

## 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:  $P0=3, P1=1, P2=4, P3=2$

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

$P0, P1$ : compare(3,1) →  $P0=1, P1=3$  [swapped]

$P2, P3$ : compare(4,2) →  $P2=2, P3=4$  [swapped]

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

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

$P1, P2$ : compare(3,2) →  $P1=2, P2=3$  [swapped]

$P0$  and  $P3$  have no partner → no change

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

Round 2 — Odd phase:  $P0, P1$ : compare(1,2) → no swap;  $P2, P3$ : compare(3,4) → no swap

Round 2 — Even phase:  $P1, P2$ : 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

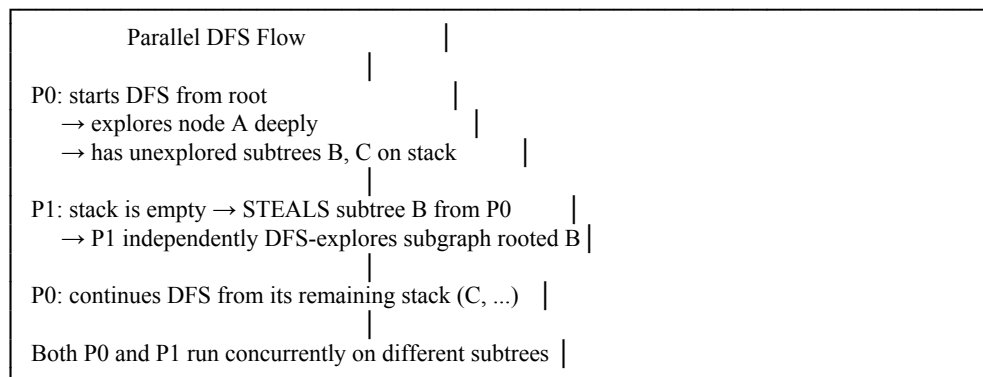
In sequential DFS, a graph  $G = (V, E)$  is explored by starting at a root, marking it visited, and recursively exploring all unvisited neighbours before backtracking. It produces a DFS tree and runs in  $O(V + E)$  time. The inherently recursive, stack-based nature of DFS makes parallelisation challenging — 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  $t_{\text{steal}}$  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. [ $O(p \log p)$  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  $\text{next\_frontier}$

// Synchronise: collect all  $\text{next\_frontier}$  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.