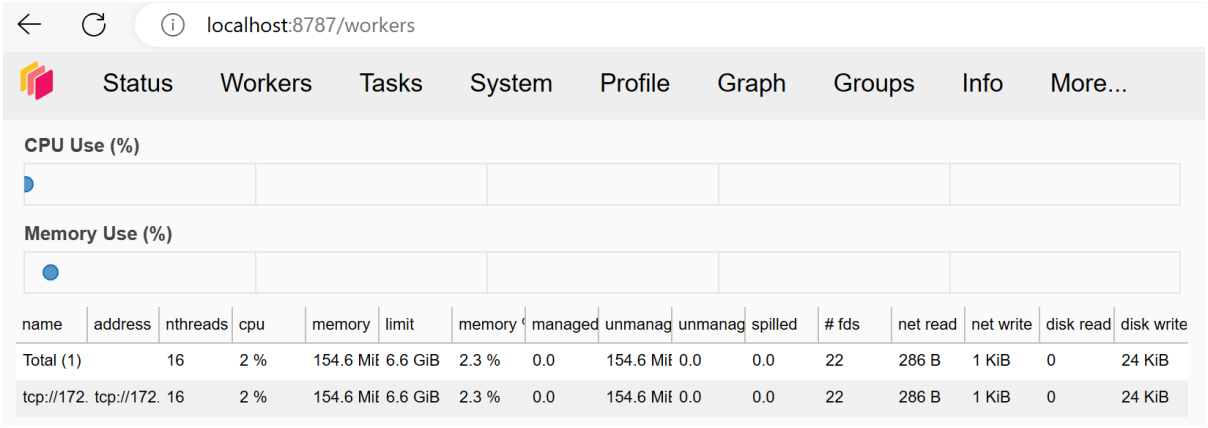
Running Dask Cluster

```
docker-compose up --scale worker=4
```

Remove Dask Cluster

```
docker-compose down
```

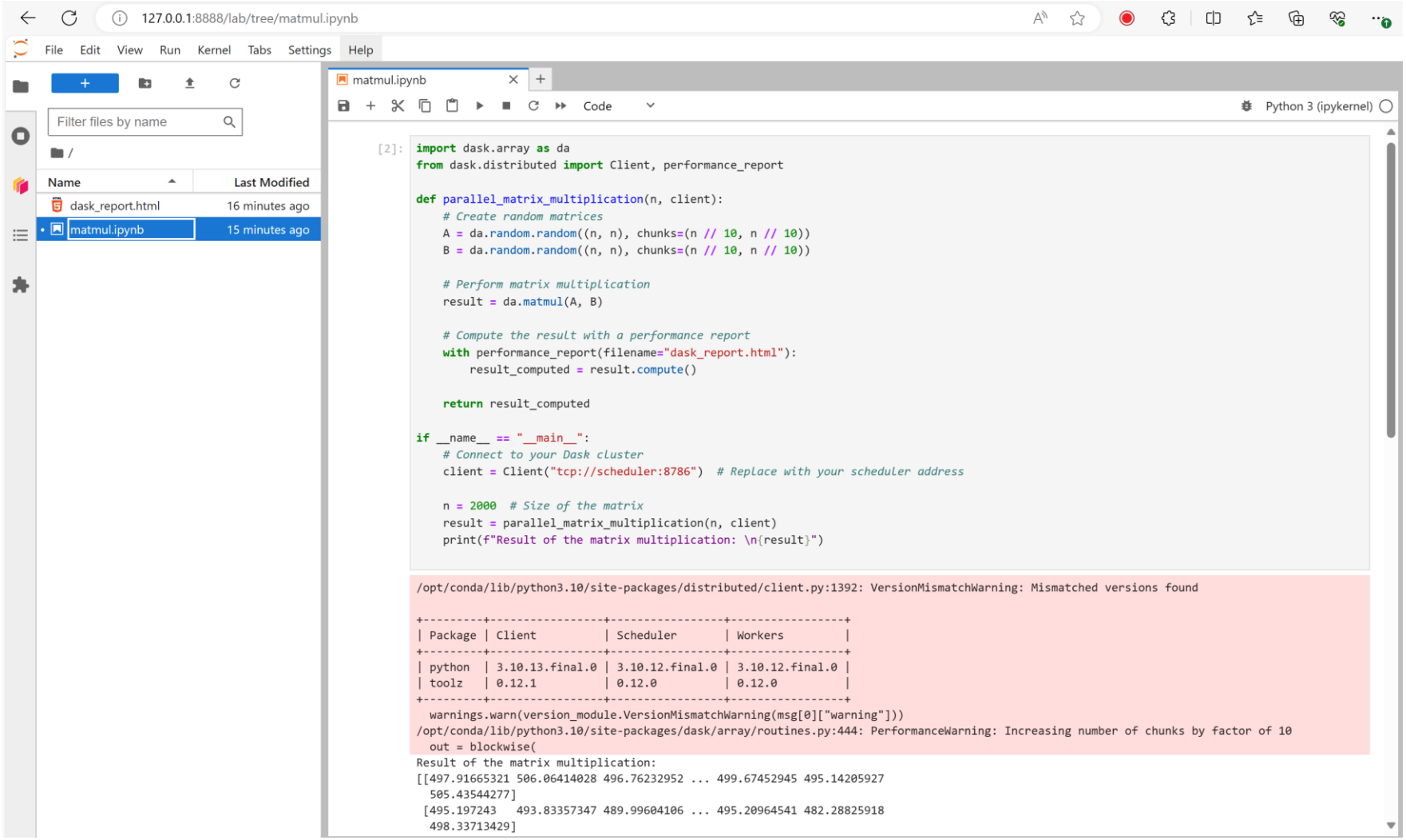
```
docker-compose up --scale worker=1
```



name	address	nthreads	cpu	memory	limit	memory	managed	unmanag	unmanag	spilled	# fds	net read	net write	disk read	disk write
Total (1)		16	2 %	154.6 Mil	6.6 GiB	2.3 %	0.0	154.6 Mil	0.0	0.0	22	286 B	1 KiB	0	24 KiB
tcp://172.	tcp://172.	16	2 %	154.6 Mil	6.6 GiB	2.3 %	0.0	154.6 Mil	0.0	0.0	22	286 B	1 KiB	0	24 KiB

```
docker-compose up --scale worker=4
```

name	address	nthreads	cpu	memory	limit	memory	managed	unmanag	unmanag	spilled	# fds	net read	net write	disk read	disk write
Total (4)		64	3 %	619.3 Mil	26.5 GiB	2.3 %	0.0	235.4 Mil	383.9 Mil	0.0	88	1 KiB	6 KiB	0	192 KiB
tcp://172.	tcp://172.	16	2 %	155.1 Mil	6.6 GiB	2.3 %	0.0	58.8 MiB	96.3 MiB	0.0	22	286 B	1 KiB	0	48 KiB
tcp://172.	tcp://172.	16	6 %	153.4 Mil	6.6 GiB	2.3 %	0.0	58.7 MiB	94.7 MiB	0.0	22	286 B	1 KiB	0	48 KiB
tcp://172.	tcp://172.	16	2 %	155.7 Mil	6.6 GiB	2.3 %	0.0	59.1 MiB	96.6 MiB	0.0	22	287 B	1 KiB	0	48 KiB
tcp://172.	tcp://172.	16	2 %	155.1 Mil	6.6 GiB	2.3 %	0.0	58.7 MiB	96.3 MiB	0.0	22	287 B	1 KiB	0	48 KiB


127.0.0.1:8888/lab/tree/matmul.ipynb
File Edit View Run Kernel Tabs Settings Help
Filter files by name
/
Name
Last Modified
dask_report.html
16 minutes ago
matmul.ipynb
15 minutes ago
matmul.ipynb
Code
Python 3 (ipykernel)
[2]:
import dask.array as da
from dask.distributed import Client, performance_report
def parallel_matrix_multiplication(n, client):
# Create random matrices
A = da.random.random((n, n), chunks=(n // 10, n // 10))
B = da.random.random((n, n), chunks=(n // 10, n // 10))
# Perform matrix multiplication
result = da.matmul(A, B)
# Compute the result with a performance report
with performance_report(filename="dask_report.html"):
result_computed = result.compute()
return result_computed
if __name__ == "__main__":
# Connect to your Dask cluster
client = Client("tcp://scheduler:8786") # Replace with your scheduler address
n = 2000 # Size of the matrix
result = parallel_matrix_multiplication(n, client)
print(f"Result of the matrix multiplication: \n{result}")
/opt/conda/lib/python3.10/site-packages/distributed/client.py:1392: VersionMismatchWarning: Mismatched versions found
+---------+-----------------+-----------------+-----------------+
| Package | Client | Scheduler | Workers |
+---------+-----------------+-----------------+-----------------+
| python | 3.10.13.final.0 | 3.10.12.final.0 | 3.10.12.final.0 |
| toolz | 0.12.1 | 0.12.0 | 0.12.0 |
+---------+-----------------+-----------------+-----------------+
warnings.warn(version_module.VersionMismatchWarning(msg[0]["warning"]))
/opt/conda/lib/python3.10/site-packages/dask/array/routines.py:444: PerformanceWarning: Increasing number of chunks by factor of 10
out = blockwise(
Result of the matrix multiplication:
[[497.91665321 506.06414028 496.76232952 ... 499.67452945 495.14205927
505.43544277]
[495.197243 493.83357347 489.99604106 ... 495.20964541 482.28825918
498.33713429]

dask_report.html

Summary | Task Stream | System | Scheduler Logs | Worker Profile (compute) | Worker Profile (administrative) | Scheduler Profile (administrative) | Bandwidth (Workers) | Bandwidth (Types)

Dask Performance Report

Select different tabs on the top for additional information

Duration: 5.97 s

Tasks Information

- number of tasks: 1600
- compute time: 65.32 s
- disk-read time: 266.02 ms
- disk-write time: 322.09 ms
- transfer time: 128.91 s

Scheduler Information

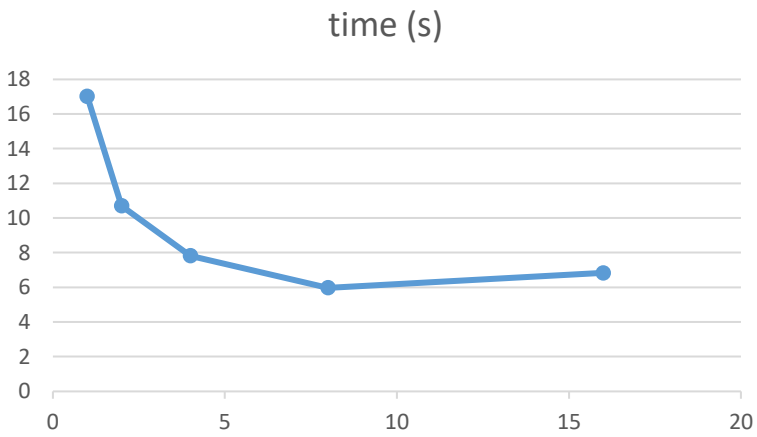
- Address: tcp://172.19.0.2:8786
- Workers: 8
- Threads: 128
- Memory: 4.00 GiB
- Dask Version: 2024.2.1
- Dask.Distributed Version: 2024.2.1

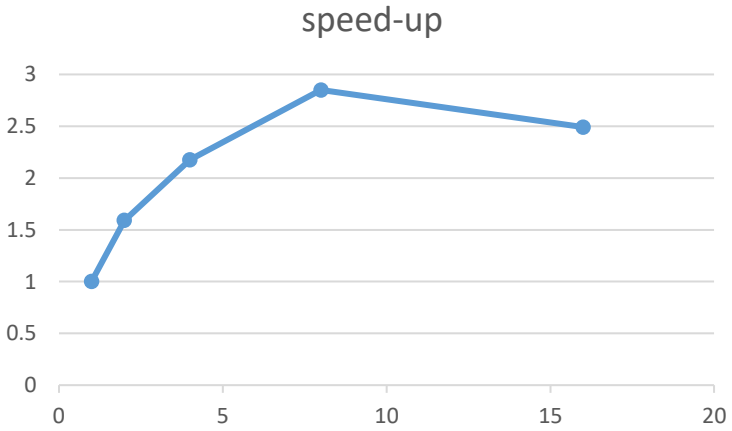
Calling Code

```
def parallel_matrix_multiplication(n, client):
    # Create random matrices
    A = da.random.random((n, n), chunks=(n // 10, n // 10))
    B = da.random.random((n, n), chunks=(n // 10, n // 10))

    # Perform matrix multiplication
    result = da.matmul(A, B)

    # Compute the result with a performance report
    with performance_report(filename="dask_report.html"):
        result_computed = result.compute()

    return result_computed
```

Speed-up

workers	time (s)	speed-up
1	17.01	1
2	10.7	1.58972
4	7.82	1.368286
8	5.97	1.309883
16	6.83	0.874085





Computing intensive : ดีหลายเลยอะ 7

Parallel Computing for Big Data

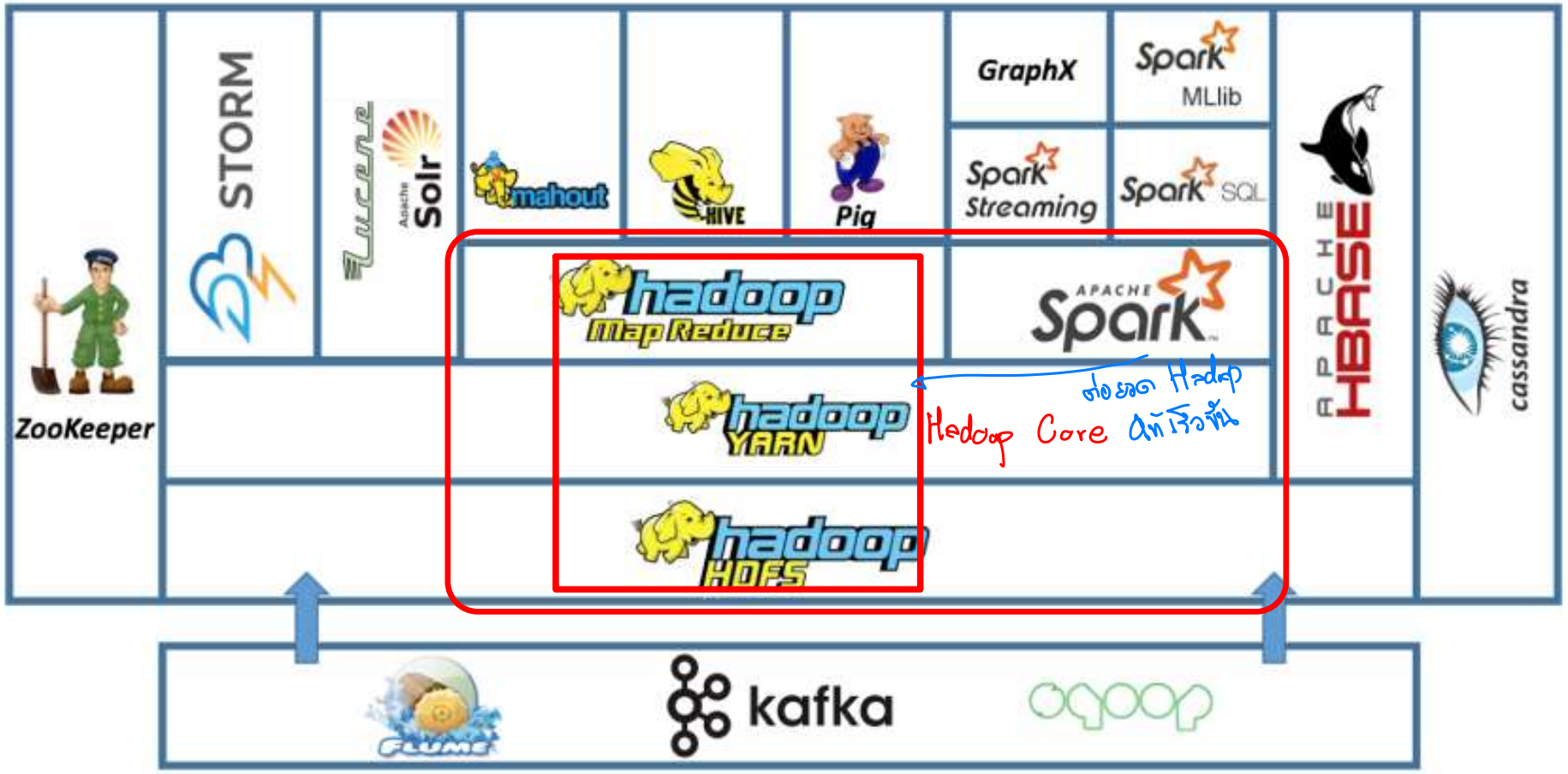
- Data-Parallel Model
- Instead of explicit communication between nodes (as in Message-Passing Model), the focus is on partitioning large datasets and applying operations on these partitions in parallel.
- The programming model works on top of distributed file systems and distributed data processing frameworks.
  - Ex: Apache Hadoop (with HDFS), Apache Spark (with RDDs)
- Key concepts:
  - Large datasets are partitioned and distributed across cluster nodes
  - Same operation or function is applied to each partition concurrently
  - Framework handles distribution, communication, and coordination

Apache Hadoop – Big Data Computing หลักการทำงาน

- An open-source framework for distributed storage and processing that led to Big Data era FW ที่ช่วยกลุ่มงาน distribute storage กับการประมวลผลแยกจากกัน
- Key Features: – HADOOP เก็บข้อมูลหลายๆ หรือรองรับ Parallel
  - Scalability: Easily scales out by adding more nodes to the cluster. สร้างเครื่องใหญ่ๆ เพิ่ม CPU, node
  - Fault Tolerance: Automatically handles failures by replicating data blocks. สำรองกรณี HW เสียอยู่แล้ว Hadoop บอกว่าบาง node พัง แต่ทั้งระบบยังทำงานต่อได้
- Core Components: องค์ประกอบ Hadoop
  - Data storage → HDFS (Hadoop Distributed File System): Scalable, highly available storage system that splits large data files into blocks distributed across nodes. ข้อมูลเก็บยังไง HDFS file ใหญ่แบ่งเป็น block ย่อยๆ แล้วกระจายไปยังข้าม node
  - Resource Manager → YARN (Yet Another Resource Negotiator): Manages and schedules resources across clusters. ทำหน้าที่กระจายงาน ตัดสินใจว่าจะส่งงานไปที่ node ไหนโดยใช้ Yarn
  - Programming → MapReduce: Programming model for processing large data sets with a parallel, distributed algorithm. Based on map and reduce operations. เขียน map, reduce function ลงไปที่ข้อมูล

พิจารณาจากที่เก็บข้อมูล, load

Apache Hadoop Ecosystem



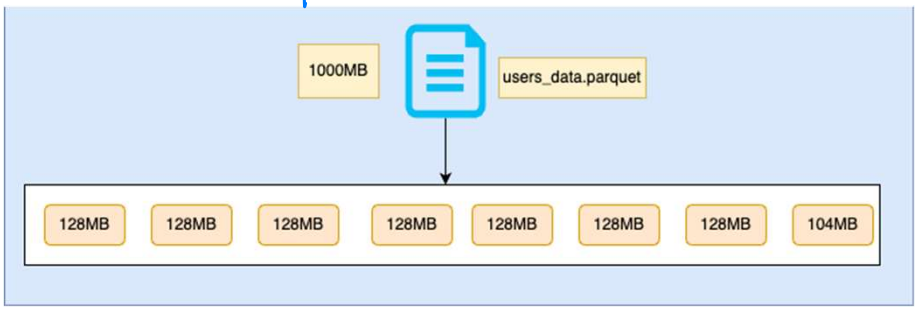
Hadoop Distributed File System (HDFS)

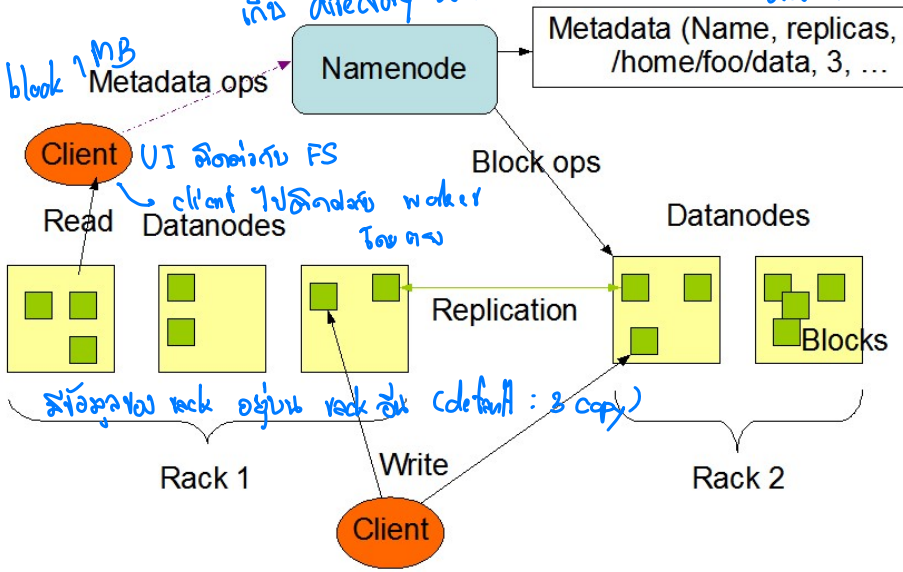
มี storage บน node ต่างๆ แล้วใช้ Hadoop มองเป็นก้อนเดียวกัน

- Distributed file system for storing large datasets across a cluster
- Support a traditional hierarchical file organization (directories and files)
- Run on commodity hardware หาซื้อ HW ได้ทั่วไป ตรงข้ามกับ Proprietary HW ที่ต้องระบุยี่ห้อ
- Focus on fault-tolerance and high throughput
- Optimized for
  - large files รองรับ file ใหญ่ ถ้าเป็นไฟล์เล็กหลายไฟล์จะไม่เหมาะ
  - write-once-read-many access pattern log data, archeive สร้างข้อมูลมาแล้วไม่สั่งการ เปลี่ยนแปลง
  - batch processing เหมาะกับงานที่ไม่ใช่ real time

HDFS Architecture

- A file is divided into blocks (default size = 128MB).
- Each block is replicated and stored on multiple nodes.
- Files are write-once (except for appends and truncates)
- Components
  - HDFS Client
    - The interface through which applications interact with HDFS
  - Name Node (master)
    - Manages file system metadata and coordinates accesses
    - Keeps track of block locations
  - Data Nodes (workers)
    - Store and serve the actual file data blocks

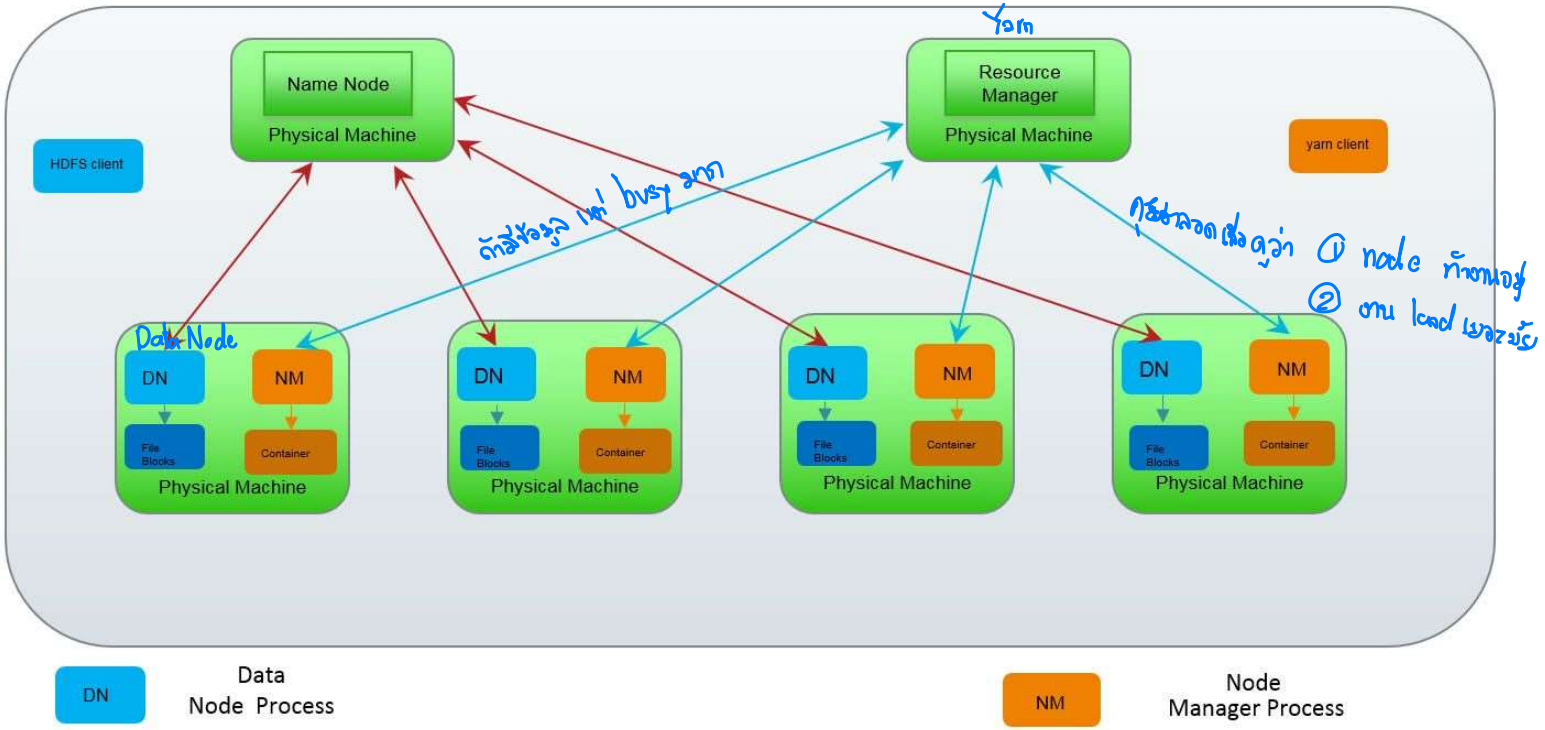




https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

YARN (Yet Another Resource Negotiator)

- Cluster resource management layer in Hadoop
- YARN schedules tasks across cluster nodes, considering resource availability, data locality for input splits, and efficient utilization of cluster resources.
- ResourceManager (RM):
  - Central process that manages and allocates cluster resources. Schedules jobs and allocates containers
- NodeManager (NM):
  - Runs on each node in the cluster and manages the resources on that node. Monitors the life-cycle of containers running on the node.

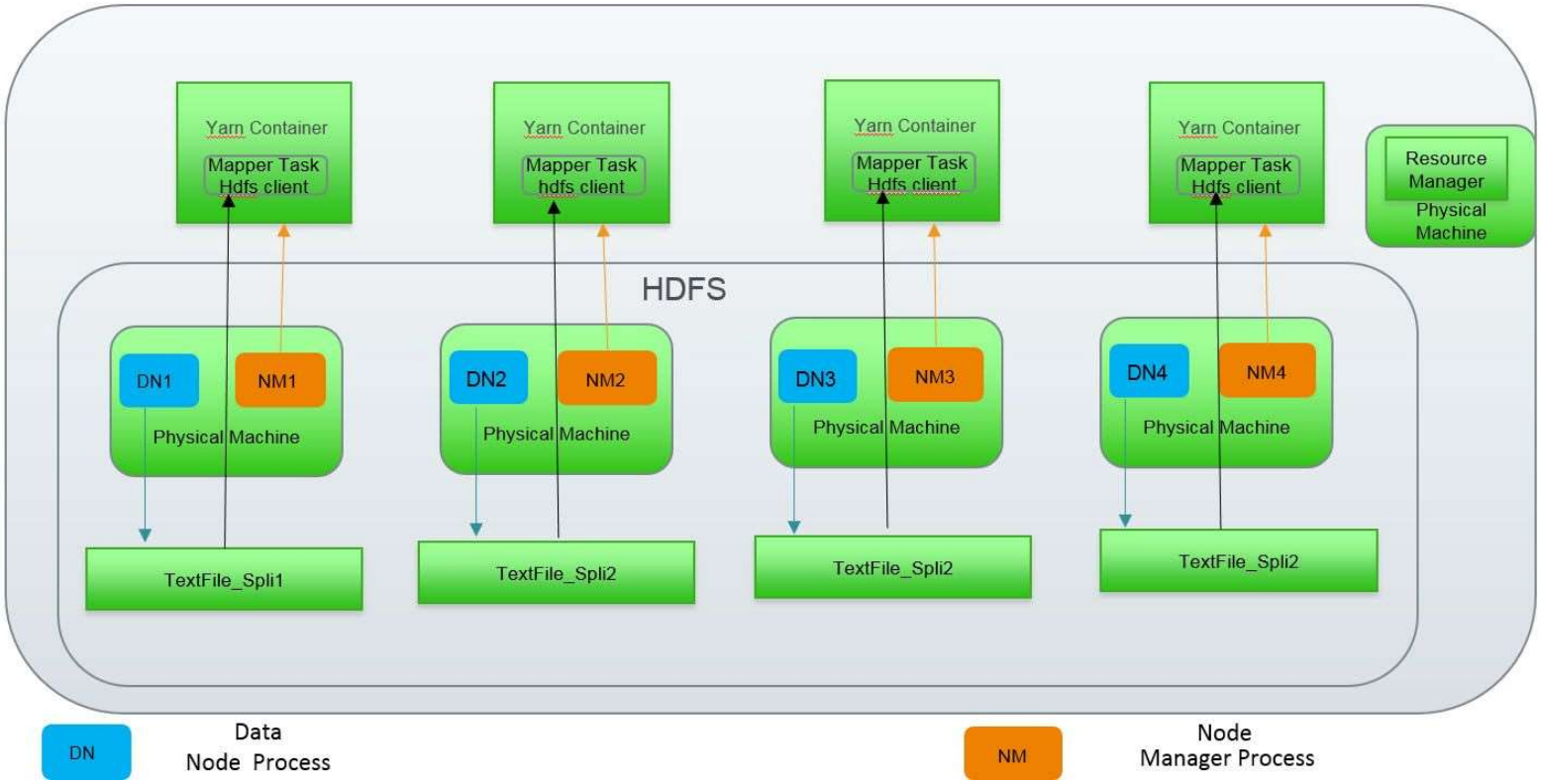


https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

YARN Scheduling

YARN schedules MapReduce tasks (in this case, Mapper tasks) across containers running on different nodes.

- ResourceManager allocates YARN containers to NodeManagers across the cluster nodes.
- Each container runs a process (Mapper Task) that reads input data from HDFS splits distributed across DataNodes.
- YARN tries to achieve data locality by scheduling Mapper Tasks on nodes where the input data resides, to minimize data movement across the network.
- If data locality is not possible, YARN schedules tasks on available nodes, and the Mapper Tasks read data from remote DataNodes.



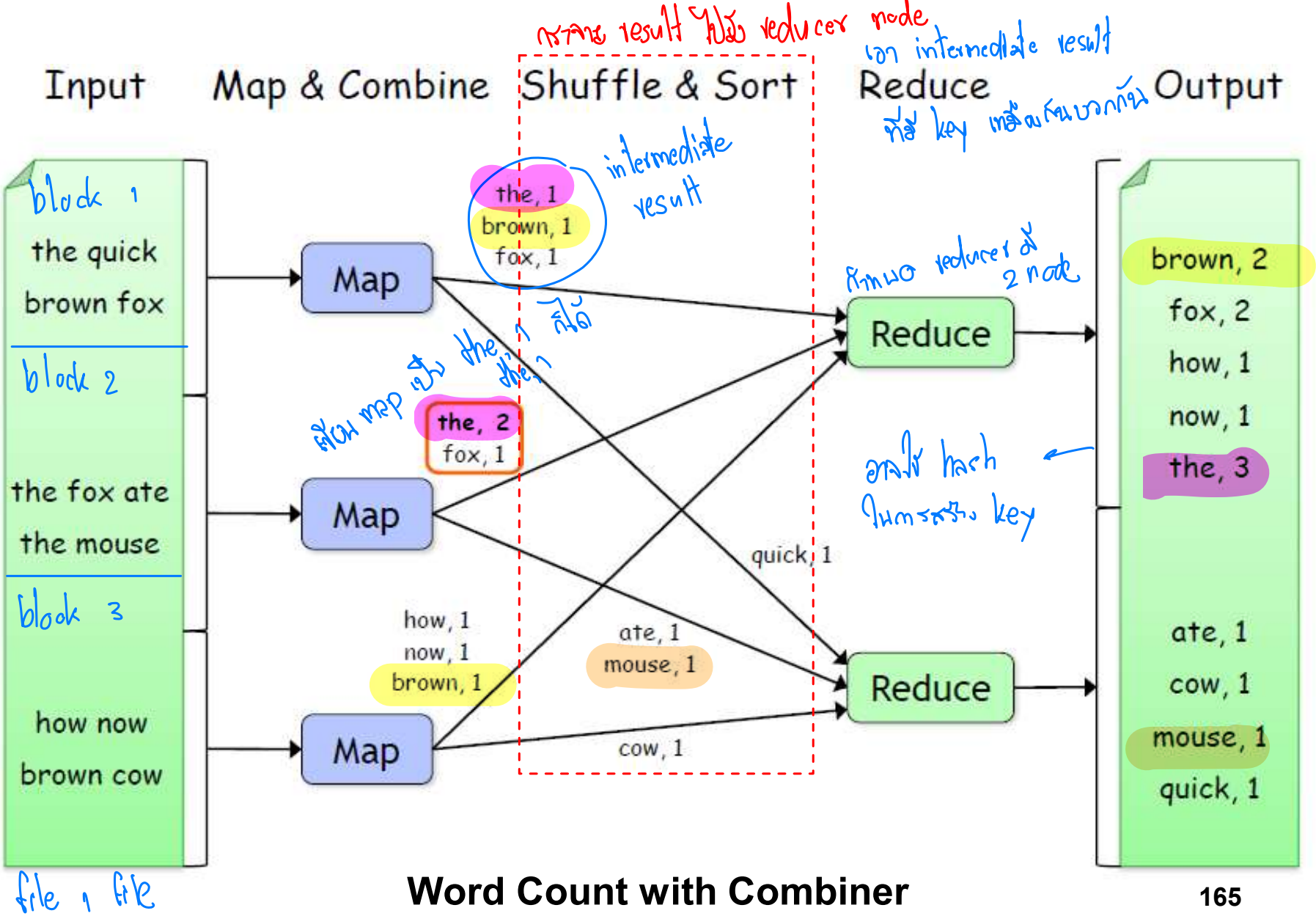
https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

MapReduce Programming Model

- A framework for processing large datasets in a parallel and distributed manner across a cluster.
- It consists of two main phases: the Map phase and the Reduce phase.
- **Map Phase**:
  - The input data is split into multiple chunks (key-value pairs) and distributed across the cluster nodes.
  - The *Map function* is applied to each key-value pair, producing intermediate key-value pairs as output.
  - The intermediate key-value pairs are sorted and partitioned based on their keys.
- Shuffle and Sort:
  - The intermediate key-value pairs are redistributed across the cluster nodes based on their keys, so that all values with the same key are sent to the same node.
  - The values with the same key are sorted and prepared for the Reduce phase.
- **Reduce Phase**:
  - The *Reduce function* is applied to each unique key and its corresponding values (the sorted values from the Map phase).
  - The Reduce function performs operations (such as aggregation, filtering, or transformation) on the values and produces the final output key-value pairs.
  - The final output is written to the distributed file system (e.g., HDFS in Hadoop).

http://hadoop.apache.org/docs/stable3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

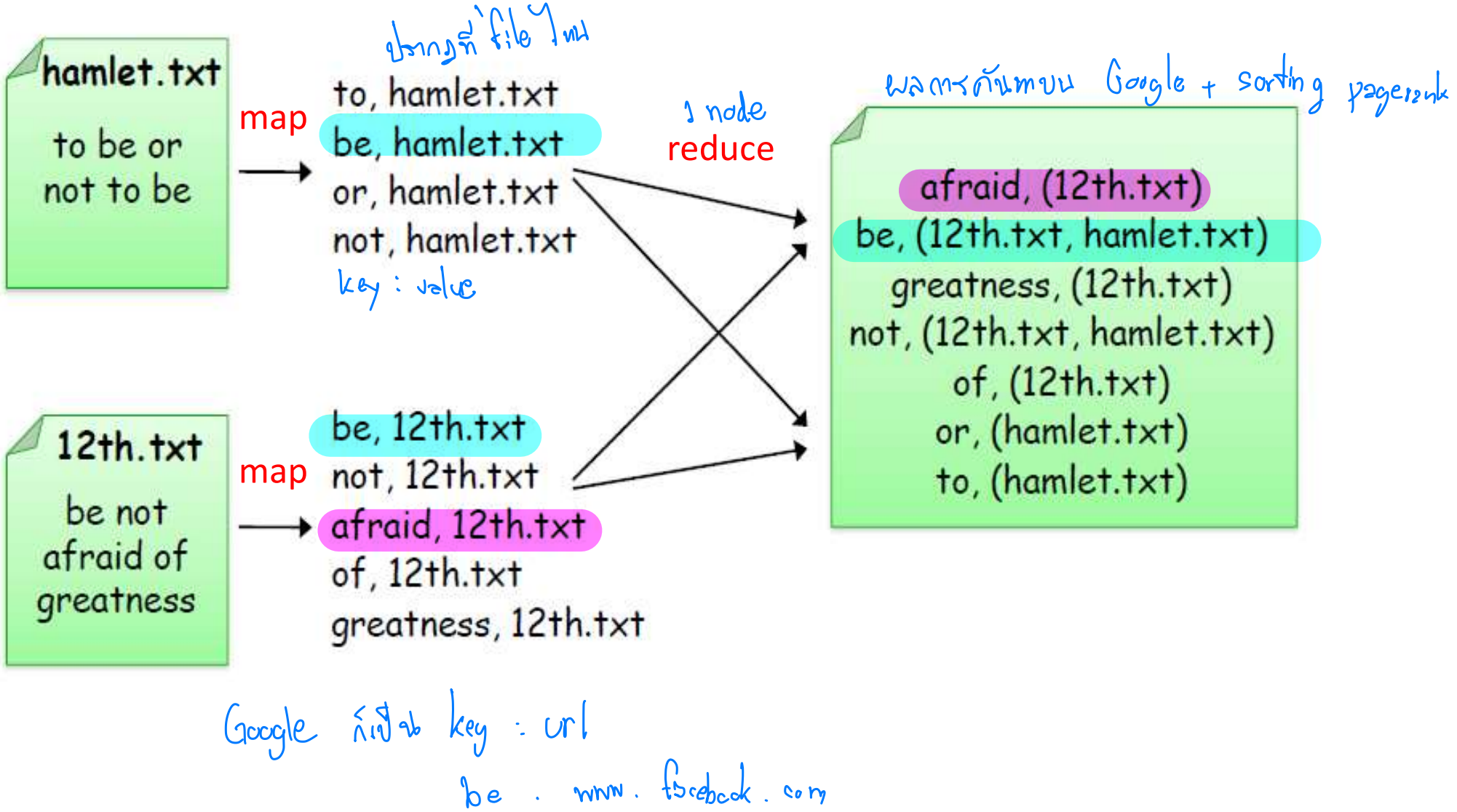
Ex: word count with MapReduce



Word Count with Combiner

keyword มีปรากฏที่ไหน, บ้างไหน

Ex. Inverted Index with MapReduce



Map-Reduce Programming Model

- Input data type: file of K/V records
- Map function: (K_{in}, V_{in}) ➔ list(K_{inter}, V_{inter})
- Reduce function: $(K_{inter}, \text{list}(V_{inter}))$ ➔ list(K_{out}, V_{out})
- Example:

```
def mapper(line):
    foreach  word in line.split();
        output(word, 1);          // (K_inter, V_inter)
```

{(the, 1)
(the, 1)
(the, 1)
(be, 2)}

```
def  reducer(key, values):
    output(key, sum(values));     // list(K_out, V_out)
```

{(the, 3)
(be, 1)}

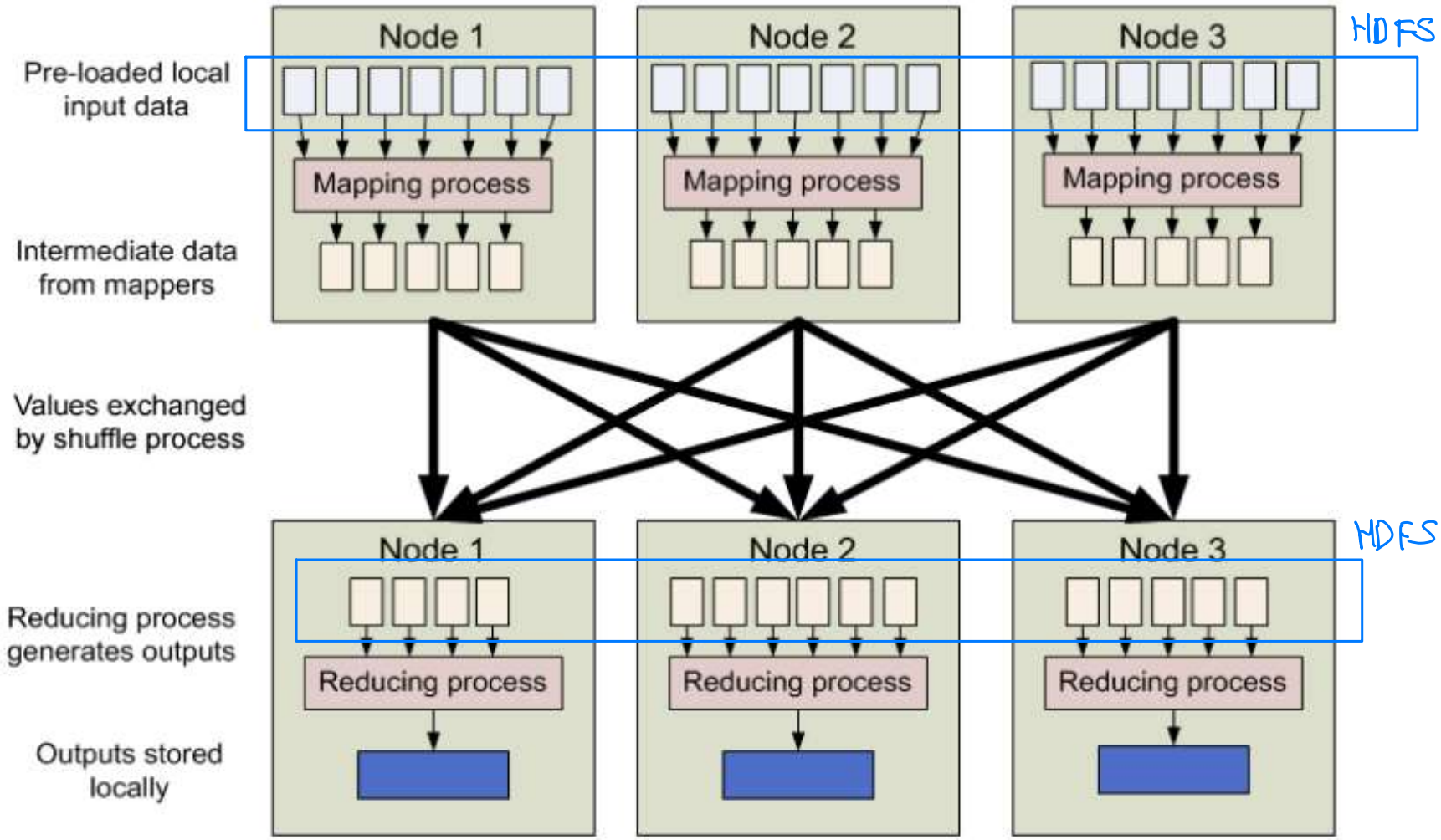
Word Count with MapReduce in Python

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class WordCount(MRJob):
    def mapper(self, _, line):
        """
        The mapper function splits the input line into words and emits (word, 1) pairs.
        """
        words = line.split()
        for word in words:
            yield word, 1

    def combiner(self, word, counts):
        """
        The combiner function performs a local aggregation of counts for each word on the mapper node.
        """
        yield word, sum(counts)

    def reducer(self, word, counts):
        """
        The reducer function aggregates the counts from all mapper nodes and emits the final (word, count) pairs.
        """
        yield word, sum(counts)

    def steps(self):
        """
        The steps method defines the MapReduce job workflow.
        """
        return [
            MRStep(mapper=self.mapper,
                   combiner=self.combiner,
                   reducer=self.reducer)
        ]

if __name__ == '__main__':
    WordCount.run()
```

fn การรวม (the, 1) (the, 1) เป็นก้อน (the, 2)

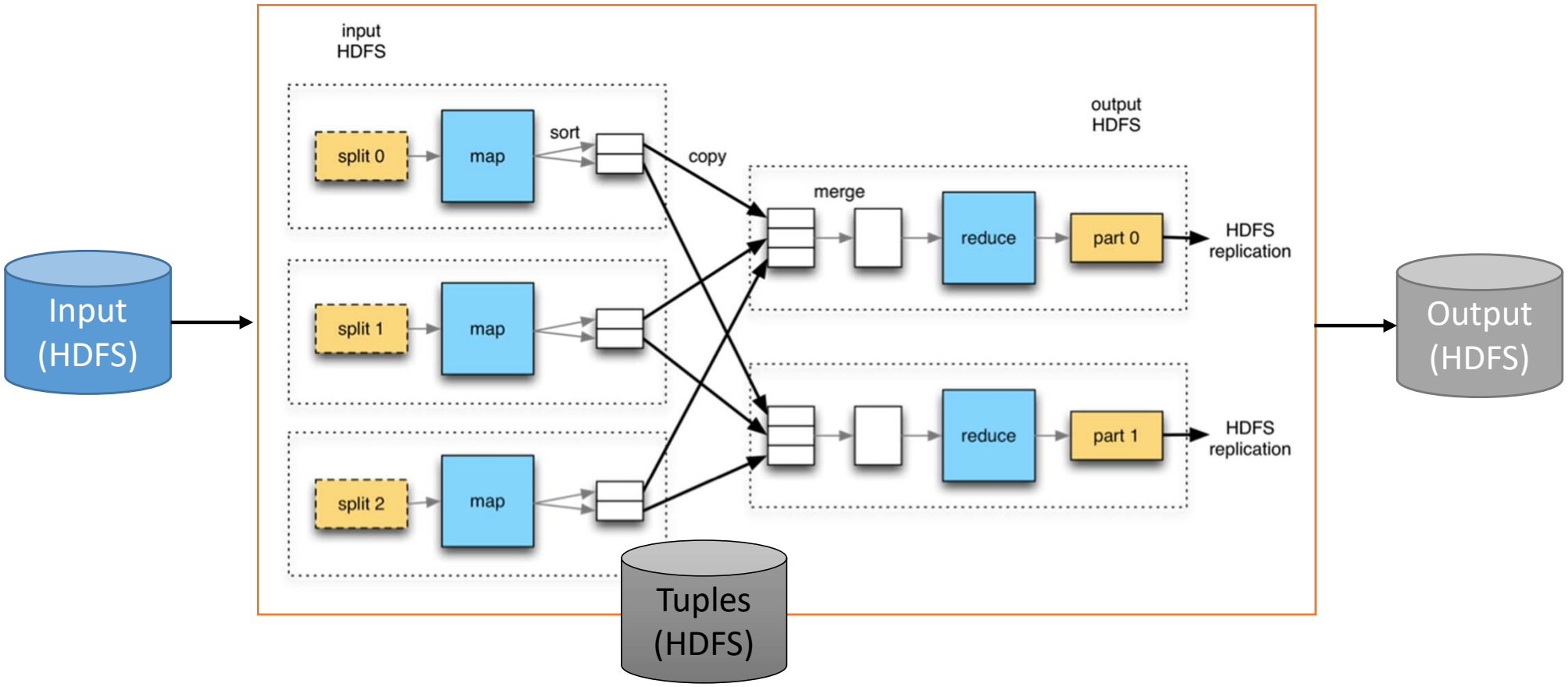
```
python word_count.py -r hadoop hdfs:///path/to/your/input_file.txt --output-dir hdfs:///path/to/output_directory/
```

Map-Reduce Processing



Hadoop Disk-oriented Processing

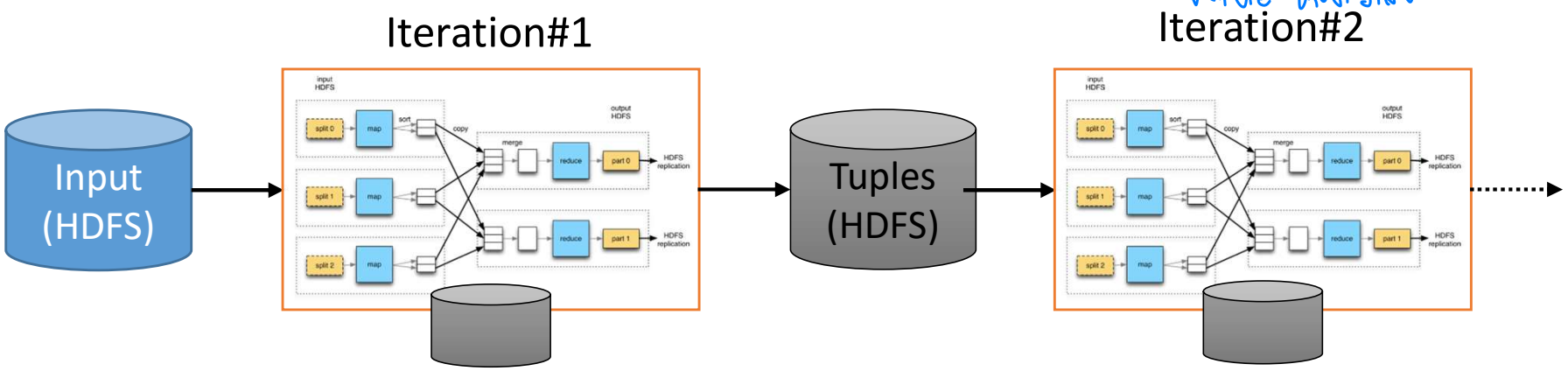
input, output, result เก็บบน HDFS ทั้งหมด



Hadoop Problems

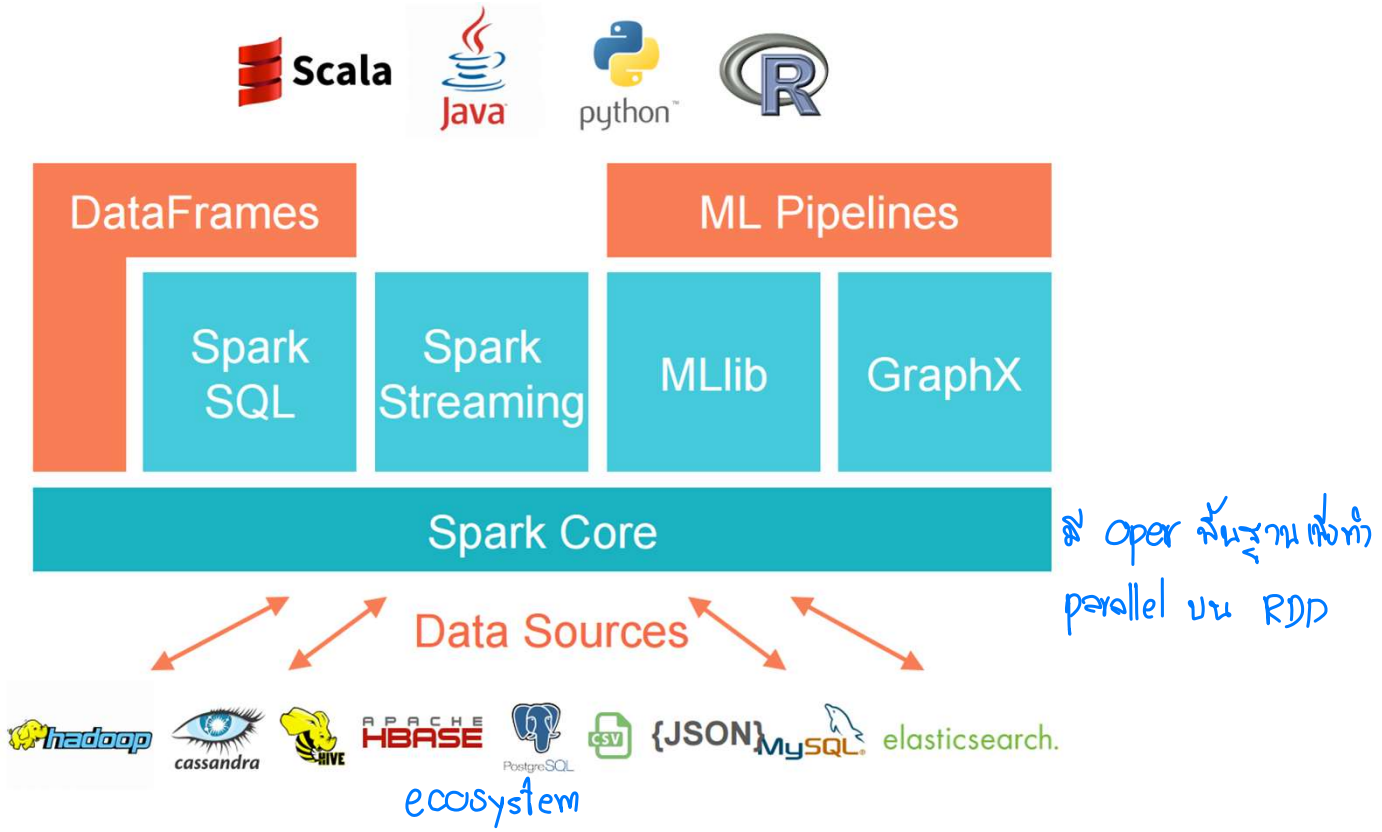
- HDFS is **slow** on write (due to replication) กระบวนการ copy ต้องทำ write จึงช้า
- Not suitable for iterative processing ทุกรอบต้องเขียนผลลัพธ์ลง HDD
- Many problems aren't easily described as map-reduce

ไม่เหมาะกับ map reduce ทุกงาน เช่น dot product กับคูณ matrix key คือ row number value คือตัวเลข



Apache Spark

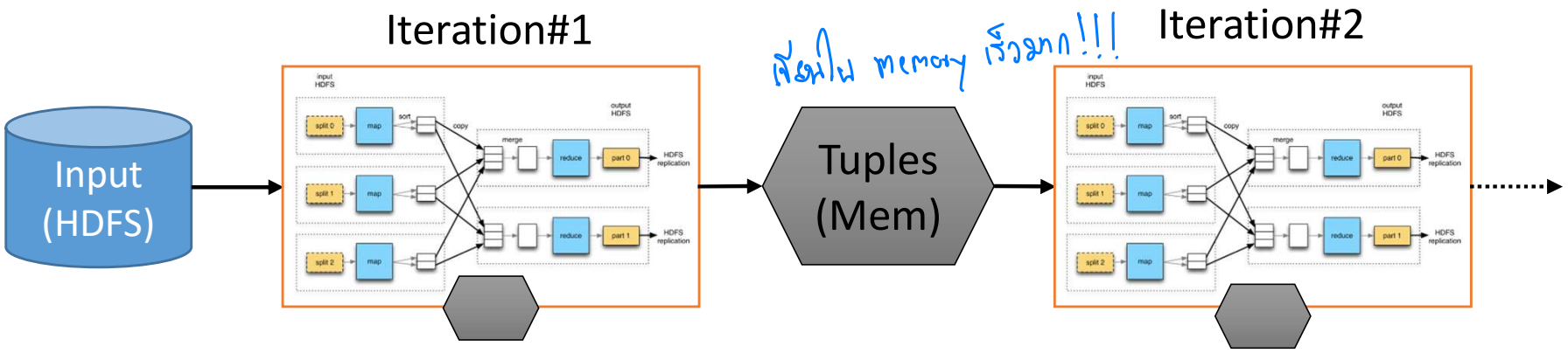
- In-memory data processing การใช้ memory ในการประมวลผลแทน HDD
- Resilient Distributed Data (RDD) เอา memory หลายเครื่องมองเป็นก้อนเดียวกัน
- Large set of operations (transformations & actions) ทำให้มี operation มากขึ้นนอกจาก mapReduce



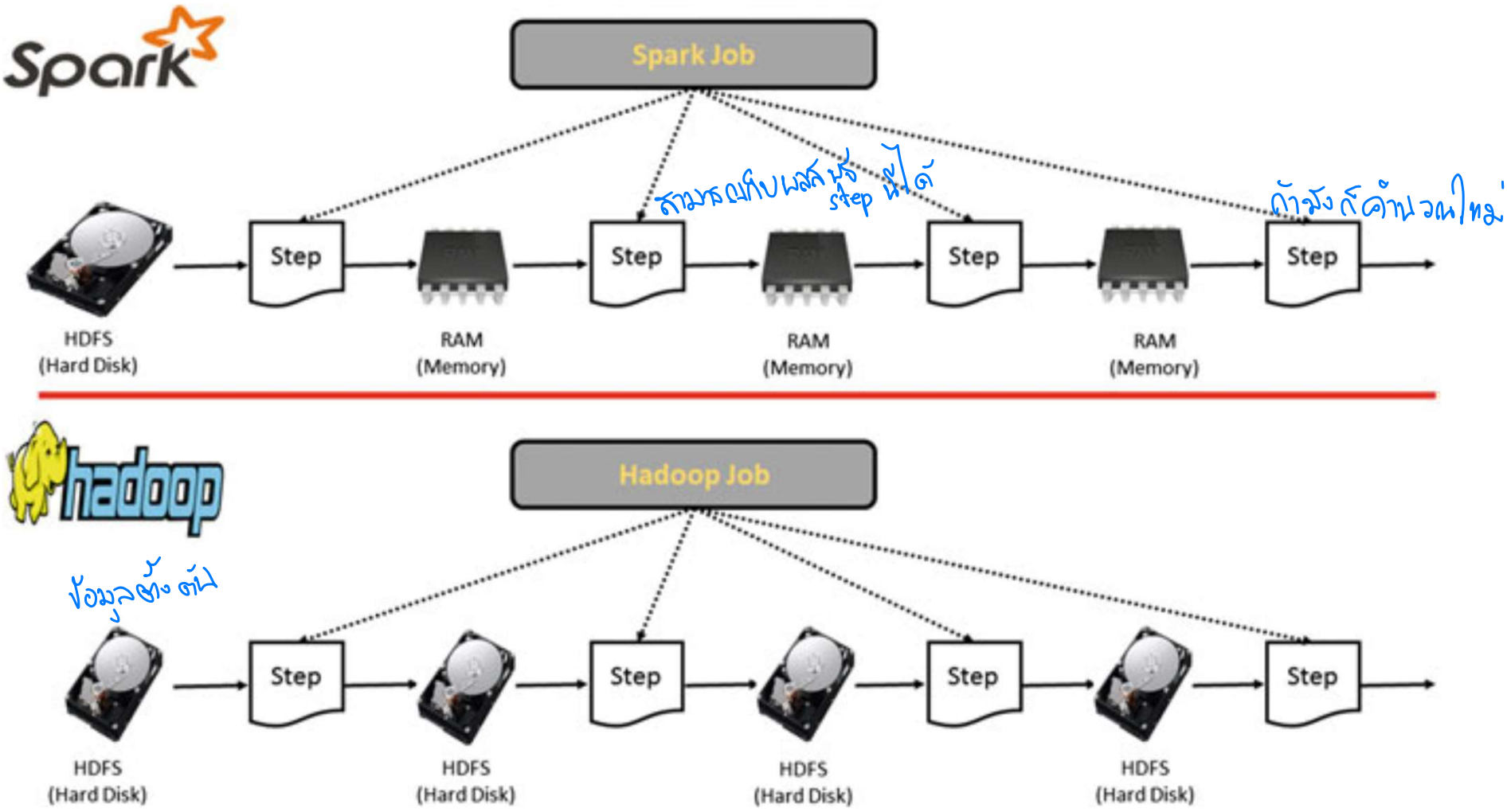
Source: M. Zaharia, “New Directions for Spark in 2015”, Spark Summit East 2015, 18 March 2015.

Spark Memory-oriented Processing

- Intermediate data are kept in memory. สามารถทำงาน iterative ได้
- Only read from disk for input and write to disk for output input, output ใช้จาก HDFS ได้



Spark vs Hadoop



Why Spark is Faster? --> Data Abstraction

ลักษณะการเก็บข้อมูล

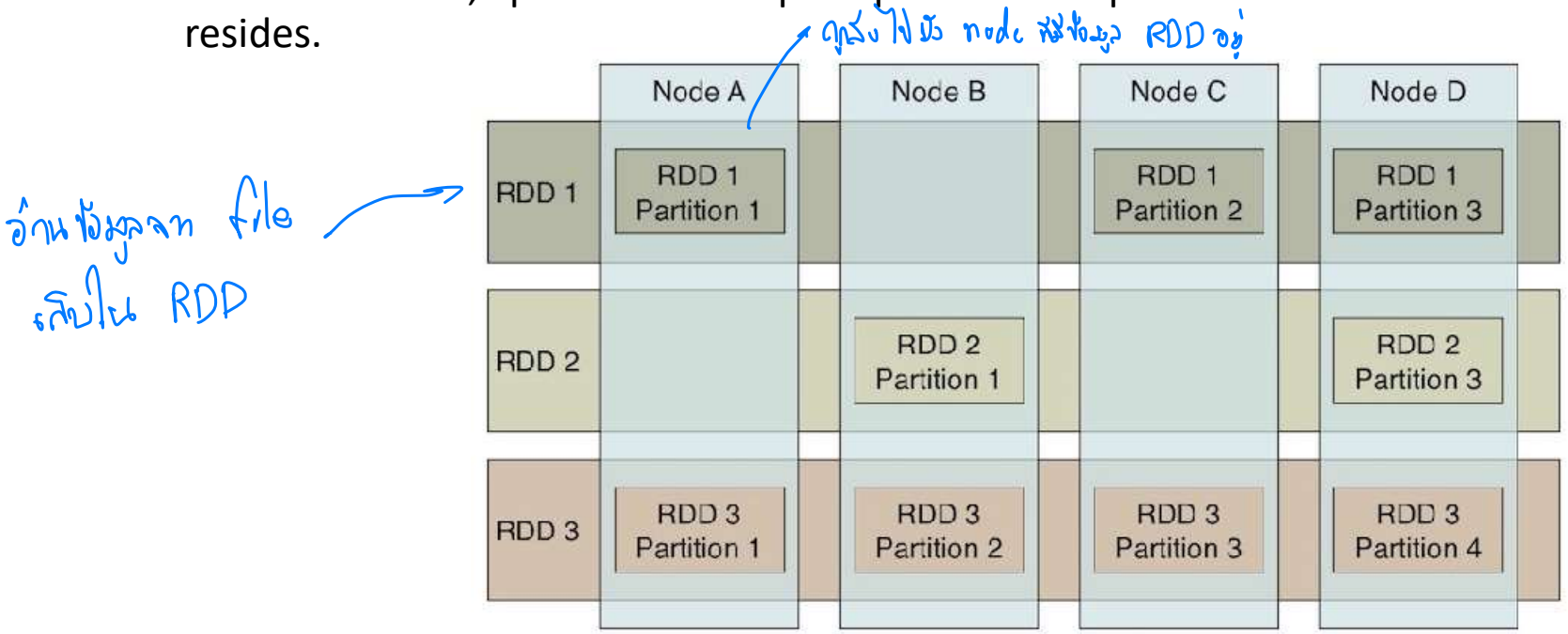
- Hadoop's basic data unit is a block in HDFS เก็บ block บน HDFS
  - Slowwww on write → must completely replicated before continue
  - Fault tolerance by prevention (block replication) ใช้หลักการป้องกัน (replication)
- Spark's basic data unit is RDD
  - Array of "read-only" data resided in memory ถ้าไฟล์ข้อมูลใน memory หาย
  - Fault tolerance by recovery (recompute RDDs) ใช้การกู้ข้อมูล (ต้นฉบับไฟล์เช่น HDFS)
- Why must RDD be read-only (immutable)? read-only
  - In a cluster, multiple nodes may operate on the data in parallel. If the data were mutable, nodes would need to synchronize to each other to ensure consistency, which can be a significant overhead.

memory มี cache เหมือนมี multiple memory มีโอกาส inconsistency

เหมือน HDFS ที่ต้องทำสำเนาให้ครบ

RDD: Resilient Distributed Data

- Immutable Distributed Collection of Objects
- RDD can be created by parallelizing an existing collection (data structure) or loading a dataset from a storage.
- RDD partitioning is based on the specified number of partitions, cluster configuration, or storage partitioning.
- Partitions are distributed across different nodes in the Spark cluster.
- The Spark driver sends tasks to executors. Each task processes data in one partition.
- Spark tries to place partitions on nodes in a way that respects data locality preferences. If data is read from HDFS, Spark will attempt to perform computations on the nodes where the data resides.



Creating RDDs

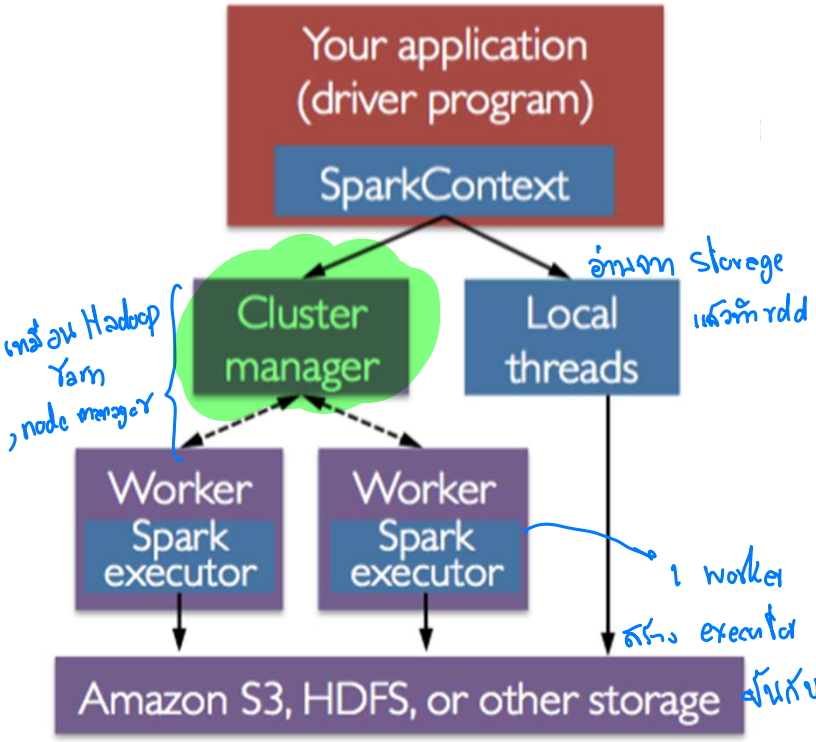
- `sc.parallelize(data)` – create an RDD from data

```
lines = sc.parallelize(List("a", "an", "the"))
```

- `sc.textFile(filename)` – create an RDD from
  - HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory
  - Can be wildcard: /data/file*

```
rdd = sc.textFile("/home/user01/data/file*")
rdd.count()
```

Spark components



- Driver program
  - The user program that creates SparkContext and communicates with it to submit jobs and receive the results.
- SparkContext process
  - Entry point to Spark functionality.
  - Coordinates execution, communicates with cluster manager for resouce allocation and task scheduling.
- Worker program
  - Executes tasks assigned by the driver program.
  - Run on cluster nodes or local threads (cluster mode vs local mode)
    - If you run a Spark program from a Jupyter notebook on your laptop, it will run in local mode. This means Spark driver, executor, and all tasks will run on the same machine as Jupyter notebook.
    - If you want to run the program in cluster mode, you need to configure Spark to connect to a remote Spark cluster and submit your program to the cluster using a cluster manager like YARN or Spark Master.

- Cluster Manager allocates and manage resources across the cluster
  - E.g. YARN, Mesos, Kubernetes, Spark Master (in standalone mode)

- Executor
  - A process that runs on a worker node, responsible for executing a task.
  - A worker program can host multiple executors.
  - Each executor can have multiple cores and run multiple tasks in parallel.

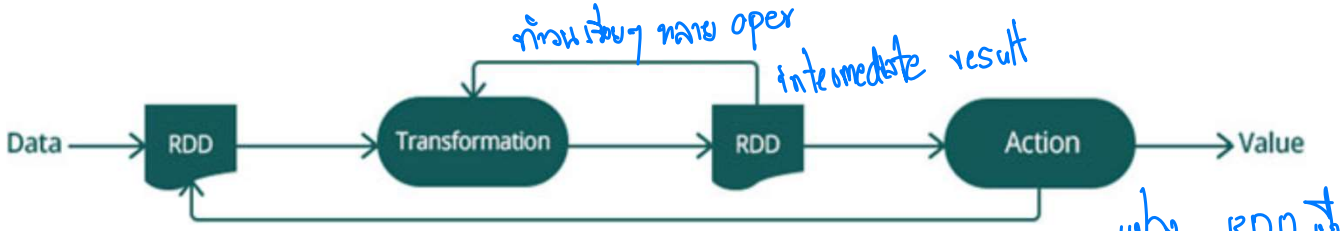
Why Spark is Faster? --> Execution Planning

- Hadoop is very straight-forward, ทำงานตรงไปตรงมา ทำตามลำดับ
  Map-then-Reduce, executed immediately
- Spark is lazy-execution, but constructs DAG (Directed Acyclic Graph) of RDDs กราฟสร้าง execute บน RDD
  (Sequence of tasks to be executed on RDDs)
  - Schedule DAG when needed

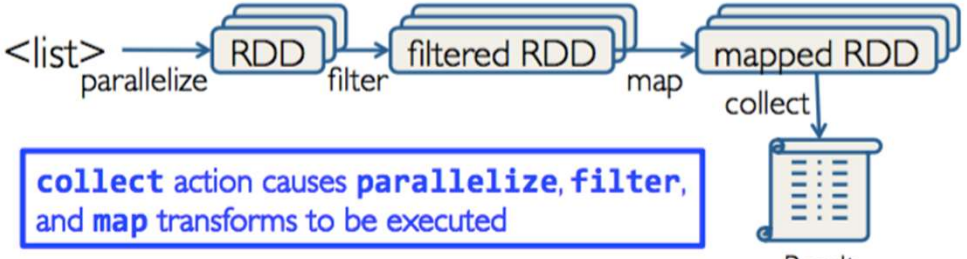
lazy - execution เวลา apply function ยังไม่ทำทันที ทำจริงๆ เมื่อต้องการผลลัพธ์

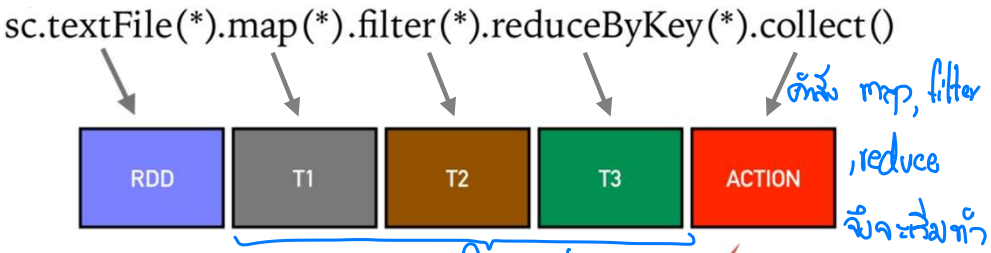
Typical Spark Program

- Create RDD from external dataset (e.g. file from HDFS) or existing collection of objects (List, Set, DataFrame)
- Transformation (Create a new RDD from existing RDDs)
- Action (Compute result from an RDD) ทำงาน RDD ก้อนนี้
- (Maybe) persist (cache) RDDs to disk or memory RDD เก็บบน disk หรือแปลงเป็น data ประเภทอื่น
- *Transformations* are **lazy** (not computed immediately) ผลลัพธ์ยังไม่ทำตอนนี้
- Transformed RDD is executed when an *action* runs on it



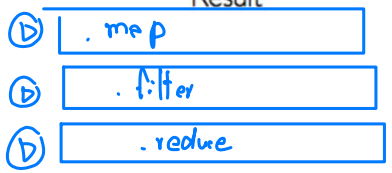
แปลง RDD เป็น data frame python เพื่อไป plot graph





ตัวสั่ง map, filter, reduce จึงจะเริ่มทำ

ใน colab



Run เร็วเพราะมันยังไม่ได้ทำ แค่จำเอาไว้ก่อน

Sample Spark Transformations

- map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
- filter(func): Return a new dataset formed by selecting those elements of the source on which func returns true
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.
- distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

https://spark.apache.org/docs/latest/programming-guide.html

Sample Spark Actions ต้อง execution transformation ทั้งหมด

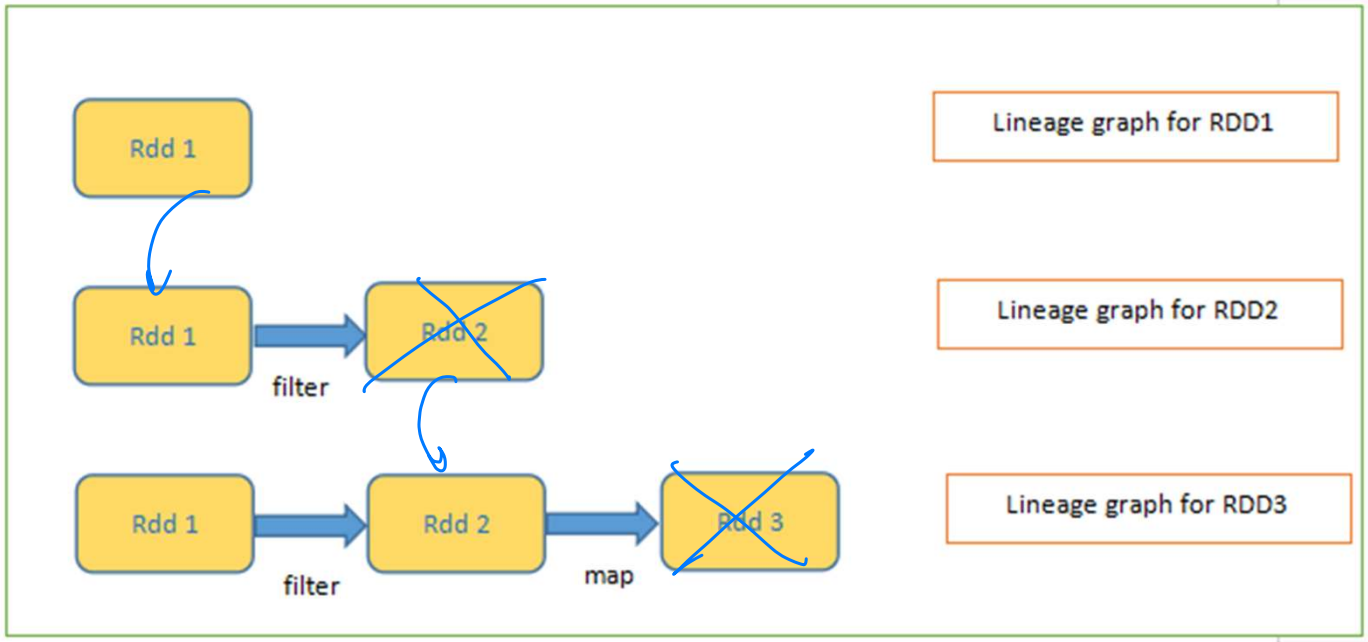
- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. รวบรวมข้อมูลทั้งหมด
- count(): Return the number of elements in the dataset.

Actions cause calculations to be performed;
transformations just set things up (lazy evaluation)

https://spark.apache.org/docs/latest/programming-guide.html

RDD Fault Tolerance

- RDD can be kept in memory or persisted to disks
- If a node fails, only the partitions on that node need to be recomputed, as Spark keeps track of the lineage of transformations of each RDD.



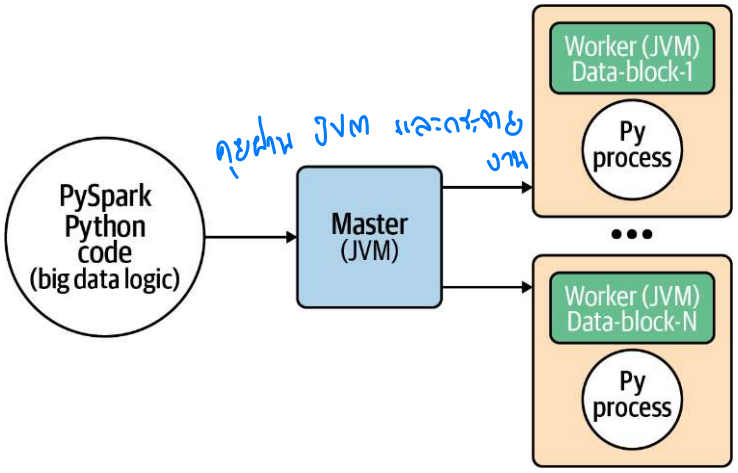
Common Communication Patterns in Spark Program

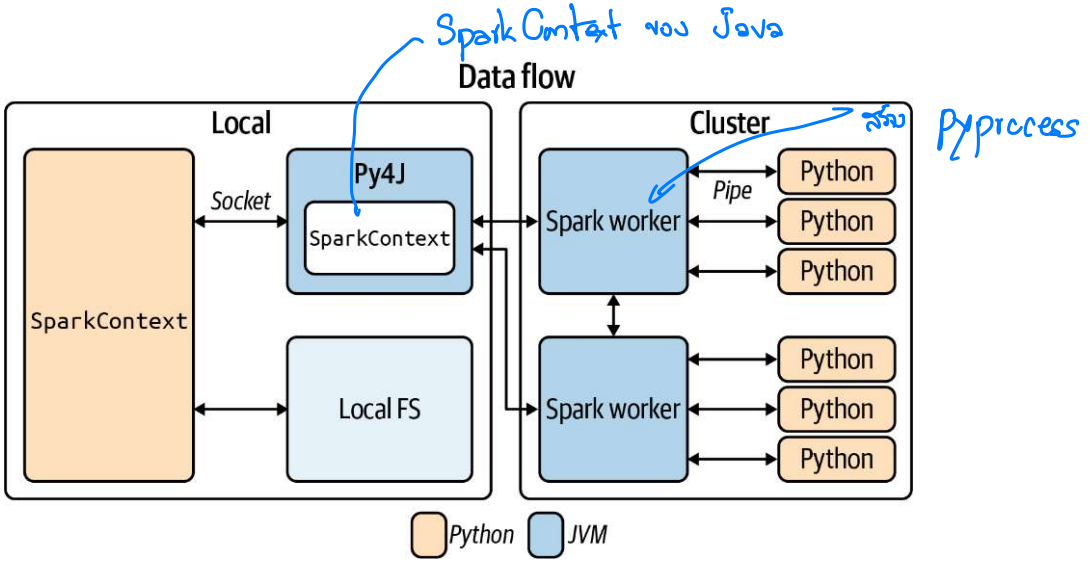
RDD operations can cause implicit communication. For example:

- Shuffling: Redistribution of data across partitions for operations like *reduceByKey*, *groupBy*, *join*, *sortBy*, and *repartition*, causing significant network and disk I/O.
- Broadcasting: Efficiently sending a large, read-only variable to all nodes, commonly used in join operations with disparate dataset sizes.
- All-to-All: Each node communicates with all others, often triggered by repartition.
- Map-Side: Operations like *map* and *filter* apply functions to RDD elements independently, with no cross-node communication.
- Reduce-Side: Aggregations like *reduce* and *fold* bring data together for processing, requiring data movement for combiners and reducers.
- Gather to Driver: The *collect* operation pulls data from all worker nodes back to the driver program.

PySpark Architecture

- PySpark is a Python API for Apache Spark, built on top of Spark's Java API.
- PySpark uses the Py4J library, which acts as a bridge between the Python driver program and the Spark Java core.
- RDD operations in user's Python program are sent to Spark workers, which create Python processes to execute the tasks.





https://spark.apache.org/docs/latest/api/python/index.html
https://www.oreilly.com/library/view/data-algorithms-with/9781492082378/ch01.html

PySpark Examples

USERS.CSV (MOVIELENS DATABASE)

```
id | age | gender | occupation | zip
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
4|24|M|technician|43537
5|33|F|other|15213
6|42|M|executive|98101
7|57|M|administrator|91344
8|36|M|administrator|05201
```

M = 670
F = 273

Spark df ≠ python df

With RDD and MapReduce

```
from pyspark import SparkContext
sc = SparkContext.getOrCreate()

print sc.textFile( "users.csv" ) \
    .map( lambda x: (x.split("|")[2], 1) ) \
    .reduceByKey(lambda x,y:x+y).collect()

sc.stop()
```

M, 1
M, 1
F, 1
M, 1

[(u'M', 670), (u'F', 273)]

With Spark DataFrame and built-in functions

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
sc = SparkContext.getOrCreate()
spark = SparkSession(sc)

spark.read.load( "users.csv", format="csv", sep="|" ) \
        .toDF( "id","age","gender","occupation","zip" ) \
        .groupby( "gender" ).count().show()

sc.stop()
```

Word Count with PySpark

```
from pyspark import SparkConf, SparkContext

# create a SparkConf object and set the application name
conf = SparkConf().setAppName("WordCount")

# create a SparkContext object
sc = SparkContext.getOrCreate(conf)

# load the input text file as an RDD
input_file = sc.textFile("input.txt")

# split the text into individual words
words = input_file.flatMap(lambda line: line.split())

# count the occurrence of each word
word_count = words.map(lambda word: (word,
1)).reduceByKey(lambda a, b: a + b)

# print the word count result
for word, count in word_count.collect():
    print("{}: {}".format(word, count))

# stop the SparkContext
sc.stop()
```

- Load the input text file as an RDD using the textFile()
- Split the text into individual words using flatMap()
- Create a new RDD of word-count pairs using map()
- Group the word-count pairs by word and sum the counts using reduceByKey()
- Collect word count result using collect() and print

Dot Product with PySpark

```
from pyspark import SparkConf, SparkContext
import findspark
import random
import datetime

# findspark sets environment variables for pyspark path
findspark.init()

# Create a SparkConf and SparkContext
conf = SparkConf().setAppName("DotProduct")
sc = SparkContext.getOrCreate(conf)

n = 10
# Define two vectors as RDDs
rdd1 = sc.parallelize([random.random() for i in range(n)])
rdd2 = sc.parallelize([random.random() for i in range(n)])

start = datetime.datetime.now()

# Compute dot product using zip, map, and reduce operations
dot_product = rdd1.zip(rdd2).map(lambda x:
x[0]*x[1]).reduce(lambda x, y: x+y)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")

# Print the dot product
print("The dot product is:", dot_product)

# Stop the SparkContext
sc.stop()
```

```
0:00:09.607832 h:mm:ss
The dot product is: 2.7423063387967446
```

- Two vectors are defined as RDDs using the parallelize().
- Dot product is computed by:
  - zip() pairs up corresponding elements of the two vectors
  - Then, map() multiplies the paired elements together
  - Finally, reduce() sums the resulting products.

- Try increasing n to 1 million and more
- Notice the execution time and overhead

Dot Product with PySpark

```
from pyspark import SparkConf, SparkContext
import findspark
import random
import datetime

findspark.init()

# Create a SparkConf and SparkContext
conf = SparkConf().setAppName("DotProduct")\
                  .set("spark.driver.memory", "16g") \
                  .set("spark.executor.memory", "16g")
sc = SparkContext.getOrCreate(conf)

n = 100000000
# Define two vectors as RDDs
rdd1 = sc.parallelize([random.random() for i in range(n)])
rdd2 = sc.parallelize([random.random() for i in range(n)])

start = datetime.datetime.now()

# Compute the dot product using zip, map, and reduce
operations
dot_product = rdd1.zip(rdd2).map(lambda x:
x[0]*x[1]).reduce(lambda x, y: x+y)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")

# Print the dot product
print("The dot product is:", dot_product)

# Stop the SparkContext
sc.stop()
```

```
0:00:26.120495 h:mm:ss
The dot product is: 24999888.363398004
```

- If data is too large, an error like *"OutOfMemoryError: Java heap space"* may occur.
- Default size of driver memory and executor memory is 1 GB, try increasing them.
- pyspark does not support restarting the Spark context.
  So, to change the settings, you will need to restart the Jupyter kernel.

Matrix Multiplication with PySpark

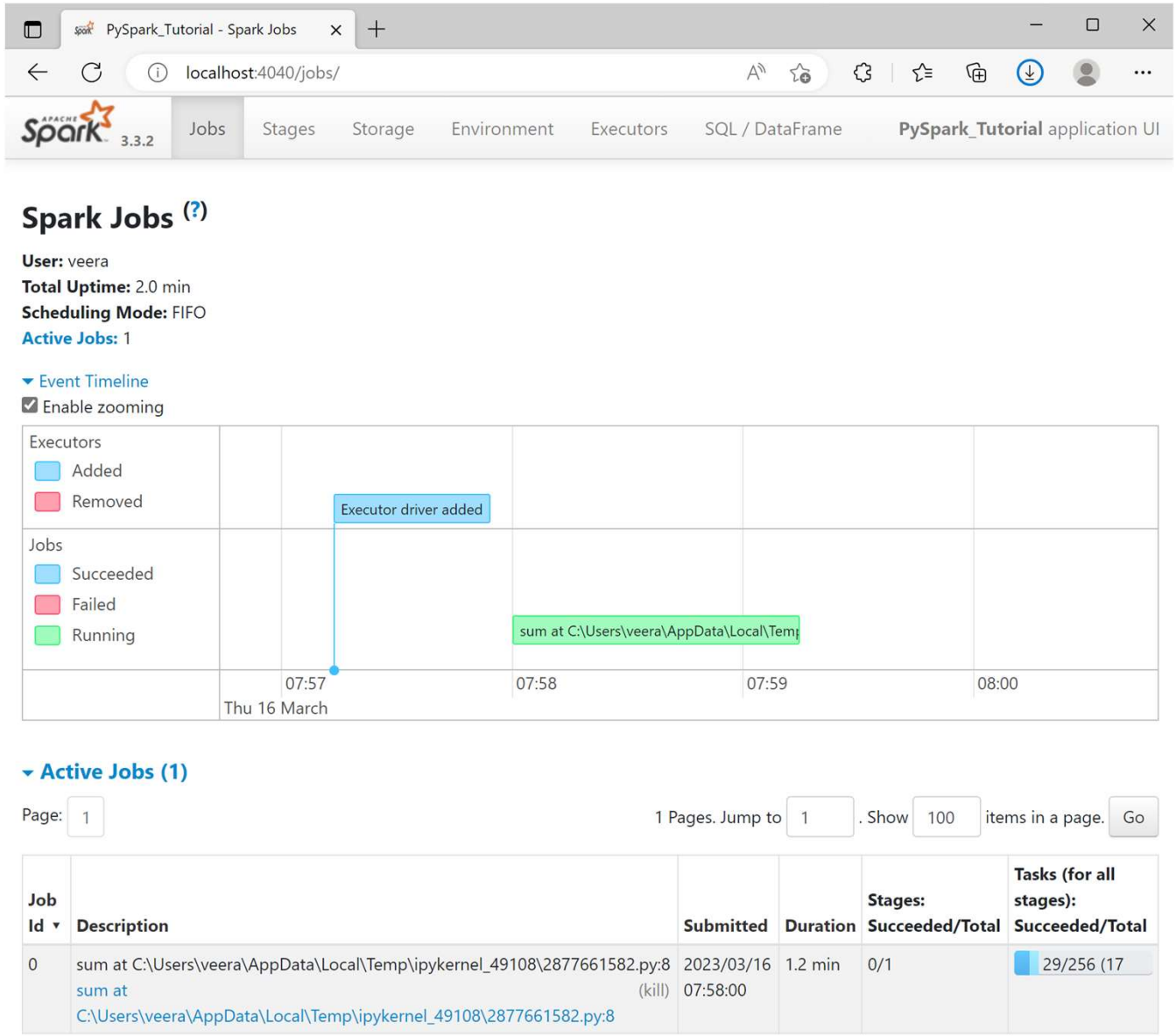
```
from pyspark import SparkConf, SparkContext
import findspark
import numpy as np
import datetime

findspark.init()

# Create a SparkConf and SparkContext
conf = SparkConf().setAppName("MatrixMultiplication")
sc = SparkContext.getOrCreate(conf)

n = 100
# Define the matrices as numpy arrays
matrix1 = np.random.rand(n, n)
matrix2 = np.random.rand(n, n)

# Broadcast the second matrix to all nodes
broadcast_matrix = sc.broadcast(matrix2)

start = datetime.datetime.now()

# Convert the first matrix to an RDD of rows
rows_rdd = sc.parallelize(matrix1)

# Perform the matrix multiplication using map and reduce operations
result = rows_rdd.map(lambda row: np.dot(row,
broadcast_matrix.value)).reduce(lambda x, y: x + y)

end = datetime.datetime.now()
print(end - start, "h:mm:ss")

# Print the result
print("The result of matrix multiplication is:\n", result)

# Stop the SparkContext
sc.stop()
```

```
0:00:09.086623 h:mm:ss
The result of matrix multiplication is:
 [2512.5996501  2674.34912392 2493.76738227 2669.46479685 2706.17381393
```

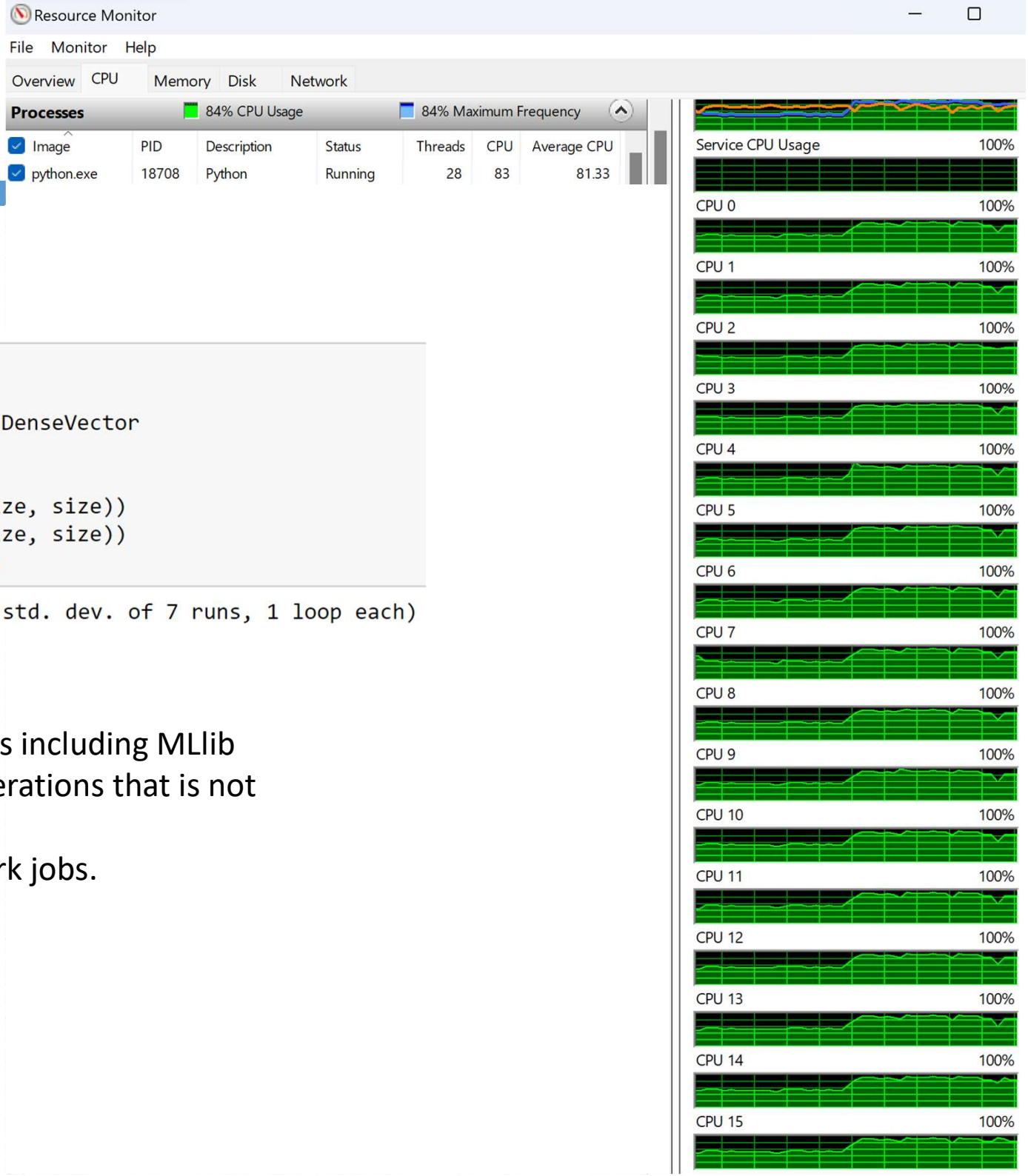
- First, define two 2-dimensional matrices matrix1 and matrix2 as numpy arrays.
- Broadcast matrix2 to all nodes using sc.broadcast().
- Convert matrix1 to an RDD of rows using sc.parallelize().
- Perform matrix multiplication using map and reduce operations.
- In the map operation, use numpy.dot() to compute the dot product between each row of the first matrix and the entire second matrix (which has been broadcasted).
- In the reduce operation, add up all the results to get the final result of matrix multiplication.

Spark UI

http://localhost:4040/jobs/



Matrix Multiplication with PySpark




```
!pip install -q pyspark
import numpy as np
from pyspark.mllib.linalg import DenseVector

size = 10000
A = DenseVector(np.random.rand(size, size))
B = DenseVector(np.random.rand(size, size))
%timeit C = DenseVector.dot(A, B)
```

```
12.6 s ± 361 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- PySpark has many libraries including MLlib
- MLlib supports vector operations that is not implemented with RDD
- So, it does not create Spark jobs.