1. Simple Multithreaded Matrix Multiplication Algorithm

Algorithm Concept: In a conventional matrix multiplication of two $n \times n$ matrices A and B to produce C, we use three nested loops. To multithread this, we can parallelize the outer loops. Since the calculation of each element $C[i][j]$ is independent of the others, we can compute them in parallel.

Algorithm (Using `parallel for` keyword): The `parallel for` keyword indicates that the iterations of the loop can be executed concurrently by multiple threads.

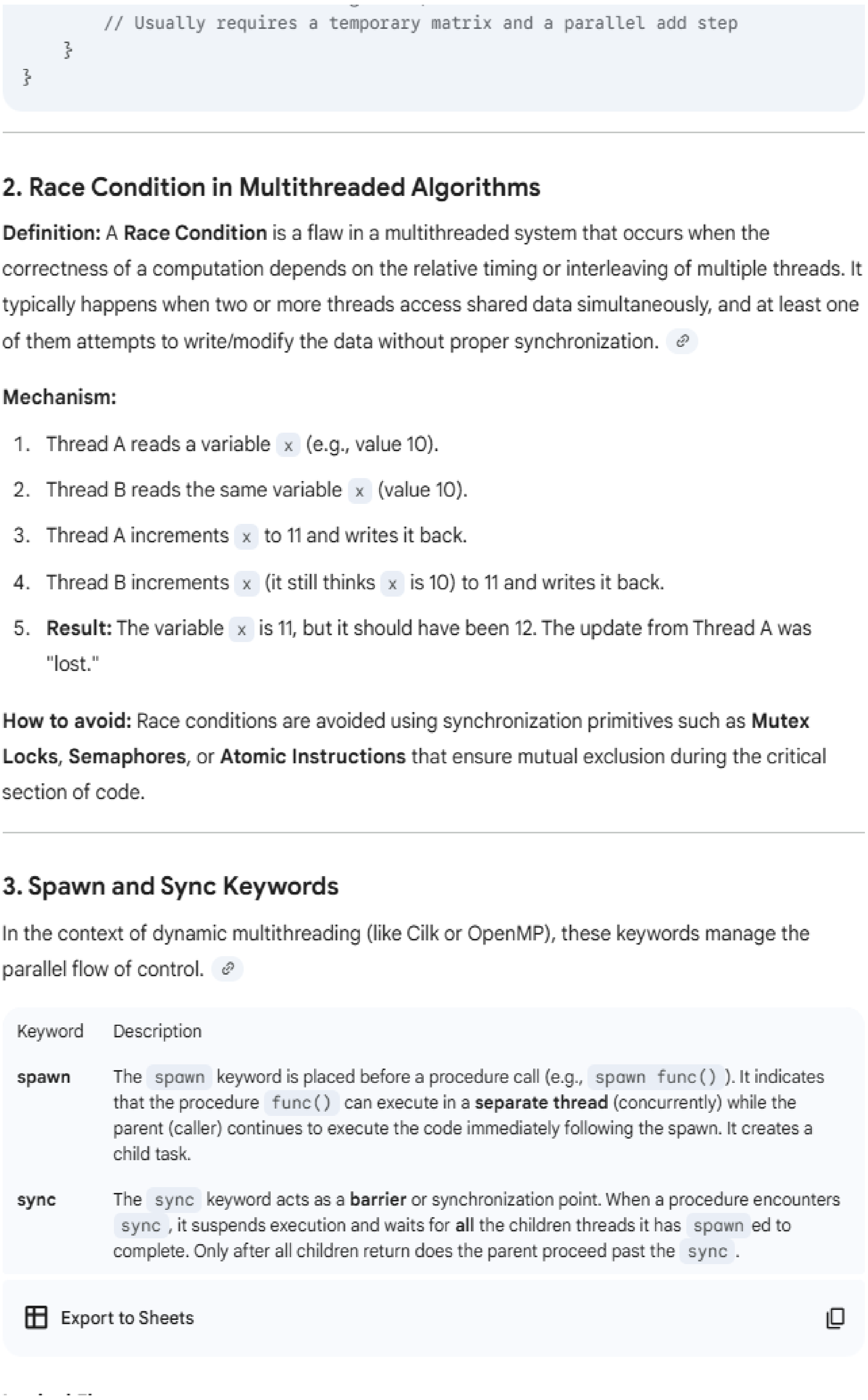
C++

```
Algorithm Parallel-Matrix-Multiply(A, B, C, n)
{
    parallel for i = 1 to n
    {
        parallel for j = 1 to n
        {
            C[i][j] = 0;
            for k = 1 to n
            {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
}
```

Algorithm (Divide and Conquer with `spawn`): A more robust theoretical approach (used in textbooks like CLRS) uses recursive divide-and-conquer.

C++

```
Algorithm P-Mat-Multiply-Recursive(C, A, B, n)
{
    if n == 1
        C[1][1] = C[1][1] + A[1][1] * B[1][1]
    else
    {
        partition A, B, and C into n/2 x n/2 submatrices
        spawn P-Mat-Multiply-Recursive(C11, A11, B11, n/2)
        spawn P-Mat-Multiply-Recursive(C12, A11, B12, n/2)
        spawn P-Mat-Multiply-Recursive(C21, A21, B11, n/2)
        spawn P-Mat-Multiply-Recursive(C22, A21, B12, n/2)
        sync // Wait for first set of multiplications

        // ... (Perform remaining multiplications and additions)
```



```
        // Usually requires a temporary matrix and a parallel add step
    }
}
```

2. Race Condition in Multithreaded Algorithms

Definition: A **Race Condition** is a flaw in a multithreaded system that occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads. It typically happens when two or more threads access shared data simultaneously, and at least one of them attempts to write/modify the data without proper synchronization.

Mechanism:

1. Thread A reads a variable `x` (e.g., value 10).
2. Thread B reads the same variable `x` (value 10).
3. Thread A increments `x` to 11 and writes it back.
4. Thread B increments `x` (it still thinks `x` is 10) to 11 and writes it back.
5. **Result:** The variable `x` is 11, but it should have been 12. The update from Thread A was "lost."

How to avoid: Race conditions are avoided using synchronization primitives such as **Mutex Locks**, **Semaphores**, or **Atomic Instructions** that ensure mutual exclusion during the critical section of code.

3. Spawn and Sync Keywords

In the context of dynamic multithreading (like Cilk or OpenMP), these keywords manage the parallel flow of control.

Keyword	Description
spawn	The `spawn` keyword is placed before a procedure call (e.g., `spawn func()`). It indicates that the procedure `func()` can execute in a **separate thread** (concurrently) while the parent (caller) continues to execute the code immediately following the spawn. It creates a child task.
sync	The `sync` keyword acts as a **barrier** or synchronization point. When a procedure encounters `sync`, it suspends execution and waits for **all** the children threads it has `spawn`ed to complete. Only after all children return does the parent proceed past the `sync`.

Logical Flow:

Plaintext

```
      Parent
        |
      spawn Child ----------------> Child runs
        |                              |
   Parent continues                    |
        |                              |
       sync  <-------------------- Child finishes
        |
   Parent resumes
```

---

4. Distributed/Parallel Breadth-First Search (BFS)

Algorithm: In a parallel or distributed BFS, we process nodes layer by layer. All nodes at distance d (Layer L_d) are processed in parallel to discover nodes at distance $d + 1$.

C++

```
Algorithm Parallel-BFS(G, s)
{
    L[0] = {s}; // Layer 0 contains only source
    d = 0;
    while (L[d] is not empty)
    {
        L[d+1] = empty set;

        parallel for each vertex u in L[d]
        {
            parallel for each neighbor v of u
            {
                // Attempt to add v to next layer atomically
                if (v is unvisited)
                {
                    mark v as visited;
                    add v to L[d+1];
                }
            }
        }
        d = d + 1;
    }
}
```

Advantage over Conventional Approach:

1. **Speedup:** On large graphs with high branching factors, processing a layer in parallel significantly reduces the wall-clock time compared to sequential processing.
2. **Diameter-based Complexity:** The time complexity depends more on the **diameter** of the graph (number of layers) rather than the total number of vertices (V) or edges (E), making it faster for shallow, dense graphs.

---

5. Rabin-Karp String Matching Algorithm

Concept: The Rabin-Karp algorithm uses **hashing** to find any one of a set of pattern strings in a text. For a single pattern, it calculates the hash value of the pattern and then rolls a window through the text, calculating the hash of the current window. If the hashes match, it performs a character-by-character check.

Algorithm: Input:

- Text T of length n.
- Pattern P of length m.
- Radix d (size of alphabet, e.g., 256).
- Prime q (to perform modulo arithmetic).

C++

```
Rabin-Karp-Matcher(T, P, d, q)
{
    n = length(T);
    m = length(P);
    h = d^(m-1) mod q;
    p = 0; // Hash value for pattern
    t = 0; // Hash value for text window

    // Preprocessing: Calculate hash of P and first window of T
    for i = 1 to m
    {
        p = (d * p + P[i]) mod q;
        t = (d * t + T[i]) mod q;
    }

    // Matching
    for s = 0 to n - m
    {
        if (p == t) // Candidate match
        {
            if (P[1..m] == T[s+1..s+m])
                print "Pattern found at shift" s;
        }
        // Calculate hash for next window using Rolling Hash
        if (s < n - m)
        {
            t = (d * (t - T[s+1]*h) + T[s+m+1]) mod q;
            if (t < 0) t = t + q;
        }
    }
}
```

Runtime Analysis:

- **Preprocessing:** $O(m)$.
- **Expected Runtime:** $O(n + m)$.
  - Ideally, the hash values are unique enough that we rarely have "spurious hits" (hash match but string mismatch). The loop runs n times, and the inner check rarely runs.
- **Worst-Case Runtime:** $O(nm)$.
  - This occurs if there are many spurious hits (e.g., $P = a^m$ and $T = a^n$, or a bad hash function where every window hashes to the same value). The verification loop runs for every shift.

---

6. Performance Measures of Multithreaded Algorithms

These metrics quantify how well a parallel algorithm utilizes resources.

1. **Work (T_1):** The total time taken to execute the entire computation on a single processor.
2. **Span (T_∞):** The longest time to execute any path of dependencies (critical path) in the computation graph. It represents the best possible time on infinite processors.
3. **Speedup:** The ratio of sequential execution time to parallel execution time with P processors.

$$\text{Speedup} = \frac{T_1}{T_P}$$

   - **Linear Speedup:** If speedup $\approx P$.
4. **Efficiency:** The speedup per processor. It measures how busy the processors are kept.

$$\text{Efficiency} = \frac{\text{Speedup}}{P} = \frac{T_1}{P \cdot T_P}$$

5. **Throughput:** The rate at which the system processes tasks (e.g., tasks per second). In matrix ops, it might be FLOPS (Floating Point Operations Per Second).

6. **Contention:** Occurs when multiple threads try to access the same shared resource (memory, lock, bus) simultaneously. High contention degrades performance due to waiting times.
7. **Latency:** The time elapsed between the initiation of a request/task and its completion. In multithreading, synchronization adds latency.

---

7. Stepwise Distributed BFS Trace

Given Adjacency Matrix:

- **A** is connected to: B, C
- **B** is connected to: A, D, E
- **C** is connected to: A, F, G
- **D, E** connected to: B
- **F, G** connected to: C

Graph Structure (Visual): This forms a tree structure rooted at A:

- Level 0: A
- Level 1: B, C
- Level 2: D, E, F, G

Process (Layer-by-Layer):

- **Initialization:**
  - `Visited` array: all 0.
  - Current Frontier $L_0 = \{A\}$.
  - Next Frontier $L_{next} = \{\}$.
- **Step 1 (Process Layer 0):**
  - **Thread 1** picks Node **A**.
  - Marks A visited.
  - Checks neighbors of A: **B, C.**
  - Both B and C are unvisited.
  - Add B and C to L_{next}.
  - $L_1 = \{B, C\}$.
- **Step 2 (Process Layer 1 - Parallel):**

  - The system spawns threads for nodes in L_1.
  - **Thread 1 (Node B):** Checks neighbors (A, D, E). A is visited. D, E are unvisited. Adds D, E to L_{next}.
  - **Thread 2 (Node C):** Checks neighbors (A, F, G). A is visited. F, G are unvisited. Adds F, G to L_{next}.
  - (Note: These threads run simultaneously).
  - $L_2 = \{D, E, F, G\}$.
- **Step 3 (Process Layer 2 - Parallel):**
  - Threads process D, E, F, G.
  - **Node D:** Neighbor B (Visited).
  - **Node E:** Neighbor B (Visited).
  - **Node F:** Neighbor C (Visited).
  - **Node G:** Neighbor C (Visited).
  - No new nodes added. L_{next} is empty.
- **Termination:** Frontier is empty. Algorithm stops.

---

8. Matrix Multiplication Time Complexity Comparison

Scenario: Multiplying Matrix X $(m \times n)$ and Matrix Y $(n \times p)$.

1. Conventional Approach (Sequential):

- Three nested loops:
  1. Outer loop runs m times (rows of X).
  2. Middle loop runs p times (cols of Y).
  3. Inner loop runs n times (dot product calculation).
- **Time Complexity:** $O(m \cdot n \cdot p)$.
  - If matrices are square $(n \times n)$, it is $O(n^3)$.

2. Multithreaded Approach (Parallel):

- **Work (T_1):** The total number of operations remains the same. $T_1 = O(m \cdot n \cdot p)$.
- **Span (T_∞):** The critical path.
  - If we parallelize the two outer loops (m and p), the span is dominated by the inner loop (summation) and the overhead of spawning.

- Using a parallel reduction (tree summation) for the inner dot product (size n), the span for one element is $O(\log n)$.
- Since all elements ($m \times p$) are computed in parallel, the total span is: $T_\infty = O(\log n)$ (assuming highly parallel recursion) or $O(n)$ (if inner loop is sequential).
- Usually, for Divide & Conquer Matrix Multiplication: $T_\infty = O(\log^2 n)$.

Discussion: The multithreaded approach does not reduce the *total work* (CPU cycles) required; in fact, it adds slight overhead for thread management. However, it drastically reduces the **wall-clock time** (Span) by utilizing multiple cores simultaneously.

---

9. Multithreaded Merge Sort Algorithm

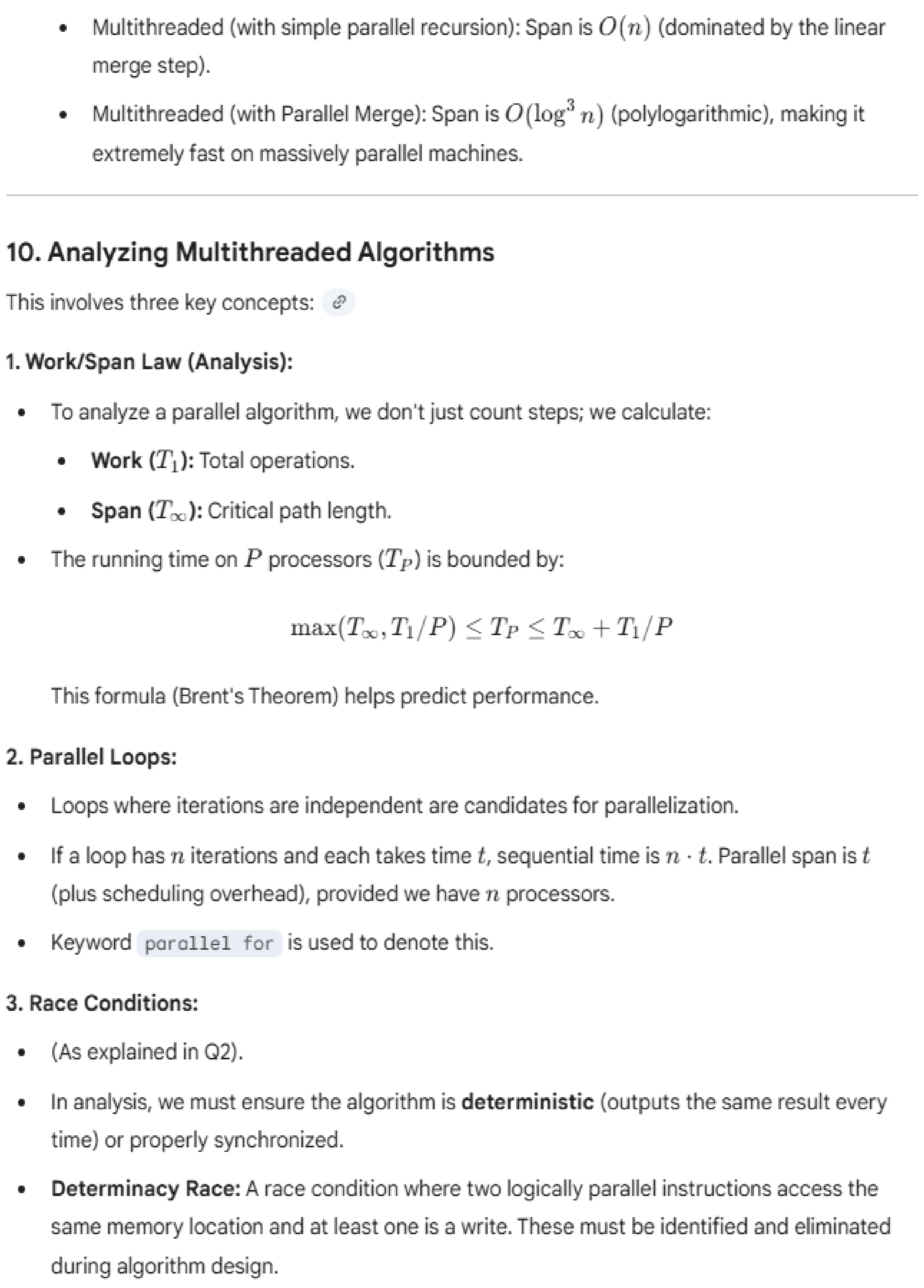
Algorithm: Merge Sort is naturally suited for parallelism because the two recursive calls operate on disjoint parts of the array.

C++

```
Algorithm Parallel-Merge-Sort(A, p, r)
{
    if p < r
    {
        q = (p + r) / 2
        // Spawn the first recursive call in a new thread
        spawn Parallel-Merge-Sort(A, p, q)

        // Call the second recursive call in the current thread (or spawn it too)
        Parallel-Merge-Sort(A, q + 1, r)

        // Wait for the spawned thread to finish
        sync

        // Merge the results (Can also be parallelized)
        Parallel-Merge(A, p, q, r)
    }
}
```

Difference from Conventional Merge Sort:

1. **Execution Flow:** Conventional Merge Sort performs the left sort completely, returns, and *then* performs the right sort. Multithreaded Merge Sort executes the left and right sorts **simultaneously**.
2. **Span:**
   - Conventional: $T(n) = O(n \log n)$.



- Multithreaded (with simple parallel recursion): Span is $O(n)$ (dominated by the linear merge step).
- Multithreaded (with Parallel Merge): Span is $O(\log^3 n)$ (polylogarithmic), making it extremely fast on massively parallel machines.

---

10. Analyzing Multithreaded Algorithms

This involves three key concepts:

1. Work/Span Law (Analysis):

- To analyze a parallel algorithm, we don't just count steps; we calculate:
  - **Work (T_1):** Total operations.
  - **Span (T_∞):** Critical path length.
- The running time on P processors (T_P) is bounded by:

$$\max(T_\infty, T_1/P) \leq T_P \leq T_\infty + T_1/P$$

  This formula (Brent's Theorem) helps predict performance.

2. Parallel Loops:

- Loops where iterations are independent are candidates for parallelization.
- If a loop has n iterations and each takes time t, sequential time is $n \cdot t$. Parallel span is t (plus scheduling overhead), provided we have n processors.
- Keyword `parallel for` is used to denote this.

3. Race Conditions:

- (As explained in Q2).
- In analysis, we must ensure the algorithm is **deterministic** (outputs the same result every time) or properly synchronized.
- **Determinacy Race:** A race condition where two logically parallel instructions access the same memory location and at least one is a write. These must be identified and eliminated during algorithm design.