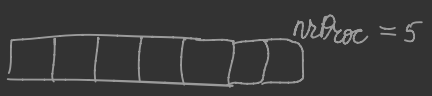
FEBRUARIE 2023 , S.1



(1)

mS(7,0,5): hL = 3, hP = 2, ch = 2, mS(4,0,3)
mS(4,0,3): hL = 2, hP = 1, ch = 1, mS(2,0,2)
mS(2,0,2): hL = 1, hP = 1, ch = 1, mS(1,0,1)
mS(1,0,1): mSLoc
mS(3,2,2): hL = 1, hP = 1, ch = 3, mS(2,2,1)
mS(?,2,1): mSLoc

A - true, because when we have a vector of size 1 and 5 processes, in the 2nd line because of the condition dataSize <= 1 it will call mergeSortLocal and all the childs will wait forever and will not receive anything

C - true, because for some processes is send more than once, and we have only one receive and some of them will not be used
D - → this problems are caused by computing the child incorrectly

(2) D - true, because if we have only one item and both dequeues reach line 23 at the same time they pop simultaneously so the items list is corrupted

E - true, because if we have 1 item in items dequeue acquires the lock, doesn't satisfy the condition items.empty(), then releases the lock and enqueue acquires the lock, and dequeue pops in exactly the same time when enqueue pushes

H - true, because it will fix the problem with the data corruption in E

FEBRUARIE 2023, S.2

S = 7 Proc = 5

① mS(7,0,5): hL = 3, hP = 3, ch = 3, mS(3,0,2)
mS(3,0,2): hL = 1, hP = 1, ch = 1, mS(1,0,1)
mS(1,0,1): mSLocal
mS(3,3,3): hL = 1, hP = 2, ch = 5 → inexistent child

B - true, because when we call mergeSort recursively (line 11), some elements of the vector will not be sorted because we call with halfLen and not with dataSize - halfLen

fixes : l. 11 : mergeSort(v + halfSize, dataSize - halfLen, myId, nrProc - halfProc)

C - true, because in some cases we will compute an inexistent child and we are trying to send to it ⇒ the program crashes before all childs are used

D - true, because for some processes is send more than once, and we have only one receive and some of them will not be used

→ this problems are caused by computing the child incorrectly

fix: l. 7 : int child = myId + (nrProc/2);

② A - true, because let's say dequeue reaches line 25 (the queue is empty) and then enqueue acquires the lock, pushes the item, notifies and then release the lock, so only now dequeue can acquire the lock, and now will wait forever because it lost enqueue's notify.

D - true, let's say we have only one element and both dequeues are trying to pop in the same time

E - true, if enqueue and dequeue are pushing and popping at the same time

F, G - true, because it will solve the issues with the data corruption and deadlocks

FEBRUARIE 2023, S.3

① E - true, because when the size of the final result is not divisible with the nb. of processes, some coefficients from the end will not be computed

for example: $p = x^3 + x^2 + x + 1$, $q = x^2 + x + 1$, nbProcesses = 4

finalSize = 6 => chunkSize = 1

=> 2 coefficients will not be computed

F - true, the same reason as above, and for the coefficients that will not be computed we will have 0, because of the resize in line 3

fixes: if (myId == nrProc - 1) {

chunkSize += (p.size() + q.size - 1) - (nrProc * chunk)

in the order: 26 - 28 - if - 27

this will fix the issues described above

② A - true, set notifies, then get acquires the lock, enters the while and will wait forever because it lost the notify

E - true, when they are called on the same thread, get reaches the wait and sleeps forever because set cannot be executed because the whole thread is sleeping

H - true, but it doesn't matter where the notify is, only to be after the lock

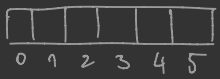
I - true, because the main idea is to have the lock before notify

FEBRUARIE 2023, S.4

p: $X^3 + X^2 + X + 1$ nrProc = 4

q: $X^2 + X + 1$

out size = 6



chunkSize = (6 + 4 - 1) / 4 => 9 / 4 = 2

① A - true, because if the out size is 6, the resize in l.9 will make the result vector of size 6, and gather will receive 8 elements, because each process sends 2 elements

fixes:

```
int chunkSize = (p.size() + q.size() - 1) / nrProc;
l.26
if (myId == nrProc - 1){
    chunkSize += (p.size() + q.size() - 1) - (nrProc * chunkSize);
}
```

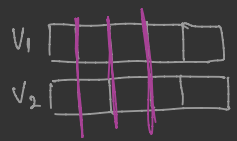
② A - true, because if get enters the while, but set is the first to acquire the lock, notifies and sets the variables, then releases the lock, then get can acquire the lock and waits forever because it will not be notified

C - true, if set gets to the line where hasValue = true, then get reaches the return, but val wasn't already initialized with v

I - true, it will fix the problem with the deadlock from point A and also the one from C

FEBRUARIE 2023, S.5

① E - true, because Scatter always sends equally sized chunks



nrProc = 3

chunkSize = 5/3 ⇒ 1

⇒ 2 elements not computed

F - true, because some elements are not computed and added to the final result

fixes: → we have to use ScatterV because it allows to send chunks of different sizes

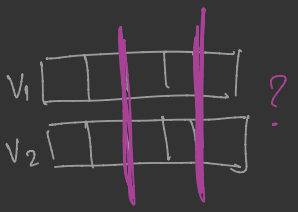
→ we will have a vector for the chunkSizes for each process and another one in which we store the starting positions for each chunk

② E - true, because if addContinuation is first and reaches l. 22, then it takes the lock, executes the instructions and releases the lock and executes the for before addContinuation takes the lock and pushes into the list. > so the for. last added will never be executed

I - true, fixes the issue described above

FEBRUARIE 2023, S.6

① A - true, because in the way we compute the chunkSize, Scatter which always sends equally sized chunks is trying to access elements that do not exist



nr.Proc = 3

chunkSize = (5 + 3 - 1) / 3 = 7/3 = 2

fixes: pad the vectors with 0 so that the vectors have the size nrProc * chunkSize

② J - true, because we have the following flow:

→ set executes line 10

→ then addContinuation reaches line 22 and executes f(val) and val is undefined

another issue fixed is:

→ set exec. l. 10 - 11

→ then addContinuation acquires the lock first reaches l. 21, releases the lock and set acquires the lock and the order of the continuations is not respected

FEBRUARIE 2023, S.7

① B - true, because myId = 0 does not send anything but it waits to receive because of the first if at line 19

E / F - true, because of the chunkSize's style of computation, some elems will not be computed & the sum will be computed incorrect

fixes: → add to the ifs at l. 19 & 23 : || myId != 0

→ add a new Receive after l. 17

```
if (myId == 0) {
    MPI_Recv(&t, 1, MPI_INT, 1, 1, MPI_COMM_WORLD,
}                                  MPI_STATUS_IGNORE);
```

② D - true, because every worker executes with the lock on it

H - true, fixes the issue above

FEBRUARIE 2022, S.2

① A – true, because we have in the first if the condition dataSize ≤ 1 and if we have 5 processes and the length of the vector is 1, neither of the 4 child processes will reiceive, and we will have a deadlock

fixes: remove "|| dataSize <= 1" from the if

② E – true, because if we have 1 item in items dequeue acquires the lock, doesn't satisfy the condition items.empty(), then releases the lock and enqueue acquires the lock, and dequeue pops in exactly the same time when enqueue pushes

H – true, because it will fix the problem with the data corruption in E

---

mS(5,0,4) → hL = 2, hP = 2, ch = 2, mS(3,0,2)

mS(3,0,2) ⇒ hL = 1, hP = 1, ch = 1, mS(2,0,1)

mS(2,0,1) ⇒ mSLocal

mS(2,2,2) ⇒ hL = 1, hP = 1, ch = 3, mS(1,2,1)

mS(1,2,1) ⇒ mSLocal

mS(1,1,1) ⇒ mSLocal

mS(1,3,1) ⇒ mSLocal

FEBRUARIE 2022, S. 3

① p 4 $x^3 + x^2 + x + 1$ (3 2 1 0)

q 3 $x^2 + x + 1$ (3 2 1 0)

$x^5 + x^4 + x^3 + x^2 + x + 1$

Proc = 4

6/4 = 1

	0	1	2	3
deg	i = 0 ⇒ 0 j = 0 r[0] += p0·q0	i = 0 ⇒ 1 j = 0, 1 r1 = p0·q1 + p1·q0	i = 0 ⇒ 2 j = 0, 1, 2 r2 = p0·q2 + p1·q1 + p2·q0	i = 0 ⇒ 3 j = 0, 1, 2, 3 r3 = p1·q2 + p2·q1 + p3·q0

j = 0 1 deg = 3, p.size = 4, q.size = 3 ⇒ 0 < 4 si 3 < 3 fals

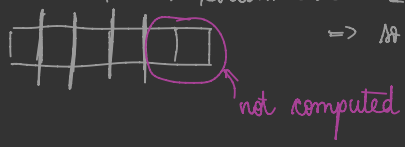
1 < 4 si 2 < 3

① E - true, because when the size of the final result is not divisible with the nb. of processes, some coefficients from the end will not be computed

for example: $p = x^3 + x^2 + x + 1$, $q = x^2 + x + 1$, nbProcesses = 4

finalSize = 6 ⇒ chunkSize = 1

⇒ 2 coefficients will not be computed



F - true, the same reason as above, and for the coefficients that will not be computed we will have 0, because of the resize in line 3

fixes: if (myId == nrProc - 1) {

chunkSize += (p.size() + q.size - 1) - (nrProc * chunk)

in the order: 26 - if - 27

this will fix the issues described above

(2) D – true, in the case when addContinuations reaches l.22 and when wants to acquire the lock it cannot because set acquires the lock first and addContinuation waits, set executes the continuations and when it finishes it releases the lock, then addContinuations acquires the lock and puts a new function in the list which will not be executed

I – true, it fixes the problem given in D & also fixes another problem in which val is undefined, hasValue is true and addContinuation executes f with an undefined value