

HPC – Unit 6: Parallel Algorithms, Sorting, Distributed Computing & Kubernetes

May–June 2023 (Paper [6004]-493)

Q7 a) Odd-Even Transposition in Parallel Bubble Sort

[8 Marks]

Background: Sequential Bubble Sort

In sequential bubble sort, adjacent elements are compared and swapped if out of order, in repeated passes until the array is sorted. It takes $O(n^2)$ time in the worst case.

Parallel Odd-Even Transposition Sort

Odd-even transposition adapts bubble sort for p processors, each holding one element. It alternates between two phases per round, and after exactly p rounds, the array is fully sorted — making the parallel complexity $O(p) = O(n)$ rounds when $n = p$.

The algorithm consists of p phases alternating between:

- Odd phase: Processors at positions 1,3,5,... compare-and-swap with their right neighbour (2,4,6,...).
- Even phase: Processors at positions 0,2,4,... compare-and-swap with their right neighbour (1,3,5,...).

In each compare-and-swap, the processor with the smaller index keeps the minimum and the other keeps the maximum.

Step-by-Step Example: Sort [3, 1, 4, 2] on 4 processors

Initial: $P_0=3, P_1=1, P_2=4, P_3=2$

Round 1 — Odd phase (compare positions 0-1 and 2-3):

P_0, P_1 : compare(3,1) → $P_0=1, P_1=3$ [swapped]

P_2, P_3 : compare(4,2) → $P_2=2, P_3=4$ [swapped]

After odd: [1, 3, 2, 4]

Round 1 — Even phase (compare positions 1-2):

P_1, P_2 : compare(3,2) → $P_1=2, P_2=3$ [swapped]

P_0 and P_3 have no partner → no change

After even: [1, 2, 3, 4] ← Already sorted!

Round 2 — Odd phase: P_0, P_1 : compare(1,2) → no swap; P_2, P_3 : compare(3,4) → no swap

Round 2 — Even phase: P_1, P_2 : compare(2,3) → no swap

After 4 rounds: [1, 2, 3, 4] — Confirmed sorted ✓

Complexity Analysis

- Time complexity: $O(n)$ rounds ($p = n$ processors), $O(1)$ work per round → $O(n)$ parallel time.
- Total work (cost): $O(n) \times O(n) = O(n^2)$ — matches sequential bubble sort (cost-optimal).
- Communication: Each round, each processor sends and receives one value — $O(1)$ messages of size $O(1)$.

Note: Odd-even transposition is provably correct (it can be shown by a 0-1 principle: if it sorts all binary sequences, it sorts all sequences). It is simple and regular, making it ideal for systolic array implementations

| *in hardware.*

Q7 b) Parallel Depth-First Search (DFS) Algorithm

[6 Marks]

Sequential DFS Recap

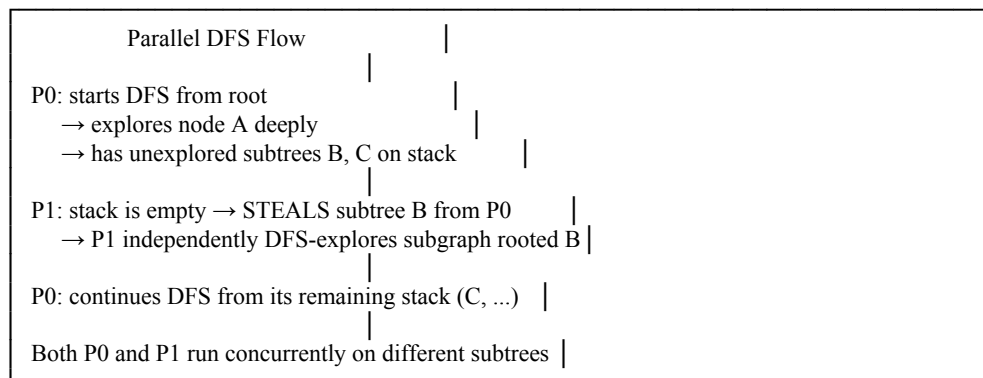
In sequential DFS, a graph $G = (V, E)$ is explored by starting at a root, marking it visited, and recursively exploring all unvisited neighbours before backtracking. It produces a DFS tree and runs in $O(V + E)$ time. The inherently recursive, stack-based nature of DFS makes parallelisation challenging — each step depends on the previous one.

Challenges in Parallel DFS

- DFS has deep sequential dependencies: the choice of which node to explore next depends on what was explored just before. This limits parallelism.
- Unlike BFS, where an entire frontier of nodes can be processed simultaneously, DFS tends to explore one path deeply before backtracking — hard to parallelize along the critical path.
- Work stealing is the primary technique used to extract parallelism from DFS.

Parallel DFS with Work Stealing

Each processor maintains its own local DFS stack. Initially, the root is placed on P0's stack. When a processor's stack is empty, it steals a subgraph from another processor's stack (specifically, it takes the deepest unprocessed subtree — the bottom of the victim's stack, which represents the most independent subproblem).



Algorithm pseudocode:

```

Each processor  $P_i$ :
while graph not fully explored:
  if local_stack is not empty:
    node = local_stack.pop()
    mark node as visited
    for each unvisited neighbour v of node:
      local_stack.push(v)
  else:
    victim = randomly chosen processor
    steal one unvisited subtree from victim's stack
    if steal successful: process it
    else: idle (try again)
  
```

Performance

- The speedup achievable depends on the graph's structure. Trees and DAGs with balanced subtrees parallelise well. Graphs with many long linear paths (bamboo-like) offer little parallelism.
- Expected parallel time with p processors: $T_p = O((V + E) / p + p \times t_{\text{steal}})$, where t_{steal} is the cost of a work-stealing operation.
- In practice, parallel DFS achieves near-linear speedup on irregular graphs when the graph has high branching factor and balanced subtrees.

Note: Parallel DFS is widely used in model checking (verifying state machines), game tree search (parallel alpha-beta pruning in chess engines), and connectivity algorithms in distributed graph processing frameworks like Pregel and GraphX.

Q7 c) Kubernetes: Features and Applications

[4 Marks]

What is Kubernetes?

Kubernetes (K8s) is an open-source container orchestration platform originally developed by Google (based on their internal Borg system) and now maintained by the Cloud Native Computing Foundation (CNCF). It automates the deployment, scaling, and management of containerised applications across a cluster of machines. In the HPC and distributed computing context, Kubernetes is increasingly used to manage GPU workloads, ML training jobs, and microservices at scale.

Key Features of Kubernetes

- Automatic scheduling: Kubernetes automatically assigns containers (pods) to available nodes based on resource requirements (CPU, memory, GPU) and constraints, maximising utilisation.
- Self-healing: If a container crashes or a node fails, Kubernetes automatically restarts the container, reschedules it on a healthy node, and replaces failed nodes — without manual intervention.
- Horizontal auto-scaling: Kubernetes can automatically increase or decrease the number of running container replicas based on CPU/memory load or custom metrics (e.g., HTTP request rate), using the Horizontal Pod Autoscaler (HPA).
- Service discovery and load balancing: Kubernetes assigns a stable DNS name and IP address to each service and automatically load-balances traffic across all pods in the service.
- Rolling updates and rollbacks: New versions of applications can be deployed with zero downtime using rolling updates. If the new version has issues, an instant rollback restores the previous version.
- Storage orchestration: Automatically mounts storage systems (local disk, NFS, cloud block storage like AWS EBS or GCP Persistent Disk) to containers that need persistent data.
- Secret and configuration management: Kubernetes can store and manage sensitive information (API keys, passwords) separately from application code using Secrets and ConfigMaps.

Applications of Kubernetes in HPC/ML

- ML training infrastructure: Running distributed ML training jobs (with GPU pods) at scale — e.g., Kubeflow orchestrates TensorFlow/PyTorch distributed training across GPU nodes.
- Microservices deployment: Deploying and managing large-scale web services and APIs with automatic scaling.
- CI/CD pipelines: Running continuous integration build and test pipelines in isolated containers.
- Data processing: Orchestrating batch data processing jobs (Spark, Flink) in containerised

environments.

Note: Kubernetes is relevant to HPC because modern AI/ML infrastructure (like Google Kubernetes Engine, AWS EKS, and on-premise GPU clusters) is almost universally managed with Kubernetes. Understanding it is essential for anyone working in cloud-based HPC.

Q8 a i) Parallel Merge Sort

[4 Marks]

Sequential Merge Sort Recap

Sequential merge sort divides the array in half recursively, sorts each half, and then merges the two sorted halves. It runs in $O(n \log n)$ time and $O(n)$ space.

Parallel Merge Sort

Parallelism is introduced at the divide step: both halves can be sorted concurrently by different processors. The merge step can also be parallelised using a parallel merge algorithm.

Parallel Merge Sort on n elements with p processors:

Phase 1 — Distribution: Divide array into p equal chunks.

Each processor sorts its n/p elements locally $\rightarrow O((n/p) \log(n/p))$ time.

Phase 2 — Parallel Merge (Binary Tree Merge):

Step 1: $p/2$ pairs of processors merge their sorted chunks $\rightarrow p/2$ sorted lists

Step 2: $p/4$ groups merge again $\rightarrow p/4$ sorted lists

...

Step $\log_2(p)$: Final merge $\rightarrow 1$ fully sorted list

Total parallel time: $O((n/p) \log(n/p)) + O(\log p \times n/p)$

$= O((n/p) \log n)$ [dominant term for large n]

Speedup $\approx O(p / \log p)$ — slightly sub-linear due to merge overhead.

An improved version uses parallel merge (using binary search to split merge boundaries) to reduce merge time to $O(\log^2 n)$ per level, giving overall parallel time $O(\log^3 n)$ for unlimited processors.

Complexity Comparison

Algorithm	Sequential Time	Parallel Time (p processors)	Speedup
Merge Sort	$O(n \log n)$	$O((n/p) \log n)$	$O(p / \log p)$
Bitonic Sort	$O(n \log^2 n)$	$O(\log^2 n)$	$O(n)$
Parallel Odd-Even	$O(n^2)$	$O(n)$	$O(n)$

Note: Parallel merge sort is commonly implemented with OpenMP (using task directives for recursive decomposition) or MPI (where each rank sorts its chunk and then participates in a tournament-style merge tree).

Q8 a ii) GPU Applications

[4 Marks]

- Deep Learning: Training CNNs, RNNs, Transformers on GPUs — NVIDIA A100/H100 GPUs are the standard hardware for training large language models (GPT, BERT) and vision models

(ResNet, ViT).

- Scientific Computing: Molecular dynamics (GROMACS, AMBER), quantum chemistry (GPAW), weather modelling (ECMWF), and finite-element analysis all use GPU acceleration.
- Real-time Graphics: GPUs were originally designed for rendering 3D graphics — ray tracing, rasterisation, and shader pipelines in gaming and virtual reality.
- Medical Imaging: CT/MRI reconstruction, ultrasound image processing, and AI-based radiological diagnosis systems run on GPUs for near-real-time results.
- Autonomous Vehicles: Sensor fusion, perception pipelines (LIDAR + camera), and path planning algorithms run on embedded GPU platforms (NVIDIA Jetson, DRIVE) in real time.
- Computational Finance: Monte Carlo risk simulations, portfolio optimisation, and high-frequency trading signal computation.
- Video Processing: GPU-accelerated video encoding (NVENC), transcoding, real-time video super-resolution, and streaming on platforms like Netflix and YouTube.

Q8 b) Issues in Sorting on Parallel Computers

[6 Marks]

1. Load Imbalance

In comparison-based parallel sorting, the number of elements assigned to each processor after a partitioning step (e.g., quicksort pivot selection) may be unequal. If one processor gets significantly more elements, it becomes a bottleneck and other processors sit idle. This is especially problematic with naive pivot selection on skewed data distributions.

2. Communication Overhead

After local sorting, elements must be redistributed across processors (the 'all-to-all personalised communication' or shuffle step in parallel quicksort/sample sort). Sending and receiving large amounts of data over the interconnect is expensive. For n elements on p processors, the total communication volume is $O(n)$ per processor in the worst case.

3. Comparison vs. Non-comparison Sorts

Comparison-based sorting (merge sort, quicksort) has a sequential lower bound of $\Omega(n \log n)$. This same lower bound does not apply to non-comparison sorts (e.g., radix sort, counting sort) which can sort in $O(n)$ sequentially but may have higher communication costs in parallel.

4. Scalability of the Merge Step

Even if local sorting is perfectly parallelised, the final global merge of p sorted lists can become a bottleneck. A naive merge of p lists takes $O(n \log p)$ time, and the merge itself must eventually be done sequentially (at least partially), limiting speedup per Amdahl's Law.

5. Data Dependency and Ordering Guarantees

Some parallel sort algorithms (e.g., bitonic sort) require the input size to be a power of 2, imposing constraints on problem size. Additionally, maintaining stable sort order (equal elements preserve original relative order) is more complex in parallel implementations.

Example: Issues in Parallel Sample Sort

1. Each of p processors sorts its n/p local elements. [OK: $O((n/p) \log(n/p))$]
2. Each processor picks s samples $\rightarrow p \times s$ total samples. [communication]
3. Root sorts $p \times s$ samples, selects $p-1$ splitters. [sequential bottleneck]

4. Splitters broadcast to all processors. $[O(p \log p) \text{ cost}]$
5. Each processor routes elements to their target processor. [all-to-all: $O(n/p)$ per proc]
6. Each processor merges its received elements. $[O((n/p) \log p)]$

Main issues: step 3 is sequential; step 5 is communication-heavy; load imbalance occurs if splitter choice is poor for skewed input.

Q8 c) Parallel BFS Algorithm (Brief)

[4 Marks]

Sequential BFS Recap

In sequential BFS, a queue is used to explore nodes level by level from a source. All neighbours of the current level are enqueued and explored before moving to the next level. Time complexity: $O(V + E)$.

Parallel BFS

The key insight for parallelising BFS is that all nodes at the same BFS level (the 'frontier') can be processed simultaneously, since they are all equidistant from the source and their processing is independent.

Parallel BFS Algorithm (Level-Synchronous):

Assign source node s to processor $P(s)$. Mark s as visited.
 $\text{frontier} = \{s\}$

while frontier is not empty:

// Distribute frontier nodes across processors
 distribute frontier nodes to processors

// Each processor expands its assigned frontier nodes in parallel

$\text{next_frontier} = \{\}$

for each node v in $\text{my_partition_of_frontier}$ (in parallel):

for each neighbour u of v :

if u not yet visited (atomic check):

mark u as visited

add u to next_frontier

// Synchronise: collect all next_frontier nodes

$\text{frontier} = \text{global_union}(\text{next_frontier})$ [collective communication]

$\text{level} = \text{level} + 1$

- Parallel time per level = $O(|\text{frontier}| / p + \text{communication cost for synchronisation})$.
- Total parallel BFS time across all levels = $O((V + E) / p + D \times t_{\text{comm}})$, where D = diameter of the graph.
- For small-world or power-law graphs (like social networks), $D = O(\log V)$, so the overhead is manageable.

Note: The key challenge in parallel BFS is the 'visited' check — when multiple processors discover the same node simultaneously, only one should mark it visited. This requires atomic operations or careful partitioning (e.g., owner-compute rule: node u is always processed by processor $u \bmod p$).

May–June 2024 (Paper [6263]-94)

Q7 a) Issues in Sorting on Parallel Computers

[8 Marks]

[REPEATED] – See Q8 b) in May–June 2023 above for the complete answer, including all five major issues and the sample sort example.

Q7 b) BFS for Parallel Execution & Complexity Analysis

[6 Marks]

Parallel BFS Algorithm

[REPEATED (core algorithm)] – See Q8 c) in May–June 2023 for the complete level-synchronous parallel BFS algorithm and pseudocode.

Complexity Analysis (Extended)

Here we provide a more detailed complexity analysis for the 2024 exam's additional requirement:

- Sequential BFS complexity: $O(V + E)$ — each vertex and each edge is visited exactly once.
- Parallel BFS with p processors: Assuming the graph is distributed so each processor owns V/p vertices and E/p edges on average.

Work per level L (size of frontier = F_L):

Computation: $O(E_L / p)$ where E_L = edges from frontier nodes at level L

Communication (frontier exchange): $O(F_L + p)$ [gather + scatter of next frontier]

Total parallel time:

$$\begin{aligned} T_p &= \sum_L [O(E_L / p) + O(F_L + p)] \\ &= O((V + E) / p) + O(D \times (V/D + p)) \\ &= O((V + E) / p + V + D \times p) \end{aligned}$$

where D = diameter of graph (number of BFS levels).

- Best case (well-connected, balanced graphs, $D = O(\log V)$): $T_p \approx O((V+E)/p + V)$ — good speedup.
- Worst case (linear chain graph, $D = V$): $T_p \approx O(V)$ — no speedup, same as sequential. This is because every level has only one frontier node; no parallelism exists.
- Speedup: $S = O(V+E) / T_p$ — depends heavily on graph topology.

Note: Modern parallel BFS implementations (like the ones used in the Graph500 benchmark) use direction-optimised BFS: for dense frontiers, they switch to a 'pull' model (each unvisited node checks if any of its neighbours are in the frontier) which reduces communication and is better suited to compressed graph representations.

Q7 c) Kubernetes (Short Note)

[4 Marks]

[REPEATED] – See Q7 c) in May–June 2023 above for the complete answer covering definition, features, and applications.

Q8 a) Sequential vs Parallel Merge Sort – Algorithm Comparison and Complexity [8 Marks]

Sequential Merge Sort

```
mergeSort(arr, left, right):
  if left >= right: return      // base case: single element
  mid = (left + right) / 2
  mergeSort(arr, left, mid)     // sort left half
  mergeSort(arr, mid+1, right)  // sort right half
  merge(arr, left, mid, right)  // merge the two sorted halves
```

merge() runs in $O(n)$ time.

Recurrence: $T(n) = 2T(n/2) + O(n)$

Solution (Master Theorem): $T(n) = O(n \log n)$

- Space complexity: $O(n)$ extra space for the merge buffer.
- Stable sort (equal elements maintain their relative order).

Parallel Merge Sort

Phase 1 — Local Sort:

Distribute n/p elements to each of p processors.

Each processor independently sorts its chunk using sequential sort.

Time: $O((n/p) \log(n/p))$

Phase 2 — Parallel Merge Tree ($\log_2 p$ rounds):

Round 1: $p/2$ pairs merge $\rightarrow p/2$ sorted lists of size $2n/p$ each

Round 2: $p/4$ groups merge $\rightarrow p/4$ sorted lists of size $4n/p$ each

...

Round $\log_2 p$: 1 group merges \rightarrow final sorted list

Each merge of two lists of size k takes $O(k)$ time sequentially,
or $O(k/p + \log^2 k)$ with parallel merge (binary search splitting).

Total parallel time (sequential merge in merge tree):

$$T_p = O((n/p) \log n) + O(n/p \times \log p) = O((n/p) \log n)$$

With parallel merge at each level:

$$T_p = O(\log^2 n \times \log p) \text{ — much faster for large } n.$$

Complexity Comparison Table

Aspect	Sequential Merge Sort	Parallel Merge Sort (p procs)
Time Complexity	$O(n \log n)$	$O((n/p) \log n)$
Space Complexity	$O(n)$	$O(n)$ total, $O(n/p)$ per processor
Speedup	—	$O(p / \log p)$ practical; $O(p)$ ideal
Communication Cost	None	$O(n \log p)$ for merge tree messages
Scalability	—	Good for large n ; degrades for small n/p
Stability	Stable	Stable (if merge is implemented stably)
Suitable for	Single-core, general use	Multi-core, GPU, MPI clusters

Q8 b) Parallel Depth-First Search in Detail

[6 Marks]

[REPEATED] – See Q7 b) in May–June 2023 above for the complete parallel DFS answer, including challenges, the work-stealing algorithm, pseudocode, and performance analysis.

Q8 c) GPU Applications (Short Note)

[4 Marks]

[REPEATED] – See Q8 a ii) in May–June 2023 above for GPU applications across deep learning, scientific computing, medical imaging, autonomous vehicles, finance, and video processing.

Additional Concepts & Quick Reference

Graph Search Algorithms: Parallel Comparison

Aspect	Parallel BFS	Parallel DFS
Parallelism model	Level-synchronous (frontier parallel)	Work stealing (subtree parallel)
Synchronisation	Required after each level	Minimal (asynchronous stealing)
Best for	Shortest path, social graphs	Game trees, model checking, connectivity
Sequential complexity	$O(V + E)$	$O(V + E)$
Parallel complexity	$O((V+E)/p + D \times p)$	$O((V+E)/p + p \times t_{\text{steal}})$
Challenge	Visited check atomicity	Load balance on irregular graphs

Sorting Algorithm Comparison

Algorithm	Sequential	Parallel (p procs)	Notes
Bubble Sort	$O(n^2)$	$O(n)$ – odd-even transposition	Simple, hardware-friendly
Merge Sort	$O(n \log n)$	$O((n/p) \log n)$	Stable, general-purpose
Bitonic Sort	$O(n \log^2 n)$	$O(\log^2 n)$	Requires $p = n$, power-of-2
Sample Sort	$O(n \log n)$	$O((n/p) \log(n/p) + p \log p)$	Practical for MPI clusters
Radix Sort	$O(nd)$	$O(nd/p + p)$	Non-comparison; d = digit count

Kubernetes Architecture Quick Reference

- **Control Plane:** API Server (all requests go through here), etcd (distributed key-value store for cluster state), Scheduler (assigns pods to nodes), Controller Manager (runs control loops for desired state).
- **Worker Nodes:** Kubelet (agent on each node that manages pods), kube-proxy (handles networking and load balancing), Container runtime (Docker, containerd, CRI-O).
- **Pod:** The smallest deployable unit in Kubernetes — a group of one or more containers sharing a network namespace and storage. GPU pods request GPU resources via 'resources.limits.nvidia.com/gpu: 1'.

Note: For HPC specifically, Kubernetes extensions like Volcano (batch scheduler), KubeFlow (ML workflows), and MPI Operator (distributed MPI jobs) are important tools that extend Kubernetes to HPC workloads beyond simple microservices.

Nov–Dec 2025 (Paper [6584]-81)

Q7 a i) Parallel Merge Sort

[4 Marks]

[REPEATED] – Full answer with sequential vs parallel comparison, complexity analysis $O((n/p) \log n)$, and OpenMP/MPI implementation notes is in: Unit 6 Answer Doc → May–June 2023 → Q8 a i).

Q7 a ii) GPU Applications

[4 Marks]

[REPEATED] – Full answer covering Deep Learning, Scientific Simulation, Medical Imaging, Autonomous Vehicles, Computational Finance, Cryptography, and Genomics is in: Unit 6 Answer Doc → May–June 2023 → Q8 a ii).

Q7 b) Communication Strategies for Parallel BFS

[6 Marks]

Parallel BFS needs a strategy for how processors communicate frontier information — specifically, how they share the set of newly discovered nodes at each level. Three communication strategies have been studied in the literature and are commonly asked in SPPU exams. The core challenge is that when processor P_i discovers node v (a neighbour of a frontier node it owns), v might be 'owned' by P_j , so P_i must tell P_j about the discovery. The efficiency of this message passing determines overall BFS performance.

1. Random Communication Strategy

In the random strategy, nodes are distributed across processors uniformly at random (node v is assigned to processor $v \bmod p$, or hashed). When processor P_i discovers that node v should be added to the next frontier, it simply sends a message to $P(v)$ (the owner of v) saying ' v is reachable at this distance'.

Frontier at level L : $\{v_1, v_2, v_3, \dots\}$

For each frontier node u owned by P_i :

For each edge (u, v) :

P_i sends message 'visit v ' to $P(v)$ = processor that owns v

$P(v)$ checks: if v not yet visited → mark visited, add to next frontier

if v already visited → discard message

After all messages are processed: collect next_frontier globally

Pros: Simple to implement, requires no special graph structure. Each processor handles $O(E/p)$ edges per level on average. Cons: The number of messages is $O(E_L)$ per level where E_L is the number of edges out of the frontier. For large, dense graphs this generates enormous message traffic. Because destination processors are random, messages cannot be batched efficiently, leading to $O(E_L)$ individual messages — very high latency on distributed-memory machines.

2. Ring Communication Strategy

In the ring strategy, processors are arranged in a logical ring ($P_0 \rightarrow P_1 \rightarrow \dots \rightarrow P_{\{p-1\}} \rightarrow P_0$). Rather than sending messages directly to the owner of a discovered node, each processor passes its 'discovery messages' around the ring one step at a time. After $p-1$ rounds, every processor has seen every discovery message.

Each processor P_i maintains a local 'message buffer' of discoveries.

Round k ($k = 1$ to $p-1$):

```

P_i sends its buffer to P_{(i+1) mod p}
P_i receives buffer from P_{(i-1+p) mod p}
P_i checks received buffer: for each 'visit v' in buffer,
    if v is owned by P_i and unvisited: mark visited, add to frontier
    else: forward in buffer (for next round) if v not yet delivered

```

After $p-1$ rounds: all messages delivered. Start next BFS level.

Pros: All communication is local (each processor only talks to its two ring neighbours), making it friendly to physically ring-like interconnects and predictable in terms of communication pattern. **Cons:** Requires $p-1$ rounds per BFS level, making it $O(p)$ times slower than direct messaging for each level. For large p , this completely dominates the computation time, making ring strategy impractical for distributed-memory systems with large p .

3. Blackboard Communication Strategy

The blackboard strategy is inspired by the metaphor of a shared blackboard that all processors can read from and write to — this is the shared-memory model. In a shared-memory parallel system (e.g., a multi-core CPU with OpenMP, or a GPU), all processors/threads can directly read and write to a globally-shared 'visited' array and 'next frontier' queue, without explicit message passing.

Shared data structures (accessible by all threads):
 visited[0..V-1]: shared boolean array, initially all false
 next_frontier: shared concurrent queue (thread-safe)

```

Each thread T_i processes a subset of current_frontier nodes:
for each node u in my_partition_of_current_frontier:
    for each neighbour v of u:
        // Atomic compare-and-swap: ensures only one thread marks v
        if (CAS(&visited[v], false, true) == false): // I was first
            next_frontier.enqueue(v) // atomic enqueue

```

Barrier: all threads synchronise after processing their partition.
 current_frontier = next_frontier; next_frontier = empty
 Repeat for next BFS level.

Pros: No explicit message passing — all 'communication' is simply shared memory reads and writes. This makes it extremely fast on shared-memory machines (multi-core CPUs, NUMA systems, GPUs). The atomic CAS ensures correctness without needing coordination. **Cons:** Only applicable to shared-memory systems. Does not scale to distributed-memory clusters (where there is no single shared memory space). Atomic operations on a single shared 'visited' array can become a bottleneck if too many threads try to write to nearby memory locations simultaneously (false sharing).

Comparison of BFS Communication Strategies

Strategy	Communication Model	Messages per Level	Rounds per Level	Best Platform
Random	Direct point-to-point	$O(E_L)$ individual msgs	1 (direct)	Distributed memory, sparse graphs
Ring	Local ring passing	$O(E_L)$ total, $O(p)$ rounds	$p - 1$	Ring interconnects, small p

Blackboard	Shared memory writes	None (shared write)	1 (atomic)	Shared-memory, GPU, multi-core
------------	----------------------	---------------------	------------	--------------------------------

Note: Modern high-performance BFS implementations like those used in Graph500 (a benchmark for graph processing) typically use a hybrid approach: intra-node communication uses the blackboard (shared memory) model, while inter-node communication uses the random model with message aggregation (bundling multiple 'visit v' messages into one network packet to reduce per-message overhead). This hybrid approach is why Graph500 systems can process trillion-edge graphs at billions of edges per second.

Q7 c) Kubernetes: Features and Applications

[4 Marks]

[REPEATED] – Full answer with complete Kubernetes definition, all features (scheduling, self-healing, auto-scaling, service discovery, rolling updates, storage orchestration, secret management), and HPC/ML applications is in: Unit 6 Answer Doc → May–June 2023 → Q7 c).

Q8 a) Recursive Decomposition in Parallelising Quicksort

[8 Marks]

Sequential Quicksort Recap

Sequential quicksort selects a pivot element, partitions the array into elements smaller than the pivot (left partition) and elements larger than the pivot (right partition), then recursively sorts both partitions. Its average-case time is $O(n \log n)$ and worst-case $O(n^2)$ (when the pivot is always the minimum or maximum element).

```
quicksort(arr, low, high):
    if low >= high: return          // base case
    pivot = partition(arr, low, high) // place pivot at correct position
    quicksort(arr, low, pivot-1)    // sort left partition — SEQUENTIAL
    quicksort(arr, pivot+1, high)   // sort right partition — SEQUENTIAL
```

Key Observation: Where Parallelism Lives

The two recursive calls (sort left partition, sort right partition) are completely independent of each other — neither affects the other's input, since the pivot has already been placed in its final position. This independence is the source of parallelism. In the recursive decomposition model, we assign these two independent subproblems to different processors.

Phase 1: Pivot Selection and Broadcast

One processor (initially the root) selects a pivot and broadcasts it to all other processors. Each processor then classifies its local data elements as 'less than pivot' or 'greater than pivot'. This phase costs $O(\log p)$ for the broadcast and $O(n/p)$ for local classification.

Phase 2: All-to-All Personalised Communication (Data Exchange)

After classification, each processor must send its 'greater than pivot' elements to the processors that will handle the upper half, and keep its 'less than pivot' elements for the lower half. This is an all-to-all personalised communication (each processor sends a different message to each other processor) and is the most expensive step.

Phase 3: Recursive Parallelism via Processor Halving

After the data exchange, the p processors are divided into two groups of $p/2$. The lower group handles

the left (smaller) partition, and the upper group handles the right (larger) partition. Each group independently applies the same algorithm recursively, halving the number of processors at each level of recursion. This continues until each processor has a single element or handles its local subarray independently.

Parallel Quicksort — Recursive Decomposition:

Level 0: All p processors, full array $[0..n-1]$

- Select pivot P_0 , broadcast to all p processors
- Each proc classifies its n/p elements as $< P_0$ or $\geq P_0$
- Redistribute: procs $0..p/2-1$ get elements $< P_0$ (left partition)
procs $p/2..p-1$ get elements $\geq P_0$ (right partition)

Level 1: Two groups of $p/2$ procs, each sorting one partition

- Each group independently selects its own pivot
- Each group redistributes among its $p/2$ procs

Level 2: Four groups of $p/4$ procs, each sorting one sub-partition

...

Level $\log_2(p)$: p groups of 1 proc each

- Each proc sorts its local subarray using sequential quicksort

After $\log_2(p)$ levels: all local arrays are sorted in-place.

Complexity Analysis

Each level involves:

Pivot broadcast: $O(\log(\text{procs_in_group}) \times t_s) \rightarrow O(\log p)$ total across levels

Data redistribution: $O(n/p \times t_w)$ per level — $O(n \times t_w)$ total across all levels

Local classification: $O(n/p)$ per level

Total parallel time (assuming balanced pivot selection):

$$T_p = O(n/p \times \log n) + O(n \times t_w \times \log p) \\ = O((n \log n)/p) \text{ [computation]} + O(n t_w \log p) \text{ [communication]}$$

Speedup (ignoring comm): $S \approx p$ for large n

Efficiency: $E = O(1)$ when $n \gg p \times t_w \times \log p$

Worst case (always bad pivot): $T_p = O(n^2/p)$ — same as sequential quicksort degradation

The Pivot Selection Problem

The performance of parallel quicksort depends critically on pivot quality. A bad pivot that splits the array 99:1 instead of 50:50 creates very unbalanced sub-problems, with some processors having much more work than others (severe load imbalance). Solutions include: median-of-three (pick the median of three random samples), random pivot selection (often used in practice), or sample sort (take p samples from each processor, sort them, use the median as the pivot).

Note: Recursive decomposition is a general parallelisation strategy, not just for quicksort. The key template is: identify the recursive case, check whether the sub-problems are independent, and if so, assign them to different processors. The halving of processor groups at each level is characteristic of this approach and gives it the $O(\log p)$ communication depth.

Q8 b) Shared Address Space vs Message Passing Formulation of Quicksort

[6 Marks]

Quicksort can be implemented in two distinct parallel programming paradigms, and understanding their trade-offs is important for choosing the right approach for a given machine architecture.

Shared Address Space Formulation

In shared-memory quicksort (e.g., implemented with Pthreads or OpenMP), all threads have direct access to the single global array being sorted. There are no explicit messages — threads communicate by reading and writing shared memory locations.

```
void parallel_quicksort_shared(int *arr, int low, int high) {
    if (low >= high) return;

    int pivot_pos = partition(arr, low, high); // partition in shared array

    // Both sub-problems work on the SAME shared array arr[]
    // but on non-overlapping ranges — no data copying needed
    #pragma omp task // spawn a new thread for left partition
    parallel_quicksort_shared(arr, low, pivot_pos - 1);

    #pragma omp task // spawn a new thread for right partition
    parallel_quicksort_shared(arr, pivot_pos + 1, high);

    #pragma omp taskwait // wait for both sub-tasks to finish
}
```

The partition step itself modifies the shared array in-place. Since the left and right partitions are disjoint memory ranges after partitioning, the two recursive calls can proceed concurrently without any synchronisation between them (only the parent needs to wait for both children via taskwait). Dynamic load balancing is handled by OpenMP's task scheduler automatically.

Message Passing Formulation

In distributed-memory quicksort (e.g., implemented with MPI), each processor owns a local subarray in its own private memory. No processor can directly read or write another's data — all coordination happens via explicit MPI_Send and MPI_Recv calls.

```
void parallel_quicksort_mpi(int *local_arr, int local_n, MPI_Comm comm) {
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
    if (size == 1) {
        sequential_quicksort(local_arr, local_n); // base case
        return;
    }

    // Step 1: root selects pivot and broadcasts
    int pivot;
    if (rank == 0) pivot = select_pivot(local_arr, local_n);
    MPI_Bcast(&pivot, 1, MPI_INT, 0, comm);

    // Step 2: each proc partitions its local array around pivot
    int *smaller, *larger, n_smaller, n_larger;
    local_partition(local_arr, local_n, pivot, &smaller, &n_smaller,
                    &larger, &n_larger);
}
```

```
// Step 3: upper half of procs send their 'smaller' to lower half,
//         lower half of procs send their 'larger' to upper half
//         (all-to-all personalised communication via MPI_Alltoallv)
// Each proc now has a balanced portion of one partition
```

```
// Step 4: split communicator and recurse
MPI_Comm sub_comm;
int color = (rank < size/2) ? 0 : 1; // 0=lower half, 1=upper half
MPI_Comm_split(comm, color, rank, &sub_comm);
parallel_quicksort_mpi(my_partition, my_n, sub_comm);
MPI_Comm_free(&sub_comm);
}
```

Detailed Comparison

Aspect	Shared Address Space	Message Passing (MPI)
Memory model	Single shared array; all threads see it	Each proc owns private subarray; explicit copies
Communication	Implicit (shared read/write)	Explicit MPI_Send, MPI_Recv, MPI_Alltoallv
Pivot broadcast	Write to shared variable; all read it	MPI_Bcast — explicit collective call
Data redistribution	No copying — partitions are disjoint ranges in shared memory	Must physically send/receive data over network
Synchronisation	__syncthreads() / taskwait — lightweight	MPI_Barrier — heavier, network round-trip
Load balancing	Dynamic (OpenMP task scheduler)	Static (at each level, halve processor groups)
Scalability	Limited to single node (shared memory only)	Scales to thousands of distributed nodes
Ease of programming	Simpler — no data movement logic	Complex — must manage send/recv buffers
Performance bottleneck	False sharing, cache coherence traffic	Data redistribution latency (network bandwidth)
Platform	Multi-core CPUs, GPU shared memory	Clusters, supercomputers, NUMA across nodes

The fundamental trade-off is this: shared address space is simpler and faster within a single node because communication is just a memory read — there is no serialisation, buffering, or network overhead. However, it cannot scale beyond the physical memory bandwidth of a single node. Message passing requires more code (explicit data movement, buffer management) but scales horizontally to arbitrarily many nodes, making it the only viable approach for sorting billions of elements on a supercomputer.

Note: In practice, modern high-performance sorting implementations use a hybrid: MPI for inter-node communication (the distributed-memory coarse-grained structure) and OpenMP or CUDA for intra-node parallelism (the shared-memory fine-grained structure). This hybrid MPI+OpenMP approach gets the best of both worlds.

Q8 c) Parallel DFS Algorithm in Detail**[4 Marks]**

[REPEATED] – Full answer with work-stealing algorithm, pseudocode, challenges, and performance analysis is in: Unit 6 Answer Doc → May–June 2023 → Q7 b). The extended version with formal complexity analysis is also in Unit 5 2025 Answer Doc → May 2025 → Q6 a).

May–June 2025 (Paper [6404]-94)

Q7 a) Parallel BFS – Short Note**[4 Marks]**

[REPEATED] – Level-synchronous parallel BFS with pseudocode, visited check atomicity, and $O((V+E)/p + D \times p)$ complexity is in: Unit 6 Answer Doc → May–June 2023 → Q8 c).

Q7 b) Communication Strategies in BFS**[5 Marks]**

[REPEATED] – Full answer with all three strategies (Random, Ring, Blackboard), their pseudocode, trade-offs, and comparison table is immediately above: Nov–Dec 2025 → Q7 b) of this document.

Q7 c) Notes on: i) Random Communication Strategy ii) Ring Communication Strategy iii) Blackboard Communication Strategy**[9 Marks]**

This question asks for extended notes on each strategy individually — building on Q7 b)'s comparison. Here each strategy is explained with greater depth, including its theoretical underpinning, implementation details, and when to choose it.

i) Random Communication Strategy — Extended

The random strategy is grounded in the principle that in a distributed-memory system, the 'owner-computes' rule distributes both computation and communication uniformly. Node v is owned by processor $P(v) = v \bmod p$ (or a hash function of v). When any processor discovers that v should be visited, it sends exactly one message to $P(v)$. Because ownership is based on a hash, every processor owns roughly V/p vertices, ensuring good load balance in expectation.

The strategy is 'random' in the sense that the destination of any given message is unpredictable from the sender's perspective — it depends on the graph structure, not on the processor IDs. This means the communication is an irregular all-to-all pattern. For sparse graphs, each processor sends and receives $O(E/p)$ messages per BFS level on average, but the variance can be high for power-law graphs (social networks) where high-degree hub nodes receive far more messages than average-degree nodes.

The random strategy is the default in most distributed-memory BFS implementations because it requires no special graph structure, achieves good average-case load balance, and allows messages to be batched (aggregated into one network packet per destination processor per level) to reduce per-message startup cost. When combined with message buffering, it can achieve nearly 100% network utilisation.

ii) Ring Communication Strategy — Extended

The ring strategy is designed for ring-topology interconnects (physically common in torus-based

supercomputers like IBM Blue Gene or Cray XC systems). Its correctness relies on the fact that if a message circulates through all p processors in the ring, every processor will eventually see it and the owner will process it. The message is effectively forwarded until it reaches its owner.

The implementation assigns each node v to processor $P(v)$ just as in the random strategy. But instead of direct point-to-point messaging, discovery messages travel around the ring one hop at a time. This means that on a physical ring interconnect, every communication step uses only the directly connected links — there are no long-distance hops that might contend with other traffic.

However, the $O(p)$ rounds-per-level cost is crippling for large p . If the BFS diameter is D levels, the total time is $O(D \times p \times (t_s + E_{avg} \times m \times t_w))$, which grows linearly with p rather than $1/p$. This makes the ring strategy strictly worse than the random strategy for large p unless the physical topology precisely matches the ring communication pattern and direct routing would be even more expensive.

iii) Blackboard Communication Strategy — Extended

The term 'blackboard' is a metaphor for a shared public space where any agent can write observations and any other agent can read them. In parallel computing, the 'blackboard' is the shared address space — all processors can write to and read from a common memory pool. The BFS 'visited' array and the 'next frontier' queue are the blackboard.

On modern multi-core CPUs and NUMA (Non-Uniform Memory Access) systems, the blackboard model maps directly to the hardware. All cores share an L3 cache and main memory. The critical synchronisation primitive is the atomic compare-and-swap (CAS) instruction, which is a hardware-level operation supported by all modern processors (x86 LOCK CMPXCHG, ARM STXR). When two threads simultaneously try to visit the same node, CAS guarantees exactly one of them succeeds in marking the node as visited, and the other simply discards its discovery.

On a GPU, the blackboard model maps to GPU global memory with CUDA's `atomicCAS()` function. Since GPUs have thousands of threads, the contention on hot blackboard entries (e.g., nodes adjacent to many frontier nodes) can cause serialisation. This is why GPU BFS implementations use a 'frontier queue' with atomic pushback rather than a flat visited array — each thread atomically appends to the queue only if its CAS succeeds.

The blackboard model's key advantage is zero-cost communication — a memory write is not 'sent' anywhere; it is immediately visible to all processors sharing that memory. This makes the blackboard model orders of magnitude faster than message passing for intra-node BFS. Its key limitation is that it requires physically shared memory, which in practice means a single multi-core processor or a GPU — not a cluster of independent machines.

Note: These three strategies represent a spectrum from fully distributed (random/ring, message-passing) to fully shared (blackboard). Modern HPC systems use all three levels: Blackboard within GPU SMs (shared memory), blackboard across the GPU (global memory with atomics), random/message-passing between GPUs or nodes. Understanding which strategy applies at which level of the memory hierarchy is crucial for writing high-performance graph algorithms.

Q8 a) Odd-Even Transportation in Bubble Sort with Suitable Example [6 Marks]

[REPEATED] – Full answer with complete 4-element worked example [3,1,4,2] showing all rounds, the odd/even phase structure, and $O(n)$ parallel complexity analysis is in: Unit 6 Answer Doc → May–June 2023 → Q7 a).

Q8 b) Parallel Formulation for CRCW PRAM**[6 Marks]****What is a PRAM?**

A Parallel Random Access Machine (PRAM) is a theoretical model for parallel computation designed to abstract away the messy details of real interconnect topologies, memory hierarchies, and synchronisation. In a PRAM, p processors share a single global memory, and all processors execute synchronously (in lock-step). At each step, every processor can perform a read, a write, or a computation. The PRAM model is used to reason about the inherent parallelism of algorithms independently of machine-specific concerns.

Types of PRAM and the CRCW Variant

PRAM models differ in how they handle simultaneous memory accesses by multiple processors:

EREW — Exclusive Read Exclusive Write:

Most restrictive. No two processors may read or write the same memory location simultaneously. Requires careful coordination.

CREW — Concurrent Read Exclusive Write:

Multiple processors may read the same location simultaneously, but only one may write at a time. Models read-only shared data.

CRCW — Concurrent Read Concurrent Write:

Most powerful. Multiple processors may BOTH read AND write the same location simultaneously. Write conflicts are resolved by one of three sub-models:

- COMMON CRCW: all concurrent writers must write the same value
- ARBITRARY CRCW: one arbitrarily chosen writer succeeds
- PRIORITY CRCW: the highest-priority (lowest-index) processor wins

Why CRCW is the Most Powerful Model

CRCW PRAM can solve some problems faster than EREW or CREW because it allows implicit 'reduction' through concurrent writes. A classic example is the OR of n boolean values: in CRCW, p processors can compute $\text{OR}(x_1, \dots, x_n)$ in $O(1)$ time by each writing 1 to a shared location if their value is 1 — at least one write succeeds if any $x_i = 1$.

Parallel Formulation for CRCW PRAM — Example: Finding Maximum

Finding the maximum of n numbers sequentially takes $O(n)$ time. On a CRCW PRAM with n^2 processors, we can find the maximum in $O(1)$ time:

Algorithm: Maximum of n numbers using CRCW PRAM (COMMON variant)

Input: n numbers $A[0..n-1]$

Output: $\text{max_val} = \text{maximum of } A$

Assign processor $P(i,j)$ to compare $A[i]$ with $A[j]$, for all $i \neq j$.

(Total processors needed: $n(n-1)/2 \approx n^2$ pairs)

Step 1 ($O(1)$): For each pair (i,j) where $A[i] < A[j]$:

$P(i,j)$ writes 'not_max' to shared $\text{flag}[i]$.

(Multiple writes to $\text{flag}[i]$ all write the same value — COMMON CRCW)

Step 2 ($O(1)$): Each processor $P(i,i)$ reads $\text{flag}[i]$.

If $\text{flag}[i]$ is not 'not_max': $A[i]$ is the maximum.

$P(i,i)$ writes $A[i]$ to shared 'result'.

Total time: $O(1)$ [just 2 synchronous steps!]
 Total processors: $O(n^2)$
 Total work: $O(n^2)$ [not work-optimal, but demonstrates $O(1)$ time]

For comparison:

EREW PRAM (tree-based comparison): $O(\log n)$ time, $O(n)$ processors
 Sequential: $O(n)$ time, 1 processor

Parallel Sum on CRCW PRAM

Another classic CRCW PRAM algorithm is prefix-sum (scan). The CRCW model allows the following elegant formulation:

Parallel prefix-sum on CRCW PRAM:

Phase 1 ($\log n$ steps): Build a binary tree of partial sums.

Step k ($k=1 \dots \log n$): processor i computes sum of 2^k elements ending at i , by adding sum ending at i and sum ending at $i - 2^{(k-1)}$.

Read: processor i reads $\text{value}[i]$ and $\text{value}[i - 2^{(k-1)}]$ [CONCURRENT READ]

Write: processor i writes their sum back to $\text{value}[i]$ [EXCLUSIVE WRITE]

This is actually CREW (concurrent reads are needed in each step).
 CRCW would allow further optimisation with concurrent writes for reduction-based patterns.

Time: $O(\log n)$ with n processors — cost-optimal (work = $O(n \log n)$)

Comparison of PRAM Models

PRAM Model	Concurrent Read?	Concurrent Write?	Power Level	Example Problems
EREW	No	No	Weakest	Sorting ($O(\log^2 n)$), prefix sum
CREW	Yes	No	Medium	Matrix multiply, search
CRCW (Common)	Yes	Yes (same value)	Strong	Maximum, OR, bitwise ops
CRCW (Arbitrary)	Yes	Yes (one wins)	Stronger	Graph connectivity
CRCW (Priority)	Yes	Yes (lowest ID wins)	Strongest	Pointer jumping, list ranking

Note: PRAM is a theoretical model — no real machine is a pure PRAM. Real shared-memory machines are approximations: multi-core CPUs implement CREW (concurrent reads are hardware-cached; concurrent writes require atomic instructions, which are more expensive than reads). GPUs implement CRCW in global memory via atomicCAS/atomicAdd but with significant performance cost. PRAM complexity results give lower bounds on what is achievable — if a problem requires $\Omega(\log n)$ time on a CREW PRAM, no shared-memory parallel algorithm can do better regardless of how many processors are used.

Q8 c) Distributed Computing for Document Classification

[6 Marks]

Document classification is the task of automatically assigning category labels to text documents (e.g.,

labelling news articles as 'sports', 'politics', 'technology'). It is a core problem in information retrieval, spam filtering, and content moderation. When the document corpus is too large for a single machine — think of classifying all of Twitter's daily tweets, or crawling the entire web — distributed computing is essential.

The Core Task

Formally: given a labelled training set of (document, category) pairs and an unlabelled test document, predict the most likely category for the test document. Common algorithms include Naive Bayes, Logistic Regression, SVM, and neural text classifiers (BERT, etc.). All these algorithms share a common structure: they first compute features from documents (e.g., TF-IDF term frequencies), then use those features to score against a model.

Why Distributed Computing Is Needed

Three aspects of large-scale document classification require distributed computing: the data volume (terabytes of documents that don't fit on one machine), the feature computation (building term frequency matrices requires counting words across billions of documents), and the model training (fitting a classifier on millions of documents with millions of features is computationally intensive). MapReduce / Spark frameworks provide the programming model that distributes all three tasks transparently.

Distributed Classification Using MapReduce

The MapReduce paradigm, pioneered by Google, breaks the classification pipeline into Map and Reduce stages that can run in parallel across many machines. Here is how it applies to document classification:

Stage 1: Feature Extraction (MapReduce Job 1)

Input: corpus of documents D_1, D_2, \dots, D_N (distributed across HDFS)

MAP (runs on each chunk of documents in parallel):

```
for each document  $D_i$  in my_chunk:
    tokenise  $D_i$  into words  $[w_1, w_2, \dots, w_k]$ 
    for each word  $w_j$ :
        emit ( $w_j$ , {doc_id:  $D_i$ .id, count: 1})
```

SHUFFLE (automatic): group all values by key (word)

REDUCE (runs on each unique word in parallel):

```
for each word  $w$ , list of (doc_id, count) pairs:
    compute  $\text{TF-IDF}(w, D_i) = \text{tf}(w, D_i) \times \log(N / \text{df}(w))$ 
    emit ( $D_i$ .id, {feature:  $w$ , value:  $\text{TF-IDF}(w, D_i)$ })
```

Output: feature vectors $\{\text{doc_id} \rightarrow [(\text{word}, \text{tfidf_score}), \dots]\}$

Stage 2: Model Training (MapReduce Job 2 — Naive Bayes example)

MAP:

```
for each labelled training document ( $D_i$ , label_i):
    for each feature (word, score) in  $D_i$ :
        emit (label_i, {word: word, score: score})
```

REDUCE:

```
for each label, accumulate  $P(\text{word} | \text{label}) = \text{count}(\text{word in label}) / \text{count}(\text{label})$ 
emit learned model parameters:  $P(\text{word} | \text{label})$  for all words and labels
```

Stage 3: Classification (MapReduce Job 3)

MAP (for each unlabelled document D_{test}):

load model parameters (broadcast or distributed cache)

for each candidate label L :

$\text{score}(L) = \log P(L) + \sum w \text{score}(w, D_{\text{test}}) \times \log P(w|L)$

emit ($D_{\text{test}}.\text{id}$, $\text{argmax}_L \text{score}(L)$)

No Reduce needed — each document is classified independently.

Parallelism Structure

The MapReduce approach achieves parallelism at multiple levels. In Stage 1, every document is processed independently by a different mapper — perfect data parallelism with no synchronisation needed between mappers. In Stage 2, every label (category) can be handled by a different reducer — again, perfect parallelism. In Stage 3, every test document is classified independently. The only coordination point is the shuffle phase, which redistributes intermediate key-value pairs across the network, but this is handled automatically by the MapReduce framework.

Modern Approach: Spark and Distributed ML

While MapReduce works well for batch processing, modern distributed document classification uses Apache Spark, which keeps intermediate data in memory (rather than writing to disk after each stage). Spark's MLlib library provides distributed implementations of TF-IDF, Naive Bayes, Logistic Regression, and even neural text models, all parallelised across a cluster.

- For very large corpora (CommonCrawl, social media), parameter servers (used in TensorFlow distributed, PyTorch DDP) are used to distribute model parameters across machines during neural network fine-tuning.
- Document sharding: documents are divided into shards and each shard is processed by a different worker. Within a shard, documents can be further parallelised on a GPU (batched inference).
- For real-time classification (e.g., spam filtering incoming emails), streaming frameworks like Apache Kafka + Flink classify documents as they arrive, with pre-trained models loaded into memory on each worker node.

Note: Document classification is a good exam answer because it combines multiple HPC concepts: data parallelism (processing many documents simultaneously), functional decomposition (feature extraction → model training → inference as separate distributed stages), and the MapReduce programming model. Always draw the MapReduce pipeline diagram (Map → Shuffle → Reduce) when explaining this topic.

Additional Concepts & Quick Reference

Parallel Quicksort vs Parallel Merge Sort — Which to Choose?

Both algorithms achieve $O((n/p) \log n)$ parallel time in the best case, but they differ in practice. Quicksort is cache-friendlier (in-place partitioning), but its performance depends heavily on pivot quality — a bad pivot creates load imbalance that merge sort avoids. Merge sort always makes exactly the same number of operations regardless of input order (it's deterministic), making it the preferred choice for guaranteed performance. In HPC, sample sort (a generalisation of quicksort that uses $p-1$ splitters chosen from a random sample) is often used for distributed sorting because it combines quicksort's good cache behaviour with better load balance guarantees.

Quick Revision: All Unit 6 Algorithms

Algorithm	Sequential Time	Best Parallel Time	Key Technique
Bubble Sort (odd-even)	$O(n^2)$	$O(n)$ rounds	Alternating odd/even compare-exchange
Merge Sort	$O(n \log n)$	$O((n/p) \log n)$	Binary tree merge, local sort + merge
Quicksort (recursive)	$O(n \log n)$ avg	$O((n/p) \log n)$ avg	Pivot broadcast + processor halving
BFS	$O(V+E)$	$O((V+E)/p + D \times p)$	Level-sync, atomic frontier expansion
DFS	$O(V+E)$	$O((V+E)/p + p \cdot t_{\text{steal}})$	Work stealing from stack bottom
Dijkstra	$O(V^2)$	$O(V^2/p + V \log p)$	Partitioned priority queue + all-reduce

Partitioned priority queue + all-reduce