Agenda

5	9-10	**Distributed Computing - Design Strategy: Divide-and-conquer for Parallel / Distributed Systems - Basic scenarios and Implications.** **Programming Patterns: Data-parallel programs and map as a construct; Tree-parallelism, and reduce as a construct; Map-reduce model: Examples (of map, reduce, map-reduce combinations, and Iterative map-reduce)**	AR

Contents

Distributed Computing Design Strategy: Divide-and-Conquer

Overview

- Divide-and-conquer is a foundational design strategy in distributed and parallel computing.
- It involves breaking a problem into smaller, independent subproblems, solving each subproblem concurrently, and then combining the results to form the final solution.
- This strategy leverages parallelism and distribution to achieve scalability, fault tolerance, and improved performance.

Basic Scenarios of Divide-and-Conquer

- **Sorting Algorithms**:
  - **Merge Sort**: Divides the array into two halves, recursively sorts each half, and merges the sorted halves.
  - **Quick Sort**: Selects a pivot element, partitions the array around the pivot, and recursively sorts the partitions.
- **Matrix Multiplication**:
  - **Strassen's Algorithm**: Reduces the complexity of matrix multiplication by dividing matrices into smaller submatrices and combining their products.
- **Searching Algorithms**:
  - **Binary Search**: Divides a sorted array into halves to locate a target value efficiently.
- **Computational Geometry**:
  - Finding the closest pair of points by recursively dividing the set of points into smaller subsets.

Key Scenarios

1. Data Processing

Scenario:

Massive datasets need to be processed, such as in big data analytics or real-time streaming systems.

Approach:

Partition the data into smaller chunks.

Process each chunk independently on different nodes.

Aggregate the results (e.g., Hadoop MapReduce or Apache Spark).

Implications:

Scalability: Adding more nodes increases processing capacity.

Fault Tolerance: Failed nodes can be retried or excluded without halting the entire process.

Data Skew: Imbalanced partitions can lead to bottlenecks.

2. Task Execution

Scenario:

Complex computations with multiple tasks, such as simulations, matrix multiplications, or distributed algorithms.

Approach:

Decompose the task into smaller subtasks.

Assign subtasks to different processors or nodes.

Combine intermediate results (e.g., FFT, matrix operations).

Implications:

Load Balancing: Effective scheduling is critical to minimize idle time.

Communication Overhead: Excessive data exchange can degrade performance.

Dependency Management: Tasks must account for dependencies to avoid deadlocks.

3. Recursive Problem Solving

Scenario:

Problems inherently recursive in nature, such as sorting (e.g., merge sort), search trees, or divide-and-conquer algorithms.

Approach:

Recursively split the problem into subproblems.

Solve subproblems concurrently.

Merge results to obtain the final output.

Implications:

Parallelism: Recursive splitting allows maximum utilization of available resources.

Stack Management: Deep recursion might cause stack overflow or excessive memory usage.

Termination Condition: Incorrect termination logic can lead to infinite recursion.

Design Considerations

1.Granularity:

Fine-grained decomposition maximizes parallelism but increases overhead.

Coarse-grained decomposition reduces overhead but may underutilize resources.

2.Communication vs. Computation:

Optimize for low communication costs relative to computation time.

3.Fault Tolerance:

Use checkpointing and redundancy to handle node failures.

- **Scalability**:

Ensure the strategy works efficiently as the number of nodes increases.

- **Data Dependency**:

Identify and minimize dependencies to maximize concurrency.

Implications for Distributed Systems

Performance Optimization: Divide-and-conquer strategies enable efficient use of hardware, reducing execution time.

Resource Utilization: Balanced partitioning ensures all nodes contribute equally.

Adaptability: These strategies are adaptable across various frameworks (e.g., Hadoop, MPI, Spark).

Resilience: Fault-tolerant designs allow systems to recover from partial failures without significant performance degradation.

Examples

Example 1: MapReduce (Big Data)

Divide: Split input data into key-value pairs.

Conquer: Process pairs independently (map phase).

Combine: Aggregate results (reduce phase).

Example 2: Parallel Matrix Multiplication

Divide: Partition matrices into submatrices.

Conquer: Compute submatrix products in parallel.

Combine: Assemble the resulting matrix from sub-products.

Challenges

Data Skew: Uneven distribution of work can lead to inefficiencies.

Network Overhead: High inter-node communication can offset gains.

Algorithmic Complexity: Designing efficient combine steps can be challenging.

Conclusion

- Divide-and-conquer is a versatile strategy that empowers distributed and parallel systems to tackle large-scale problems efficiently.
- While it introduces challenges like load balancing and communication overhead, thoughtful design and implementation ensure robust and scalable solutions.

Questions & Discussion Points:

What strategies can mitigate data skew in divide-and-conquer?

How does fault tolerance influence the choice of distributed frameworks?

Programming Patterns in Parallel Computing

Programming Patterns: Data-parallel programs and *map* as a construct;

Tree-parallelism, and *reduce* as a construct;

Map-reduce model: Examples (of map, reduce, map-reduce combinations, and

Iterative map-reduce)

1. Data-Parallel Programs and Map as a Construct

Definition:

Data-parallel programs involve performing the same operation concurrently on elements of a dataset. The map construct is central to data parallelism, allowing operations to apply independently to each data element.

Characteristics:

Input: A collection of data elements.

Operation: A function applied independently to each element.

Output: A transformed collection of elements.

Example:

Input Dataset: [2, 4, 6, 8]

Operation: Square each number.

Code (Python-like):

```
def square(x):
    return x * x


result = map(square, [2, 4, 6, 8])
print(list(result))  # Output: [4, 16, 36, 64]
```

Applications:

Image processing (e.g., applying filters).

Financial computations (e.g., portfolio analysis).

Scientific simulations (e.g., grid-based data).

2. Tree-Parallelism and Reduce as a Construct

Definition:

Tree-parallelism involves aggregating data hierarchically, where intermediate results are combined in a tree-like structure. The reduce construct is key, performing a reduction operation (e.g., summation, maximum) on a dataset.

Characteristics:

Input: A collection of elements.

Operation: A binary function that reduces two elements to one.

Output: A single aggregated result.

Example:

Input Dataset: [1, 2, 3, 4]

Operation: Sum all elements.

Code (Python-like):

```
from functools import reduce


def add(x, y):
    return x + y
result = reduce(add, [1, 2, 3, 4])
print(result)  # Output: 10
```

Applications:

Summarizing data (e.g., computing totals, averages).

Finding extrema (e.g., maximum or minimum).

Aggregating logs or distributed metrics.

3. Map-Reduce Model

Definition:

Map-reduce combines map and reduce constructs to enable distributed processing of large datasets by mapping computations to independent tasks and reducing intermediate results to final outputs.

Workflow:

1. **Map**: Transform and distribute data.
2. **Shuffle**: Organize intermediate results (group by keys).
3. **Reduce**: Aggregate results by keys.

Examples of Map-Reduce Combinations

Example 1: Word Count

Problem: Count occurrences of each word in a text.

Steps:

- **Map**: Emit key-value pairs for each word (key = word, value = 1).
- **Reduce**: Sum values for each word.

Code:

```
# Map
words = ["hello", "world", "hello"]
mapped = [(word, 1) for word in words]
# Reduce
from collections import defaultdict
counts = defaultdict(int)
for word, count in mapped:
    counts[word] += count
print(counts)  # Output: {'hello': 2, 'world': 1}
```

Example 2: Distributed Matrix Multiplication

Problem: Multiply two matrices A and B.

Steps:

- **Map**: Compute partial products for each matrix entry.
- **Reduce**: Sum products to calculate final entries.

Iterative Map-Reduce

Some problems require iterative refinement, where the output of one map-reduce step becomes the input for the next.

Example: PageRank Algorithm

Problem: Rank webpages based on importance.

Steps:

1. **Initialize**: Assign equal rank to all pages.
2. **Map**: Distribute rank contributions to linked pages.
3. **Reduce**: Aggregate contributions to compute new ranks.

Repeat: Iterate until ranks converge.

Advantages of Map-Reduce Model

Scalability: Processes large datasets across many nodes.

Fault Tolerance: Handles node failures with task retries.

Abstraction: Simplifies parallel programming by hiding low-level details.

Conclusion

The programming patterns of data-parallelism, tree-parallelism, and the map-reduce model form the backbone of many distributed and parallel computing applications.

These patterns not only enable scalable and efficient solutions but also simplify the design of complex computational workflows.

Discussion Points:

1. What are the trade-offs between map-reduce and other parallel programming models?

2. How can iterative map-reduce be optimized for convergence speed?

Summary of Key Constructs

Construct	Description	Example Usage
Map	Applies a function to each element independently	Squaring elements in an array
Reduce	Combines results from multiple computations	Summing values from an array
Map-Reduce	Combines map and reduce for processing large datasets	Counting word frequencies across documents
Iterative Map-Reduce	Repeatedly applies map-reduce for refinement	Calculating PageRank through multiple iterations

IMP Note to Self