CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

AD4305 - PARALLEL PROGRAMMING THROUGH PYTHON

UNIT III DISTRIBUTED COMPUTING IN PYTHON

Introduction to distributed computing with Python- Setting up and managing worker processes with multiprocessing and concurrent futures- Using message passing with Celery for distributed tasks - Introduction to Dask for scalable analytics and parallel computing.

Introduction to distributed Computing With Python

What is Distributed Computing?

Distributed computing is a computing paradigm where large or complex problems are divided into smaller subtasks and executed concurrently across multiple computers (nodes) connected via a network. These systems work together to produce a unified result, enabling greater efficiency, scalability, and reliability.

"Divide and conquer at machine scale."

Simple Analogy

Imagine solving a 1,000-piece puzzle:

- A **single person** takes hours to complete it.
- If **10 friends** each solve a section, the task completes much faster.

Distributed computing mirrors this concept by distributing computing work to multiple systems.

Core Idea

Divide and Conquer — Break big problems into smaller, manageable subtasks and solve them in parallel across several computers for greater speed, efficiency, and reliability.

Key Benefits of Distributed Computing

Feature	Description
Scalability	Easily add more computers (nodes) as workloads grow.
Fault Tolerance	If one computer fails, others can continue the work.
Performance	Many tasks run in parallel, drastically accelerating computation.
Cost Efficiency	Multiple affordable machines can replace a supercomputer.
Geographical	Distribute resources worldwide for faster, local access (e.g., CDNs).

Components of a Distributed System

- **Nodes (Workers)**: Physical or virtual machines that execute tasks.
- **Master (Coordinator)**: Assigns, monitors, and aggregates subtasks (can be distributed or redundant).
- **Network**: Facilitates node communication, often over local or global networks.
- **Middleware**: Software that abstracts complexities, handles discovery, messaging, and coordination (e.g., Dask, Celery).
- **Data Storage**: Distributed file systems or databases that ensure fast, consistent access across all nodes.

Types of Distributed Computing

Type	Description	Example
Parallel Computing	Multiple cores/CPUs handle concurrent tasks	Scientific simulations, matrix multiplication
Cloud Computing	On-demand, remote resources via the internet	AWS Lambda, Google Cloud Functions
Cluster Computing	Interconnected machines as a unified system	Hadoop, Kubernetes, SLURM cluster
Peer-to-Peer (P2P)	Decentralized resource sharing	BitTorrent, file-sharing, blockchains
Grid Computing	Heterogeneous systems across organizations	Folding@home, SETI@home

Key Concepts and Terminology

- **Concurrency:** Multiple tasks make progress at overlapping times.
- **Parallelism:** Tasks are executed truly simultaneously, often on different processors.
- **Load Balancing:** Evenly spreads out tasks to prevent any node from being overloaded.
- **Task Scheduling:** Management of when and where subtasks run.
- **Latency:** Delay in sending data between machines, critical in global systems.
- **Throughput:** The amount of work completed in a fixed time.
- **Consistency:** Ensures all nodes have a synchronized view of data.
- **Transparency:** Users experience the distributed system as a single system.

Architecture Models in Distributed Computing

Model	**Description**	**Example**
Client-Server	Clients request resources; servers respond	Web apps, databases
Peer-to-Peer (P2P)	All nodes act as both client and server	File-sharing, blockchains
Three-Tier	Presentation, application, and data layers distributed	Enterprise business apps
Microservices	Application is split into small, independently deployable services	Netflix, Kubernetes
Service-Oriented Arch	Services communicate over the network, each providing a specific business function	Enterprise integration

Advanced Use Cases and Applications

- **Internet and Web Infrastructure**: Search engines, content delivery networks (CDNs), online marketplaces.
- **High-Speed Financial Trading**: Real-time synchronization of stock trades and risk calculations.
- **Healthcare & Life Sciences**: Drug discovery, genomics, remote diagnostics, and robotic surgery.
- **Smart Grids & Energy**: Real-time monitoring, optimization, and automation across power networks.
- **Manufacturing & IoT**: Distributed control for automated factories and edge device networks.
- **Online Gaming & VR**: MMO game servers handling massive concurrent players globally.
- **Big Data Analytics**: Hadoop, Spark for terabyte and petabyte scale processing.
- **Scientific Research**: Protein folding, space simulations, climate modeling.
- **Blockchain & Decentralized Apps**: Secure, tamper-proof transactions and distributed ledgers.

Consistency Models in Distributed Systems

- **Strong Consistency**: Guarantees all users see the same data at the same time (harder to achieve).
- **Eventual Consistency**: Updates will reach all nodes eventually—favored in large, global systems.
- **Causal Consistency**: Ensures that cause-and-effect relationships are preserved in data updates.
- **CAP Theorem**: In distributed systems, you can choose only two of Consistency, Availability, and Partition tolerance at the same time.

Security in Distributed Computing

- **Access Control**: Ensures only authorized devices/users can access resources.
- **Encryption**: Safeguards data in transit between nodes.
- **Authentication & Authorization**: Confirms identity and permissions for communication.
- **Auditing/Logging**: Tracks and reviews distributed system activities for compliance.

Emerging Trends and Future Directions

Trend	**Description**
Edge Computing	Processing data near the source on IoT devices for lower latency.
Serverless Computing	Functions executed in the cloud, hardware abstracted away.
Federated Learning	Decentralized ML model training with data privacy (e.g., mobile devices).
Microservices & Service Mesh	Fine-grained control, resilience, and monitoring of modern cloud apps.
Distributed AI/ML	Training huge models across GPU clusters and TPU pods.
Blockchain/DLT	Decentralized consensus and tamper-resistant ledgers.

Python's Role in Distributed Computing

Python's simplicity and vast ecosystem have made it a tool of choice, offering:

- Clean syntax, widespread adoption in data science and engineering.
- Libraries like multiprocessing, concurrent.futures, Celery, Dask, Ray, PySpark, mpi4py.
- Integration with big data and machine learning pipelines for scalable solutions.

Example Code: Using Dask for Distributed Data Processing

```
import dask.array as da
# Create a large random array (1 billion elements), split into chunks
x = da.random.random(size=int(1e9), chunks=int(1e7))
# Compute the sum in parallel across all available cores
total = x.sum().compute()
print("Sum:", total)
```

Dask splits data and computation so multiple workers process chunks in parallel, scaling seamlessly from a laptop to a large cluster.

Recap & Real-World Impact

Distributed computing powers:

- The backbone of the internet and global web services.
- Real-time financial transactions and trading platforms.
- Personalized medical diagnostics, gene sequencing, and smart grid energy management.
- Modern AI/ML applications, from GPT-style models to federated learning.
- Global-scale online gaming, video streaming, and content delivery.

Modern organizations depend on distributed systems for **scalability, reliability, and efficiency**—and new trends (edge, serverless, federated learning, distributed AI) continuously expand what's possible.

By introducing these advanced subtopics and expanding on architectural models, use cases, security, and emerging trends, this overview offers a thorough and up-to-date foundation for learning, teaching, or applying distributed computing in real-world scenarios.

Add to follow-up

Check sources

1. https://www.geeksforgeeks.org/computer-networks/examples-and-applications-of-distributed-systems-in-real-life/
2. https://www.couchbase.com/blog/distributed-applications/
3. https://en.wikipedia.org/wiki/Distributed_computing
4. https://konghq.com/blog/engineering/what-are-distributed-applications
5. https://aws.amazon.com/what-is/distributed-computing/
6. https://www.supermicro.com/en/glossary/distributed-computing
7. https://hazelcast.com/foundations/distributed-computing/distributed-computing/
8. https://www.geeksforgeeks.org/devops/what-is-distributed-computing/
9. https://www.ibm.com/think/insights/distributed-computing-use-cases
10. https://www.techtarget.com/whatis/definition/distributed-computing

11. https://vfunction.com/blog/distributed-application/
12. https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture
13. https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/x2d2f703b37b450a3:parallel-and-distributed-computing/a/distributed-computing
14. https://www.ibm.com/think/topics/distributed-computing

Why Is Distributed Computing Needed?

1. To Handle Massive Data Volumes (Big Data)

- Today's systems generate terabytes to petabytes of data (e.g., sensors, social media, e-commerce).
- A single machine cannot process or store all of it efficiently.
- **Distributed systems divide and conquer** — each node handles a portion.

Example: Analyzing user behavior logs from billions of website clicks daily.

2. To Improve Speed and Performance

- Time-critical tasks (e.g., fraud detection, stock trading) demand **low latency and fast execution**.
- Multiple machines running in parallel reduce response time drastically.

Example: Fraud detection systems must identify anomalies in real time.

3. To Ensure System Availability and Fault Tolerance

- In centralized systems, a failure in one component can bring everything down.
- **Distributed systems can recover automatically** by shifting tasks to other nodes.

Example: Cloud storage systems like Google Drive replicate your data across multiple servers.

4. To Enable Scalability

- As businesses grow, so do their processing needs.
- Distributed systems allow **horizontal scaling** (adding more nodes) rather than upgrading one large server.

Example: Amazon Web Services (AWS) adds or removes servers dynamically to meet demand.

5. To Serve Global Users

- Placing servers in multiple geographic locations **reduces latency** for users around the world.
- Improves user experience for applications like video streaming or gaming.

Example: Netflix uses distributed edge servers (CDNs) to serve content faster worldwide.

6. To Support Modern Applications

- Emerging technologies like **IoT, AI/ML, blockchain, smart cities**, etc., require systems that are distributed by design.
- Devices and services are inherently spread out and need coordination.

Example: Self-driving cars use edge computing to process sensor data in real time — a form of distributed processing.

Where Is Distributed Computing Used?

Sector	**Use Case Example**
AI/ML	Training large models like GPT across GPU clusters
E-Commerce	Handling millions of concurrent orders and inventory systems
Finance	Risk modeling, fraud detection, high-frequency trading
Healthcare	Remote diagnostics, genome sequencing, AI-driven drug discovery
Energy	Smart grid automation, load balancing across distributed stations
Gaming	Real-time multiplayer experiences with global server distribution
IoT & Mobility	Edge computing in autonomous cars, smart traffic systems
Telecom	Distributed routing, call/data management across global networks
Web Services	Google Search, Facebook, Amazon, Netflix—all rely on massive distributed systems
Scientific Research	Protein folding, weather simulation, astrophysics simulations

What Happens If We Don't Use Distributed Computing?

1. Limited Performance and Slow Execution

- A single machine (even a powerful one) has **finite CPU, RAM, and I/O capacity**.
- As data grows, performance **degrades** significantly.

Example: A weather simulation for a continent may take **days** instead of **hours** without distributed processing.

2. No Scalability

- Scaling up one system (called *vertical scaling*) hits limits quickly.
- Distributed computing enables *horizontal scaling*—adding more machines.

Without it: You can't serve millions of users or handle peak loads efficiently (e.g., Black Friday traffic on Amazon).

3. Single Point of Failure (SPOF)

- If the only server goes down, the entire system crashes.
- Distributed systems provide **fault tolerance**—if one node fails, others continue.

Without it: Imagine if Google Search went down worldwide due to one server crashing.

4. Inability to Handle Big Data

- Today's datasets are **too big to fit in the memory or storage** of one machine.
- Tasks like data mining, real-time analytics, and deep learning models would become impossible or extremely slow.

Without it: Social media analysis, recommendation engines, or fraud detection would break or lag massively.

5. Geographical Latency

- A centralized system located in one region results in **high latency** for distant users. *Without distributed computing*: Users in Asia experience delays accessing a server in the US (slow video streaming, high ping in games).

6. No Support for Real-Time Systems

- Real-time systems (e.g., IoT, self-driving cars, banking) need fast decision-making using distributed, decentralized processing.

Without it: Real-time fraud detection or live vehicle tracking fails due to slow processing.

7. Lack of Resilience in AI/ML

- Training AI models on massive datasets **requires splitting workloads** across GPUs, TPUs, or machines.
- A single machine cannot handle deep learning at scale.

Without it: GPT, DALL·E, BERT, etc., couldn't be trained or deployed.

In Simple Terms:

Without Distributed Computing...	**Result**
Increasing users or data	Slows or crashes system
A machine failure	Brings down the whole service
Global user access	Causes high delays
Training large models	Becomes impractical
Data-driven apps	Cannot scale or function in real time

Setting Up and Managing Worker Processes

Using multiprocessing and concurrent.futures in Python

Why Worker Processes?

- **Worker processes** enable **parallel execution** of tasks across CPU cores.
- Ideal for **CPU-bound** operations (e.g., math calculations, simulations).
- Improves performance by utilizing multi-core systems.

1. Using multiprocessing Module

🔷 **Step-by-Step Setup:**

Example: Using multiprocessing.Pool

```
from multiprocessing import Pool
def square(n):
    return n * n
if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]

    with Pool(processes=4) as pool:  # Create a pool of 4 workers
        results = pool.map(square, numbers)
    print("Squared:", results)
```

Key Concepts

Function	Description
Pool(processes=n)	Starts n worker processes
map(func, iterable)	Applies func to each item in iterable
close() & join()	Cleanly shut down workers

2. Using concurrent.futures.ProcessPoolExecutor

Example: Clean, High-Level API for Multiprocessing

```
from concurrent.futures import ProcessPoolExecutor

def factorial(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

if __name__ == "__main__":
    numbers = [5, 6, 7, 8]

    with ProcessPoolExecutor(max_workers=4) as executor:
        results = list(executor.map(factorial, numbers))

    print("Factorials:", results)
```

Key Features

Feature	Description
ProcessPoolExecutor	Creates worker processes
submit(func, arg)	Schedules a single function for execution
map(func, iterable)	Parallel map like multiprocessing.Pool.map
shutdown(wait=True)	Gracefully exits the pool

Comparison Table

Feature	multiprocessing	concurrent.futures
API Style	Low-level	High-level (cleaner)
Suitable For	More control, customization	Quick setup, cleaner syntax

Feature	multiprocessing	concurrent.futures
Parallelism Type	Process-based	Process-based (or thread-based)
Best Use	Complex process management	Simple CPU-bound parallel tasks

Managing and Monitoring Workers

Best Practices:

- Always use if __name__ == "__main__" in multiprocessing to prevent infinite spawning (especially on Windows).
- Use with statement to manage pool lifetimes (auto close() and join()).
- Avoid sharing mutable data between processes — use Queue, Pipe, or shared Value/Array.
- Catch exceptions using try/except inside the worker function or via future.result().

Real-World Use Cases

Use Case	Why Multiprocessing?
Image Processing	Handle many files in parallel
Scientific Simulations	Each simulation run is CPU-intensive
File Compression/Conversion	Distribute file jobs among processes
Machine Learning Preprocessing	Apply transformations across large datasets

Summary

- Python's multiprocessing and concurrent.futures provide robust tools for parallelizing CPU-bound tasks.
- multiprocessing offers control and flexibility.
- concurrent.futures offers simplicity and cleaner syntax.
- Always handle worker setup and shutdown carefully to avoid resource leaks.

Heavy CPU Workload Examples for Worker Processes

1. Prime Number Checker (Heavy Computation)

```
from multiprocessing import Pool
import math
import time

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

if __name__ == "__main__":
    numbers = list(range(10_000_000, 10_000_100))  # Check 100 large numbers
    start = time.time()

    with Pool(processes=4) as pool:
        results = pool.map(is_prime, numbers)

    end = time.time()
    print("Primes:", [n for n, is_p in zip(numbers, results) if is_p])
    print("Time taken:", end - start)
```

2. Monte Carlo Simulation to Estimate π

```
from concurrent.futures import ProcessPoolExecutor
import random
import time

def monte_carlo_pi_part(samples):
    count = 0
    for _ in range(samples):
        x, y = random.random(), random.random()
        if x*x + y*y <= 1:
            count += 1
    return count

if __name__ == "__main__":
    total_samples = 10_000_000
    workers = 4
    samples_per_worker = total_samples // workers
    start = time.time()

    with ProcessPoolExecutor(max_workers=workers) as executor:
        results = executor.map(monte_carlo_pi_part, [samples_per_worker]*workers)

    inside_circle = sum(results)
    pi_estimate = 4 * inside_circle / total_samples

    end = time.time()
    print(f"Estimated π: {pi_estimate}")
    print("Time taken:", end - start)
```

3. Matrix Multiplication Simulation

```
import numpy as np
from multiprocessing import Pool
import time

def multiply_row(row_data):
    row, matrix_b = row_data
    return np.dot(row, matrix_b)

if __name__ == "__main__":
    size = 500  # 500x500 matrices
    matrix_a = np.random.rand(size, size)
    matrix_b = np.random.rand(size, size)

    start = time.time()
    with Pool(processes=4) as pool:
        result = pool.map(multiply_row, [(row, matrix_b) for row in matrix_a])

    end = time.time()
    print("Matrix multiplication complete.")
    print("Time taken:", end - start)
```

4. Image Processing Simulation (e.g., filter large files)

```
from multiprocessing import Pool
import numpy as np
import time

def process_image(img):
    # Simulate heavy processing (e.g., blur, edge detection)
    return np.fft.fft2(img)

if __name__ == "__main__":
    images = [np.random.rand(1024, 1024) for _ in range(8)]  # 8 fake HD images

    start = time.time()
    with Pool(processes=4) as pool:
        results = pool.map(process_image, images)
    end = time.time()
    print("Processed 8 images.")
    print("Time taken:", end - start)
```

5. Inefficient Recursive Fibonacci (for benchmarking)

Warning: Intentionally inefficient to generate heavy CPU load.

```
from concurrent.futures import ProcessPoolExecutor
import time

def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    numbers = [35, 36, 37, 38]  # Takes time
    start = time.time()

    with ProcessPoolExecutor(max_workers=4) as executor:
        results = list(executor.map(fib, numbers))
    end = time.time()
    print("Fibonacci Results:", results)
    print("Time taken:", end - start)
```

Using Message Passing with Celery for Distributed Tasks

What is Celery?

Celery is a powerful, production-grade **distributed task queue** that lets you **run time-consuming or background tasks** asynchronously across one or many worker nodes.

It uses **message passing** between:

- A **producer** (your main application)
- A **broker** (like Redis or RabbitMQ)
- A **consumer** (Celery workers)

Why Message Passing?

In **distributed computing**, components often run on **separate machines**. They need to communicate, coordinate, and share work **without direct function calls**.

Instead, they **exchange messages** asynchronously using a **message broker**.

This enables:

- Loose coupling
- Fault tolerance
- Scalability
- Asynchronous background execution

System Architecture

```
[ Producer (FastAPI/Django/Script) ]
          |
    Task Function Call
          ▼
    [ Message Broker (Redis) ]
          ▼
[ Celery Worker(s) (Multiple Nodes) ]
          |
    Task Executed
```

```
          ▼
     [ Results Backend (Optional) ]
```

Basic Setup with Redis + Celery

1. Install Requirements

```
pip install celery redis
```

2. Define Celery App (celery_app.py)

```
from celery import Celery

# Create a Celery app with Redis as broker
app = Celery('tasks', broker='redis://localhost:6379/0')

# Optional: Store results
app.conf.result_backend = 'redis://localhost:6379/0'
```

3. Create a Task Module (tasks.py)

```
from celery_app import app
import time

@app.task
def long_running_task(name):
    time.sleep(5)
    return f"Task completed by {name}"
```

4. Start Redis Server (in separate terminal)

```
redis-server
```

5. Start Celery Worker

```
celery -A tasks worker --loglevel=info
```

6. Run a Task (Producer Script)

```
from tasks import long_running_task

# Sends a message to the broker
```

```
result = long_running_task.delay("Worker-1")

# Retrieve result asynchronously
print("Result (Async):", result.get(timeout=10))
```

Behind the Scenes: How Message Passing Works

1. delay() → Sends a **message** with task info to Redis.
2. Redis → Queues the task in a **FIFO structure**.
3. Celery Worker → Subscribes to Redis, picks up the task.
4. Task → Is executed in a **separate process/thread**.
5. Result → Sent back to Redis (if configured).

Example: Parallel Image Processing

```
# tasks.py
from celery_app import app
from PIL import Image, ImageFilter

@app.task
def apply_blur(image_path, output_path):
    img = Image.open(image_path)
    blurred = img.filter(ImageFilter.GaussianBlur(10))
    blurred.save(output_path)
    return f"Blurred image saved to {output_path}"
```

Usage:

```
from tasks import apply_blur

apply_blur.delay("input.jpg", "output.jpg")
```

Multiple images can be processed in **parallel across worker nodes**.

Example: Web Scraping Tasks via Flask + Celery

```
# tasks.py
import requests
from celery_app import app

@app.task
```

```
def fetch_url(url):
    r = requests.get(url)
    return len(r.text)
```

Flask Route to Trigger Scraping

```
from flask import Flask, request
from tasks import fetch_url

app = Flask(__name__)

@app.route('/scrape', methods=['POST'])
def scrape():
    url = request.json['url']
    result = fetch_url.delay(url)
    return {"task_id": result.id}, 202
```

Use /scrape to trigger background scraping from a UI or API.

Real-World Use Cases

Domain	**Use Case**
E-Commerce	Asynchronous email sending, invoice generation
Web Development	Background file uploads, thumbnail generation
AI/ML	Model training jobs dispatched via Celery
IoT Systems	Sensor data ingestion from multiple devices
Finance	Parallel risk computation, fraud analysis
Education	PDF generation, plagiarism checks asynchronously

Monitoring and Management

- Use **Flower**: A web-based Celery dashboard

```
pip install flower
celery -A tasks flower
```

Accessible at http://localhost:5555

Advanced Tips

- **Retrying failed tasks**:

```
@app.task(bind=True, max_retries=3)
```

```
def download_file(self, url):
    try:
        # Your download logic
        pass
    except Exception as e:
        raise self.retry(exc=e, countdown=5)
```

- **Routing tasks to specific queues**:

```
app.conf.task_routes = {
    'tasks.high_priority_task': {'queue': 'high'},
    'tasks.low_priority_task': {'queue': 'low'},
}
```

- **Task chaining / workflows**:

```
from celery import chain
chain(task1.s(), task2.s(), task3.s())()
```

Introduction to Dask for Scalable Analytics and Parallel Computing

What is Dask?

Dask is an **open-source, parallel and distributed computing framework for Python** that enables large-scale analytics, numerical computations, and machine learning to run efficiently on datasets that are too big to fit into memory or workloads that require faster execution.

Unlike many big-data frameworks (e.g., Apache Spark) that require learning new APIs or languages, Dask **mirrors familiar Python interfaces**:

- dask.array → works like NumPy arrays.
- dask.dataframe → works like pandas DataFrames.
- dask.bag → works like Python iterables.
- Integrates with **scikit-learn**, **XGBoost**, **PyTorch**, **TensorFlow**, and HPC tools like **mpi4py**.

Its core goal is **scaling Python code from one CPU to many CPUs/GPUs or an entire cluster** with minimal changes.

Why Dask is Needed in Distributed Computing

a) Problem Without Dask

- Pandas and NumPy **load entire data into RAM**, so a dataset larger than memory will crash or swap to disk (slow).

- Python's **GIL (Global Interpreter Lock)** limits multi-threaded CPU-bound performance.
- Processing terabyte-scale datasets on a single machine is **impractical**.

b) How Dask Solves This

- **Out-of-core processing**: Splits data into smaller partitions and processes sequentially or in parallel.
- **Task graph execution**: Breaks work into a **Directed Acyclic Graph (DAG)** of small tasks.
- **Multi-environment scalability**: Runs on:
  - A single laptop (multi-core parallelism).
  - Multi-core servers.
  - HPC clusters.
  - Cloud environments (AWS, GCP, Azure).

Core Components

Component	Purpose
Dask Arrays	Distributed NumPy-like arrays for large numerical computations.
Dask DataFrames	Distributed pandas-like DataFrames for tabular data analysis.
Dask Bags	Handles semi/unstructured data like JSON logs.
Dask Delayed	Build custom task graphs from Python functions.
Dask Distributed	Scheduler + workers for distributed cluster execution.

Dask in Distributed Computing Theory

Distributed Computing Term	Dask Equivalent
Node (Worker)	Dask Worker
Master Node	Dask Scheduler
Distributed Data Structure	Dask Array/DataFrame
Task Scheduling	Task Graph + Scheduler
Load Balancing	Scheduler assigns tasks evenly

Scheduler Types:

- **Single-threaded** (debugging)
- **Threaded** (GIL-releasing tasks, I/O-bound)
- **Multiprocessing** (CPU-bound tasks)

- **Distributed** (multiple machines, remote clusters)

Code Example 1: Local Parallel Computation

```
python
CopyEdit
import dask.array as da
# Create a 1 billion element array in chunks of 10 million
x = da.random.random(size=int(1e9), chunks=int(1e7))
# Parallel mean computation
mean_value = x.mean().compute()
print("Mean:", mean_value)
```

Explanation:

- **Chunking** divides the array into pieces.
- A **DAG** is built internally to compute the mean.
- Execution happens in **parallel across all CPU cores**.

Code Example 2: Distributed DataFrame Processing

```
from dask.distributed import Client
import dask.dataframe as dd
# Start local cluster
client = Client()
# Load large CSV in parallel
df = dd.read_csv("large_dataset.csv")
# Data transformation
df['total'] = df['quantity'] * df['price']
# Group and aggregate in parallel
result = df.groupby("category")["total"].sum().compute()
print(result)
```

Key Points:

- Client() manages the scheduler and workers.
- Data is loaded in **partitions**.
- Grouping and aggregation happen in **parallel**.

Code Example 3: Custom Task Graph

```
from dask import delayed

@delayed
def square(x):
    return x * x
@delayed
def add(a, b):
    return a + b
# Build graph
result = add(square(2), square(3))
# Trigger execution
print(result.compute())
```

Explanation:

- Each @delayed function call creates a **node** in the DAG.
- The scheduler decides **execution order** for efficiency.

Code Example 4: Integrating Dask with Scikit-learn

```
import dask.dataframe as dd
from dask_ml.model_selection import train_test_split
from dask_ml.linear_model import LogisticRegression
# Load dataset
df = dd.read_csv("big_dataset.csv")
# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(
    df.drop('target', axis=1), df['target'], test_size=0.2
)
# Train logistic regression in parallel
model = LogisticRegression()
model.fit(X_train, y_train)
```

```
print("Accuracy:", model.score(X_test, y_test))
```

Typical Dask Workflow

1. **Define Computation**: Create Dask objects or use delayed.
2. **Build Task Graph**: Automatically generated.
3. **Execute**: Scheduler coordinates worker processes.
4. **Aggregate Results**: Return to Python.

Real-World Use Cases

- **Data Science**: Large ETL pipelines.
- **ML Training**: Distributed algorithms.
- **Scientific Research**: Simulations.
- **Business Intelligence**: Real-time reporting.

Celery & concurrent.futures in Python Distributed Computing

Celery: Managing Task Queues and Execution Order

Celery is an **asynchronous, distributed task queue system** that lets you **run tasks in the background**, schedule jobs, and execute workloads across multiple machines or CPU cores.

It's widely used in **web applications, ETL pipelines, and real-time processing systems** to **offload heavy work** from the main application thread and handle **scalable, fault-tolerant distributed execution**.

Celery Architecture

Celery operates using the **Producer–Broker–Consumer** model:

- **Producer (Client)** → Sends tasks to a broker.
- **Broker** → Message broker that holds and distributes tasks (e.g., **RabbitMQ**, **Redis**).
- **Workers** → Processes that pick tasks from the broker and execute them.
- **Result Backend** (optional) → Stores task results (e.g., Redis, SQL, MongoDB).
- **Task Queue** → Logical queue holding pending tasks.

Workflow:

1. **Application (Producer)** sends tasks to the **Broker**.

2. **Broker** queues tasks.
3. **Workers** pull tasks and execute them.
4. **Results** (if needed) are sent to the **Result Backend**.

How Celery Ensures Task Execution Order

Celery by default does **not** guarantee strict **FIFO** across multiple workers, but you can control execution order via:

a) Task Priorities

- Assign higher priority to critical tasks (RabbitMQ supports priorities).

```
task.apply_async(priority=10)  # Higher priority
```

b) Task Routing & Multiple Queues

- Route tasks to specific queues to enforce order inside each queue.

```
CELERY_QUEUES = {
    "queue1": {"exchange": "tasks", "routing_key": "task.queue1"},
    "queue2": {"exchange": "tasks", "routing_key": "task.queue2"},
}
CELERY_ROUTES = {
    "app.task1": {"queue": "queue1"},
    "app.task2": {"queue": "queue2"}
}
```

c) Task Chaining (Dependencies)

- Ensure sequential execution:

```
from celery import chain
chain(task1.s(), task2.s(), task3.s())()
```

d) Groups & Chords

- **Groups**: Run tasks in parallel, collect results.
- **Chords**: Run a callback after all group tasks finish.

```
from celery import group, chord
chord((task1.s(), task2.s(), task3.s()), final_task.s())()
```

e) ETA & Countdown Scheduling

- Delay execution to enforce sequence:

```
task.apply_async(countdown=10)  # Run after 10 seconds
```

Celery Example: Distributed Task Execution

```
# tasks.py
from celery import Celery
import time
app = Celery('my_tasks', broker='redis://localhost:6379/0', backend='redis://localhost:6379/0')
@app.task
def process_data(x):
    time.sleep(2)
    return x * x
# producer.py
from tasks import process_data
for i in range(10):
    process_data.delay(i)
# Terminal 1 - Start Redis
redis-server
# Terminal 2 - Start Celery worker
celery -A tasks worker --loglevel=info
```

concurrent.futures: High-Level Concurrency in Python

The concurrent.futures module is a **high-level API for parallel execution**, supporting:

- **ThreadPoolExecutor** → For **I/O-bound** tasks.
- **ProcessPoolExecutor** → For **CPU-bound** tasks.

It uses the **Future** object to represent the result of an asynchronous computation.

ThreadPoolExecutor

- Best for **I/O-bound tasks** (e.g., web scraping, file I/O, network requests).
- Uses **threads** — shares memory but affected by the GIL.

Example:

```
from concurrent.futures import ThreadPoolExecutor
import time

def download_file(n):
    print(f"Downloading file {n}")
    time.sleep(2)
    return f"File {n} downloaded"

if __name__ == "__main__":
    with ThreadPoolExecutor(max_workers=3) as executor:
        futures = [executor.submit(download_file, i) for i in range(5)]
        for future in futures:
            print(future.result())
```

ProcessPoolExecutor

- Best for **CPU-bound tasks** (e.g., image processing, mathematical computations).
- Uses **processes** — bypasses GIL, separate memory space.

Example:

```
from concurrent.futures import ProcessPoolExecutor
import time

def compute_factorial(n):
    print(f"Computing factorial for {n}")
    time.sleep(2)
    fact = 1
    for i in range(1, n+1):
        fact *= i
    return fact

if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=3) as executor:
        futures = [executor.submit(compute_factorial, i) for i in range(5, 10)]
        for future in futures:
            print(future.result())
```

Key Differences

Feature	**ThreadPoolExecutor**	**ProcessPoolExecutor**
Best For	I/O-bound tasks	CPU-bound tasks
Uses	Threads	Processes
Memory Usage	Shared memory space	Separate memory
GIL Impact	Affected	Not affected
Startup Overhead	Low	Higher

Celery vs concurrent.futures in Distributed Computing

Feature	**Celery**	**concurrent.futures**
Scope	Distributed & Parallel	Local Parallelism
Broker Required	Yes	No
Task Scheduling	Advanced (queues, priorities)	Simple (FIFO)
Fault Tolerance	Yes	No
Best For	Large-scale, distributed tasks	Local CPU/I/O-bound tasks

Mini Project: Big Data Analysis with Dask (Parallel & Scalable)

📌 **Objective**

We will simulate an **E-commerce analytics system** where we:

1. Generate a **huge synthetic dataset** (100 million+ records).
2. Perform **parallel computations** for:
   - Mean sales
   - Maximum & minimum prices
   - Sales aggregation by category
   - Finding top-performing products
3. Visualize results using **Matplotlib**.
4. Optimize resource usage using **Dask task graph**.

Install Required Libraries

```
pip install dask[complete] matplotlib
```

Python Code Implementation

```python
import dask.array as da
import dask.dataframe as dd
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from dask.distributed import Client

# Step 1: Start a Local Dask Client for parallel processing
client = Client()
print(client)
# Step 2: Simulate large e-commerce dataset using pandas first
np.random.seed(42)
categories = ['Electronics', 'Clothing', 'Books', 'Toys', 'Home']
n_records = 50_000_000  # 50 million rows
pdf = pd.DataFrame({
    'product_id': np.arange(n_records),
    'category': np.random.choice(categories, n_records),
    'price': np.random.uniform(5, 500, n_records),
    'units_sold': np.random.randint(1, 50, n_records)
})
# Step 3: Convert pandas dataframe to Dask DataFrame for parallelism
df = dd.from_pandas(pdf, npartitions=50)  # Partition data for parallel execution
# Step 4: Compute total sales per row
df['total_sales'] = df['price'] * df['units_sold']
# Step 5: Parallel computations
mean_sales = df['total_sales'].mean().compute()
max_price = df['price'].max().compute()
min_price = df['price'].min().compute()
```

```
print(f"📊 Mean Sales: {mean_sales:,.2f}")
print(f"💰 Max Price: {max_price:,.2f}")
print(f"💲 Min Price: {min_price:,.2f}")
# Step 6: Aggregate total sales by category
sales_by_category = df.groupby('category')['total_sales'].sum().compute()
print("\nCategory-wise Total Sales:\n", sales_by_category)
# Step 7: Find top 5 products by sales
top_products = df.nlargest(5, 'total_sales')[['product_id', 'total_sales']].compute()
print("\n🏆 Top 5 Products:\n", top_products)
# Step 8: Visualization (Parallel results → plotted locally)
sales_by_category.plot(kind='bar', color='skyblue', title="Total Sales by Category")
plt.ylabel("Sales ($)")
plt.xlabel("Category")
plt.grid(axis='y')
plt.tight_layout()
plt.show()
# Step 9: Example of Dask Array computation for heavy math
x = da.random.random(size=(1_000_000_000,), chunks=(10_000_000,))
mean_value = x.mean().compute()
print(f"\n⚡ Mean of 1 Billion Random Numbers: {mean_value:.4f}")
# Step 10: Shut down client
client.close()
```

How This Uses Dask's Power

a) Chunking & Partitioning

- npartitions=50 ensures dataset is split for parallel execution.
- Dask processes only chunks in memory.

b) Out-of-Core Execution

- Handles **50 million+ rows** without memory overload.

c) Parallel Scheduling

- Client() uses all CPU cores for simultaneous processing.

d) Task Graph Optimization

- Dask builds an **execution graph** before running computations, removing redundant steps.

e) Integration with Existing Libraries

- Works seamlessly with **Pandas** & **NumPy** API.

Sample Output (May Vary)

```
📊 Mean Sales: 6,335.72

💰 Max Price: 499.99

💲 Min Price: 5.00

Category-wise Total Sales:

category

Books         3.15e+11

Clothing      3.13e+11

Electronics   3.12e+11

Home          3.14e+11

Toys          3.15e+11

Name: total_sales, dtype: float64

🏆 Top 5 Products:

   product_id  total_sales

0   34156312  23919.410775

1   20143523  23752.533768

2   42156191  23618.284516

3   11524689  23561.129239

4   38257149  23499.122835


⚡ Mean of 1 Billion Random Numbers: 0.5001
```

Possible Extensions

- Deploy this with **Dask on Kubernetes** or AWS/GCP.
- Add **real-time streaming data** with Kafka.
- Integrate with **Scikit-learn** for parallel ML model training.