CCA3012: Parallel and Distributed Algorithms

Assignment-2

NAME = Prajal Pareta

# REG NO = 22BSA10239

**Q1Explain the divide-and-conquer strategy used in parallel matrix multiplication.**

ANS The divide-and-conquer strategy in parallel matrix multiplication involves breaking the matrix multiplication task into smaller subproblems that can be solved concurrently:

1. Matrix Partitioning: Divide matrices A and B into four submatrices each (e.g., A11,A12,A21,A22).
2. Recursive Subproblems: Compute submatrices of the result C (e.g., C11=A11B11+A12B21) independently.
3. Parallel Execution: Perform the eight smaller multiplications and subsequent additions concurrently across multiple processors.
4. Combine Results: Assemble the final matrix C from the computed submatrices.

This approach is efficient, scalable, and well-suited for parallel execution, with optimizations like Strassen’s algorithm further reducing computational complexity. However, challenges include communication overhead and load balancing among processors.

#### Q2) =Define parallel algorithms for sorting. Apply hyper quick sort algorithm to sort following

#### elements: 2, 1, 9, 5, 3, 8, 4

ANS Parallel algorithms for sorting are algorithms designed to arrange elements of a list in a specified order (e.g., ascending or descending) by dividing the task into smaller subtasks that can be executed simultaneously across multiple processors. These algorithms exploit concurrency to achieve faster execution times compared to sequential sorting.

Key Features of Parallel Sorting Algorithms:

1. Concurrency: Multiple parts of the list are processed at the same time.
2. Divide-and-Conquer: The list is often divided into smaller segments, sorted independently, and then merged.
3. Interprocessor Communication: Sorting requires exchanging data between processors to ensure global order.

3. Explain Euler tour technique for parallel computation. Construct a Euler tour for the

given tree:

Ans: The Euler Tour Technique is a method used in parallel computation for processing tree structures efficiently. It transforms tree problems into problems over arrays or sequences, enabling the application of parallel algorithms. This technique is particularly useful for solving problems like parenthesis matching, tree traversal, and finding properties such as depth or size of subtrees.

#### *4. Compare DFS and BFS with suitable Example.*

Ans: Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental graph traversal algorithms. They differ in their approach to exploring nodes in a graph. Here's a comparison with an example:

Depth-First Search (DFS)

* Approach: Explores as far as possible along each branch before backtracking.
* Data Structure: Stack (can be implemented using recursion or an explicit stack).
* Traversal Order: Goes deep into one path before exploring others.
* Use Cases: Finding connected components, topological sorting, and solving puzzles like mazes.
* Time Complexity: O(V+E), where V is vertices and E is edges.
* Space Complexity: O(V) (due to the recursion stack).

Breadth-First Search (BFS)

* Approach: Explores all neighbors at the current depth level before moving to the next depth.
* Data Structure: Queue.
* Traversal Order: Processes nodes layer by layer.
* Use Cases: Shortest path in unweighted graphs, peer-to-peer networks, and level-order tree traversal.
* Time Complexity: O(V+E).
* Space Complexity: O(V) (due to the queue).

Example

Graph:

1

/ \

2 3

/ \ \

4 5 6

1. DFS Traversal (Starting from Node 1):

Using the recursive approach, the traversal proceeds as:

1 → 2 → 4 → (backtrack) → 5 → (backtrack) → 3 → 6

Order: 1, 2, 4, 5, 3, 6

2. BFS Traversal (Starting from Node 1):

Using a queue, the traversal proceeds as:

1 → 2 → 3 → 4 → 5 → 6

Order: 1, 2, 3, 4, 5, 6

Comparison Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Feature | DFS | BFS |  |  |  |
| Exploration | Depth-first (go deep). | Breadth-first (go wide). |  |  |  |
| Data Structure | Stack (recursion/explicit). | Queue. |  |  |  |
| Pathfinding | Finds a path, not shortest. | Finds the shortest path in unweighted graphs. |  |  |  |
| Memory Usage | Less (stack depth). | More (queue size grows with breadth). |  |  |  |
| Example Traversal | 1, 2, 4, 5, 3, 6. | 1, 2, 3, 4, 5, 6. |  |  |  |

Summary:

* Use DFS when you want to explore a path deeply (e.g., puzzles or searching for connected components).
* Use BFS when you need the shortest path in unweighted graphs or level-wise exploration.

#### 5. Explain the concept of arithmetic pipelining and its significance in parallel computing.

#### Design and implement a pipelined architecture for floating-point addition and

#### multiplication operations.

ANS: Concept of Arithmetic Pipelining

Arithmetic pipelining is a technique in parallel computing where arithmetic operations are divided into smaller, sequential stages, enabling multiple operations to be processed simultaneously. Each stage performs a specific sub-operation, and as one operation progresses through the pipeline, new operations can enter, improving throughput.

Significance in Parallel Computing

1. Increased Throughput: Pipelines allow multiple operations to overlap, leading to higher computational efficiency.
2. Reduced Latency for Bulk Operations: Once the pipeline is filled, results are produced at regular intervals, minimizing idle time.
3. Scalability: Pipeline stages can be optimized or parallelized further, enhancing performance in systems like GPUs or vector processors.

Design of a Pipelined Architecture for Floating-Point Addition and Multiplication

Floating-point operations involve complex steps such as alignment, normalization, and rounding. A pipelined architecture breaks these into distinct stages to enable parallel execution.

Pipeline Stages

1. Floating-Point Addition:
   * Stage 1: Exponent Difference Calculation  
     Align the smaller number to the larger exponent by adjusting the mantissa.
   * Stage 2: Mantissa Addition  
     Add the aligned mantissas.
   * Stage 3: Normalization  
     Normalize the result to ensure the mantissa is within the valid range.
   * Stage 4: Rounding and Formatting  
     Round the result and adjust to conform to the floating-point format.
2. Floating-Point Multiplication:
   * Stage 1: Exponent Addition  
     Add the exponents of the two numbers, subtracting the bias.
   * Stage 2: Mantissa Multiplication  
     Multiply the mantissas.
   * Stage 3: Normalization  
     Normalize the result.
   * Stage 4: Rounding and Formatting  
     Round and adjust the final result.

Implementation Example

Floating-Point Pipeline Simulation

Below is a simple implementation in Python to simulate pipelined floating-point addition and multiplication.

Results

1. Floating-Point Addition:
   * Result: Mantissa = 1.25, Exponent = 128  
     This corresponds to the floating-point representation of 2.52.5.
2. Floating-Point Multiplication:
   * Result: Mantissa = 1.5, Exponent = 127  
     This corresponds to the floating-point representation of 1.51.5.

Explanation of Results

* Addition: The mantissas were aligned by adjusting the smaller number's exponent. After addition, normalization increased the exponent of the result.
* Multiplication: Exponents were combined, and the product of mantissas was normalized to ensure the result fits the floating-point format.

This pipelined architecture demonstrates how arithmetic operations can be split into stages to enable parallelism and efficiency in computation.