УНИВЕРСИТЕТ ИТМО

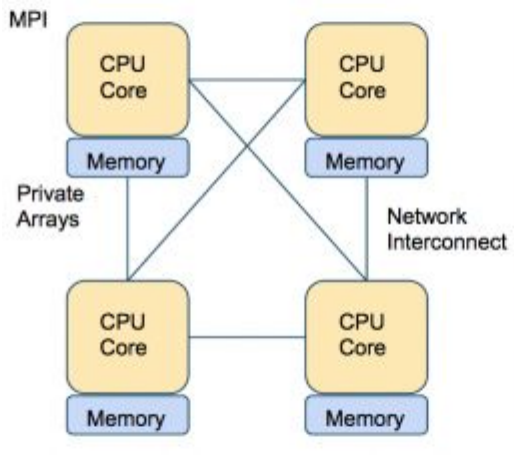
Parallel algorithms for the analysis and synthesis of data

L2

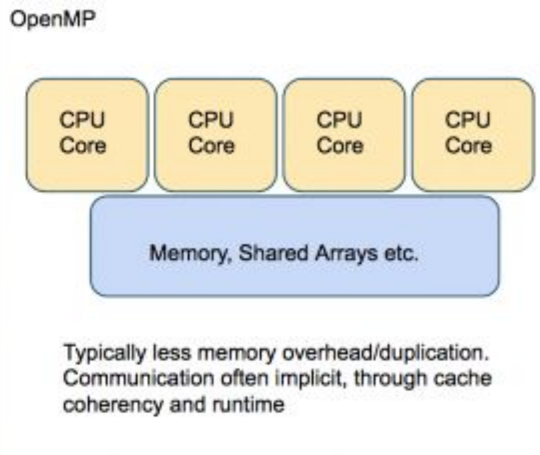
Sokhin Timur, PhD student
t.me/Qwinpin, qwinpin@gmail.com

Intro

Distributed memory



Shared memory

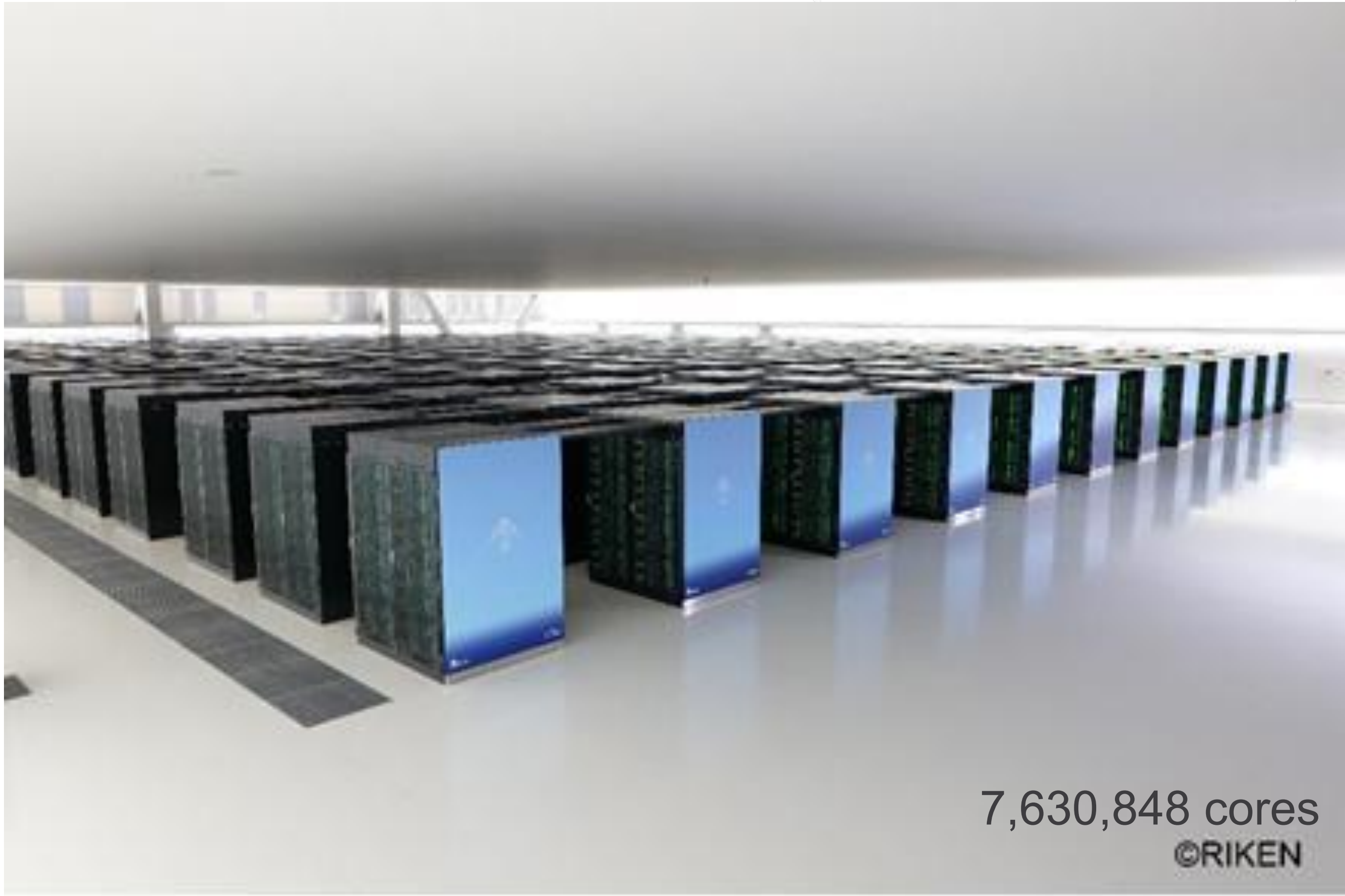


MPI (Message Passing Interface) is a standardized and portable message-passing standard designed to function on parallel computing architectures.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming.

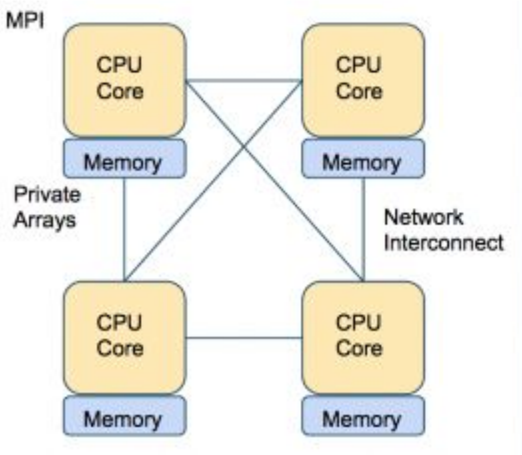
Intro (2)

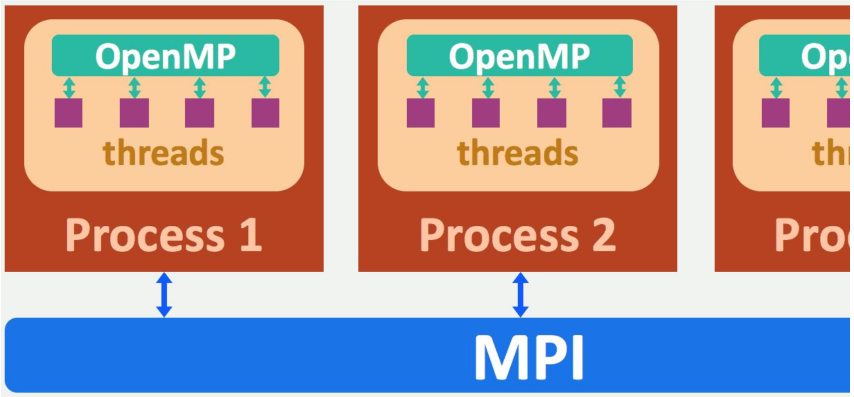
How they communicate?



Intro (2)

MPI - standard for message passing that is used for clusters, supercomputers, etc.
It is not a replacement for openmp, but a solution to the problem of applying parallel processing without shared memory.



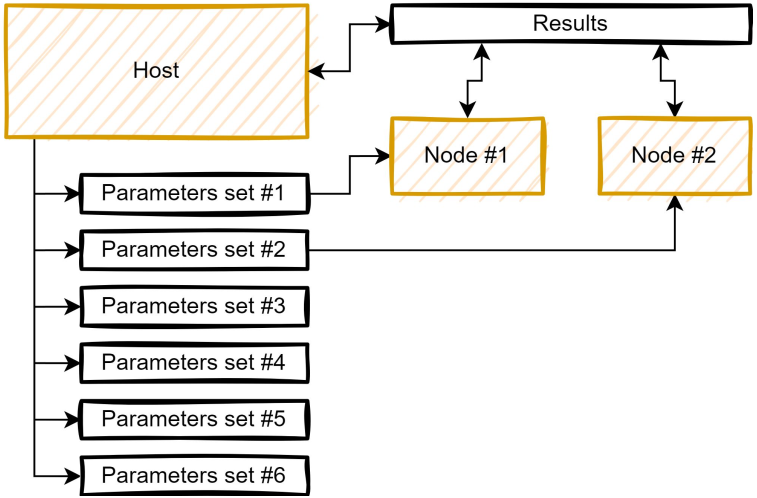


https://www.mpi-forum.org/

Tasks

Tasks solved by MPI are not only limited to paralleling by data, but also paralleling by tasks:

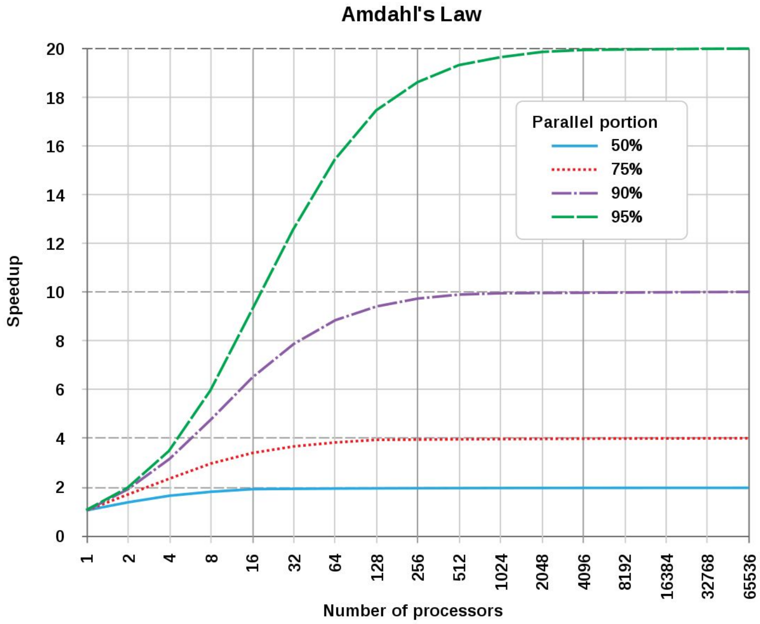
- train multiple algorithms with different parameters
- apply different algorithms to the same data
- apply different algorithms to different data samples



Limitations

Amdahl's law. The acceleration achieved by paralleling the algorithm has its limits, depending on the fraction of the program that can be executed in parallel, this efficiency can quickly come to zero.

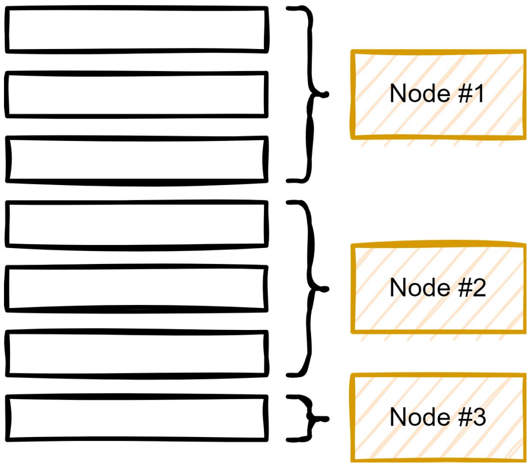
Additional factor we should take into account: communication latency of network, data processing.

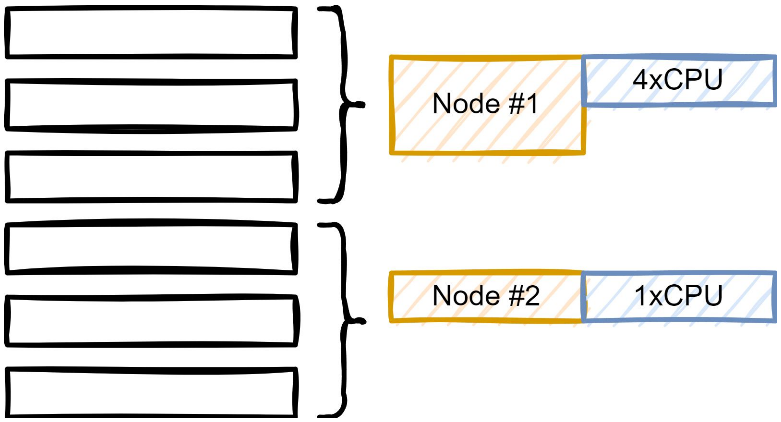


Possible bottlenecks

It is a trivial paralleling task to distribute a large amount of data among separate nodes. However, even this task can be fraught with a number of problems:

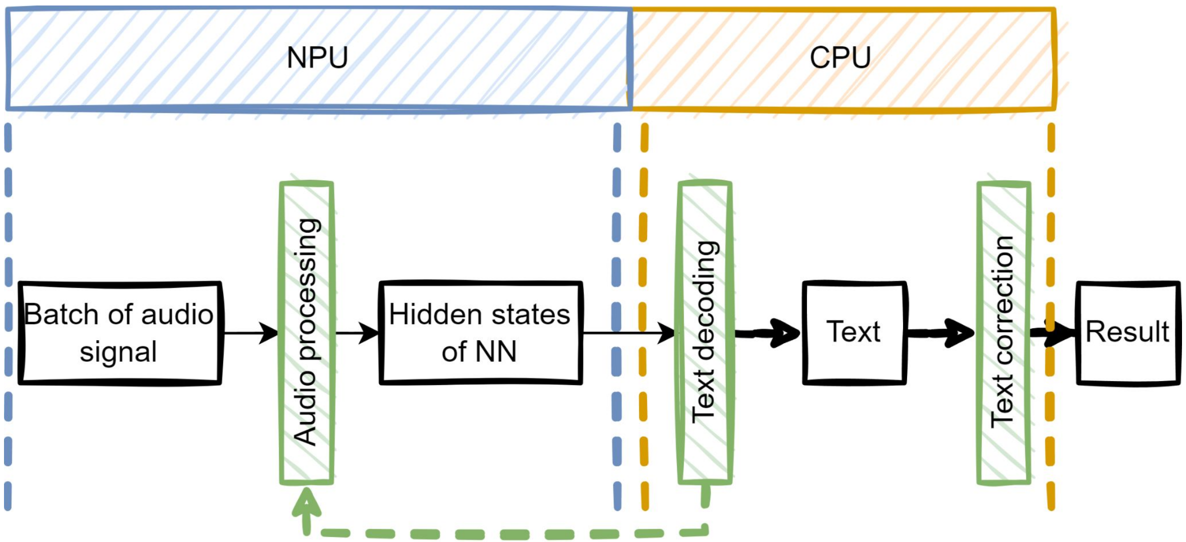
- splitting residuals
- difference in computing power of machines
- possible early termination of execution





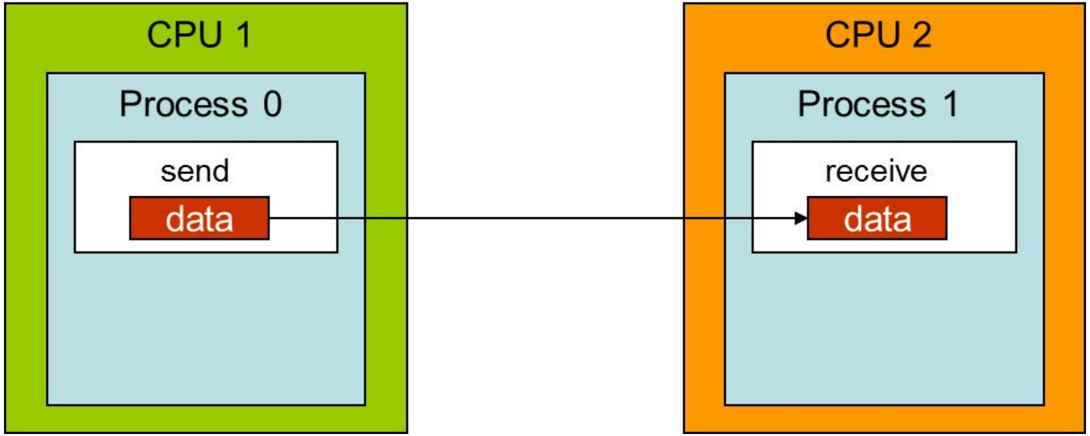
Non-trivial tasks

Sometimes we need to separate over part of task not the data



Basics

MPI main concepts:
- communicator
- rank
- send/receive operations



MPI Hello World

УНИВЕРСИТЕТ ИТМО

C example

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);  // Initialization of communicator

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);


    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);


    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);


    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Python example

```python
from mpi4py import MPI  # Initialization of communicator

comm = MPI.COMM_WORLD  # get communicator object
rank = comm.Get_rank()
```

Python pure example

```python
import mpi4py

mpi4py.rc.initialize = False  # do not initialize MPI automatically
mpi4py.rc.finalize = False   # do not finalize MPI automatically

from mpi4py import MPI # import the 'MPI' module

MPI.Init()  # Initialization of communicator
comm = MPI.COMM_WORLD  # get communicator object
rank = comm.Get_rank()
world_size = comm.Get_size()
processor_name = MPI.Get_processor_name()


print("Hello world from processor {}, rank {} out of {} processors\n".format(
processor_name, rank, world_size))

MPI.Finalize()  # manual finalization of the MPI environment
```

https://mpi4py.readthedocs.io/en/stable/intro.html

MPI Hello World (2)

УНИВЕРСИТЕТ ИТМО

C example

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // get some of parameters

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  // get some of parameters

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);  // get some of parameters

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Python example

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # get some of parameters
```

Python pure example

```python
import mpi4py

mpi4py.rc.initialize = False  # do not initialize MPI automatically
mpi4py.rc.finalize = False   # do not finalize MPI automatically

from mpi4py import MPI # import the 'MPI' module

MPI.Init()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # get some of parameters
world_size = comm.Get_size()  # get some of parameters
processor_name = MPI.Get_processor_name()  # get some of parameters


print("Hello world from processor {}, rank {} out of {} processors\n",
        processor_name, rank, world_size)

MPI.Finalize()  # manual finalization of the MPI environment
```

MPI Hello World (3)

Python example

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # get some of parameters
```

Python pure example

```
import mpi4py

mpi4py.rc.initialize = False  # do not initialize MPI automatically
mpi4py.rc.finalize = False   # do not finalize MPI automatically

from mpi4py import MPI # import the 'MPI' module

MPI.Init()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # get some of parameters
world_size = comm.Get_size()  # get some of parameters
processor_name = MPI.Get_processor_name()  # get some of parameters


print("Hello world from processor {}, rank {} out of {} processors\n",
         processor_name, rank, world_size)

MPI.Finalize()  # manual finalization of the MPI environment
```

MPI Send/Receive

УНИВЕРСИТЕТ ИТМО

Simple communication

```
from mpi4py import MPI

# Get my rank
rank = MPI.COMM_WORLD.Get_rank()

if rank == 0:
    message = "Hello, world!"
    MPI.COMM_WORLD.send(message, dest=1, tag=0)   # send message from current rank == 0 to destination rank == 1


if rank == 1:
    message = MPI.COMM_WORLD.recv(source=0, tag=0)
    print(message)  # receive message from rank == 0 and print
```

Deadlock communication

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
n_ranks = MPI.COMM_WORLD.Get_size()

message_to_send = list(range(15000))

MPI.COMM_WORLD.send(message_to_send, dest=1 - rank, tag=0)  # send message from current rank to rank 0 or 1. The program is stuck here


recieved_message = MPI.COMM_WORLD.recv(source=1 - rank, tag=0)



print(recieved_message)
```

MPI Send/Receive (2)

УНИВЕРСИТЕТ ИТМО

Deadlock communication

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
n_ranks = MPI.COMM_WORLD.Get_size()

message_to_send = list(range(15000))

MPI.COMM_WORLD.send(message_to_send, dest=1 - rank, tag=0)  # send message from current rank to rank 0 or 1. The program is stuck here


recieved_message = MPI.COMM_WORLD.recv(source=1 - rank, tag=0)



print(recieved_message)
```

MPI isend/ireceive

Simple communication

```
from mpi4py import MPI

# Get my rank
rank = MPI.COMM_WORLD.Get_rank()

if rank == 0:
    message = "Hello, world!"
    req = MPI.COMM_WORLD.isend(message, dest=1, tag=0)  # non-blocking communication, if here we have any code, it will be executed


if rank == 1:
    req = MPI.COMM_WORLD.irecv(source=0, tag=0)
    message = req.wait()
    print(message)
```

Non-Deadlock communication

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
n_ranks = MPI.COMM_WORLD.Get_size()

message_to_send = list(range(15000))

req = MPI.COMM_WORLD.isend(send_message, dest=1 - rank, tag=0)  # non-blocking communication, both 0 and 1 nodes will proceed to receiving operation


recieved_message = MPI.COMM_WORLD.recv(source=1 - rank, tag=0)  # here we have request object, not the message itself
# recieved_message = req.wait()  # wait for the message

print(recieved_message)
```

MPI isend/ireceive (2)

Non-Deadlock communication

```
from mpi4py import MPI

rank = MPI.COMM_WORLD.Get_rank()
n_ranks = MPI.COMM_WORLD.Get_size()

message_to_send = list(range(15000))

req = MPI.COMM_WORLD.isend(send_message, dest=1 - rank, tag=0)  # non-blocking communication, both 0 and 1 nodes will proceed to receiving operation


recieved_message = MPI.COMM_WORLD.recv(source=1 - rank, tag=0)  # here we have request object, not the message itself
# recieved_message = req.wait()  # wait for the message

print(recieved_message)
```

MPI isend/ireceive (2)

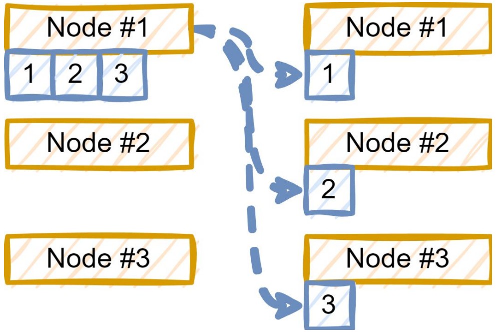
Service notifications

```
from mpi4py import MPI

# Get my rank
rank = MPI.COMM_WORLD.Get_rank()

common_func_call()

if rank == 0:
    master_node_func_call()  # take some time
    req = MPI.COMM_WORLD.irecv(source=0, tag=0)
    message = req.wait()  # wait for the message

    worker_node_func_call()  # continue with some task

if rank == 1:
    req = MPI.COMM_WORLD.isend(message, dest=1, tag=0)  # send
message and move forward

    worker_node_func_call()
```

MPI scatter/gather

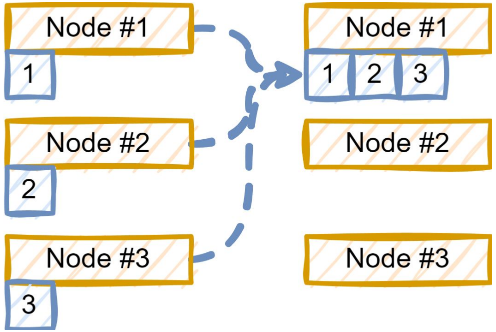
Scatter

```
from mpi4py import MPI

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

if rank == 0:
   data = [(x+1)**x for x in range(size)]
else:
   data = None

# note that rank == 0 will receive data as well and we lose original data list
for rank == 0
data = MPI.COMM_WORLD.scatter(data, root=0)
print ('rank {} has data: {}'.format(rank, data))
```



Gather

```
from mpi4py import MPI

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

send_message = "Hello World, I'm rank {:d}".format(rank)
receive_message = comm.gather(send_message, root=0)

if rank == 0:
    for i in range(size):
        print(receive_message[i])
```

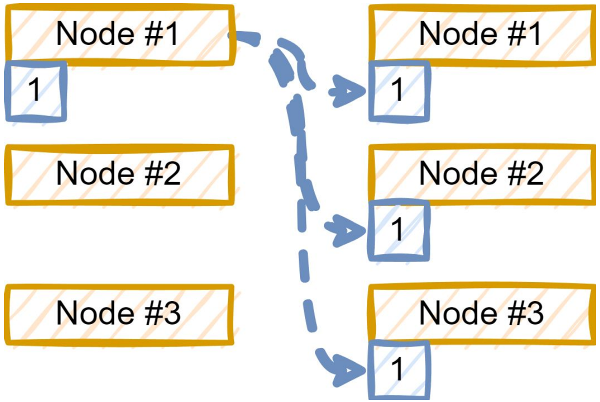


MPI bcast

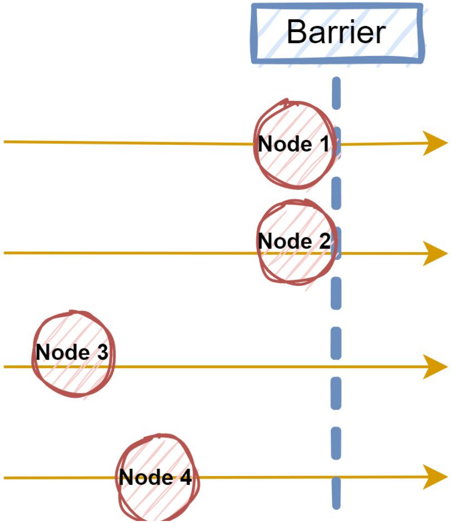
УНИВЕРСИТЕТ ИТМО

Bcast

```
from mpi4py import MPI

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

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None

data = comm.bcast(data, root=0)
```



Barrier



Pytorch MPI

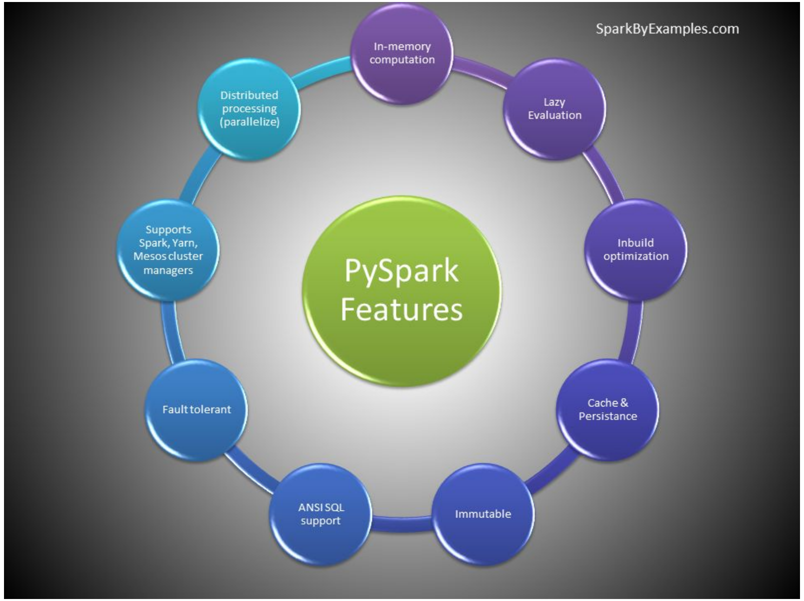
Simple PyTorch MPI program

```
import os
import socket
import torch
import torch.distributed as dist

from torch.multiprocessing import Process

def run(rank, size, hostname):
    print(f"I am {rank} of {size} in {hostname}")
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
        print('Rank ', rank, ' has data ', tensor[0])

def init_processes(rank, size, hostname, fn, backend='tcp'):
    """ Initialize the distributed environment. """
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size, hostname)

if __name__ == "__main__":
    world_size = int(os.environ['OMPI_COMM_WORLD_SIZE'])
    world_rank = int(os.environ['OMPI_COMM_WORLD_RANK'])
    hostname = socket.gethostname()
    init_processes(world_rank, world_size, hostname, run, backend='mpi')
```

PySpark

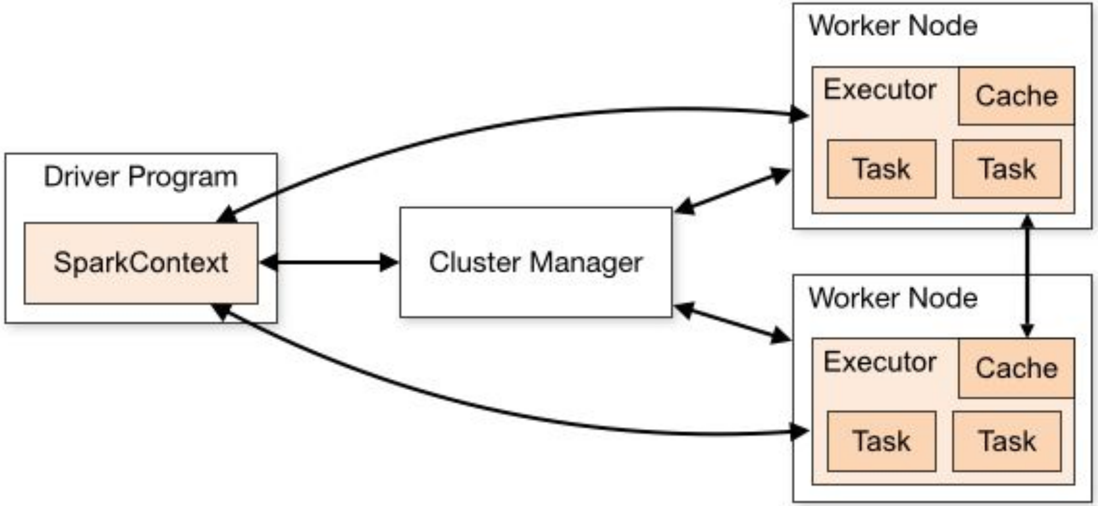
PySpark is a Python API for **Apache Spark**.
Apache Spark is an analytical processing engine for large scale powerful distributed data processing and machine learning applications.

- In-memory computation
- Distributed processing
- Fault-tolerant
- Immutable
- Lazy evaluation
- Cache & persistence



Architecture

Apache Spark works in a master-slave architecture where the master is called “Driver” and slaves are called “Workers”. Cluster manager responsible for resources managment.



Entry point

УНИВЕРСИТЕТ ИТМО

Simple

```
# Import SparkSession
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .master("local[1]") \ # create local driver with 1 core active
    .appName("YourNameOfApp") \
    .getOrCreate()
```

Multiple configs

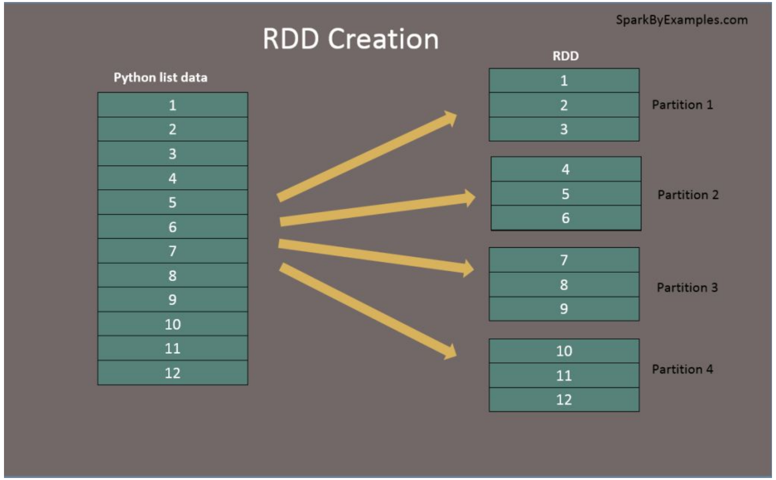
```
# Import SparkSession
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .master("local[4]") \ # create local driver with 1 core active
    .appName("YourNameOfApp") \
    .config("spark.driver.maxResultSize", "1g") \
    .config("spark.driver.memory", "4g") \
    .config("spark.executor.memory", "2g") \
    .config("spark.broadcast.blockSize", "12m") \
    .config("spark.files.maxPartitionBytes", "134217728") \
    .config("spark.dynamicAllocation.enabled", “true”) \
    .getOrCreate()
```

https://spark.apache.org/docs/latest/configuration.html

ITsMOre than a UNIVERSITY

RDD

```
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data, 10)
rdd.getNumPartitions()
repartitioned_rdd = rdd.repartition(4)
```



DataFrame

Data loading

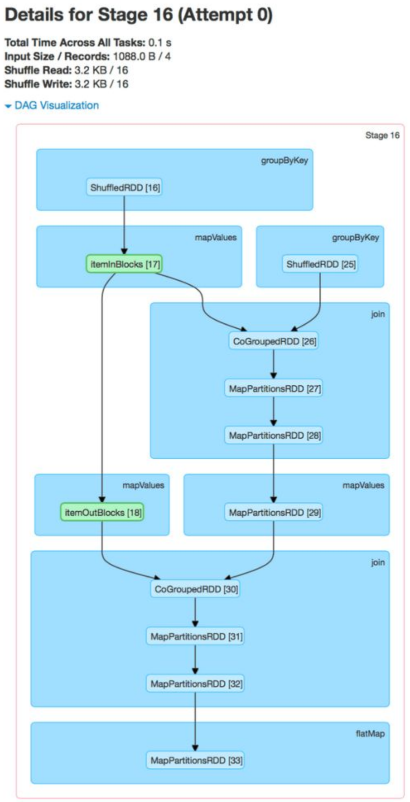
```
data = [('James','','Smith','1991-04-01','M',3000),
  ('Michael','Rose','','2000-05-19','M',4000),
  ('Robert','','Williams','1978-09-05','M',4000),
  ('Maria','Anne','Jones','1967-12-01','F',4000),
  ('Jen','Mary','Brown','1980-02-17','F',-1)
]

columns =
["firstname","middlename","lastname","dob","gender","salary"]
# sometimes
df = spark.createDataFrame(data=data, schema=columns)

df.show()
df.printSchema()

df2 = spark.read.csv("path/to/file.csv")  # in this case schema will be created from file columns
```

Manual schema definition

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType  # we need types definitions

schema = schema = StructType([
   StructField("name", StringType(), True),
   StructField("age", IntegerType(), False)])  # create schema of dataframe, we set name, type and possibility to be None
```

DAG

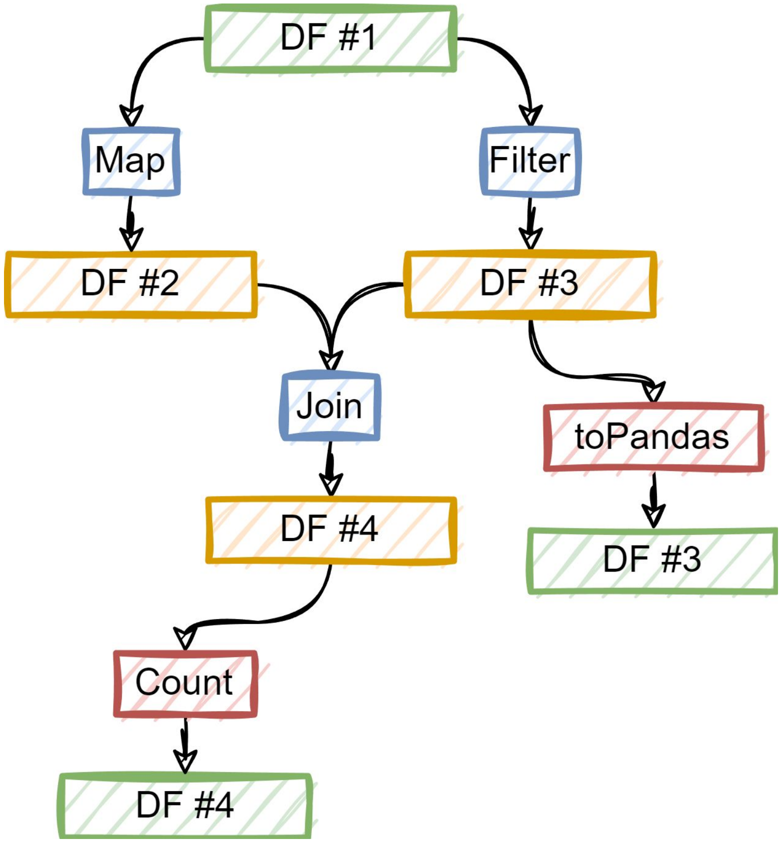
PySpark work with DAG of different actions



DataFrame operations

PySpark has two main kinds of operations:
- transformation

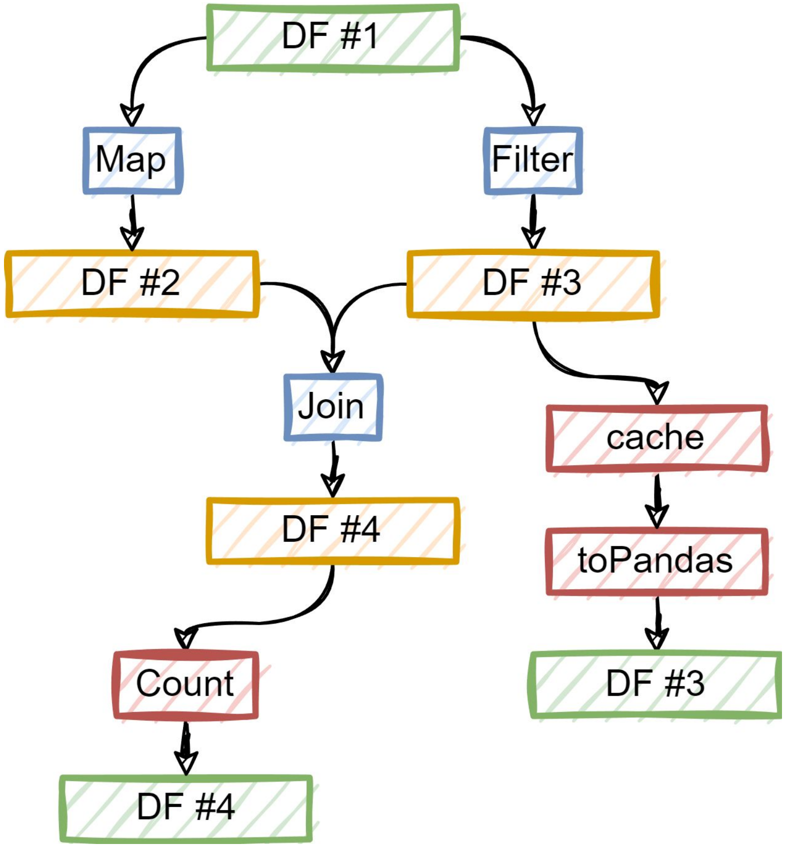
- action



DataFrame operations (2)

Transformations - lazy operations. They are not involve any calculation, but add operations for Spark DAG

Action - return the values of the whole calculation graph triggering transformations to execute..



Select

Select columns from dataframe (creates new DataFrame)

```
from pyspark.sql.functions import col

df.select(“col1”, “col2”).show()
df.select(col(“col1”), col(“col2”)).show()
```

Collect

Collect is action operation - all previous transformations of DataFrame will be calculated.
This operation send whole DataFrame to driver node and should be used on a small subset of data (after filter/where operations). Result of command is Array of RowType.

```
df_material = dataframe.collect()

df_material_col = dataframe.select("col1").collect()
```

toPandas

Collect is action operation - all previous transformations of DataFrame will be calculated.
This operation send whole DataFrame to driver node and should be used on a small subset of data (after filter/where operations).

toPandas() do almost the same, but returns pandas dataframe, which is more useful usually.

```
df_material = dataframe.collect()

df_material_col = dataframe.select("col1").collect()

pandas_df = dataframe.toPandas()
pandas_df_sample =
dataframe.sample(withReplacement=False, \
   0.5).toPandas()
```

UDF

PySpark have sql and dataframe level of user defined function.
UDF allows to enclose some python function into an transformation operation.

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, IntegerType

udf_function1 = udf(lambda x: x + 1, IntegerType())

dataframe = dataframe.withColumn(“number_of_elements_p”, \
   udf_function1(“number_of_elements”))

def int_to_string(val):
   return str(val)
udf_function2 = udf(lambda x: int_to_string(x), StringType())
dataframe = dataframe.withColumn(“str_number_of_elements_p”, \
   udf_function2(“number_of_elements_p”))

dataframe.select(col(“number_of_elements”), \
   udf_function2(col(“str_number_of_elements_p”)).alias(“res”))

@udf(returnType=StringType())
def some_fnc(val):
   …
```

Filter

Filter - do filtering operation (alias - where).

```
dataframe_small  = dataframe.filter(dataframe.col_name == “OFF”)

dataframe.filter((dataframe.col_name == “OFF”) &
(dataframe.col_name2 != “G”))

lst = [0, 4, 5]
dataframe.filter(dataframe.col_name3.isin(lst))
dataframe.filter(~dataframe.col_name3.isin(lst))

dataframe.filter(dataframe.col_name.like(“%FF”))
```

Join

Join is important operation for filtering over other dataframe or to merge to frames with additional information.

Types of join:
- inner
- outer
- left
- right
- cross

and many others.

dataframe.join(dataframe2, dataframe.col1 == dataframe.col2, “inner”)

dataframe.join(dataframe2, dataframe.col1 == dataframe.col2, “left”)

Save on storage

Write - action operation

```
dataframe.write.save("test.parquet", format="parquet")
dataframe.write.parquet("test.parquet")
dataframe.write.save("test.json", format="json")
dataframe.write.mode('append').parquet("test.parquet")
```

Parquet

УНИВЕРСИТЕТ ИТМО

Parquet is a columnar storage format - files organized by column. This format supports efficient compression and encoding schemes.

Dataset	Size on Amazon S3	Query Run Time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	$5.75
Data stored in Apache Parquet Format	130 GB	6.78 seconds	2.51 GB	$0.01
Savings	87% less when using Parquet	34x faster	99% less data scanned	99.7% savings

```
dataframe.write.save("test.parquet", format="parquet")
dataframe.write.parquet("test.parquet")
dataframe.write.save("test.json", format="json")
```

https://parquet.apache.org/docs/file-format/

Parquet

Parquet is a columnar storage format - files organized by column. This format supports efficient compression and encoding schemes.

4-byte magic number "PAR1"
<Column 1 Chunk 1 + Column Metadata>
<Column 2 Chunk 1 + Column Metadata>
...
<Column N Chunk 1 + Column Metadata>
<Column 1 Chunk 2 + Column Metadata>
<Column 2 Chunk 2 + Column Metadata>