SUBJECT 1 – 2025

1. Example:

p.size() = 11;

nrProc = 4; (0, 1, 2, 3)

chunkSize = 2;

\*in the example, the vector length is NOT A MULTIPLE OF THE MPI PROCESSES\*

Example: p.size() = 1000

In this example, each process will get 2 elements that it shall compute => 4 \* 2 = 8 elements will be computed. This is not complete and there is a mistake since the size of the vector is 11 and thus the size of the final vector should be 11 too. We want to somehow compute the elements that are left out too. This can be done by handling the remaining elements to the master process. A for-loop can be done to compute these elements in the master process:

For(int const& v : partResults) sum += v;

For(int I = nrProc \* chunkSize; I <= p.size(); i++)

Sum += p[i] \* q[i];

Thus the response E.

2.

p.size() = 5

q.size() = 6

nrProc = 4

chunkSize = 3

partRes.size() =

SUBIECT 3 – 2025

**Exercise 1**

The issue in the computation of the product of the two polynomials using MPI distributed programming lies in the actual computation of the coefficients for the resulting polynomial. Normally, in the nested loop that begins at line 29 the index i should handle the position for each of a processes’ chunk, and the index j should handle the position in the final polynomial result. Instead in the given example j goes up until i at line 30, thus unfulfilling its job(mention myId is not used at all). For example: chunkSize=4, finalResult.size()=9 => in this scenario and considering the problematic code section, no computation will go further than position 4 in the 2 polynomials (p and q) and thus the final elements will not be reached for computation. The problem thus will be “E” (I think all coefficients are computed but wrong, so no D) and a solution would be to add an “offset” to properly align the polynomials and change the boundaries of j.

Between 28-29 => int offset = chunkSize \* myId;

30 => For(int j = 0; j <= i + offset; j++)

31 =>r[i] += p[j] + q[i + offset - j];

**Exercise 2**

The answer is (E) because if we have multiple threads that will call wait(), seeing the count is still non-zero they will of course enter the wait and be put to sleep, each one of them releasing the mutex they acquired. This would not be a problem, if in the done() function, upon reaching count = 0, the condition variable would wake up ALL the threads. However in our scenario it wakes up only one of them, resulting in a forever waiting for something that won’t execute again (done gets called exactly once). Another answer can be (A) (vedeti raspuns Gabi pe whatsapp cu ordinea executarii).  
To fix this we would need both to replace notify\_one() with notify\_all() as I said above, so that all the threads will get notified when countdown reaches 0, and also put a lock before line 10 and unlock it after line 13 to ensure that the operations in both functions are atomic and can’t result in a deadlock.

Subject 4 - 2024