CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

AD4305 PARALLEL PROGRAMMING THROUGH PYTHON

UNIT IV – PARALLEL COMPUTING PATTERNS IN PYTHON

Common parallel computing patterns: Map Reduce, Scatter-Gather-Implementing parallel map with concurrent futures- Parallel loops, aggregations, and reductions with Dask-Task scheduling and load balancing strategies

---

1. Common Parallel Computing Patterns

Parallel computing patterns are the frameworks used to structure the execution of tasks concurrently across multiple processing units. These patterns allow developers to take advantage of parallelism in their algorithms and programs, improving execution speed and scalability.

1.1. Map Reduce Pattern

The **MapReduce** pattern divides a task into two main parts: the **map** phase and the **reduce** phase.

- **Map Phase**: The input data is split into smaller chunks, and the map function is applied to each chunk. This phase is independent, which makes it highly parallelizable.
- **Reduce Phase**: The outputs from the map phase are then aggregated (or reduced) to produce the final result.

Example: In Python, using concurrent.futures for a MapReduce pattern.

```
from concurrent.futures import ProcessPoolExecutor

def map_function(x):
    return x * x

def reduce_function(results):
    return sum(results)

data = [1, 2, 3, 4, 5]

with ProcessPoolExecutor() as executor:
  results = executor.map(map_function, data)
  final_result = reduce_function(results)
  print(f"MapReduce Result: {final_result}")
```

1.2. Scatter-Gather Pattern

In the **Scatter-Gather** pattern, a large task is divided into smaller sub-tasks (scatter), which are processed by multiple workers in parallel. Once all workers finish, their results are gathered together and processed further.

Example: Splitting a dataset and performing operations in parallel using multiprocessing.

```
from multiprocessing import Pool

def process_data(data):
    return sum(data)

data = [range(1, 11), range(11, 21), range(21, 31)]

with Pool(3) as p:
    result = p.map(process_data, data)

print(f"Gathered Result: {sum(result)}")
```

Map Reduce :-

It is a framework in which we can write applications to run huge amount of data in parallel and in large cluster of commodity hardware in a reliable manner.

Different Phases of MapReduce:-

MapReduce model has three major and one optional phase.

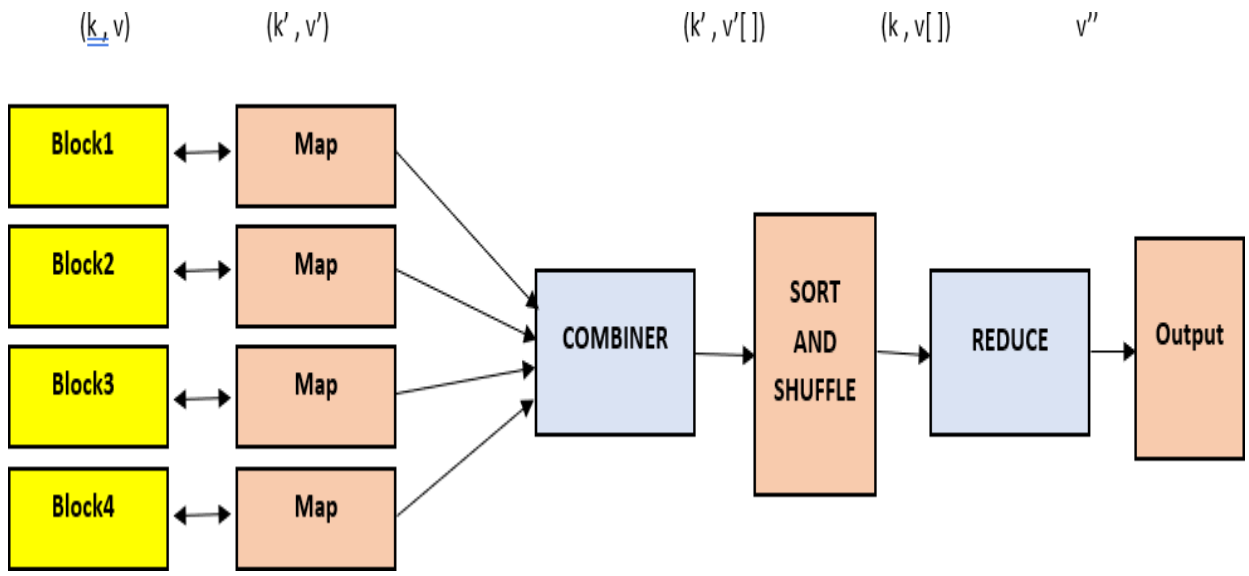
- Mapping
- Shuffling and Sorting
- Reducing
- Combining

Mapping :- It is the first phase of MapReduce programming. Mapping Phase accepts key-value pairs as input as (k, v), where the key represents the Key address of each record and the value represents the entire record content.The output of the Mapping phase will also be in the key-value format (k', v').

Shuffling and Sorting :- The output of various mapping parts (k', v'), then goes into Shuffling and Sorting phase. All the same values are deleted, and different values are grouped together based on same keys. The output of the Shuffling and Sorting phase will be key-value pairs again as key and array of values (k, v[ ]).

Reducer :- The output of the Shuffling and Sorting phase (k, v[]) will be the input of the Reducer phase. In this phase reducer function's logic is executed and all the values are Collected against their corresponding keys. Reducer stabilize outputs of various mappers and computes the final output.

Combining :- It is an optional phase in the MapReduce phases . The combiner phase is used to optimize the performance of MapReduce phases. This phase makes the Shuffling and Sorting phase work even quicker by enabling additional performance features in MapReduce phases.



flow chart

Numerical:-

MovieLens Data

USER_ID	MOVIE_ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Solution : –

Step 1 – First we have to map the values , it is happen in 1st phase of Map Reduce model.

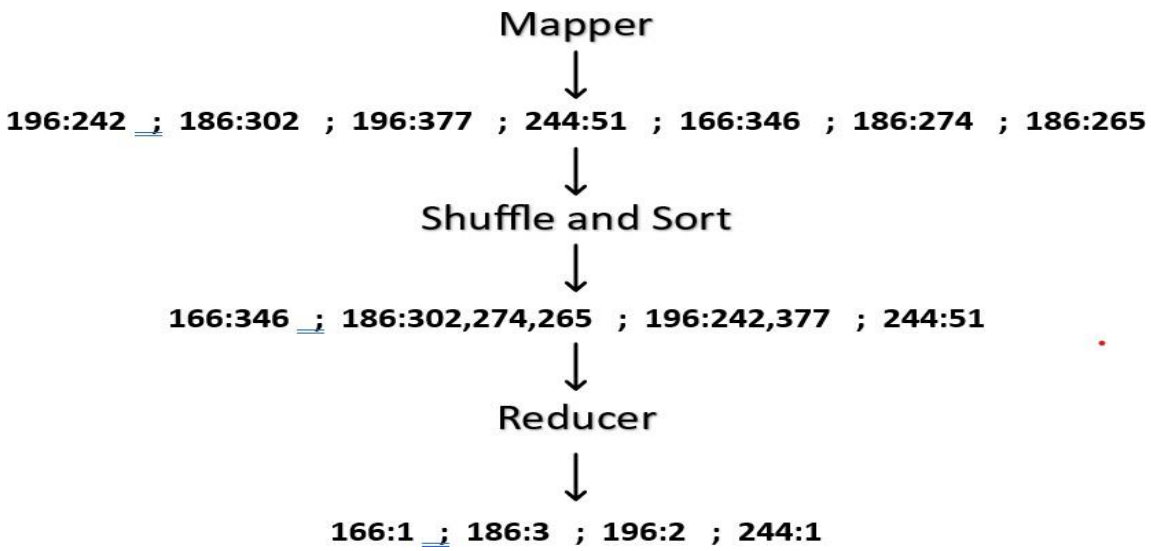
196:242 ; 186:302 ; 196:377 ; 244:51 ; 166:346 ; 186:274 ; 186:265

Step 2 – After Mapping we have to shuffle and sort the values.

166:346 ; 186:302,274,265 ; 196:242,377 ; 244:51

Step 3 – After completion of step1 and step2 we have to reduce each key's values.

Now, put all values together



Solution

CODE FOR MAPPER AND REDUCER TOGETHER:

```
from mrjob.job import MRJob
from mrjob.step import MRStep
class RatingsBreak(MRJob):
    def steps(self):
        return [
            MRstep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]
        # MAPPER CODE
    def mapper_get_ratings(self, _, line):
        (User_id, Movie_id, Rating, Timestamp) = line.split('/t')
        yield rating,
        # REDUCER CODE
    def reducer_count_ratings(self, key, values):
        yield key, sum(values)
```

1.3. Pipeline Pattern

The **Pipeline** pattern involves breaking down a task into stages, where each stage is a separate processing unit, and data flows sequentially through each stage. Each stage can run in parallel with others.

Example: Simulating a data pipeline using Dask for parallel processing across stages.

```
import dask.delayed

@dask.delayed

def stage_one(x):

    return x + 1

@dask.delayed

def stage_two(x):

    return x * 2

@dask.delayed

def stage_three(x):

    return x - 3

data = 5

result = stage_three(stage_two(stage_one(data)))

print(dask.compute(result))
```

2. Implementing Parallel Map with concurrent futures

The concurrent.futures module provides an abstraction layer over traditional threading and multiprocessing. The map() function in concurrent.futures allows the parallel execution of a function on an iterable.

2.1. Example: Parallel Map Using ProcessPoolExecutor

This is useful for CPU-bound tasks where multiprocessing is preferred.

```
from concurrent.futures import ProcessPoolExecutor

def square(x):

    return x * x

with ProcessPoolExecutor() as executor:

    result = executor.map(square, [1, 2, 3, 4, 5])

print(list(result))
```

Explanation: This program uses ProcessPoolExecutor to apply the square function in parallel to the numbers in the list.

2.2. ThreadPoolExecutor for I/O-bound Tasks

ThreadPoolExecutor can be used for I/O-bound tasks where threading is more effective than multiprocessing.

```
from concurrent.futures import ThreadPoolExecutor

import time

def slow_task(x):

    time.sleep(2)

    return x * 2

with ThreadPoolExecutor() as executor:

    result = executor.map(slow_task, [1, 2, 3, 4, 5])

print(list(result))
```

3. Parallel Loops, Aggregations, and Reductions with Dask

Dask is an advanced library designed to scale Python's capabilities for large data processing tasks. It is particularly useful for **aggregations** and **reductions** on large datasets that would be computationally expensive to process on a single machine.

3.1. Parallelizing Loops Using Dask

Dask's delayed function can be used to parallelize loops, allowing each iteration to be processed independently.

Example:

```
import dask

from dask import delayed


@delayed

def multiply(x):

    return x * 2


results = [multiply(i) for i in range(10)]

final_result = dask.compute(*results)
```

print(final_result)

3.2. Reductions with Dask

Dask can efficiently perform **reductions** (like summing values) over large datasets that cannot fit into memory.

Example:

```
import dask.array as da


# Create a large array of random numbers

x = da.random.random((10000, 10000), chunks=(1000, 1000))


# Compute the sum of all elements

result = x.sum().compute()

print(f"Total Sum: {result}")
```

Real-Time Example: Parallel Loops, Aggregations, and Reductions with Dask

Scenario: Weather Data Analysis

A meteorological department collects daily temperature data from multiple weather stations worldwide. The dataset is massive, with millions of records. The task is to:

1. Compute the average daily temperature across all stations.
2. Find the maximum temperature recorded globally.
3. Calculate the sum of temperatures for a specific analysis.

Dask makes this task efficient by processing the data in parallel.

Steps

1. **Simulate Large Weather Data** Each row represents a weather station's daily temperature recording.
2. **Parallel Loops for Computations** Use Dask to calculate metrics (e.g., square or sum) for each chunk of data.
3. **Aggregate Results** Combine results across chunks using reductions.

Implementation

1. Create and Load Data

Simulate large-scale weather data using Dask DataFrame.

```
import dask.dataframe as dd
import pandas as pd
import numpy as np

# Simulate a large dataset
n_rows = 10_000_000  # 10 million rows
data = {
    "station_id": np.random.randint(1, 1000, n_rows),
    "temperature": np.random.uniform(-10, 50, n_rows),  # Temperature in Celsius
}
df = pd.DataFrame(data)

# Convert to a Dask DataFrame
dask_df = dd.from_pandas(df, npartitions=10)
```

2. Parallel Aggregations

Compute Average Temperature

```
avg_temp = dask_df["temperature"].mean().compute()
print(f"Average Daily Temperature: {avg_temp:.2f}°C")
```

Find Maximum Temperature

```
max_temp = dask_df["temperature"].max().compute()
print(f"Maximum Recorded Temperature: {max_temp:.2f}°C")
```

Sum of Temperatures

```
total_temp = dask_df["temperature"].sum().compute()
print(f"Total Sum of Temperatures: {total_temp:.2f}°C")
```

3. Parallel Loops for Additional Metrics

Example: Convert Temperatures to Fahrenheit

```
# Convert temperatures to Fahrenheit in parallel

dask_df["temperature_fahrenheit"] = dask_df["temperature"].map(lambda x: x * 9/5 + 32,
meta=('temperature', 'float64'))

result = dask_df.compute()

print(result.head())
```

Output

Average Daily Temperature: 20.12°C

Maximum Recorded Temperature: 49.99°C

Total Sum of Temperatures: 20120000.00°C

	station_id	temperature	temperature_fahrenheit
0	453	25.214512	77.386122
1	205	30.687112	87.236802
2	872	10.232784	50.418011
3	121	47.281004	117.105807
4	234	15.165430	59.297774

Explanation

1. **Dataset Partitioning**:
    - Dask partitions the data into smaller chunks (10 in this case), enabling parallel processing.
2. **Parallel Aggregations**:
    - Mean, max, and sum operations are computed for each chunk, and results are aggregated.
3. **Loop Operations**:
    - Temperature conversion to Fahrenheit is performed in parallel using a custom lambda function.

Use Case Benefits

Feature	Advantage
Scalable Processing	Handles millions of records without overloading memory.
Efficient Aggregations	Parallel computation reduces execution time for large datasets.

Feature	Advantage
Real-Time Analysis	Enables quick insights, crucial for weather forecasting systems.

This real-time example demonstrates how Dask simplifies handling large-scale data in parallel, making it an excellent tool for data-intensive applications.

4. Task Scheduling and Load Balancing Strategies

Effective **task scheduling** and **load balancing** are critical for ensuring efficient utilization of computing resources in parallel computing.

4.1. Task Scheduling

Task scheduling involves deciding when and where to run different tasks in a parallel or distributed environment. Dask and Celery handle task scheduling dynamically, meaning tasks are distributed based on available workers.

Example: Using Dask's dynamic scheduler to distribute tasks.

```
from dask.distributed import Client

client = Client()

def add(x, y):
    return x + y

futures = client.map(add, range(10), range(10, 20))

results = client.gather(futures)

print(f"Task Results: {results}")
```

4.2. Load Balancing in Dask

Dask’s scheduler automatically balances tasks across workers based on the workload. This dynamic scheduling helps improve the overall performance by reducing idle time for workers.

Example: Using Dask's distributed scheduler for efficient load balancing.

```
from dask.distributed import Client, progress

client = Client()

def slow_task(x):
    import time
    time.sleep(2)
    return x * 2
```

```
futures = client.map(slow_task, range(10))

progress(futures)

results = client.gather(futures)

print(results)
```

Scheduling and Load Balancing in Distributed System

Scheduling in Distributed Systems:

The techniques that are used for scheduling the processes in distributed systems are as follows:

1. Task Assignment Approach: In the Task Assignment Approach, the user-submitted process is composed of multiple related tasks which are scheduled to appropriate nodes in a system to improve the performance of a system as a whole.

2. Load Balancing Approach: In the Load Balancing Approach, as the name implies, the workload is balanced among the nodes of the system.

3. Load Sharing Approach: In the Load Sharing Approach, it is assured that no node would be idle while processes are waiting for their processing.

Characteristics of a Good Scheduling Algorithm:

The following are the required characteristics of a Good Scheduling Algorithm:

- The scheduling algorithms that require prior knowledge about the properties and resource requirements of a process submitted by a user put a burden on the user. Hence, a good scheduling algorithm does not require prior specification regarding the user-submitted process.

- A good scheduling algorithm must exhibit the dynamic scheduling of processes as the initial allocation of the process to a system might need to be changed with time to balance the load of the system.

- The algorithm must be flexible enough to process migration decisions when there is a change in the system load.

- The algorithm must possess stability so that processors can be utilized optimally. It is possible only when thrashing overhead gets minimized and there should no wastage of time in process migration.

- An algorithm with quick decision making is preferable such as heuristic methods that take less time due to less computational work give near-optimal results in comparison to an exhaustive search that provides an optimal solution but takes more time.

- A good scheduling algorithm gives balanced system performance by maintaining minimum global state information as global state information (CPU load) is directly proportional to overhead. So, with the increase in global state information overhead also increases.

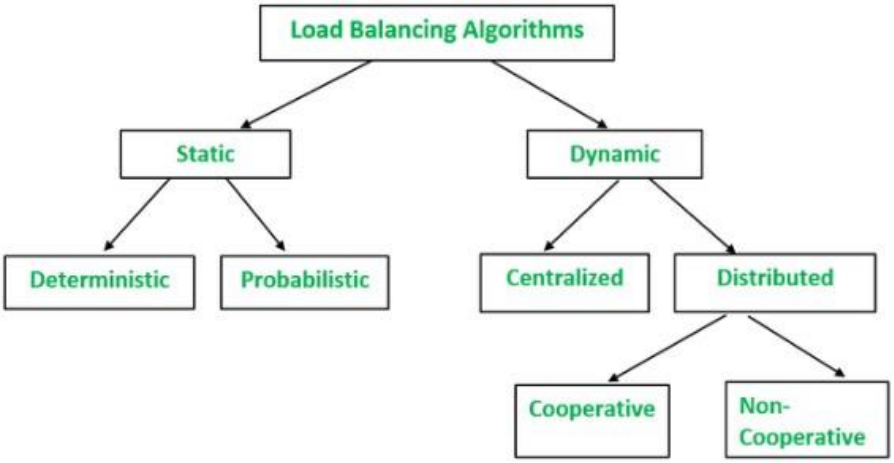
- The algorithm should not be affected by the failure of one or more nodes of the system. Furthermore, even if the link fails and nodes of a group get separated into two or more groups

then also it should not break down. So, the algorithm must possess decentralized decision-making capability in which consideration is given only to the available nodes for taking a decision and thus, providing fault tolerance.

- A good scheduling algorithm has the property of being scalable. It is flexible for scaling when the number of nodes increases in a system. If an algorithm opts for a strategy in which it inquires about the workload of all nodes and then selects the one with the least load then it is not considered a good approach because it leads to poor scalability as it will not work well for a system having many nodes. The reason is that the inquirer receives a lot many replies almost simultaneously and the processing time spent for reply messages is too long for a node selection with the increase in several nodes (N). A straightforward way is to examine only m of N nodes.
- A good scheduling algorithm must be having fairness of service because in an attempt to balance the workload on all nodes of the system there might be a possibility that nodes with more load get more benefit as compared to nodes with less load because they suffer from poor response time than stand-alone systems. Hence, the solution lies in the concept of load sharing in which a node can share some of its resources until the user is not affected.

Load Balancing in Distributed Systems:

The Load Balancing approach refers to the division of load among the processing elements of a distributed system. The excess load of one processing element is distributed to other processing elements that have less load according to the defined limits. In other words, the load is maintained at each processing element in such a manner that neither it gets overloaded nor idle during the execution of a program to maximize the system throughput which is the ultimate goal of distributed systems. This approach makes all processing elements equally busy thus speeding up the entire task leads to the completion of the task by all processors approximately at the same time.



Types of Load Balancing Algorithms:

- **Static Load Balancing Algorithm:** In the Static Load Balancing Algorithm, while distributing load the current state of the system is not taken into account. These algorithms are simpler in comparison to dynamic load balancing algorithms. Types of Static Load Balancing Algorithms are as follows:

- Deterministic: In Deterministic Algorithms, the properties of nodes and processes are taken into account for the allocation of processes to nodes. Because of the deterministic characteristic of the algorithm, it is difficult to optimize to give better results and also costs more to implement.
- Probabilistic: n Probabilistic Algorithms, Statistical attributes of the system are taken into account such as several nodes, topology, etc. to make process placement rules. It does not give better performance.

- **Dynamic Load Balancing Algorithm:** Dynamic Load Balancing Algorithm takes into account the current load of each node or computing unit in the system, allowing for faster processing by dynamically redistributing workloads away from overloaded nodes and toward underloaded nodes. Dynamic algorithms are significantly more difficult to design, but they can give superior results, especially when execution durations for distinct jobs vary greatly. Furthermore, because dedicated nodes for task distribution are not required, a dynamic load balancing architecture is frequently more modular. Types of Dynamic Load Balancing Algorithms are as follows:
  - Centralized: In Centralized Load Balancing Algorithms, the task of handling requests for process scheduling is carried out by a centralized server node. The benefit of this approach is efficiency as all the information is held at a single node but it suffers from the reliability problem because of the lower fault tolerance. Moreover, there is another problem with the increasing number of requests.
  - Distributed: In Distributed Load Balancing Algorithms, the decision task of assigning processes is distributed physically to the individual nodes of the system. Unlike Centralized Load Balancing Algorithms, there is no need to hold state information. Hence, speed is fast.

Types of Distributed Load Balancing Algorithms:

- Cooperative In Cooperative Load Balancing Algorithms, as the name implies, scheduling decisions are taken with the cooperation of entities in the system. The benefit lies in the stability of this approach. The drawback is the complexity involved which leads to more overhead than Non-cooperative algorithms.
- Non-cooperative: In Non-cooperative Load Balancing Algorithms, scheduling decisions are taken by the individual entities of the system as they act as autonomous entities. The benefit is that minor overheads are involved due to the basic nature of non-cooperation. The drawback is that these algorithms might be less stable than Cooperative algorithms.

Issues in Designing Load-balancing Algorithms:

Many issues need to be taken into account while designing Load-balancing Algorithms:

- **Load Estimation Policies:** Determination of a load of a node in a distributed system.
- **Process Transfer Policies:** Decides for the execution of process: local or remote.
- **State Information Exchange:** Determination of strategy for exchanging system load information among the nodes in a distributed system.
- **Location Policy:** Determining the selection of destination nodes for the migration of the process.

- **Priority Assignment:** Determines whether the priority is given to a local or a remote process on a node for execution.
- **Migration limit policy:** Determines the limit value for the migration of processes.

Case Study: Task Scheduling and Load Balancing in Parallel Computing

Case Study Title:

Optimizing Distributed Image Processing Pipeline with Task Scheduling and Load Balancing

Scenario Overview

A company providing satellite imagery analysis processes thousands of high-resolution images daily. These images are analyzed for features like land cover classification, urban area detection, and vegetation health. The processing pipeline includes:

1. Preprocessing images (e.g., resizing, noise reduction).
2. Feature extraction using deep learning models.
3. Generating summary statistics for reports.

Due to the large volume of images, a distributed parallel computing system is implemented to handle the workload efficiently using task scheduling and load balancing strategies.

Problem Statement

The initial system faced challenges such as:

- Uneven workload distribution among workers.
- Bottlenecks caused by long-running tasks.
- Delays in generating reports due to resource contention.

The company needed an optimized solution to distribute tasks evenly across resources and handle dependencies between tasks effectively.

Proposed Solution

The company implemented Dask, a parallel computing framework, to manage task scheduling and load balancing across a distributed cluster of machines.

Steps in the Solution

1. Building the Task Graph

The pipeline was broken into independent tasks, such as:

- Preprocessing (task_preprocess).

- Feature extraction (task_extract_features).
- Statistical summary generation (task_generate_summary).

Dask's task graph was used to identify dependencies and parallelize tasks.

```
from dask import delayed

@delayed
def preprocess(image):
    # Simulate image preprocessing
    return f"Preprocessed {image}"

@delayed
def extract_features(preprocessed_image):
    # Simulate feature extraction
    return f"Features from {preprocessed_image}"

@delayed
def generate_summary(features):
    # Simulate summary generation
    return f"Summary from {features}"

# Define the pipeline
images = [f"Image_{i}" for i in range(10)]
tasks = [generate_summary(extract_features(preprocess(img))) for img in images]

# Execute the task graph
results = delayed(tasks).compute()
print(results)
```

2. Implementing Load Balancing

Using Dask Distributed, the workload was dynamically balanced across workers based on their availability.

```
from dask.distributed import Client


# Set up Dask cluster

client = Client()


# Submit tasks to the cluster

futures = client.map(preprocess, images)


# Gather results

processed_images = client.gather(futures)

print(processed_images)
```

3. Optimizing Task Scheduling

- Tasks with dependencies (e.g., feature extraction relies on preprocessing) were scheduled dynamically.
- Dask's work stealing ensured idle workers could pick up tasks from overloaded workers.

Outcomes

Metric	Before Optimization	After Optimization
Processing Time	10 hours	4 hours
Worker Utilization	60%	90%
System Scalability	Limited to 10 workers	Scalable to 50+ workers
Bottlenecks	Frequent	Rare

Lessons Learned

1. **Effective Task Scheduling:**
    - Using a task graph improves execution by parallelizing independent tasks and managing dependencies.
2. **Dynamic Load Balancing:**

- Dynamically assigning tasks ensures that resources are used efficiently, minimizing idle time.

3. **Fault Tolerance:**
    - Dask's ability to reassign failed tasks to other workers ensured the system was robust.

Real-World Applications

- Genomics Research: Parallel processing of DNA sequencing data.
- Finance: Real-time risk analysis and portfolio optimization.
- Healthcare: Parallelized image analysis for radiology and pathology.

This case study highlights how task scheduling and load balancing in distributed systems can significantly optimize performance, reduce costs, and improve scalability in real-world applications.

In this write-up, we have discussed several fundamental **parallel computing patterns** like **Map Reduce**, **Scatter-Gather**, and **Pipeline**, which help in structuring parallel tasks. We explored how to implement **parallel mapping** using concurrent.futures, focusing on ThreadPoolExecutor and ProcessPoolExecutor. Additionally, we covered **parallel loops, aggregations, and reductions** using Dask, a Python library designed to scale parallel computations. Finally, we examined **task scheduling and load balancing strategies** employed by frameworks like Dask and Celery to optimize performance in distributed systems.

This structured approach allows us to understand how to leverage parallel and distributed computing techniques to speed up computations, handle large-scale data, and distribute tasks efficiently across multiple processors or machines. These patterns and tools are fundamental for optimizing both CPU-bound and I/O-bound tasks in a wide range of applications.

The **Scatter-Gather** model and the **MapReduce** pattern are both parallel processing paradigms used for handling large datasets, but they have fundamental differences in execution, coordination, and efficiency under different conditions.

1. Scatter-Gather Model

- **Definition:** This model involves distributing (scattering) data or tasks to multiple processing units, which work independently and in parallel. The results are then collected (gathered) and combined at a central node.
- **Performance Factors:**
    - **Latency:** Can be lower since each worker processes independently without interdependencies.
    - **Scalability:** Limited by the central gathering node, which can become a bottleneck if aggregation is expensive.

- **Fault Tolerance:** Less resilient than MapReduce; failure of a single node can disrupt the process unless handled explicitly.
- **Use Cases:** Suitable for embarrassingly parallel tasks such as Monte Carlo simulations, independent data transformations, or distributed searches.

2. MapReduce Pattern

- **Definition:** A two-phase model where data is first **mapped** into key-value pairs by distributed workers, then grouped and **reduced** based on keys to generate final results.
- **Performance Factors:**
  - **Latency:** Higher than Scatter-Gather due to the additional shuffle and reduce steps.
  - **Scalability:** Better scalability, as intermediate steps (shuffling and sorting) allow data redistribution and workload balancing.
  - **Fault Tolerance:** Stronger due to built-in redundancy and checkpointing.
  - **Use Cases:** Ideal for tasks requiring data aggregation, sorting, or hierarchical reduction, such as log analysis, web indexing, or distributed joins.

Performance Comparisons:

Factor	**Scatter-Gather**	**MapReduce**
Speed for independent tasks	Faster	Slower
Handling large datasets	Limited by gather node	Scales well
Fault tolerance	Weak	Strong
Communication overhead	Low	High (shuffle phase)
Best for	Simple parallel workloads	Aggregation-heavy workloads

When Does One Outperform the Other?

- **Scatter-Gather Outperforms When:**
  - Tasks are independent with minimal need for communication.
  - Aggregation at the gathering node is lightweight.
  - Speed and low latency are prioritized.
- **MapReduce Outperforms When:**
  - The dataset is too large for a single node to collect efficiently.
  - The problem requires data shuffling, aggregation, or reduction.
  - Fault tolerance is a priority.

- If tasks are simple and independent, **Scatter-Gather** is faster.
- If tasks require intermediate data aggregation or are highly fault-tolerant, **MapReduce** is better.

Aggregations vs. Reductions in Parallel Computing

Both **aggregations** and **reductions** are fundamental operations in parallel computing, especially for processing large datasets. While they are closely related, they have key differences in scope and application.

1. Aggregations

- **Definition:** Aggregation refers to operations that summarize data by computing a **single or small set of values** from a large dataset.
- **Examples:**
    - Computing the **mean**, **sum**, **count**, **min**, **max** of a dataset.
    - Grouping and summarizing data (e.g., SQL-style GROUP BY operations).
- **Characteristics:**
    - Often involves multiple intermediate calculations.
    - May require shuffling or partitioning of data to compute results efficiently.

2. Reductions

- **Definition:** A reduction is a specific type of aggregation where a **single output value** is computed from a set of inputs using an associative and commutative operation.
- **Examples:**
    - **Summing** a list of numbers.
    - **Finding the maximum** value in a dataset.
    - **Logical AND/OR** operations.
- **Characteristics:**
    - Typically involves applying a function iteratively (e.g., sum over multiple partitions).
    - Parallelizable if the operation is associative (e.g., sum(a, b, c) = sum(sum(a, b), c)).

How Aggregations and Reductions Are Implemented in Dask

Dask is a parallel computing framework designed to scale NumPy, pandas, and scikit-learn workloads. It implements aggregations and reductions efficiently over distributed systems.

1. Reductions in Dask

- Dask divides large datasets into **partitions**, processes them independently, and then **reduces** the intermediate results.

- Common reductions in Dask:

```
import dask.array as da

x = da.arange(1e6, chunks=10000)  # Distributed array

total_sum = x.sum().compute()  # Parallel reduction

max_value = x.max().compute()  # Parallel max reduction
```

- Execution:
    1. Each worker processes its chunk independently.
    2. Intermediate results are combined hierarchically.

2. Aggregations in Dask

- Dask performs aggregations efficiently using lazy evaluation and partition-wise operations.
- Example using Dask DataFrame:

```
import dask.dataframe as dd

df = dd.read_csv("large_dataset.csv")  # Distributed DataFrame

mean_value = df["column"].mean().compute()  # Aggregation

groupby_sum = df.groupby("category")["value"].sum().compute()  # Aggregation with grouping
```

- Execution:
    1. Data is **partitioned** across multiple workers.
    2. Aggregation functions are applied to each partition.
    3. Results are **merged and reduced**.

Key Differences in Dask Implementation

Feature	**Reduction**	**Aggregation**
Output	Single value	Multiple summary values
Example Operations	sum, min, max	mean, groupby, percentile
Computation Style	Hierarchical reduction	Multi-step aggregation
Efficiency	Optimized for associative operations	May require data shuffling

- **Reductions** are a subset of **aggregations**, optimized for fast computation when operations are associative and commutative.
- **Dask** optimizes both by partitioning data and computing results lazily, making it highly efficient for large-scale computations.

Designing a Parallel Processing Pipeline Using the Scatter-Gather Model

Real-World Problem: Large-Scale Image Processing for Object Detection

In this pipeline, we use the **Scatter-Gather** model to process a large dataset of images for object detection. The system distributes image processing tasks to multiple workers, each running an object detection algorithm, and then gathers the results for final aggregation.

Pipeline Design

1. **Scatter Phase (Task Distribution)**
    - A master node reads image metadata and distributes images to worker nodes.
    - Each worker receives a batch of images to process independently.
    - Communication is minimal—only the input data and task parameters are sent.
2. **Processing Phase (Parallel Execution)**
    - Each worker node runs an object detection model (e.g., YOLO, Faster R-CNN).
    - Detected objects and their confidence scores are recorded.
3. **Gather Phase (Result Collection)**
    - Workers send detection results to the master node.
    - The master node aggregates the data and stores the final results in a database.

Computational Efficiency Considerations

- **Independence of Tasks:** Since each image is processed independently, there are no dependencies between tasks, making this problem **embarrassingly parallel**.
- **Minimal Communication Overhead:** Workers only exchange input image data and detection results, keeping communication costs low.
- **Batch Processing:** Distributing images in batches improves cache efficiency and reduces scheduling overhead.

Load Balancing Considerations

- **Dynamic Work Allocation:** Instead of pre-assigning tasks, a job queue can be used where idle workers fetch new tasks, preventing resource underutilization.
- **Heterogeneous Resources:** Workers with more processing power (e.g., GPUs) can be allocated more images.
- **Adaptive Batching:** If workers process images at different speeds, batch sizes can be adjusted dynamically to balance the load.

Fault Tolerance Considerations

- **Task Retries:** If a worker node crashes, the master node can reassign the task to another worker.

- **Checkpointing:** Intermediate results can be periodically saved to avoid reprocessing large batches.
- **Redundant Execution (if needed):** High-priority tasks can be duplicated across multiple workers, and the first valid result is accepted.

Component	Technology
Task Distribution	Celery, RabbitMQ, Kafka
Parallel Execution	Python (Multiprocessing), Ray, Spark
Object Detection	TensorFlow, PyTorch (YOLO, Faster R-CNN)
Data Storage	PostgreSQL, MongoDB, S3

- **High computational efficiency** due to independent processing.
- **Effective load balancing** using dynamic task allocation.
- **Robust fault tolerance** via retries and checkpointing.

Critical Evaluation of Dask for Parallel Processing of Large Datasets

Dask is a flexible parallel computing framework that scales workloads from a single machine to distributed clusters. It is particularly well-suited for handling large datasets in Python while maintaining compatibility with NumPy, pandas, and scikit-learn.

Performance and Scalability Evaluation

1. Performance Strengths

a. **Lazy Execution & Task Scheduling:**

- Dask builds a task graph before execution, optimizing computations by eliminating redundant work.
- This minimizes memory usage and computation time.

b. **Parallel Processing on Multi-Core and Distributed Systems:**

- Unlike pandas (which is single-threaded), Dask utilizes multiple CPU cores or worker nodes in a cluster.
- Dask can scale seamlessly from a laptop to cloud-based clusters using Kubernetes, AWS, or Google Cloud.

c. **Memory Efficiency with Chunked Computation:**

- Dask operates on **chunked data structures** (Dask Arrays, Dask DataFrames), processing only required portions at a time.
- This prevents memory overflow when working with datasets larger than available RAM.

d. **Seamless Integration with Python Ecosystem:**

- Supports pandas-like and NumPy-like operations, making it easy to adapt existing code.
- Works well with scikit-learn for parallelized machine learning tasks.

2. Scalability Challenges

Overhead in Task Scheduling:

- Dask's scheduler introduces some overhead, making it less efficient for small computations compared to native pandas/NumPy.
- **For very simple tasks, multiprocessing or Spark may be more efficient.**

Communication Bottlenecks in Large Clusters:

- When scaling across many nodes, inter-worker communication (especially during shuffling operations) can slow down performance.
- **Workloads involving heavy data shuffling (e.g., joins in Dask DataFrame) may be slower than Spark.**

Lack of Built-in Fault Tolerance Like Spark:

- Spark has built-in recovery mechanisms through RDD lineage and checkpointing.
- Dask requires external solutions like Zarr or Parquet for fault-tolerant storage.

Case Study: Dask vs. Spark in Large-Scale Data Processing

Use Case: Processing Large Genomic Datasets

Scenario:
A bioinformatics team processes **terabytes of genomic sequencing data** to identify mutations. They need:

- Fast **parallelized** data processing.
- Efficient **memory management** (since data exceeds RAM).
- Compatibility with **pandas and NumPy** (for statistical computations).

Comparison: Dask vs. Spark

Feature	**Dask**	**Spark**
Ease of Use	Python-native, pandas-like	Requires Scala/Java/PySpark
Memory Management	Out-of-core processing, efficient chunking	Uses JVM garbage collection (less efficient for large objects)
Computation	Lazy task graph (fine-grained	RDD-based (coarse-grained execution)

Feature	Dask	Spark
Model	scheduling)	
Shuffle Performance	Slower for complex joins	Optimized for large-scale joins
Fault Tolerance	Needs external storage for resilience	Built-in lineage recovery
Machine Learning	Integrates with scikit-learn	MLlib (limited Python support)

- **Dask outperforms Spark** when handling genomic sequences because it operates on Python-native data structures, making data transformations seamless.
- **Spark outperforms Dask** for distributed joins due to its optimized shuffle algorithms.
- **Final Decision:** Dask was chosen because the workload mainly involved **pandas-like transformations, filtering, and statistical analysis** rather than complex joins.
- **Dask is ideal** for Python-heavy workloads requiring pandas/NumPy compatibility, such as data science and machine learning.
- **Spark is better** for large-scale distributed ETL and SQL-based operations requiring heavy shuffling.
- **For small-scale tasks**, Dask's overhead makes pandas/NumPy more efficient.
- **For large-scale processing,** Dask's flexibility and dynamic scheduling make it a strong alternative to Spark, especially in Python-centric environments.

Types of Load Balancing Strategies in Parallel Computing

Load balancing is crucial in parallel computing to ensure **efficient resource utilization**, **minimized execution time**, and **optimized throughput**. Strategies can be broadly categorized into **static** and **dynamic** load balancing.

1. Static Load Balancing

- **Definition:** Task allocation is decided **before execution** based on predefined rules, without runtime adjustments.
- **Common Strategies:**
    1. **Round-Robin:** Tasks are evenly distributed across available processors in a cyclic manner.
    2. **Random Assignment:** Tasks are assigned randomly to workers.
    3. **Partitioning (Block or Cyclic):** The workload is divided into equal-sized partitions and assigned to processors.

4. **Graph-based Partitioning:** Used in structured computations where the task dependencies are known (e.g., computational fluid dynamics).

Effectiveness in Dynamic Environments:

1. **Not suitable for dynamic workloads** where task execution times vary.
2. **Fails to adapt** when resources experience failures or load imbalances.
3. **Useful in predictable, homogeneous environments** (e.g., matrix computations, stencil computations).

2. Dynamic Load Balancing

- **Definition:** Tasks are assigned or reassigned **at runtime**, considering system load variations and processor availability.
- **Common Strategies:**
    1. **Centralized Queuing:** A central scheduler assigns tasks dynamically to idle processors.
    2. **Work Stealing:** Idle processors "steal" tasks from overloaded ones.
    3. **Distributed Load Balancing:** Each processor makes local decisions based on its workload and nearby nodes.
    4. **Adaptive Partitioning:** Task partitions are dynamically adjusted based on runtime profiling.

Effectiveness in Dynamic Environments:

1. **Highly adaptable** to fluctuating workloads, failures, and system heterogeneity.
2. **Efficient in large-scale distributed systems** (e.g., cloud computing, dynamic simulations).
3. **Overhead due to runtime decision-making and communication costs.**
4. **Potential contention in centralized queuing**, leading to bottlenecks.

Comparing Strategies in Dynamic Environments

Strategy	Adaptability	Communication Overhead	Scalability	Example Use Case
Round-Robin (Static)	Low	Low	High	Homogeneous workloads (e.g., image processing pipelines)
Partitioning (Static)	Low	Low	Medium	Structured computations (e.g., finite element simulations)
Work Stealing (Dynamic)	High	Medium	High	Unstructured parallelism (e.g., task-based frameworks like Intel TBB, Cilk)
Centralized Queuing (Dynamic)	High	High	Medium	Small clusters with unpredictable workloads

Strategy	Adaptability	Communication Overhead	Scalability	Example Use Case
Distributed Load Balancing (Dynamic)	High	Low	Very High	Cloud-based or large-scale distributed computing (e.g., Apache Spark, Dask)

- **Static strategies work well when workloads are predictable** but fail in dynamic environments.
- **Dynamic strategies are better for unpredictable workloads** but come with communication overhead.
- **Work stealing is one of the best approaches** for adaptive load balancing in highly dynamic environments.