**S3:**

**- "Local": THREADS**

**- "Distributed": MPI**

-------------------------------------------------------------------------------------------------------------------------

Before joining some threads you need to unlock the mutex otherwise, you get a deadlock

unique\_lock<mutex> lck(mtx);

closed = true;

cv.notify\_all();

lck.unlock();

for(thread& th: threads) th.join();

-------------------------------------------------------------------------------------------------------------------------

If you want to work on some items executing the program in parallel, then you need to unlock before wi() and lock right after it

unique\_lock<mutex> lck(mtx);

if(!work.empty()) {

function<void()> wi = work.front();

work.pop\_front();

lck.unlock();

wi();

lck.lock();

}

2023-7&8 B

---------------------------------------------------------------------------------------------------------------------------

unique\_lock<mutex> lck(mtx);

while(items.empty() && !isClosed) {

cv.wait(lck);

}

deadlock if you don’t put lck in wait()

---------------------------------------------------------------------------------------------------------------------------

is deadlock before close a deadlock??

- nu mai ajunge la secventa de cod care apeleaza metoda "close()", deci da, este deadlock

---------------------------------------------------------------------------------------------------------------------------

- we can assume certain method calls can be made before the statement (issue)

---------------------------------------------------------------------------------------------------------------------------

**Orice poate fi modificat TREBUIE sa se afle sub lock.**

---------------------------------------------------------------------------------------------------------------------------

**Notify trebuie sa fie sub lock (mai departe, nu conteaza neaparat unde).**

---------------------------------------------------------------------------------------------------------------------------

**"After" in cerinta inseamna ca intai se termina executia primei metode SAU se da release la lock (de exemplu, de catre un wait), dupa care se executa a doua metoda.**

---------------------------------------------------------------------------------------------------------------------------

**???Daca un wait nu se afla sub un lock, programul e gresit.???**

---------------------------------------------------------------------------------------------------------------------------

**Chiar daca e pus bine lockul in set, tot putem avea deadlock daca se apeleaza simultan get cu set: in get ajungem in while (prima linie), dupa care se executa set(). Din acel moment, wait ramane blocat in while => DEADLOCK / WAITING. (ref: 2023: 4)**

---------------------------------------------------------------------------------------------------------------------------

**Daca o variabila e atomica, NU trebuie neaparat protejata de lockuri. ("NU LE FUTE THREADURILE")**

---------------------------------------------------------------------------------------------------------------------------

Daca cealalta metoda NU are lock, se poate intra din metoda initiala in ea, chiar daca in metoda initiala e pus lock. (vezi subiect moldo)

- P.S.: "last" call to done == se face countul 0 (indiferent daca incepem de la ccount = 13, de exemplu)

---------------------------------------------------------------------------------------------------------------------------

Doar UNIQUE MUTEX da release cand se termina de executat metoda (cel "clasic" nu da release by default la terminarea executiei)

---------------------------------------------------------------------------------------------------------------------------

A mutex can be held by a single thread at a time. If a second thread tries to get the mutex, it waits until the mutex is released by the first thread.

I WANT TO COMMIT DESPAWN (ma iei de carici)

2023 - 1

2023 - 2

2023 - 3

2023 - 4 A, I

2023 - 5 E, F, G, I

2023 - 6 F, G, J

2023 - 7 D, H

2023 - 8   
ANU ASTA - A D E G I K

– MPI SHT –

|| dataSize <= 1

**the application can deadlock if the length of the vector is smaller than the number of MPI processes.**

int halfLen = dataSize / 2;

mergeSort(v+halfSize, halfLen, myId, nrProc-halfProc);

**the application can produce a wrong result if the input vector size is not a power of 2.**

int child = myId + halfProc;

int halfProc = nrProc / 2;

mergeSort(v+halfSize, dataSize-halfSize, myId, nrProc-halfProc);

mergeSort(v+halfSize, dataSize-halfSize, myId,nrProc - nrProc/2);

**some worker processes are not used if the number of processes is not a power of 2.**

int chunkSize = (p.size()+q.size()-1) / nrProc;

int totalSize = p.size() + q.size() - 1;

int chunkSize = totalSize / nrProc;

int remainingSize = totalSize - chunkSize \* nrProc;

if (myId == nrProc - 1) {

chunkSize += remainingSize;

}

This is because the partProd function divides the work among the MPI processes by splitting the output vector into chunks of equal size. If the total number of coefficients to compute is not evenly divisible by the number of processes, then some coefficients will not be computed by any process.

**some coefficients are not computed at all.**

int chunkSize = (p.size()+q.size()-1+nrProc-1) / nrProc;

int totalSize = p.size() + q.size() - 1;

int chunkSize = totalSize / nrProc;

int remainingSize = totalSize - chunkSize \* nrProc;

if (myId == nrProc - 1) {

chunkSize += remainingSize;

}

When chunkSize is calculated, it is always rounded up => it leads to MPI\_Gather() to "overfill" the allocated destination vector, which leads to memory corruption.

**the application can have memory corruption.**

int chunkSize = p.size() / nrProc;

**some terms are not added at all.**