


# National University of Computer and Emerging Sciences, Lahore Campus

	Course Name:	Parallel and Distributing Computing	Course Code:	CS3006
	Degree Program:	BS (CS)	Semester:	Fall 2022
	Exam Duration:	60 Minutes	Total Marks:	35
	Paper Date:	10/11/22	Weight	15
	Exam Type:	Midterm II	Page(s):	4

**Student : Name:** \_\_\_\_\_ **Roll No.** \_\_\_\_\_ **Section: 7A**

**Instruction:** Attempt all questions on the question paper. Rough sheets can be used but it should not be attached. If you think some information is missing then assume it and mention it clearly.

**Question 1: For each MCQ, please encircle the correct letter option only.**

**[Marks: 8, CLO: 2]**

- i. For communication (e.g., all-to-all broadcast or prefix sum) within a hypercube, we used an expression ( $my\_id \oplus 2^i$ ) within the for loop. What was its purpose?
  - a. To decide the size of the message to be sent in this iteration
  - b. To decide the total dimensions existing within the hypercube
  - c. **To decide the ID of the neighbor with which we will exchange messages in this iteration**
  - d. To decide the ID of the farthest node with which we will exchange messages in this iteration
- ii. When OMP\_NESTED is set to FALSE, this means:
  - a. For each new level of nesting within a parallel region, a new team of threads is created
  - b. Only the master thread will execute any code within the parallel region
  - c. We cannot use any for loops within the parallel region
  - d. **For each new level of nesting within a parallel region, no new team of threads is created**
- iii. What does the `#pragma omp parallel sections` clause do?
  - a. Creates a team of threads which executes all critical sections in parallel
  - b. Creates a team of threads which executes all preceding loops in parallel
  - c. **Creates a team of threads which executes the sections in the region in parallel**
  - d. Creates a team of threads which executes the sections in the region sequentially
- iv. To parallelize traversal of a linked list within OpenMP, we may:
  - a. Use the OpenMP `parallel while` construct
  - b. **Find the size of the list, create an array of pointers that points to each list item, then use the OpenMP `parallel for` construct**
  - c. Use the OpenMP `single` clause and the `firstprivate` clause within OpenMP `parallel while`
  - d. Use the OpenMP `critical` clause
- v. All-reduce could also be considered as equivalent to:
  - a. One-to-all broadcast by process x and then all-to-one reduction by process x
  - b. All-to-all broadcast
  - c. All-to-one reduction by process 0, then one-to-all broadcast by process p-1
  - d. **All-to-one reduction by process x and then one-to-all broadcast by process x**
- vi. The *gather* communication operation is similar to *all-to-one reduction*, because:
  - a. **In both, 1 process gathers unique elements from all processes**
  - b. In both, 1 process gathers unique elements from all processes and applies an associative operator
  - c. In both, 1 process gathers unique elements from all processes then does a 1-to-all broadcast
  - d. In both, all processes send a unique element to its right neighbor

- vii. When we use the (guided, C) form of scheduling in OpenMP, loop iterations are scheduled such that:
- Each thread is always assigned a chunk size equal to C
  - Each thread is always assigned a chunk size not larger than C
  - Each thread is assigned a chunk size of size  $C^2$
  - Each thread is always assigned a chunk size of size C or greater**
- viii. We apply `#omp single` within a parallel region to make sure?
- Only the master thread executes the code block succeeding this line
  - Only one thread executes the code block succeeding this line**
  - Only one thread has access to the shared variables
  - Only p-1 threads will execute the code block succeeding this line

**Question 2: (2+3+3)**

**[Marks: 8, CLO: 3]**

Assume we have a 4\*4 matrix with the following values:

```

2  4  6  8
1  3  5  7
10 12 14 16
11 13 15 17

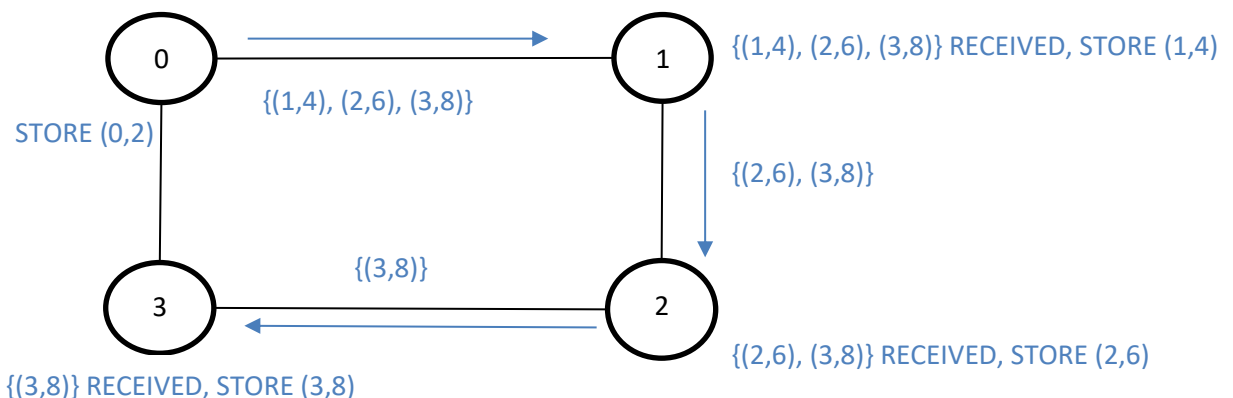
```

Assume each row is stored at different processes, row 1 (2, 4, 6, 8) is stored at process P0, row 2 at P1, row 3 at P2, and row 4 (11, 13, 15, 17) at P3. We want to apply a matrix transpose. Describe:

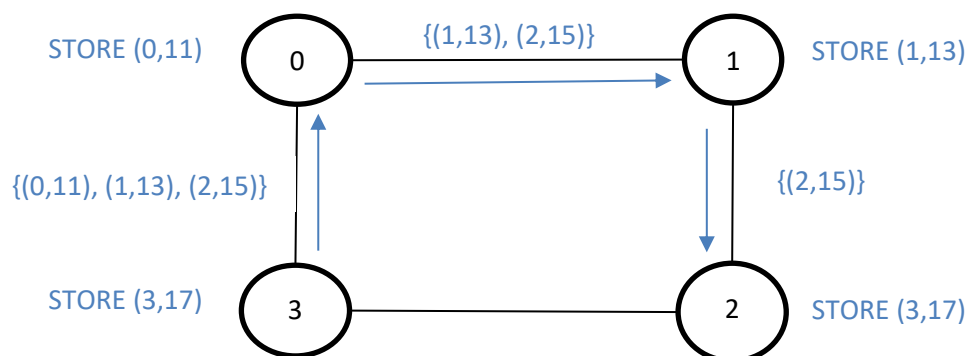
- (i) The operation that needs to take place

**Total exchange (a form of all-to-all personalized communication)**

- (ii) Draw the message originating from process P0 and show what happens at each step with this message

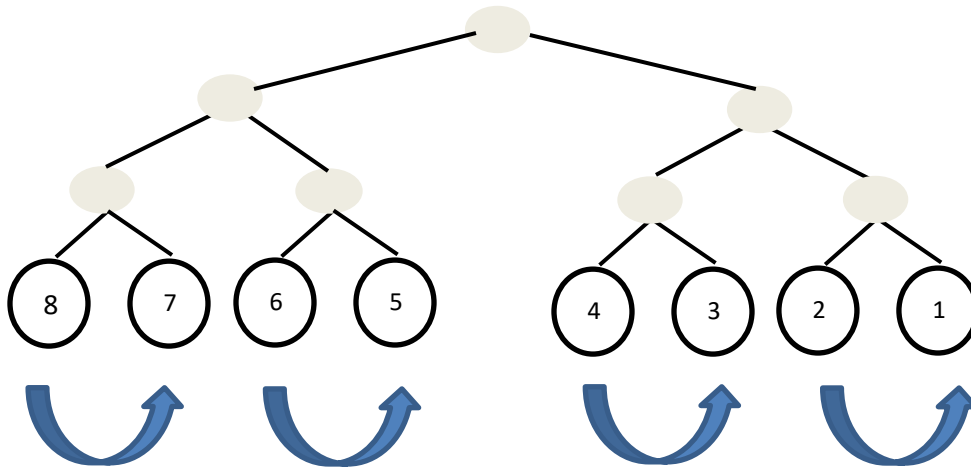


- (iii) Draw the message originating from process P3 and show what happens at each step with this message



**Question 3:****[Marks: 10, CLO: 3]**

Briefly explain the concept of “Prefix-sum”. Now consider the following complete binary tree where we have to perform the operation “Prefix-sum”. Assume that the value to be contributed by each node is equal to  $(8 - ID)$ . Show the calculation at each step and provide the final value at each node at the end of the operation.

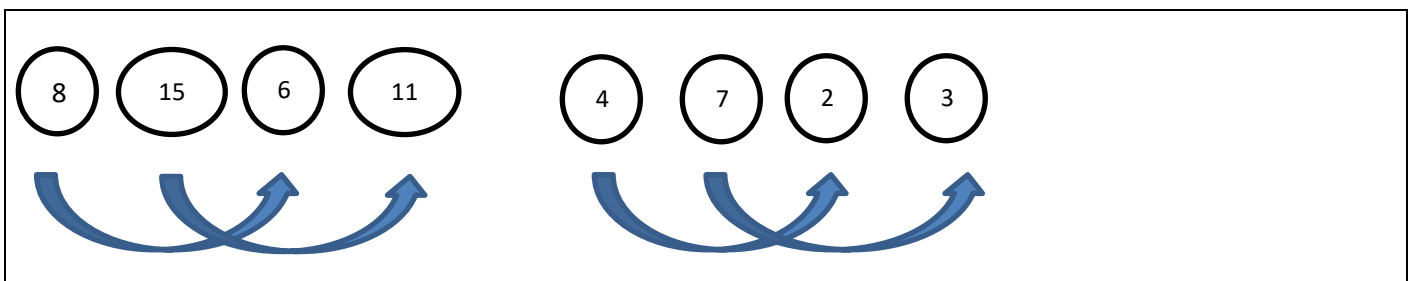
**Stage 1:**

Node 0 (value: 8) and node 1 (value: 7) exchange with each other, after which node 0 has (value: 8, stored: 7, 8) and node 1 has (value: 15, stored: 7, 8). Reasoning: since value 8 was received by node 1 from node 0 (smaller rank), it will add this to its value (7+8).

Node 2 (value: 6) and node 3 (value: 5) exchange with each other, after which node 2 has (value: 6, stored: 5, 6) and node 3 has (value: 11, stored: 5, 6). Reasoning: since value 6 was received by node 3 from node 2 (smaller rank), it will add this to its value (5+6).

Node 4 (value: 4) and node 5 (value: 3) exchange with each other, after which node 4 has (value: 4, stored: 3, 4) and node 5 has (value: 7, stored: 3, 4). Reasoning: since value 4 was received by node 5 from node 4 (smaller rank), it will add this to its value (3+4).

Node 6 (value: 2) and node 7 (value: 1) exchange with each other, after which node 6 has (value: 2, stored: 1, 2) and node 7 has (value: 3, stored: 1, 2). Reasoning: since value 2 was received by node 7 from node 6 (smaller rank), it will add this to its value (1+2).

**Stage 2:**

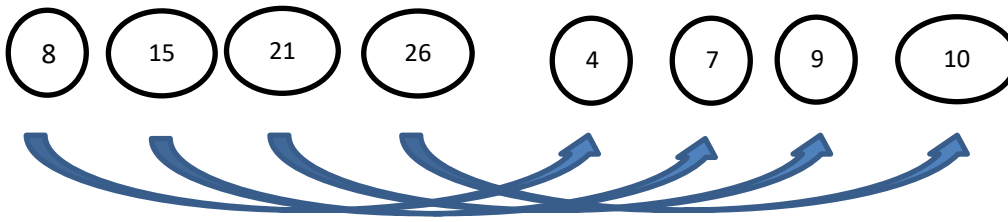
Node 0 (value: 8) and node 2 (value: 6) exchange with each other, after which node 0 has (value: 8, stored: 5, 6, 7, 8) and node 2 has (value: 21, stored: 5, 6, 7, 8). Reasoning: since (7, 8) was received by node 2 from node 0 (smaller rank), it will add this to its value (6+7+8).

Node 1 (value: 15) and node 3 (value: 11) exchange with each other, after which node 1 has (value: 15, stored: 5, 6, 7, 8) and node 3 has (value: 26, stored: 5, 6, 7, 8). Reasoning: since (7, 8) was received by node 3 from node 1 (smaller rank), it will add this to its value ( $5+6+7+8$ ).

Node 4 (value: 4) and node 6 (value: 2) exchange with each other, after which node 4 has (value: 4, stored: 1, 2, 3, 4) and node 6 has (value: 9, stored: 1, 2, 3, 4). Reasoning: since (3, 4) was received by node 6 from node 4 (smaller rank), it will add this to its value ( $2+3+4$ ).

Node 5 (value: 7) and node 7 (value: 3) exchange with each other, after which node 5 has (value: 7, stored: 1, 2, 3, 4) and node 7 has (value: 10, stored: 1, 2, 3, 4). Reasoning: since (3, 4) was received by node 7 from node 5 (smaller rank), it will add this to its value ( $1+2+3+4$ ).

Stage 3:

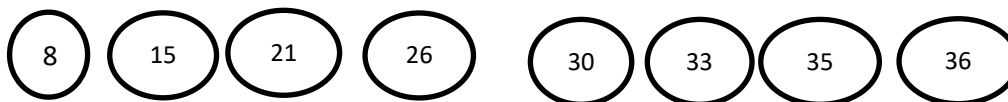


Node 0 (value: 8) and node 4 (value: 4) exchange with each other, after which node 0 has (value: 8, stored: 1 to 8) and node 4 has (value: 30, stored: 1 to 8). Reasoning: since (5, 6, 7, 8) was received by node 4 from node 0 (smaller rank), it will add this to its value ( $4+5+6+7+8$ ).

Node 1 (value: 15) and node 5 (value: 7) exchange with each other, after which node 1 has (value: 15, stored: 1 to 8) and node 5 has (value: 33, stored: 1 to 8). Reasoning: since (5, 6, 7, 8) was received by node 5 from node 1 (smaller rank), it will add this to its value ( $3+4+5+6+7+8$ ).

Node 2 (value: 21) and node 6 (value: 9) exchange with each other, after which node 2 has (value: 21, stored: 1 to 8) and node 6 has (value: 35, stored: 1 to 8). Reasoning: since (5, 6, 7, 8) was received by node 6 from node 2 (smaller rank), it will add this to its value ( $2+3+4+5+6+7+8$ ).

Node 3 (value: 26) and node 7 (value: 10) exchange with each other, after which node 3 has (value: 26, stored: 1 to 8) and node 7 has (value: 36, stored: 1 to 8). Reasoning: since (5, 6, 7, 8) was received by node 7 from node 3 (smaller rank), it will add this to its value ( $1+2+3+4+5+6+7+8$ ).



**Question 4: (5+4)****[Marks: 9, CLO: 2]**

(i) Show the output of the following program:

```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    int nums[14] = { 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

#pragma omp parallel for num_threads(5) schedule(static, 2)
    for (int j = 0; j < 14; j++) {
        nums[j] += j + 1;
        int x = omp_get_thread_num();
        cout << "At thread: " << x << " iteration: " << j << endl;
    }

    for (int i = 0; i < 14; i++) {
        cout << nums[i] << "\t";
    }
    return 0;
}
```

**Output:**

```
At thread: 0 iteration: 0
At thread: 0 iteration: 1
At thread: 1 iteration: 2
At thread: 1 iteration: 3
At thread: 2 iteration: 4
At thread: 2 iteration: 5
At thread: 3 iteration: 6
At thread: 3 iteration: 7
At thread: 4 iteration: 8
At thread: 4 iteration: 9
At thread: 0 iteration: 10
At thread: 0 iteration: 11
At thread: 1 iteration: 12
At thread: 1 iteration: 13
15  15  15  15  15  15  15  15  15  15  15  15  15  15
```

**NOTE:** The output does not necessarily look exactly like this. However, the iterations for a specific thread (e.g., thread 0) will normally be seen in chronological order. For example, thread 0 manages iterations 0, 1, 10 and 11. So we will normally see these four lines in this order from top to bottom. But between different threads, we won't necessarily see the order shown above, it is possible that thread 1 executes and returns both its iterations before any of thread 0's iterations.

(ii) Describe what we can do to protect our programs from circular deadlock in MPI, when we have multiple processes interacting with each other?

We may use `MPI_Sendrecv()`, since it handles circular deadlock by itself.

We may also use if/else logic to alternate the pattern of calling `MPI_Send()` and `MPI_Recv()` functions (differing between odd and even threads/ranks).