SI 618 Day 12: Distributed Computing I

Introduction to concepts and dask

Outline

- Introduction to Distributed Computing
- Introduction to Python for Distributed Computing
- Getting Started with Dask
- Basic Dask Operations
- Distributed Computing with Dask: Hands-on Examples

Introduction to Distributed Computing

- Brief History of Distributed Computing
- Definitions and Fundamental Concepts
- Importance and Applications of Distributed Computing

Brief History of Distributed Computing

- development of networked computers in the 1970s made distributed computing a possibility.
- a way to improve performance and capabilities of large-scale computations.
- as processing power and network speeds increased, distributed computing became more prevalent and powerful.

Definitions and Fundamental Concepts

1. Distributed Systems:

- In distributed computing, a system consists of a collection of autonomous computers connected through a network and distribution middleware, which enables these computers to coordinate their activities and share resources/services of the system.

2. Parallel Processing:

- It is the simultaneous execution of multiple tasks required to solve a problem by a distributed system. It speeds up computation and is integral to the process of distributed processing.

3. Nodes:

- In the context of distributed computing, a node is a unit of the larger computing system, usually a computer or a processor, participating in the distributed task execution.

4. Concurrency:

- This refers to the execution of independent tasks in a distributed system at the same time. Concurrency helps optimize system resources and increase overall output.

5. Networking and Protocols:

- The backbone of distributed systems, networking allows for the interlinking of nodes. Protocols define a standardized format for data transfer, ensuring effective communication between nodes regardless of their underlying hardware or software variations.

Definitions and Fundamental Concepts (cont'd)

6. Fault Tolerance:

- Fault tolerance refers to the ability of a system to continue functioning correctly in the event of a partial system failure. In distributed computing, fault tolerance is of great importance to ensure continuity and reliability.

7. Scalability:

- Scalability is the capability of a distributed system to handle and adapt to an increasing amount of work. Scalability is achieved by adjusting the number of nodes/processes within the distributed system.

8. Latency and Bandwidth:

- Latency refers to the delays during information processing while bandwidth refers to the maximum data transfer rate of a network or internet connection. Optimization of both is crucial for efficient distributed computing.

9. Synchronization and Coordination:

- In a distributed computing environment, it's important to coordinate the processes running concurrently across different nodes. Effective synchronization ensures a certain order of execution to avoid discrepancies due to different process speeds, thus achieving the desired outcome.

10. Consistency and Replication:

- Replication of resources improves reliability and availability in distributed systems. Managing replicated data requires protocols to ensure consistency, meaning that all users see the same version of data, regardless of the node they access.

Importance of Distributed Computing

- **Scalability:** Distributed computing enables us to handle ever-increasing amounts of work by adding new machines or nodes to the system.
- **Redundancy & Reliability:** Distributed systems are typically more reliable as a failure in one component does not cause the entire system to fail, ensuring business continuity.
- **Resource Sharing:** Computers can share resources such as computing power, disk storage or data, making it more economical and beneficial for an organization.
- **Speed**: With distributed computing, tasks are divided and executed concurrently, leading to faster computation and efficient problem solving.
- **Lower Latency:** Distributed systems can serve user requests from geographically closer nodes, resulting in lower latency in delivering responses to user requests.

Applications of Distributed Computing

- **Internet Services:** With the distribution of servers, internet services like email, social networks, search engines, and databases are capable of handling enormous amounts of traffic quickly and reliably.
- **Telecommunications:** In distributed and cloud telecommunication systems, processing can take place closer to the edge of the network which reduces lag time and increases speed of services.
- **Banking & Financial Services:** Banks use distributed computing for their customer relationship management systems, transactions, processing, and for risk management analysis.
- **Scientific Research:** Distributed computing assists scientific research by enabling scientists to borrow and unite computing power to pursue research on climate study, environment modeling, protein folding study, exploring space, breaking encryption codes, etc.
- **Distributed Gaming:** Distributed systems enable massively multiplayer online games (MMOs) by allowing thousands of participants from around the world to interact in a virtual gaming environment.
- **Health Care:** In medicine, distributed computing is used to analyze complex data sets in genome mapping, cancer detection, patient diagnosis and treatment, medical imaging and more.
- **Business and Commerce:** Businesses leverage distributed computing systems to analyze customer behavior, optimize logistics, manage inventory, streamline supply chains, and execute other computation-heavy tasks.

Introduction to Python for Distributed Computing

- Python and Distributed Computing: Opportunities and Challenges
- Review of fundamental Python concepts for Distributed Computing

Python and Distributed Computing: Opportunities

Library Ecosystem: Python's extensive range of libraries like Dask, multiprocessing, PySpark, etc., allows high-level abstractions to perform distributed computing and parallel processing easily.

Ease of Use: Python's clean and simple syntax enables users to express distributed algorithms with fewer lines of code, allowing them to focus on the challenge at hand.

Versatility: Python can effortlessly handle every stage of a data pipeline, from gathering and cleaning data to visualizing results. This versatility extends to distributed computing for handling large-scale data processing tasks.

Widespread Acceptance: Python is widely accepted and used in data-intensive industries. The applicability of python in distributed computing further enhances its attractiveness.

Python and Distributed Computing: Challenges

1. **Global Interpreter Lock (GIL)**: Python wasn't initially designed for distributed computing. The Global Interpreter Lock (GIL) in CPython ensures only one thread is executed at a time. This presents a challenge in scenarios requiring multi-threading or high performance which are fundamental to distributed computing.
2. **Performance**: While Python provides greater simplicity and productivity, it may not match up to the computing speed of languages like C, C++, or Java, which might be more suitable for performance-intensive applications.
3. **Memory Management**: Python's built-in garbage collection is convenient for many applications, but for large, long-running distributed computing processes, it could create inefficiency through its memory consumption patterns.
4. **Lack of Strong Type Checking**: Python's dynamic typing is helpful for rapid application development, but in the context of distributed computing where data is shared across different nodes, it may lead to more runtime errors.

NOTE: it's crucial to appreciate that the 'right' language or tool depends on the specific requirements and constraints of the project. The key lies in using Python wisely and in line with its strengths while staying aware of the challenges.

Review of fundamental Python concepts for Distributed Computing

Python Libraries for Numerical Operations:

Python offers various libraries like Numpy and Pandas for statistical analysis and data manipulation. Familiarity with these libraries is essential, especially for handling array-based or table-like data.

Concurrency and Parallelism:

- Understanding the use of threads, processes, and coroutines which is important for distributed computing. Libraries such as concurrent.futures or multiprocessing provide tools to parallelize Python code.

Generators and Iterators:

- Essential to understand how to effectively use these tools to handle large volumes of data that may not fit into memory.

Decorators:

- Powerful tool that allows modification of the behavior of a function or class. Useful in various circumstances, such as adding a timing code, changing the input parameters, or logging.

Serialization and Deserialization (Pickle):

- To share data between different Python processes, one needs to encode it as a stream of bytes. This encoding process is known as serialization. When this data needs to be read back into a Python program, that's known as deserialization. Python's built-in library Pickle can be used for this.

Error Handling (try, except):

- Crucial for any coding exercise. The error handling mechanisms are similar in distributed computing but have additional complexities given various workers involved.

Class and Objects (OOP concept):

- Object-oriented programming (OOP) can be a useful way to write distributed programs in Python. Understanding how classes and objects work can aid in understanding interfaces to distributed systems more quickly.

Getting Started with Dask

- What is Dask?
- Installation and Setup

What is Dask?

Dask is a powerful open-source library in Python for parallel and distributed computing. It allows for dynamic task scheduling and parallel computations designed for big data processing. Dask is often a go-to tool for data scientists who need to crunch large data sets and can't do it efficiently with Pandas, Numpy, or Scikit-Learn due to memory constraints.

Features of Dask:

1. **Dynamic Task Scheduling:** Unlike traditional static scheduling systems, Dask can adapt to data that changes over time, handle complex data dependencies, and execute computations with less memory.
2. **Compatible with existing Python Libraries:** Dask integrates well with popular Python libraries like Pandas and NumPy, allowing you to work on larger-than-memory data sets using familiar APIs.
3. **Scalable:** It scales from single machines to thousand-node clusters seamlessly.
4. **Handling Big Data:** Dask has functionality to work with large datasets that can't fit into memory by breaking them into smaller, manageable parts.
5. **Lazy Evaluation:** A computation model where the execution is delayed until the result is required. This helps optimize the computation and save resources.
6. **Diverse Workloads:** Dask is capable of parallelizing different workloads like array computations, dataframe computations, machine learning, and even custom task scheduling algorithms.

Basic Dask Operations

- Dask Data Structures: Dask Array, Dask DataFrame, Dask Bag, Dask Delayed
- Understanding the Dask Compute graph and Scheduler
- Basic Operations: Map, Reduce, Filter, Aggregations etc.

Dask Data Structures: Dask Array, Dask DataFrame, Dask Bag, Dask Delayed

1. Dask Array:

- A Dask array is a large parallel array library that cooperates with NumPy arrays. It is composed of many smaller NumPy arrays, split along one or more dimensions. With it, you can perform large-scale computations on arrays that don't fit into memory.
- You can operate on a Dask array like you would with a NumPy array, but under-the-hood, computations are broken down into smaller pieces and executed either on disk or in parallel across a cluster, improving computational efficiency.

2. Dask DataFrame:

- Dask DataFrames are a large parallel DataFrame library that coordinate with Pandas DataFrames. It is split into several smaller pandas DataFrames across a defined index.
- It allows you to work with larger-than-memory datasets with familiar pandas operations, such as grouping, joining, and time series computations.

3. Dask Bag:

- Dask Bag is used to create a parallel computation on data not fitting into a DataFrame or an array structure. It is ideal for semi-structured datasets or datasets where we don't know the format beforehand.
- Dask Bag efficiently manipulates flexible, arbitrary-sized data with operations like map, filter, groupby and aggregations.

4. Dask Delayed:

- Dask Delayed is a way to parallelize your own custom code. It allows you to take existing Python code, parallelize it, and make it lazy (meaning the computations are not executed immediately). When compute() is called, the Dask scheduler executes the operations in parallel.
- This is useful for tasks that don’t fit into Dask's arrays or dataframes abstractions and require custom computations

Understanding the Dask Compute

- A Dask **compute graph** is a dictionary-like data structure that represents a sequence of computations where each node corresponds to an output, and each edge points at an operation producing that output. This non-cyclical, directed graph provides a high-level view of the computation being carried out, making it easier to conceptualize and debug.
- Key Components of a Compute Graph:
  - **Vertices/Nodes:** These represent the outputs. In Dask, these are generally arrays or numbers.
  - **Edges:** These signify operations - the computations that produce the outputs.
  - **Data Dependencies:** These are represented by directed edges, indicating the execution order.

Understanding the Dask Scheduler

- in Dask, a scheduler executes the compute graphs by taking a computation described by a graph and executing it on one or more cores or machines.
- dask has several schedulers, each suitable for different kinds of workloads:
  1. **Single-machine scheduler (threaded, multiprocessing):** Ideal for computations that are less complex and can be performed on a single machine. This scheduler provides low overhead and excellent memory management.
  2. **Distributed scheduler:** This scheduler is robust and can scale from a single machine to a cluster of several machines. It can handle sophisticated computations, provide advanced features like scaling policies, data locality, and others, making it suitable for handling larger, more complex workloads and computations.
- the selection of a scheduler depends on the scale of the problem and the nature of the computation. Regardless of the choice, the Dask scheduler will take the compute graph and execute the operations by coordinating available computational resources, carefully respecting the order of operations defined by the graph's edges.

Basic Operations: Map, Reduce, Filter, Aggregations etc.

1. Map:

- The Map operation in Dask applies a function to every element in a data collection (like Dask bag or Dask array) in parallel. This operation is handy when you need to perform a specific operation on each element of a collection independently of every other element.

2. Reduce:

- The Reduce operation applies a function consecutively to the elements of a collection (such as a Dask bag or Dask array) so as to combine them into a single output. In the context of distributed computing, reduce is typically used after a map operation to consolidate results.

3. Filter:

- The Filter function in Dask filters the elements of the data collection based on some condition. It creates a new dataset from the elements of an existing collection that satisfy a given condition.

4. Aggregations:

- Dask supports various aggregation functions like sum(), mean(), min(), max() etc. on its data structures. These functions reduce a data collection to a single value by iteratively applying a binary operation. For example, computing the sum of all elements in a Dask array.

Distributed Computing with Dask: Hands-on Examples

[see today's notebook]

Q & A and Wrap-Up