

DAA UNIT – 6 (Multithreaded and Distributed Algorithms) – END-SEM PYQ Answers► **MAY / JUN 2022****Q7) a) Write short notes on the following. [10]****i) Multithreaded matrix multiplication****iii) Distributed breadth first search****ii) Multithreaded merge sort****iv) The Rabin-Karp algorithm****i) Multithreaded Matrix Multiplication**

Matrix multiplication of two matrices A ($n \times m$) and B ($m \times p$) produces matrix C ($n \times p$). In a multithreaded approach, the computation of each element or each row of C can be assigned to a different thread.

Threading strategies:

1. **Row-level parallelism:** each thread computes one full row of C.
2. **Element-level parallelism:** each element $C[i][j]$ computed by a separate thread.
3. **Block-level parallelism:** divide matrices into smaller blocks and assign block multiplications to threads.

Advantages:

- Takes advantage of multicore processors.
- Reduces computation time dramatically.

Disadvantages:

- Overhead of thread creation.
- False sharing and cache misses.

ii) Multithreaded Merge Sort

Merge Sort is naturally recursive. In multithreading:

- Each recursive call (sorting left half and right half) is assigned to a **separate thread**.
- Once both halves are sorted, they are merged.

Pseudo-idea:

Thread Mergesort(A, l, r):

if $l < r$:

mid = $(l+r)/2$

create thread T1 to Mergesort(A, l, mid)

create thread T2 to Mergesort(A, mid+1, r)

join(T1); join(T2)

merge(A, l, mid, r)

Provides parallelism for divide phase; merging becomes the bottleneck unless *parallel merging* is used.

iii) Distributed Breadth-First Search (Distributed BFS)

Used in large graphs spread across multiple machines.

Main Idea:

- Each machine stores a part of the graph (partition of vertices).
- BFS frontier (current level nodes) is distributed across machines.
- When exploring neighbors, machines exchange messages to notify other machines about newly discovered vertices.

Characteristics:

- Uses message passing (MPI, Hadoop Graph processing).
- Requires coordination and synchronization per BFS level.
- Suitable for very large real-world graphs (social networks, web graphs).

iv) The Rabin–Karp Algorithm

Rabin–Karp is a string-matching algorithm that uses **hashing** to find occurrences of a pattern P in text T.

Idea:

- Compute hash value of pattern P.
- Slide a window over T and compute hash of each substring.
- If hash matches, verify characters (to avoid collision).

Time Complexity:

- Best/average: $O(n + m)$
- Worst-case: $O(nm)$ due to hash collisions

Widely used in plagiarism detection and multi-pattern matching.

b) With respect to Multithreaded Algorithms explain Analyzing multithreaded algorithms, Parallel loops, Race conditions. [7]

1. Analyzing Multithreaded Algorithms

We use two major metrics:

Work (T_1)

Total time taken using one processor.

Equivalent to serial cost.

Span (T_∞)

Time taken on infinite processors.

Also called *critical path length*.

Speedup: $S = \frac{T_1}{T_p}$

Parallelism: $\text{Parallelism} = \frac{T_1}{T_\infty}$

Goal: Maximize parallelism, minimize synchronizations.

2. Parallel Loops

Loops whose iterations are independent can be executed in parallel.

Example:

parfor i = 1 to n:

$$A[i] = B[i] + C[i]$$

Such parallel loops scale very well because:

- No dependencies
- Uniform workload
- No communication overhead

3. Race Conditions

A race condition occurs when:

- Two or more threads access shared data
- At least one thread writes
- Execution order affects final output

Example:

shared x = 0

Thread1: x = x + 1

Thread2: x = x + 1

Possible results: 1 or 2 (incorrect).

Solution: locks, mutexes, atomic operations.

Q8) a) Write and explain pseudo code for Multi-threaded merge sort algorithm. How parallel merging gives a significant parallelism advantage over Merge Sort? [9]

Multithreaded Merge Sort — Pseudocode

Procedure MT_MergeSort(A, l, r):

```

    if l >= r:
        return
    mid = (l + r) / 2
    spawn MT_MergeSort(A, l, mid)    // new thread
    spawn MT_MergeSort(A, mid+1, r)  // new thread
    sync                             // wait for both threads
    ParallelMerge(A, l, mid, r)

```

Parallel Merge Idea

Normally, merging two sorted arrays L and R is linear ($O(n)$) and sequential.

To parallelize:

- Pick the median of L.
- Binary search its correct position in R.
- Divide both arrays into two halves.
- Recurse merging both halves in parallel.

Why it gives significant parallelism?

Because merge sort's bottleneck is the merge step:

Serial merge time = $O(n)$

Parallel merge time $\approx O(\log n)$ depth

Overall advantage:

- Allows merge sort to exploit both divide and merge phases
- Moves parallelism closer to optimal
- Improves span from $O(n) \rightarrow O(\log^2 n)$

b) Write a pseudo code for naïve string matching algorithm and Rabin-Karp algorithm for string matching and analyze the same. [8]

1. Naïve String Matching — Pseudocode

NaiveMatch(T, P):

```

    n = length(T)

```

```

m = length(P)
for i = 0 to n-m:
    j = 0
    while j < m and T[i+j] == P[j]:
        j++
    if j == m:
        print "Match at", i

```

Time Complexity

- Best case: $O(n)$
- Worst case: $O(nm)$
- Example worst case: $T = \text{"aaaaaa..."}$, $P = \text{"aaaab"}$

2. Rabin–Karp Algorithm — Pseudocode

RabinKarp(T, P):

```

n = length(T)
m = length(P)
hP = hash(P)
hT = hash(T[0..m-1])

for i = 0 to n-m:
    if hP == hT:
        if T[i..i+m-1] == P:    // verify
            print "Match at", i
    if i < n-m:
        hT = RecomputeRollingHash(T, i, m)

```

Hash idea: Rolling Hash

$$hT_{i+1} = (d \cdot (hT_i - T[i] \cdot d^{m-1}) + T[i+m]) \bmod q$$

Time Complexity

- Average case: $O(n + m)$
- Best case: $O(n + m)$
- Worst case: $O(nm)$ (hash collisions)

► **MAY/JUN 2023**

Q7) a) Write and explain pseudo code for multi-threaded merge sort algorithm. How parallel merging gives a significant parallelism advantage over merge sort? [9]

Pseudocode for Multithreaded Merge Sort

Procedure MT_MERGESORT(A, l, r):

```

    if l >= r:
        return
    mid = (l + r) / 2
    // Spawn threads for left and right subproblems
    spawn MT_MERGESORT(A, l, mid)
    spawn MT_MERGESORT(A, mid+1, r)
    sync // wait for both threads
    PARALLEL_MERGE(A, l, mid, r)

```

Parallel Merge Procedure

Procedure PARALLEL_MERGE(A, l, mid, r):

```

    n1 = mid - l + 1
    n2 = r - mid
    L = A[l..mid]
    R = A[mid+1..r]
    // Parallel merge using divide & conquer
    if n1 < threshold or n2 < threshold:
        merge sequentially
        return
    choose pivot = median(L)
    pos = BinarySearch(R, pivot)
    // Now process two halves in parallel
    spawn PARALLEL_MERGE(left part of L, left part of R)
    spawn PARALLEL_MERGE(right part of L, right part of R)
    sync

```

Why Parallel Merging Gives Significant Advantage?

1. Normal Merge is Sequential

Traditional merge is $O(n)$ and is *not parallel*.

Even if the recursive (divide) steps are parallel, merge becomes a bottleneck.

2. Parallel Merge Uses Binary Search

- Median of one array is found.
- Binary search done in other array $\rightarrow O(\log n)$
- Problem splits into two independent subproblems.

3. Merging Achieves True Divide-and-Conquer

Parallel merge divides merging into:

- Two equal halves
- Each half merged in parallel

4. Reduced Span (Critical Path Length)

Serial merge span = $O(n)$

Parallel merge span = $O(\log^2 n)$

5. Overall Merge Sort Becomes Highly Parallel

Without parallel merge:

- Only divide phase is parallel \rightarrow limited parallelism

With parallel merge:

- Both divide and conquer phases run in parallel
- Near-optimal parallel speedup

b) i) Explain an algorithm for Distributed Minimum Spanning Tree.

ii) Write and explain Rabin-Karp algorithm for string matching.

(i) Distributed Minimum Spanning Tree (MST) Algorithm

The most famous distributed MST algorithm is **The Gallager–Humblet–Spira (GHS) Algorithm**.

Basic Idea

Each processor/node has:

- Partial view of graph
- Communicates with neighbors
- Uses message passing to build MST

Key Concepts

- Nodes form **fragments** (partial MSTs)
- Each fragment has a **level** and **leader node**
- Repeatedly find the **minimum outgoing edge (MOE)** for each fragment
- Merge fragments through these MOEs

Algorithm Steps

1. Initialization: Every node is its own fragment.

2. Find Minimum Outgoing Edge (MOE): Each fragment identifies its lightest edge leading outside the fragment.

3. Send Join Messages: Fragments send messages through MOE to merge with neighboring fragments.

4. Fragment Merging:

Two fragments merge into a single fragment with:

- New leader
- Increased level
- Updated structure

5. Repeat Until One Fragment Remains: The final fragment represents the **complete MST**.

Features

- Works on distributed graphs
- Requires no central control
- Uses message passing
- Time complexity: $O(E + V \log V)$ messages

(ii) Rabin–Karp Algorithm for String Matching

Idea: Use a **rolling hash** to compare pattern hash with substring hash.

Pseudocode

RabinKarp(T, P):

$n = \text{length}(T)$

$m = \text{length}(P)$

$hP = \text{hash}(P)$

$hT = \text{hash}(T[0..m-1])$

 for $i = 0$ to $n - m$:


```

if hP == hT:
    if T[i..i+m-1] == P:
        print "Match at", i
    if i < n - m:
        hT = RecomputeRollingHash(hT, T[i], T[i+m], m)

```

Rolling Hash Formula

$$h_{i+1} = (d(h_i - T[i]d^{m-1}) + T[i+m]) \bmod q$$

Time Complexity

- Best/Average: $O(n + m)$
- Worst: $O(nm)$ (hash collisions)
- Space = $O(1)$

Q8) a) Write short notes on the following. [10]

- i) Multithreaded matrix multiplication
- iii) Distributed breadth first search

- ii) Multithreaded merge sort.
- iv) The Rabin-Karp algorithm.

i) Multithreaded Matrix Multiplication

- Divide matrix C into blocks/rows.
- Assign each block's computation to a separate thread.
- Achieves parallelism because each element $C_{ij} = \sum A_{ik} B_{kj}$ is independent.

ii) Multithreaded Merge Sort

- Left and right recursive calls run in separate threads.
- Synchronization after child threads complete.
- Optional: parallel merging for deeper parallelism.

iii) Distributed BFS

- Graph split among machines.
- BFS frontier is shared.
- Each machine expands frontier locally and exchanges messages.
- Used in web graph, social network processing.

iv) Rabin–Karp Algorithm

- Hash pattern and each window in text.
- Use rolling hash to update hash quickly.

- Compare only when hashes match.
- Efficient for multi-pattern search.

b) With respect to Multithreaded Algorithms explain Analyzing multithreaded algorithms, Parallel loops, Race conditions. [7]

1. Analyzing Multithreaded Algorithms

Key metrics:

Work (T_1)

Total time on 1 processor (serial cost).

Span (T_∞)

Critical-path length on infinite processors.

Parallelism: $\text{Parallelism} = \frac{T_1}{T_\infty}$

Speedup: $S = \frac{T_1}{T_p}$

2. Parallel Loops

Loops where iterations are independent can be parallelized.

Example:

parfor i = 1..n:

A[i] = A[i] * 2

Advantages

- No dependencies
- Easy to scale
- High parallel efficiency

3. Race Conditions

Occurs when:

- Two or more threads access shared data
- At least one writes
- Final result depends on execution order

Example:

x = 0

Thread1: $x = x + 1$

Thread2: $x = x + 1$

May produce 1 or 2 \rightarrow incorrect.

Solution: Locks, semaphores, atomic instructions.

► **NOV/DEC 2023**

Q7) a) i) Explain an algorithm for Distributed Minimum Spanning Tree.

ii) Write and explain Rabin-Karp algorithm for string matching. [10]

(i) — Distributed Minimum Spanning Tree (GHS algorithm)

Goal: compute an MST when the graph is distributed: each node is a processor knowing only its incident edges and able to send messages to neighbors.

High-level idea (Gallager–Humblet–Spira, GHS):

1. **Fragments:** Initially each node is a fragment of size 1. Each fragment has a level and a fragment id (leader).
2. **Find Minimum Outgoing Edge (MOE):** Each fragment finds the lightest edge leaving the fragment (the minimum outgoing edge). This is done by having each node locally examine incident edges and vote/aggregate to the fragment leader.
3. **Merge:** Two fragments connected by their respective MOE merge into a single larger fragment. When two fragments merge, the merged fragment's level may increase (levels control coordination).
4. **Repeat:** Repeat MOE discovery and merges until only one fragment (the MST) remains.
5. **Message types & termination:** The algorithm uses messages like TEST, ACCEPT, REJECT, REPORT, CHANGE_ROOT, CONNECT. Termination occurs when no outgoing edges remain (single fragment).

Properties:

- Works in asynchronous message-passing model.
- Correct: merges always pick smallest crossing edges, preserving MST property.
- Complexity (roughly): $O(E \log V)$ messages and $O(V \log V)$ time (amortized across levels), depending on implementation; message complexity is often stated as $O(E + V \log V)$ in standard GHS analyses.

Why it's good: decentralized, fault-tolerant, no central coordinator — ideal for very large distributed graphs.

(ii) — Rabin–Karp string matching — algorithm and explanation

Problem: find all occurrences of pattern P (length m) in text T (length n).

Key idea: use a rolling hash to filter mismatches cheaply; only when hash matches do we verify character-by-character.

Parameters:

- Choose base d (size of alphabet, e.g. 10 for digits, 256 for bytes).
- Choose modulus q (a reasonably large prime) to reduce collisions.

Pseudocode:

RabinKarp($T[0..n-1]$, $P[0..m-1]$, d , q):

$hP = 0$

$hT = 0$

$h = (d^{(m-1)} \bmod q)$ // used to remove leading digit

 // compute initial hashes

 for $i = 0..m-1$:

$hP = (d * hP + \text{code}(P[i])) \bmod q$

$hT = (d * hT + \text{code}(T[i])) \bmod q$

 for $i = 0..n-m$:

 if $hP == hT$:

 // potential match — verify to avoid spurious due to collision

 if $T[i..i+m-1] == P[0..m-1]$:

 report match at i

 if $i < n-m$:

 // roll the hash: remove leading, add trailing

$hT = (d * (hT - \text{code}(T[i]) * h) + \text{code}(T[i+m])) \bmod q$

 if $hT < 0$: $hT += q$

Complexity:

- Average / expected: $O(n + m)$ — hash update $O(1)$ per shift, verification rare.
- Worst-case: $O(n * m)$ (many collisions causing full verifications).
- Space: $O(1)$ extra beyond input.

Notes:

- Good for single and multi-pattern matching (with multi-hash or combined).
- Spurious hits = cases when $hP == hT$ but substring \neq pattern. Their expected number is low if q large and hash well-chosen

b) With respect to Multithreaded Algorithms explain Analyzing multithreaded algorithms, Parallel loops, Race conditions. [7]

1. Analyzing multithreaded algorithms

Use two central metrics:

- **Work (T_1):** total time if run on one processor (sum of all operations).
- **Span (T_∞) / Critical-path length:** time on infinite processors — length of longest dependency chain.
From these derive:
 - **Parallelism** = T_1 / T_∞ (upper bound on achievable speedup).
 - **Speedup on P processors:** at most $\min(P, T_1 / T_\infty)$, actual $T_P \geq \max(T_1/P, T_\infty)$ by Brent's theorem.

Also consider overheads: thread creation, synchronization, cache effects, load balance.

2. Parallel loops

A loop whose iterations are independent can be run in parallel (e.g., parfor). Requirements:

- No loop-carried dependencies.
- Balanced workload among threads.
- Low synchronization: ideally none inside the loop.
Examples: element-wise vector ops, independent row computations in matrix multiply.
Benefits: almost linear scaling (until memory or other bottlenecks).

3. Race conditions

A race occurs when two or more threads access shared data and at least one writes, with no proper synchronization — result depends on timing.

- Example: two threads increment a shared counter without atomic op: lost updates possible.
- Fixes: locks/mutexes, atomic operations, barriers, lock-free algorithms. But synchronization can hurt performance (contention); design must minimize critical sections.

Q8) a) Write and explain pseudo code for Multi-threaded merge sort algorithm. How parallel merging gives a significant parallelism advantage over Merge Sort? [9]

Pseudocode (thread-spawning, divide-and-conquer; keep threshold to limit thread creation):

Procedure MT_MERGESORT(A, l, r):

if $l \geq r$: return

mid = $(l + r) // 2$

// sort halves in parallel

spawn MT_MERGESORT(A, l, mid)

```
spawn MT_MERGESORT(A, mid+1, r)
sync
```

```
PARALLEL_MERGE(A, l, mid, r)
```

PARALLEL_MERGE(A,l,mid,r) uses divide-and-conquer merging:

Procedure PARALLEL_MERGE(A, l, mid, r):

```
// lengths n1 = mid-l+1, n2 = r-mid
if n1 + n2 <= THRESHOLD:
    sequential_merge(A,l,mid,r)
    return

choose pivot index i = l + n1//2 // median element of left
pivot = A[i]
j = binary_search(A, mid+1, r, pivot) // find split in right
// pivot placed at position k = (i - l) + (j - (mid+1)) + l
spawn PARALLEL_MERGE(A, l, i-1, mid, j-1)
spawn PARALLEL_MERGE(A, i, mid, j, r)
sync
```

(Implementation adjusts indices carefully; canonical references show exact index math.)

Why parallel merging improves parallelism

- In classic parallel merge sort where only recursive sorts run in parallel, the merge step is sequential $O(n)$ and forms the critical path, bounding speedup.
- Parallel merging splits a merge into two (or more) independent merge subproblems via median + binary search — enabling the merge itself to be done in parallel.
- Result: span (T_∞) for merging drops from $O(n)$ to about $O(\log^2 n)$ (depending on recursion), and overall T_∞ for whole sort becomes polylogarithmic rather than linear in n . This increases T_1/T_∞ (parallelism), enabling much higher speedup on many cores.

Practical notes:

- Use threshold to avoid too fine-grained tasks.
- Overhead (thread creation, synchronization) must be amortized across large tasks.

b) For string matching, working module $q = 11$, how many spurious hits does the Rabin-Karp matcher encounters in Text $T = 31415926535.....[8]$

i) $T = 31415926535$

ii) $P=26$

iii) Here $T.Length = 11$ so $Q = 11$

iv) And $P \bmod Q = 26 \bmod 11 = 4$

v) Now find the exact match of $P \bmod Q...$

- Text $T = 31415926535$ (length $n = 11$)
- Pattern $P = 26$ (so $m = 2$)
- Modulus $q = 11$
- Pattern hash: $P \bmod q = 26 \bmod 11 = 4$

We need to find how many **spurious hits** (hash equal to 4 but substring \neq "26") occur when scanning all length-2 substrings of T .

All length-2 substrings of T (positions 0..9):

1. $T[0..1] = 31 \rightarrow 31 \bmod 11 = 9$
2. $T[1..2] = 14 \rightarrow 14 \bmod 11 = 3$
3. $T[2..3] = 41 \rightarrow 41 \bmod 11 = 8$
4. $T[3..4] = 15 \rightarrow 15 \bmod 11 = 4 \leftarrow$ hash match, substring "15" \neq "26" \Rightarrow **spurious hit #1**
5. $T[4..5] = 59 \rightarrow 59 \bmod 11 = 4 \leftarrow$ hash match, "59" \neq "26" \Rightarrow **spurious hit #2**
6. $T[5..6] = 92 \rightarrow 92 \bmod 11 = 4 \leftarrow$ hash match, "92" \neq "26" \Rightarrow **spurious hit #3**
7. $T[6..7] = 26 \rightarrow 26 \bmod 11 = 4 \leftarrow$ hash match, "26" == "26" \Rightarrow **true match**
8. $T[7..8] = 65 \rightarrow 65 \bmod 11 = 10$
9. $T[8..9] = 53 \rightarrow 53 \bmod 11 = 9$
10. $T[9..10] = 35 \rightarrow 35 \bmod 11 = 2$

Summary:

- Hash-equal positions: substrings "15", "59", "92", "26" \rightarrow 4 total hash matches.
- Among these, only one is the actual pattern "26".
- Therefore **spurious hits = 4 – 1 = 3**.

So the Rabin-Karp matcher encounters **3 spurious hits** for the data given.

► **MAY/JUNE 2024**

Q7) a) Write a Rabin-Karp string matching algorithm. Input to the algorithm be: Original text “t” of length n and pattern text being matched is “p” of length m. What is the expected runtime and worst-case runtime of this algorithm? [10]

We are given:

- **Text** $t[0..n-1]$ of length **n**
- **Pattern** $p[0..m-1]$ of length **m**

Rabin–Karp uses **rolling hash** to compare pattern and each substring of the text.

Rabin–Karp Algorithm (Pseudocode)

RabinKarp(t, p, n, m, d, q):

// d = size of alphabet

// q = large prime modulus

$h = (d^{(m-1)}) \bmod q$ // used for removing leading digit

p_hash = 0

t_hash = 0

// Step 1: Compute initial hash values for p and first window of t

for i = 0 to m-1:

 p_hash = (d * p_hash + code(p[i])) mod q

 t_hash = (d * t_hash + code(t[i])) mod q

// Step 2: Slide the pattern over text one position at a time

for s = 0 to n - m:

 if p_hash == t_hash:

 // possible match -> verify characters

 if t[s..s+m-1] == p[0..m-1]:

 print "Pattern found at shift", s

// Step 3: Compute rolling hash for next window

if $s < n - m$:

$$t_hash = (d * (t_hash - code(t[s]) * h) + code(t[s + m])) \bmod q$$

if $t_hash < 0$:

$$t_hash = t_hash + q$$

Explanation

1. Compute hash of pattern p .
2. Compute hash of first window of text $t[0..m-1]$.
3. Compare the two hashes.
 - o If equal \rightarrow check characters (to avoid collision).
4. Slide the window one step at a time using **rolling hash**:

$$t_hash = (d(t_hash - t[s]d^{m-1}) + t[s + m]) \bmod q$$

5. Repeat until end of text.

Time Complexity

Expected Time

$$O(n + m)$$

Because:

- Rolling hash update = $O(1)$
- Expected number of *spurious hits* is small
- Character verification rarely triggered

Worst-Case Time: $O(nm)$

Occurs when:

- Many hash collisions happen
- Every shift triggers full pattern verification

Example worst-case:

- Text: "aaaaaaaaaaaa....a"
- Pattern: "aaaab"

b) Write multi-threaded merge sort algorithm. Briefly discuss how does it differ from conventional merge sort. [8]

Multithreaded Merge Sort Algorithm (Pseudocode)

Uses **spawn** and **sync** to create parallelism.

Procedure MT_MergeSort(A, l, r):

 if l >= r:

 return

 mid = floor((l + r) / 2)

 // Sort left and right halves in parallel

 spawn MT_MergeSort(A, l, mid)

 spawn MT_MergeSort(A, mid + 1, r)

 // Wait for both halves to finish

 sync

 // Merge the two sorted halves

 Merge(A, l, mid, r)

Standard Merge Procedure

Procedure Merge(A, l, mid, r):

 create temp array

 merge left and right subarrays into temp

 copy temp back to A[l..r]

How Multithreaded Merge Sort Differs from Conventional Merge Sort

1. Parallel Execution

- Conventional merge sort: recursive calls run **sequentially**.
- Multithreaded merge sort: calls run **in parallel** using spawn.

2. Use of sync

- Ensures both recursive sorts finish before merging.

- Not needed in normal merge sort.

3. Better Use of Multi-core CPUs

- Parallel merge sort reduces run time on multi-core machines.
- Conventional merge sort runs on a single core.

4. Work vs. Span

- Total work $T_1 = O(n \log n)$ — same as normal merge sort.
- **Parallel span** $T_\infty = O(\log^2 n)$ (much smaller).

This yields near-optimal **parallel speedup**.

5. Thread-creation Overhead

- Multithreaded version must manage threads efficiently.
- A threshold is often used to stop spawning tiny tasks.

Summary

Feature	Conventional Merge Sort	Multithreaded Merge Sort
Execution	Sequential	Parallel via spawn
Synchronization	Not needed	Uses sync
Performance	$O(n \log n)$ on 1 core	Much faster on multi-core
Span	$O(n)$	$O(\log^2 n)$
Overhead	Low	Higher (threads)

Q8) a) Consider the graph represented by an adjacency matrix: [10]

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

We have the adjacency matrix:

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

Each node is considered as an **independent process**; BFS proceeds in *levels*, where each level is processed in parallel by multiple nodes.

Let **A** be the BFS root.

We show the **distributed BFS waves (levels)**:

LEVEL 0 (Initialization)

- Node A becomes the **source**
- A marks itself visited
- A sends “VISIT” messages to all neighbors: **B and C**

Messages sent in parallel:

- $A \rightarrow B$
- $A \rightarrow C$

Visited set: {A}

LEVEL 1 (Processing messages received from A)

Nodes B and C receive message from A.

Node B:

- Marks itself visited, parent = A
- Sends “VISIT” to unvisited neighbors: **D, E**

Node C:

- Marks itself visited, parent = A
- Sends “VISIT” to unvisited neighbors: **F, G**

Messages sent in parallel:

- $B \rightarrow D$
- $B \rightarrow E$
- $C \rightarrow F$
- $C \rightarrow G$

Visited set: {A, B, C}

LEVEL 2 (Processing messages received from B and C)

Nodes D, E, F, G receive messages.

Node D

- Receives from B
- Marks visited, parent=B
- No further neighbors except B (already visited)

Node E

- Same as D
- Marks visited, parent=B
- No new neighbors

Node F

- Receives from C
- Marks visited, parent=C
- No new neighbors

Node G

- Same as F
- Marks visited, parent=C
- No new neighbors

No new messages generated.

Visited set: {A, B, C, D, E, F, G}

LEVEL 3 (Termination)

No new nodes discovered.

All nodes have received their parent assignment and visited messages.

Distributed BFS Tree Output:

```

A
 /\
B  C
 /\ | \
D  E F G

```

Key Points about Distributed BFS Demonstrated

1. Each node processes incoming BFS messages independently.
2. BFS proceeds in synchronous "waves" (frontiers).
3. No centralized control — each node broadcasts to neighbors.
4. Communication pattern is message passing based on adjacency.
5. The BFS tree emerges from parent assignments.

b) What do you understand by spawn and sync keywords used in multithreaded programming? Explain with the help of suitable example.[8]

Multithreaded pseudocode languages (like Cilk, OpenMP, Intel TBB) use spawn and sync to manage parallel execution.

1. spawn Keyword

spawn indicates that the function can run in parallel with the continuation of the current function.

Example:

```

spawn MergeSort(A, l, mid)
spawn MergeSort(A, mid+1, r)

```

Meaning:

- The two recursive calls may run on two different threads/cores
- The parent does not wait for them automatically

Equivalent to launching a lightweight parallel task.

2. sync Keyword

sync makes the current function wait until all previously spawned child tasks are finished.

Example:

```

spawn MergeSort(left)
spawn MergeSort(right)

sync          // waits for both sorting tasks

```

merge(left, right)

Meaning: Ensures that the merge happens only after both children complete.

3. Full Example

Procedure ParallelFib(n):

if $n \leq 1$:

return n

x = spawn ParallelFib(n-1)

y = ParallelFib(n-2)

sync

return x + y

Explanation:

- ParallelFib(n-1) runs in parallel as a separate child task
- ParallelFib(n-2) runs in current task
- sync ensures both results computed before adding them

Summary of spawn & sync

Keyword	Meaning	Purpose
spawn	Create parallel task	Enable concurrency
sync	Wait for spawned tasks	Ensure correctness and synchronization

They form the foundation of structured parallelism in multithreaded algorithms.

► NOV/DEC 2024

Q7) a) Write a Rabin-Karp string matching algorithm. Let input to the algorithm be Original text “t” of length n and pattern text being matched is “p” of length m. What is the expected runtime and worst-case runtime of this algorithm? [10]

Let:

- Text: $t[0..n-1]$
- Pattern: $p[0..m-1]$
- Alphabet size: d
- A prime modulus: q

Rabin–Karp uses hashing + sliding window rolling hash.

Rabin–Karp Algorithm (Pseudocode)

RabinKarp(t, p, n, m, d, q):

```
// Preprocessing
h = (d^(m-1)) mod q      // used to remove leading character
p_hash = 0
t_hash = 0

// Compute initial hash values
for i = 0 to m-1:
    p_hash = (d * p_hash + code(p[i])) mod q
    t_hash = (d * t_hash + code(t[i])) mod q

// Slide the pattern over text
for s = 0 to n - m:
    // If hash matches, verify actual characters
    if p_hash == t_hash:
        if t[s..s+m-1] == p[0..m-1]:
            print "Pattern found at shift:", s

    // Compute next window hash
    if s < n - m:
        t_hash = (d * (t_hash - code(t[s]) * h) + code(t[s+m])) mod q
        if t_hash < 0:
            t_hash = t_hash + q
```

Explanation

1. Compute a numeric hash of pattern and first window of text.
2. Slide window by 1 step each iteration.
3. Use rolling hash to update in $O(1)$ time.
4. When hash values match, perform exact character check to avoid collision.

Time Complexity:

Expected Runtime: $O(n + m)$

Reason:

- Rolling hash update is $O(1)$ per shift.
- Expected number of hash collisions very small.
- Character verification done rarely.

Worst Case Runtime: $O(nm)$

Reason:

- If many hash collisions occur (e.g., adversarial strings),
- Every shift must perform full pattern comparison $\rightarrow O(m)$ each

b) Briefly explain performance measures – speedup, efficiency, throughput, contention, and latency of multithreaded algorithms. [8]

1. Speedup (S)

Speedup measures **how much faster** a multithreaded algorithm runs on P processors than on 1 processor.

$$S(P) = \frac{T_1}{T_P}$$

Where:

- T_1 = time using 1 processor
- T_P = time using P processors

Ideal speedup: **$S(P) = P$**

Realistic speedup $< P$ due to synchronization and overheads.

2. Efficiency (E)

Efficiency measures **how well processors are utilized**.

$$E(P) = \frac{S(P)}{P} = \frac{T_1}{P \cdot T_P}$$

Range = 0 to 1.

Higher efficiency = better resource usage.

3. Throughput

Throughput is the **amount of work completed per unit time**.

Examples:

- Number of tasks completed/second
- Number of operations processed per second

High throughput means more parallel work is being finished.

4. Contention

Contention occurs when **multiple threads attempt to access the same shared resource** simultaneously.

Effects:

- Locks, atomic operations, or memory access conflicts slow execution
- Causes waiting, blocking, reduced parallel speedup

Lower contention → faster multithreaded performance.

5. Latency

Latency is the **delay** experienced by a thread when:

- Waiting for data,
- Waiting for I/O,
- Waiting for another thread (sync),
- Or waiting for a critical section to unlock.

High latency reduces performance.

Low latency means fast responsiveness of the system.

Q8) a) Consider the graph represented by an adjacency matrix:

Show stepwise process how the distributed breadth first search algorithm works on the above graph.

Graph:

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	0	1	1
D	0	1	0	0	0	0	0
E	0	1	0	0	0	0	0
F	0	0	1	0	0	0	0
G	0	0	1	0	0	0	0

We assume **A is the BFS root node** and BFS is executed in a distributed manner (each vertex is a process running BFS independently).

Distributed BFS runs in *levels* (also called “waves”), where each level’s frontier broadcasts to neighbors.

LEVEL 0 — Start at Root Node A

- A marks itself **visited**.
- A sends BFS “VISIT” messages to all its neighbors:

Neighbors of A: **B, C**

Messages:

A → B

A → C

Visited so far: **{A}**

LEVEL 1 — Nodes B and C process messages from A

Node B:

- Receives “VISIT” from A
- Marks visited
- Sets parent = A
- Sends “VISIT” to unvisited neighbors: **D, E**

Node C:

- Receives “VISIT” from A
- Marks visited
- Sets parent = A
- Sends “VISIT” to unvisited neighbors: **F, G**

Messages sent in this wave:

- B → D
- B → E
- C → F
- C → G

Visited so far: **{A, B, C}**

LEVEL 2 — Nodes D, E, F, G process messages

Node D:

- Receives from B

- Marks visited
- Parent = B
- Its only neighbor is B → already visited → no new messages

Node E:

- Receives from B
- Marks visited
- Parent = B
- No new neighbors to send to

Node F:

- Receives from C
- Marks visited
- Parent = C
- No further neighbors

Node G:

- Receives from C
- Marks visited
- Parent = C
- No further neighbors

Visited so far:

{A, B, C, D, E, F, G}

No new messages → BFS terminates.

Distributed BFS Tree Obtained

```

      A
     / \
    B   C
   / \ / \
  D  E F  G

```

This is exactly the BFS tree from distributed execution.

Important points (Distributed BFS):

- Works in synchronous “waves”
- Each node independently processes BFS messages

- Parent pointers form the BFS spanning
- No central coordinator required
- Communication is via message passing

Q8) b) If we have two matrices of the order $m \times n$ and $n \times p$ then what will be the time complexity of multiplying these matrices in conventional approach and in multithreaded approach. Discuss. [8]

We multiply:

- Matrix **A** of size $m \times n$
- Matrix **B** of size $n \times p$
- Result matrix **C** of size $m \times p$

(i) Conventional (Sequential) Matrix Multiplication

Each element:

$$C[i][j] = \sum_{k=1}^n A[i][k] \cdot B[k][j]$$

For each of the $m \cdot p$ cells, we perform n multiplications & additions.

Time Complexity:

$$O(mnp)$$

(ii) Multithreaded Matrix Multiplication

Strategy:

- Different threads compute:
 - one **row** each, or
 - one **column** each, or
 - one **block** each, or
 - each **element** independently.

Since elements of C are independent, perfect parallelization is possible.

If T threads are available:

$$T_p = \frac{mnp}{T}$$

Time Complexity with T threads:

$$O\left(\frac{mnp}{T}\right)$$

If $T = m \cdot p$ (ideal case: one thread per element):

$$T_p = O(n)$$

Difference

Feature	Conventional Approach	Multithreaded Approach
Execution	Sequential	Parallel
Time Complexity	$O(mnp)$	$O(mnp / T)$
Speedup	None	Depends on number of threads
Utilization	Single core	Multi-core / multi-processor
Scalability	Poor	High
Design	Simple	Requires thread creation, scheduling, synchronization

Note: Please verify all answers before referring.