HPC – Unit 3: Parallel Communication (END-SEM PYQ Answers)

May–June 2023 (Paper [6004]-493)

Q1 a) One-to-All Broadcast on an 8-Node Ring (Recursive Doubling) + All-to-One Reduction [7 Marks]

Part 1 – One-to-All Broadcast on an 8-Node Ring

- One-to-all broadcast is a collective communication pattern where a single source node sends the same message to every other node. The recursive doubling (also called recursive halving) technique exploits the ring topology to halve the number of uninformed nodes in each step.
- Key idea: At each step, every node that already has the data sends it to exactly one new neighbour. The number of informed nodes doubles each step, so $\log_2(p)$ steps are enough for p nodes.
- 8-node ring, Node 0 is source. Nodes are labelled 0–7 connected in a ring (each node connected to its left and right neighbour).

Step-by-step process:

```
Step 0 (Initial):  Only node 0 has data.
            Informed: {0}

Step 1:  Node 0 sends to Node 4 (distance = p/2 = 4)
      Ring: 0--1--2--3--4--5--6--7--0
      Informed: {0, 4}

Step 2:  Node 0 sends to Node 2, Node 4 sends to Node 6
      Informed: {0, 2, 4, 6}

Step 3:  Node 0->1, Node 2->3, Node 4->5, Node 6->7
      Informed: {0,1,2,3,4,5,6,7}  [All nodes!]

Total steps: log2(8) = 3
```

Communication cost analysis:

- Each step involves one send + one receive (one-port model).
- Total steps = $\log_2(p) = \log_2(8) = 3$.
- Total time = $\log_2(p) \times (t_{startup} + m \times t_{data})$, where m = message size.

> *Note: The recursive doubling technique ensures optimal use of the ring links. In each step, the 'distance' used to pick the partner is halved from p/2 → p/4 → … → 1.*

Part 2 – All-to-One Reduction (Node 0 as Destination)

All-to-one reduction is the exact reverse of one-to-all broadcast. Every node has a partial result, and the destination (Node 0) must collect and combine all of them using a reduction operator (e.g., sum, max).

Process (reverse of broadcast):

- Step 1: Nodes 1,3,5,7 send to their respective partners (0,2,4,6). Each partner combines the received data with its own.
- Step 2: Nodes 2,6 send to 0,4. Partners combine again.

- Step 3: Node 4 sends to Node 0. Node 0 now has the complete reduced result.

```
Step 1: 1->0, 3->2, 5->4, 7->6   (combine at 0,2,4,6)
Step 2: 2->0, 6->4               (combine at 0,4)
Step 3: 4->0                     (Node 0 = final result)
```

- Cost is identical to broadcast: $O(\log_2 p)$ steps.

> *Note: All-to-one reduction is widely used in parallel summation, parallel dot products, and MPI_Reduce operations.*

Q1 b) Blocking and Non-Blocking Communication Using MPI [6 Marks]

MPI (Message Passing Interface) provides two fundamental modes of point-to-point communication:

1. Blocking Communication

A blocking call does not return until the communication operation is safe to reuse the communication buffer. The program is effectively paused until the operation completes.

- MPI_Send(buf, count, type, dest, tag, comm) – blocks until the message is sent (or buffered safely).
- MPI_Recv(buf, count, type, src, tag, comm, status) – blocks until a matching message is received.

Characteristics:

- Simple to program and reason about.
- Risk of deadlock if two processes each wait for the other to receive before sending (both blocked on MPI_Send).
- No overlap of computation and communication.

```
Process 0:  [SEND]---blocks---[continues after send]
Process 1:  [RECV]---blocks---[continues after receive]
```

> *Note: Deadlock example: if P0 does MPI_Send to P1 AND P1 does MPI_Send to P0 simultaneously, both block waiting for the other to post MPI_Recv first.*

2. Non-Blocking Communication

Non-blocking calls return immediately, allowing computation to overlap with communication. The actual data transfer happens in the background.

- MPI_Isend(buf, count, type, dest, tag, comm, &request) – initiates send, returns immediately.
- MPI_Irecv(buf, count, type, src, tag, comm, &request) – initiates receive, returns immediately.
- MPI_Wait(&request, &status) – blocks until the non-blocking operation completes.
- MPI_Test(&request, &flag, &status) – checks if the operation is done without blocking.

Characteristics:

- Enables computation-communication overlap (latency hiding).
- Eliminates most deadlock scenarios.
- Programmer must ensure buffer is not modified before the operation completes.

```
Process 0:  [MPI_Isend]--[COMPUTE]--[MPI_Wait]--[continue]
Process 1:  [MPI_Irecv]--[COMPUTE]--[MPI_Wait]--[use data]
                  ^
            Computation overlaps with communication
```

Comparison Table

Feature	Blocking	Non-Blocking
Returns when?	Operation fully complete / buffered	Immediately (operation pending)
Overlap with compute?	No	Yes
Deadlock risk?	Yes (must order carefully)	Lower risk
Complexity	Simple	More complex (need Wait/Test)
MPI functions	MPI_Send, MPI_Recv	MPI_Isend, MPI_Irecv, MPI_Wait

---

Q1 c) Prefix-Sum Operation [4 Marks]

The prefix-sum (also called scan) operation is a collective computation where each node i computes the sum (or another associative operator) of all values at nodes 0, 1, …, i.

Formally, if node i holds value x_i, after the prefix-sum, node i holds:

```
node 0 → x0
node 1 → x0 + x1
node 2 → x0 + x1 + x2
...
node p-1 → x0 + x1 + ... + x(p-1)
```

Algorithm – recursive doubling in $\log_2(p)$ steps:

- At each step k (k = 1, 2, …, $\log_2 p$), node i sends its current partial sum to node (i + 2^(k-1)) and receives from node (i - 2^(k-1)), then adds the received value to its own.

Example for 4 nodes with values [3, 1, 4, 2]:

```
Initial:  [3]  [1]  [4]  [2]
Step 1 (distance=1): each node adds left neighbour
      [3]  [4]  [5]  [6]   (node1=3+1, node2=1+4, node3=4+2)
Step 2 (distance=2): each node adds node-2 neighbour
      [3]  [4]  [8]  [10]  (node2=3+5, node3=4+6)
Final prefix sums: [3, 4, 8, 10] ✓
```

- Total steps = $O(\log_2 p)$, making it very efficient.
- Widely used in parallel sorting (e.g., counting sort), load balancing, and parallel list ranking.

> *Note: MPI_Scan is the standard MPI function that performs an inclusive prefix-sum across all processes.*

Q2 a) All-to-All Broadcast on an 8-Node Ring [7 Marks]

In all-to-all broadcast, every node is both a source and a destination — each node broadcasts its own message to every other node. After completion, each node holds p different messages.

Unlike one-to-all broadcast, each node sends a unique piece of data, so messages cannot be simply forwarded. Instead, each step involves moving a new batch of data around the ring.

Algorithm for an 8-node ring — each step, nodes shift messages one hop:

```
Initial state: Node i holds message Mi
Nodes: 0--1--2--3--4--5--6--7--0 (ring)

Step 1: Each node sends its own message to its right neighbour.
  Node 0 sends M0 to Node 1
```

```
Node 1 sends M1 to Node 2
...
Node 7 sends M7 to Node 0
After step 1, each node i has {Mi, M(i-1 mod 8)}

Step 2: Each node forwards the message it received in step 1.
Now each node i has 3 messages.

... (p-1 steps total) ...

Step 7 (last step): Each node i has all 8 messages M0..M7
```

First two steps in detail (showing node 0):

- After Step 1: Node 0 has {M0, M7} (received M7 from Node 7)
- After Step 2: Node 0 has {M0, M7, M6} (received M6 from Node 7 which got it from Node 6)
- After Step 7: Node 0 has {M0, M1, M2, M3, M4, M5, M6, M7}

Cost analysis:

- Number of steps = p - 1 = 7 for an 8-node ring.
- Each step transfers one extra message per node.
- Total communication time ≈ (p-1)(t_startup + m × t_data), where m = message size.

> *Note: This is more expensive than one-to-all broadcast (O(log p)) because all-to-all requires O(p) steps on a ring. Hypercube topologies are more efficient for this operation.*

Q2 b) Scatter and Gather Communication Operations [6 Marks]

Scatter

In scatter, a single root node distributes distinct pieces of data to all other nodes. Unlike broadcast (which sends the same message to everyone), scatter sends a different chunk to each node.

- MPI function: MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

```
Root (Node 0) has: [A | B | C | D]  (4 chunks for 4 nodes)

After MPI_Scatter:
Node 0 receives chunk A
Node 1 receives chunk B
Node 2 receives chunk C
Node 3 receives chunk D
```

- The root retains its own chunk (chunk A) and sends the rest.
- Useful in data-parallel programs to distribute work across processes.

Gather

Gather is the exact reverse of scatter. Each node sends its local data to the root, which assembles all pieces into a single buffer in rank order.

- MPI function: MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

```
Before MPI_Gather:
Node 0 has A, Node 1 has B, Node 2 has C, Node 3 has D
```

```
After MPI_Gather (root = Node 0):
  Node 0 has: [A | B | C | D]
```

- Only the root process receives the complete gathered data.
- MPI_Allgather is the variant where every process gets the complete result.

> *Note: Scatter + computation + Gather is the fundamental pattern of many parallel algorithms (e.g., parallel matrix-vector multiply).*

Q2 c) Circular Shift Operation **[4 Marks]**

A circular shift (also called a cyclic shift) is a collective communication operation where each node i sends its data to node (i + q) mod p, and receives data from node (i - q) mod p, for a given shift parameter q.

```
Example: 6-node ring, q = 2 (shift right by 2)

Before: Node 0=A, Node 1=B, Node 2=C, Node 3=D, Node 4=E, Node 5=F
After:  Node 2=A, Node 3=B, Node 4=C, Node 5=D, Node 0=E, Node 1=F

Each node i sends to (i+2) mod 6 and receives from (i-2+6) mod 6
```

- On a ring, circular shift requires only 2 concurrent message sends (left and right), since messages travel along the ring.
- On a mesh or hypercube, circular shift can be implemented efficiently using the topology.
- Key application: staggered matrix algorithms (e.g., Cannon's matrix multiplication uses circular shifts to align sub-matrices on a process mesh).

> *Note: Circular shift differs from a general personalized communication because the pattern is regular and predictable, allowing for highly optimised implementations.*

May–June 2024 (Paper [6263]-94)

Q1 a) One-to-All Broadcast on a Hypercube + Cost Calculation **[7 Marks]**

A hypercube with p = 2^d nodes is a d-dimensional topology where each node is connected to d neighbours, differing in exactly one bit of their binary address.

Algorithm – Recursive Doubling on a Hypercube

Node 0 (binary: 000…0) is the source. The broadcast proceeds across one dimension per step, using the XOR of node IDs to determine partners:

```
Example: 8-node hypercube (d=3 dimensions), source = Node 0

Step 1 (dimension 0, bit 0):
  Node 0 (000) sends to Node 1 (001)
  Informed: {000, 001}

Step 2 (dimension 1, bit 1):
  Node 0 (000) sends to Node 2 (010)
  Node 1 (001) sends to Node 3 (011)
  Informed: {000, 001, 010, 011}
```

```
Step 3 (dimension 2, bit 2):
  Node 0 sends to Node 4, Node 1 sends to Node 5,
  Node 2 sends to Node 6, Node 3 sends to Node 7
  Informed: {0,1,2,3,4,5,6,7} — All nodes!

General rule at step k: sender i sends to i XOR 2^(k-1)
```

Pseudocode:

```
for k = 1 to d:                    // d = log2(p) steps
  partner = my_id XOR (1 << (k-1))    // flip bit k-1
  if my_id has bit (k-1) = 0:         // I have data, I send
    send data to partner
  else:                               // I need data, I receive
    receive data from partner
```

Cost Calculation

- Number of steps: $\log_2(p)$ — one step per hypercube dimension.
- Each step involves exactly one send/receive pair per informed node.
- Time per step = $t_s + m \cdot t_w$, where t_s = startup latency, m = message size in bytes, t_w = per-byte transfer time.
- Total broadcast time: $T = \log_2(p) \times (t_s + m \cdot t_w)$
- For the 8-node hypercube: $T = 3(t_s + m \cdot t_w)$ — very efficient!

> *Note: The hypercube is optimal for one-to-all broadcast. The ring requires O(p) steps, but the hypercube only O(log p). This is why hypercubes and hypercube-like networks (fat-trees, butterfly networks) are popular in HPC clusters.*

Q1 b) Scatter and Gather Communication Operation [6 Marks]

[REPEATED] – See Q2 b) in May–June 2023 above for complete answer.

Q1 c) Circular Shift Operation [4 Marks]

[REPEATED] – See Q2 c) in May–June 2023 above for complete answer.

Q2 a) All-to-All Broadcast and All-to-All Reduction with Cost Analysis [7 Marks]

All-to-All Broadcast

Every node broadcasts its own unique message to all other nodes. After completion, every node holds all p messages. This is essentially p simultaneous one-to-all broadcasts.

On a ring, the algorithm works in p-1 steps, rotating messages one position per step (see Q2a in 2023 for the step-by-step ring example). On a hypercube with $d = \log_2(p)$ dimensions, all-to-all broadcast takes d steps, and at each step the message size doubles (since each node is accumulating more messages to forward).

```
Hypercube all-to-all broadcast:
Step 1: exchange messages across dimension 0
  Each node now has 2 messages
Step 2: exchange 2-message bundles across dimension 1
```

Each node now has 4 messages
...
Step d: exchange 2^(d-1) messages across dimension d
Each node now has p = 2^d messages

Cost Analysis – Hypercube

- Step k transmits 2^(k-1) × m bytes (message grows each step).
- Total time = $\Sigma_{k=1}^{d} (t_s + 2^{(k-1)} \cdot m \cdot t_w) = d \cdot t_s + (p-1) \cdot m \cdot t_w$
- This equals $\log_2(p) \cdot t_s + (p-1) \cdot m \cdot t_w$.

All-to-All Reduction

All-to-all reduction is the complement: each node i contributes a vector of p values (one for each destination), and node j receives and combines (reduces) all values destined for it from every node.

- MPI equivalent: MPI_Reduce_scatter — combines scatter and reduce.
- The algorithm is the time-reverse of all-to-all broadcast, using the same topology traversal but with accumulation instead of forwarding.
- Cost on hypercube: same as all-to-all broadcast — $\log_2(p) \cdot t_s + (p-1) \cdot m \cdot t_w$.

> *Note: All-to-all reduction is used in parallel matrix transpose, FFT butterfly stages, and neural network gradient aggregation.*

Q2 b) Blocking and Non-Blocking Communication Using MPI [6 Marks]

[REPEATED] – See Q1 b) in May–June 2023 above for complete answer.

Q2 c) Improving the Speed of Communication Operations [4 Marks]

Several algorithmic and hardware techniques can be used to speed up collective communication operations in parallel systems:

1. Pipeline Communication

Instead of sending the entire message as one block, the message is split into smaller packets that are pipelined through intermediate nodes. While the first packet is being processed at the next hop, the subsequent packets begin transmission.

- Reduces the effect of startup latency for large messages.
- Effective for one-to-all broadcast on rings and meshes.

2. Recursive Halving / Doubling

Communication rounds are structured so that the number of nodes involved doubles each round (for broadcast) or halves (for reduction), achieving $\log_2(p)$ total rounds instead of p-1.

3. Spanning Trees

Construct a spanning tree rooted at the source. For broadcast, the source sends to its tree children, which then forward to their children, and so on. The tree depth determines the number of steps (O(log p) for a binary tree).

4. All-Reduce Optimisation (Ring AllReduce)

The ring AllReduce algorithm (popular in deep learning, e.g., Horovod) splits data into p chunks. In the first phase (reduce-scatter), each node contributes a chunk and receives a partial sum. In the second phase (all-gather), each node sends the fully summed chunk to the next, ending with all nodes having the complete reduced result.

- Bandwidth: nearly 100% utilisation.
- Latency: O(p) steps but bandwidth-optimal.

5. Hardware Offloading

- Modern NICs (e.g., InfiniBand with RDMA) handle collective operations in firmware, bypassing the CPU.
- NVIDIA NCCL and Intel MPI use topology-aware algorithms tuned to the network fabric.

> *Note: The choice of algorithm depends on message size: short messages favour latency-optimised algorithms (recursive doubling), while long messages favour bandwidth-optimised algorithms (ring-based).*

Additional Concepts & Quick Reference

MPI Collective Functions Summary

MPI Function	Operation	Who has result?
MPI_Bcast	One-to-all broadcast (same data to all)	All processes
MPI_Scatter	Root sends different chunks to each process	Each process gets its chunk
MPI_Gather	Each process sends chunk to root	Root only
MPI_Allgather	Gather result goes to all processes	All processes
MPI_Reduce	All contribute, one root gets result	Root only
MPI_Allreduce	Reduce result goes to all	All processes
MPI_Reduce_scatter	Reduce then scatter result	Each process gets part
MPI_Scan	Prefix-sum across ranks	Each process i gets sum 0..i

Communication Cost Model

The standard model used in HPC to estimate communication cost is:

```
T_comm = t_s + n * t_w
where:
  t_s = message startup time (latency)
  n   = message size in words/bytes
  t_w = per-word transfer time (1/bandwidth)
```

- For a ring of p nodes: one-to-all broadcast takes $\log_2(p) \times (t_s + m \cdot t_w)$.
- For a hypercube of p nodes: same — $\log_2(p) \times (t_s + m \cdot t_w)$, but with better physical bandwidth.

> *Note: Always write the cost formula explicitly in exams — examiners expect it even if the diagram is the main deliverable.*