

1.

```
#include <iostream>
#include <mpi.h>
#define MAX 20

int nprocs, myrank; double a[MAX], b[MAX], c[MAX];
MPI_Status status;
//
//init MPI

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    int value = myrank + 10; int sum = 0;
    MPI_Recv(&sum, 1, MPI_INT, (myrank - 1 + nprocs) / nprocs, 10,
MPI_COMM_WORLD, &status); sum += value;
    MPI_Send(&sum, 1, MPI_INT, (myrank + 1) % nprocs, 10, MPI_COMM_WORLD); if
    (myrank == 0) printf("%d", sum); MPI_Finalize(); return 0;
}
```

Care din următoarele afirmatii sunt adevarate?

Se executa corect si afisaza 30.

Executia programului produce deadlock pentru ca procesul de la care primesc procesul 0 nu este bine definit.

Executia programului produce deadlock pentru ca nici un proces nu poate sa trimit inainte sa primeasca.

2. Care dintre urmatoarele afirmatii sunt adevarate ? count INTEGER blocked:

CONTAINER

down do

if count > 0 then count: = count -1 **else**

blocked.add(P) --P is the current process P.state:=blocked --block process P end end **up**

do if blocked.is_empty **then** cout:=count+1 **else**

Q:=blocked.remove--select some process Q

Qstate:=ready -- unblock process Q **end**

end

aceasta varianta de implementare defineste un strong semaphore

aceasta varianta de implementare defineste un weak semaphore

aceasta varianta de implementare nu este "starvation free"

aceasta varianta de implementare este "starvation free"

Weak Semaphore = container, elementele sunt luate aleatoriu
Strong Semaphore = FIFO

Starvation – este posibila pt semafoarele de tip **weak semaphores**: Pentru ca procesul de selectie este de tip random

```
3
int main(int argc, char* argv[])
{
    int nprocs, myrank;

    MPI_Init(&argc, &argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("Salutari de la %d", myrank);

    MPI_Finalize(); printf("Program finalizat cu succes!"); return 0;
}
```

Select one or more:

1.

Salutari de la 0

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

programul finalizat cu succes

programul finalizat cu succes

2.

Salutari de la 3

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

3.

Salutari de la 2

Salutari de la 0

Salutari de la 1

programul finalizat cu succes

programul finalizat cu succes

programul finalizat cu succes

4.

Salutari de la 0

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

5.

Salutari de la 3

Salutari de la 1

Salutari de la 2

programul finalizat cu succes programul finalizat cu succes

programul finalizat cu succes

6.

Salutari de la 2

Salutari de la 0

Salutari de la 1

programul finalizat cu succes

6.Ce se poate intampla la executia programului urmator?

```
public class Main { static Object l1 = new Object(); static Object l2 = new Object(); static int a = 4, b = 4;
```

```
    public static void main(String args[]) throws Exception{    T1 r1 = new T1();    T2 r2 = new T2();
        Runnable r3 = new T1(); Runnable r4 = new T2();
        ExecutorService pool = Executors.newFixedThreadPool( 1 );
        pool.execute( r1 ); pool.execute( r2 ); pool.execute( r3 ); pool.execute( r4 ); pool.shutdown();
        while ( !pool.awaitTermination(60,TimeUnit.SECONDS)){ }
```

```
        System.out.println("a=" + a + "; b="+ b);
    }
```

```
    private static class T1 extends Thread { public void run() {    synchronized (l1) {        synchronized (l2)
    {            int temp = a;            a += b;            b += temp;
        }
    }
    }
}
```

```
    private static class T2 extends Thread { public void run() {    synchronized (l2) {        synchronized (l1)
    {            a--;            b--;
        }
    }
    }
}
```

1.

2.

4.

5.

8.

Select one or more:

1.

Nr threaduri: 5; a = 5

2.

Nr threaduri: 20; Valorile finale ale lui "a" pot fi diferite la fiecare rulare din cauza "data race"

3.

Nr threaduri: 0; a = 20

4.

Nr threaduri: 20; a = 15

5.

Nr threaduri: 20; a = 5

Pentru sablonul de proiectare paralela "Pipeline" sunt adevarate urmatoarele afirmatii:

pentru a avea o performanta cat mai buna este preferabil ca numarul de subtaskuri in care se descompune calculul sa fie cat mai mic

2.

calculul se imparte in mai multe subtask-uri care se pot executa de catre unitati de procesare diferite

3.

se obtine performanta prin paralelizare daca este nevoie de mai multe traversari ale pipeline-ului

4.

pentru a obtine o performanta cat mai buna este preferabil ca impartirea pe subtaskurile sa fie cat mai echilibrata

Apare data-race la executia programului urmator?

////////////////////////////////////

```

static int sum=0; static const int MAX=10000; void f1(int a[], int s, int e){ for(int i=s; i<e; i++) sum +=
a[i];
}
int main() { int a[MAX]; thread t1(f1, ref(a), 0, MAX/2); thread t2(f1, ref(a), MAX/2, MAX); t1.join();
t2.join();
cout<<sum<<endl; return 0;
}
////////////////////////////////////
TRUE - conform Fisier final PPD + Dind

```

Care dintre urmatoarele afirmatii sunt adevarate?
 Select one or more:

- ☒ 1. un monitor este definit de un set de proceduri
- 2. toate procedurile monitorului pot fi executate la un moment dat
- ☒ 3. un monitor poate fi accesat doar prin procedurile sale
- ☒ 4. o procedura a monitorului nu poate fi apelata simultan de catre 2 sau mai multe threaduri

Conform Fisier final PPD + Dind

Care este rezultatul executiei urmatorului program?

////////////////////////////////////

```

public class Main {
    static int value=0;
    static class MyThread extends Thread { Lock l; CyclicBarrier b; public MyThread(Lock l,
        CyclicBarrier b) { this.l = l; this.b = b;
    }
    public void run(){ try{
        l.lock();
        value+=1; b.await();
    }
    catch (InterruptedException|BrokenBarrierException e) {
        e.printStackTrace();
    }
    finally { l.unlock();}
    }
}

public static void main(String[] args) throws InterruptedException {
    Lock l = new ReentrantLock(); CyclicBarrier b = new CyclicBarrier(2); MyThread t1 =
    new MyThread( l, b ); MyThread t2 = new MyThread( l, b ); t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.print(value);
}
}

```


//

Select one or more:

1. se termina si afiseaza 1 ☒ 2. nu se termina
3. se termina si afiseaza 2 ☒ conform Fisier final PPD + Dind

Care dintre urmatoarele tipuri de comunicare MPI suspenda executia programului apelant pana cand comunicatia curenta este terminata?

Select one:

1. Asynchronous ☒ 2. Blocking
3. Nici una dintre variantele de mai sus
4. Nonblocking ☒ conform Fisier final PPD + Dind

Cate threaduri se folosesc la executia urmatorului kernel CUDA?

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    ...
}
int main()
{
    int M= 16, N=8;
    ...
    VecAdd<<< M , N >>>(A, B, C);
    ...
}
```

Select one or more:

- ☒ 1. 128
2. 16
3. 8

☒ conform Fisier final PPD + Dind

Consideram executia urmatorului program MPI cu 3 procese.

//

```
int main(int argc, char *argv[]){ int nprocs, myrank; MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    int value = myrank*10; int sum=0;
```

```

        MPI_Recv(&sum, 1, MPI_INT, (myrank-1+nprocs)/nprocs, 10, MPI_COMM_WORLD,
&status); sum+=value;
        MPI_Send(&sum, 1, MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD); if (myrank
==0) printf("%d", sum); MPI_Finalize( ); return 0;
}

```

////////////////////////////////////

Care dintre urmatoarele afirmatii sunt adevarate?

Select one or more:

1. se executa corect si afiseaza 30
- ☐ 2. executia programului produce deadlock pentru ca procesul de la care primeste procesul 0 nu este bine definit
- ☒ 3. executia programului produce deadlock pentru ca nici un proces nu poate sa trimita inainte sa primeasca **conform Fisier final PPD + Dino** este posibil ca si 2 sa fie corect?

Consideram urmatorul program MPI care se executa cu 3 procese.

////////////////////////////////////

```

int main(int argc, char *argv[] ) { int nprocs, myrank;
    int i; int *a, *b;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    a = (int *) malloc( nprocs * sizeof(int)); b = (int *) malloc( nprocs* nprocs * sizeof(int)); for(int
i=0;i<nprocs; i++) a[i]=nprocs*myrank+i;

    if (myrank>0) MPI_Recv(b, nprocs*(myrank+1), MPI_INT, (myrank-1), 10,
MPI_COMM_WORLD, &status);
        MPI_Send(b, nprocs*(myrank+1), MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD);
    if (myrank==0) MPI_Recv(b, nprocs*nprocs, MPI_INT, (myrank-1), 10, MPI_COMM_WORLD,
&status);

    MPI_Finalize( ); return 0;
}

```

////////////////////////////////////

Intre ce perechi de procese se realizeaza comunicatia si in ce ordine se realizeaza comunicatiile

Select one or more:

1. (0->1), (1-2), (2->0) in ordine aleatorie
2. (2->1), (1->0), (0->2) in ordine aleatorie
- ☒ 3. (0->1) urmata de (1-2) urmata de (2->0)
4. (2->1), urmata de (1->0), urmata de (0->2)
5. comunicatiile nu sunt bine definite pentru ca nu se realizeaza corect perechile (sender, receiver) **conform Fisier final PPD + Dino**

La ce linie se creeaza/distrug thread-urile:

```
#include <stdio.h>
#include <omp.h>

void main(){
    int i,k;      int N = 3;

    int A[3][3] = {{1,2,3},{5,6,7},{8,9,10}};      int B[3][3] = {{1,2,3},{5,6,7},{8,9,10}};      int C[3][3] ;

    omp_set_num_threads(9);

    #pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)      for (i=0; i<N; i++) {
    for (k=0; k<N; k++) {
        C[i][j] = (A[i][k] + B[i][k]);
    }
    }
}
```

1. Creează: 17, distrug 18 2. Creează: 4, distrug 19
(X) 3. Creează: 14, distrug 20
4. Creează: 12, distrug 20

conform Document final PPD

La ce linie se creeaza/distrug thread-urile:

```
#include <stdio.h>
#include "omp.h"

void main() {
    int i, t, N = 12;      int a[N], b[N], c[N];

    for (i=0; i<N; i++) a[i] = b[i] = 3;

    omp_set_num_threads(3);

    #pragma omp parallel shared(a,b,c) private(i,t) firstprivate(N)
        #pragma omp single      t = omp_get_thread_num();

        #pragma omp sections
        {
            #pragma omp section
            {
                for (i=0; i<N/3; i++) {      c[i] = a[i] + b[i] + t;
            }
        }

        #pragma omp section
}
```

```

        {
            for (i=N/3; i<(N/3)*2; i++) {
                c[i] = a[i] + b[i] + t;
            }
        }

#pragma omp section
        {
            for (i=(N/3)*2; i<N; i++) {
                c[i] = a[i] + b[i]
+ t;
            }
        }
    }
}

```

1. Creează: 10, distrug 36
2. Creează: 16, distrug 36
3. Creează: 4, distrug 34
- (X) 4. Creează: 12, distrug 36

conform Dino

Care sunt variabilele shared, respectiv variabilele private, in regiunea paralela:

1.

Shared: a, b, c, i, t

2.

Shared: a, b, c / private: i, N, t (conform excel doc)

3.

Shared: a, b, c / private: i, t

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
void main(){    int i,k;    int N = 3;
```

```
    int A[3][3] = {{1,2,3},{5,6,7},{8,9,10}};    int B[3][3] = {{1,2,3},{5,6,7},{8,9,10}};    int C[3][3] ;
```

```
    omp_set_num_threads(9);
```

```
    #pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)    for (i=0; i<N; i++) {
    for (k=0; k<N; k++) {
```

```
        C[i][j] = (A[i][k] + B[i][k]);
```

```

    }
}

```

Care sunt variabilele shared, respectiv variabilele private: (X) 1. Shared: A, B, C, N / private: i, k

2. Shared: C / private: i, k, A, B, N

3. Shared: A, B, C / private: i, k, N

conform conform Fisier final PPD + Dino Cate thread-uri se vor crea:

Cate core-uri exista pe CPU

9 + 1 main

3 + 1 main

8 + 1 main (din excel fara cod?)

Se considera executia urmatoarei program MPI cu 2 procese

```

int main(int argc, char *argv[]){ int nprocs, myrank; MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    int value = myrank*10;
    if (myrank ==0) MPI_Recv( $value, 1, MPI_INIT, 0, 10, MPI_COMM_WORLD, &status); if
    (myrank ==1) MPI_Send( $value, 1, MPI_INIT, 0, 10, MPI_COMM_WORLD); if (myrank ==0)
    printf("%d", value); MPI_Finalize( ); return 0;
}

```

Select one or more:

1. programul nu se termina pentru ca nu sunt bine definite comunicatiile

2. programul se termina si afiseaza valoarea 0
- (X) 3. programul se termina si afiseaza valoarea 10

conform Dino

Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore binar? (Se ignora timpul de creare procese, distributie date, comunicatie, iar timpul necesar operatiei de adunare se considera egal cu 1.)

Select one or more:

- (X) 1. [1 , 1024 , 1, 1024]
2. [10 , 102.4 , 0.1, 10240]
- (XX) 3. [1 , 102.4 , 10, 102.4]
4. [10 , 102.4 , 10.24, 1024]

(X) - conform Fisier final PPD

(XX) - conform Dino

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
void main() {
```

```
    int i,j,k,t;        int N=4;
```

```
    int A[4][4] = { {1,2,3,4},{5,6,7,8}, {8,9,10,11}, {1,1,1,1}};    int B[4][4] = { {1,2,3,4},{5,6,7,8},
{8,9,10,11}, {1,1,1,1}};    int C[4][4] = ;
```

```
    omp_set_num_threads(3);
```

```
    #pragma omp parallel shared(A,B,C) private(i,j,k,t) firstprivate(N) {
```

```
        #pragma omp for schedule(dynamic)        for (i=0; i<N; i=i+1){        t =
omp_get_thread_num();
```

```
            for (j=0; j<N; j=j+1) {
```

```
                C[i][j]=0.;
```

```
                for(k=0; k<N; k=k+1) {
```

```
                    C[i][j] += A[i][k] * B[k][j] + t;
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

Cate thread-uri se vor crea:

1. Cate core-uri exista pe CPU (X)
2. 3 + 1 main
3. 15 + 1 main

4. 2 + 1 main

(X) - conform Dino

(A C E L E R A T I A am spus)

Acceleratia unui aplicatii paralele se defineste folosind urmatoarea formula:

(Se considera:

T_s = Complexitatea-timp a variantei secventiale

T_p = complexitatea-timp a variantei paralele

p = numarul de procesoare folosite pentru varianta paralela.

Select one or more:

(XX) 1. T_s/T_p

(X) 2. $T_s/(p \cdot T_p)$

3. T_p/T_s

4. $p \cdot T_s/T_p$

(X) - conform Fisier final PPD

(XX) - conform Dino

Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore binar? (Se ignora timpul de creare procese, distributie date, comunicatie, iar timpul necesar operatiei de adunare se considera egal cu 1.) Select one or more:

1.

[1 , 1024 , 1, 1024] (fisier final)

2.

[10 , 102.4 , 0.1, 10240]

3.

[1 , 102.4 , 10, 102.4]

4.

[10 , 102.4 , 10.24, 1024]

```
#pragma omp parallel for for (i=1; i < 10; i++)
```

```
{
```

```
    factorial[i] = i * factorial[i-1];
```

```
}
```

Avem parte de data race in exemplul de mai sus ?

1. Fals, deoarece fiecarui thread ii vor fi asociate task-uri independente astfel incat nu este posibila o suprapunere in calcule.
2. Adevarat, pentru ca paralelizarea for este dinamica daca nu se specifica explicit (X)
3. Adevarat, pentru ca exista posibilitatea ca un thread sa modifice valoarea factorial[i-1] in timp ce alt thread o foloseste pentru actualizarea elementului factorial[i] conform Fisier final PPD + Dinc

Corespunzator clasificarii Flynn arhitecturile de tip cluster se incadreaza in clasa Select one or more:

1. MISD
2. SIMD
3. MIMD (conform Fisier final PPD)
4. SISD

Un program paralel este optim din punct de vedere al costului daca:

Select one or more:

1. timpul paralel este de acelasi ordin de marime cu timpul secvential
- (X) 2. timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential
3. acceleratia inmultita cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential conform Fisier final PPD + Dinc

Overhead-ul in programele paralele se datoreaza:

Select one or more:

- (X) 1. timpului necesar crearii threadurilor/proceselor
 - (X) 2. timpului de asteptare datorat sincronizarii (X) 3. partitionarii dezechilibrate in taskuri
 4. interactiunii interproces
 5. timpului necesar distributiei de date
 6. calcul in exces (repetat de fiecare proces/thread)
- conform Fisier final PPD + Dinc

Consideram urmatorul program MPI care trebuie completat in zona specificata de comentariul "COD de COMPLETAT".

Cu care dintre variantele specificate rezultatul executiei cu 3 procese va fi 0 1 2 3 4 5 6 7 8

```
int main(int argc, char argv[]) {      int nprocs, myrank;
    int i; int *a, *b;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    a = (int *) malloc( nprocs * sizeof(int));
    b = (int *) malloc( nprocs* nprocs * sizeof(int)); for(int i=0;i<nprocs; i++) a[i]=nprocs*myrank+i;

    /*
        COD de COMPLETAT
    */

    if (myrank==0) for(i=0; i<nprocs*nprocs; i++) printf(" %d", b[i]);

    MPI_Finalize( ); return 0;
}
```

1.
MPI_Gather(a, nprocs, MPI_FLOAT, b, nprocs, MPI_FLOAT, 0, MPI_COMM_WORLD);

(X) 2.
for (i = 0; i < nprocs; i++) b[i+nprocs*myrank] = a[i];
if (myrank>0) MPI_Recv(b, nprocs*(myrank+1), MPI_INT, (myrank-1), 10, MPI_COMM_WORLD, &status);
MPI_Send(b, nprocs*(myrank+1), MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD);
if (myrank==0) MPI_Recv(b, nprocs*nprocs, MPI_INT, (myrank-1), 10, MPI_COMM_WORLD, &status);

3.
if (myrank>0)
 MPI_Send(a, nprocs, MPI_INT, 0, 10, MPI_COMM_WORLD); else { for (i = 1; i < nprocs; i++) b[i] = a[i]; for (i = 1; i < nprocs; i++)
 MPI_Recv(b + i * nprocs, nprocs, MPI_INT, i, 10, MPI_COMM_WORLD, &status);
}

conform Fisier final PPD + Dino

Conform legii lui Amdahl, acceleratia este limitata de procentul(fractia) partii secventiale a unui program. Daca pentru un caz concret avem procentul partii secventiale egal cu 25% cat este acceleratia maxima care se poate obtine (cf legii lui Amdahl)?

Select one or more:

1. 75
2. 25

(X) 3. 4

conform Dino

```
#include "omp.h"

void main() {
    int i, t, N = 12;    int a[N], b[N], c[N];

    for (i=0; i<N; i++) a[i] = b[i] = 3;

    omp_set_num_threads(3);

    #pragma omp parallel shared(a,b,c) private(i,t) firstprivate(N)
        #pragma omp single                t = omp_get_thread_num();

        #pragma omp sections
        {
            #pragma omp section
            {
                for (i=0; i<N/3; i++) {
                    c[i] = a[i] + b[i] + t;
                }
            }

            #pragma omp section
            {
                for (i=N/3; i<(N/3)*2; i++) {
                    c[i] = a[i] + b[i] + t;
                }
            }

            #pragma omp section
            {
                for (i=(N/3)*2; i<N; i++) {
                    c[i] = a[i] + b[i] + t;
                }
            }
        }
}
```

Are programul de mai sus o executie determinista ?

1. Nu, pentru ca nu vom obtine acelasi rezultat de ori cate ori am rula programul contand ordinea de executie a thread-urilor.

(X) 2. Da, pentru ca vom obtine acelasi rezultat de ori cate ori am rula programul chiar daca programul se va executa paralel.

3. Da, pentru ca block-urile de tipul section vor fi executate secvential si nu in paralel.

conform Dino

```
#include <stdio.h>
#include "omp.h"
```

```

void main(){    int i,k;    int N = 3;

    int A[3][3] = {{1,2,3},{5,6,7},{8,9,10}};    int B[3][3] = {{1,2,3},{5,6,7},{8,9,10}};    int C[3][3] ;

    omp_set_num_threads(9);

    #pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)    for (i=0; i<N; i++) {
for (k=0; k<N; k++) {
        C[i][j] = (A[i][k] + B[i][k]);
    }
}
}

```

Apelul pragma omp parallel for din exemplul de mai sus paralelizeaza executia ambelor structuri for?

- (X) 1. Fals
- (XX) 2. Adevarat
- 3. Depinde de versiunea compilatorului folosita
- (X) - conform Fisier final PPD
- (XX) - conform Dino

Ce se poate intampla la executia programului urmator?

```

//////////////////////////////// public class Main {
    static Object l1 = new Object(); static Object l2 = new Object(); static int a = 4, b = 4;

    public static void main(String args[]) throws Exception{
        T1 r1 = new T1();    T2 r2 = new T2();
        Runnable r3 = new T1(); Runnable r4 = new T2();
        ExecutorService pool = Executors.newFixedThreadPool( 1 ); pool.execute( r1 );
        pool.execute( r2 ); pool.execute( r3 ); pool.execute( r4 ); pool.shutdown();
        while ( !pool.awaitTermination(60,TimeUnit.SECONDS)){ }

        System.out.println("a=" + a + "; b="+ b);
    }

    private static class T1 extends Thread { public void run() { synchronized (l1) { synchronized
        (l2) { int temp = a; a += b; b += temp;
        }
        }
    }
}

    private static class T2 extends Thread { public void run() { synchronized (l2) { synchronized
        (l1) {
            a--; b--;
        }
    }
}
}

```

```
}  
}  
////////////////////////////////////////////////////////////////
```

Select one or more:

- ☒ 1. nu poate apareea deadlock
- ☐ 2. se afiseaza : a=14; b=14
- ☒ 3. se afiseaza : a=13; b=13
- ☐ 4. se afiseaza : a=9; b=9
- ☐ 5. se afiseaza : a=12; b=12

Conform Dino

Rezultatul executiei este nedeterminist (conform Fisier final PPD)

Care dintre afirmatiile urmatoare sunt adevarate?

Select one or more:

- ☐ 1. Partionarea prin descompunere functionala conduce in general la aplicatii cu scalabilitate mai buna decat partitionarea prin descompunerea domeniului de date.
- ☐ 2. Scalabilitatea unei aplicatii paralele este determinata de numarul de taskuri care se pot executa in paralel.
- ☒ 3. Daca numarul de taskuri care se pot executa in paralel creste liniar odata cu cresterea dimensiunii problemei atunci aplicatia are scalabilitate buna.

Conform Dino

Pentru sablonul de proiectare paralela "Pipeline" sunt adevarate urmatoarele afirmatii:

☒ 1. pentru a avea o performanta cat mai buna este preferabil ca numarul de subtaskuri in care se descompune calculul sa fie cat mai mic

☒ 2. calculul se imparte in mai multe subtask-uri care se pot executa de catre unitati de

procesare diferite

☒ 3. se obtine performanta prin paralelizare daca este nevoie de mai multe traversari ale

pipeline-ului

☒ 4. pentru a obtine o performanta cat mai buna este preferabil ca impartirea pe subtaskurile sa fie cat mai echilibrata

(X) - conform Fisier final PPD

(XX) - conform Dino

Cate thread-uri vor fi create (ce exceptia thr Main) si care este rezultatul afisat de programul de mai jos?

```
//////////////////////////////// public class Main { public static
void main(String[] args) throws InterruptedException { AtomicNr a = new AtomicNr(5);
```

```
    for (int i = 0; i < 5; i++) {
        Thread t1 = new Thread()->{ a.Add(3); });
        Thread t2 = new Thread()->{ a.Add(2); });
        Thread t3 = new Thread()->{ a.Minus(1); });
        Thread t4 = new Thread()->{ a.Minus(1); });
    t1.start(); t2.start(); t3.start(); t4.start(); t1.join(); t2.join(); t3.join(); t4.join();
    }
    System.out.println("a = " + a);
    }
};
```

```
class AtomicNr{ private int nr;
    public AtomicNr(int nr){ this.nr = nr;}

    public void Add(int nr) { this.nr += nr;}
    public void Minus(int nr){ this.nr -= nr;}

    @Override
    public String toString() { return "" + this.nr;}
};
```

```
////////////////////////////////
```

Select one or more:

1. Nr threaduri: 5; a = 5
- ☒ 2. Nr threaduri: 20; Valorile finale ale lui "a" pot fi diferite la fiecare rulare din cauza "data race"
3. Nr threaduri: 0; a = 20
4. Nr threaduri: 20; a = 15
5. Nr threaduri: 20; a = 5

Conform Fisier final PPD + Dind

Apare data-race la executia programului urmator?

```
////////////////////////////////
```

```
public class Test { static int value=0;
static class MyThread extends Thread{ public void run() { value++; }
}
public static void main(String[] args) throws InterruptedException {    MyThread t1 = new MyThread();
MyThread t2 = new MyThread();    t1.start(); t2.start();    t1.join(); t2.join();
    System.out.print(value);
}
}
```

a) DA (conform Fisier final PPD)

b) NU

Consideram urmatoarea schita de implementarea pentru un semafor:

```
count: INTEGER blocked: CONTAINER
down do if count > 0 then count := count - 1
    else
        blocked.append(P)      -- P is current process
        P.state := blocked     -- blocked process P end
    end up do if blocked.is_empty then count := count + 1
    else
        Q := blocked.remove    -- selected some process Q
        Q.state := ready       -- unblock process Q end
    end
```

Care dintre urmatoarele afirmatii sunt adevarate ?

Select one or more:

1. aceasta varianta de implementare defineste un "strong-semaphor" (semafor puternic)
- ☒ 2. aceasta varianta de implementare defineste un "weak-semaphor" (semafor slab)
- ☒ 3. aceasta varianta de implementare nu este "starvation-free"
4. aceasta varianta de implementare este "starvation-free"

☒ Conform Dino

Se considera paralelizarea sortarii unui vector cu n elemente prin metoda "merge-sort" folosind sablonul Divide&impera.

In conditiile in care avem un numar nelimitat de procesoare, se poate ajunge la un anumit moment al executie la un grad maxim de paralelizare egal cu Select one or more:

1. $\log n$
- ☒ 2. $n / \log n$
- ☒ 3. n

☒ Conform Dino

☒ Conform Fisier final PPD

Arhitecturile UMA sunt caracterizate de:

Select one or more:

1. identificator unic pentru fiecare procesor
- ☒ 2. acelasi timp de acces pentru orice locatie de memorie

☒ Conform Fisier final PPD + Dino

(U R M A T O R U L U I am zis) urmatorului program se va executa cu 3 procese. Ce valoare se va afisa?

////////////////////////////////////

```
int main(int argc, char *argv[]) { int nprocs, myrank; int i, value=0; int *a, *b;
```

```

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) { a = (int*) malloc(nprocs * sizeof(int)); for(int i=0;i<nprocs; i++) a[i]=i+1;
}
b = (int *) malloc( sizeof(int));
MPI_Scatter(a, 1, MPI_INIT, b, 1, MPI_INT, 0 ,MPI_COMM_WORLD); b[0] += myrank;
MPI_Reduce(b, &value, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD); if( myrank == 0)
{ printf("value = \"%d \n", value); } MPI_Finalize( ); return 0;
}
////////////////////////////////////

```

Select one or more:

- ☒ 1. 9
- ☒ 2. 6
- ☐ 3. 12
- ☒ Conform Dino
- ☒ Conform Fisier final PPD

2.
 - a.
 - b.**
3.
 - a. Scadere**
 - b. Crestere
 - c. Nu este legatura
4.
 - a.**
 - b.
5. "Context Switch" este mai costisitor pentru:
 - a. Procese**
 - b. Threaduri
6.
 - a. Da**
 - b. Nu
7.
 - a.
 - b.**
8.
 - a.**
 - b.

9. Granularitatea unei aplicatii paralele este
- Definite ca dimensiunea minima a unei unitati secventiale dintr-un program, exprimata in numar de instructiuni**
 - Este determinate de numarul de taskuri rezultate prin descompunerea calculului
 - Se poate aproxima ca fiind raportul din timpul total de calcul si timpul total de comunicare
10. Granularitatea unei aplicatii paralele este de droit sa fie
- Mica
 - Mare**
11. Granularitatea unui sistem paralel este de droit sa fie
- Mica**
 - Mare
12. Eficienta unui program paralel care face suma a 2 vectori de dimensiune n folosind p procese este:
- Maxim 1**
 - Minim 1
 - Egala cu p
13. Costul unei aplicatii paralele este optim daca
- $C=O(T_s \cdot \log p)$
 - $C=O(T_s)$**
 - $C=O(\Omega(T_s))$
14. Un semafor care stocheaza procesele care asteapta intr-o multime, se numeste:
- Strong Semaphore (semafor puternic)
 - Weak Semaphore (semafor slab)**
 - Semafor binar
15. Livelock descrie situatia in care:
- Un grup de procese/threaduri nu progreseaza datorita faptului ca isi cedeaza reciproc executia**
 - Un grup de procese/threaduri nu progreseaza datorita faptului ca isi blocheaza reciproc executia
 - Un grup de procese/threaduri nu progreseaza datorita faptului ca nu se termina niciunul
16. Fata de Monitor, Semaforul este o structura de sincronizare:
- De nivel inalt
 - De nivel jos**
- c. De acelasi nivel



2. Care dintre urmatoarele variante pot fi rezultatul afisat al executiei urmatorului program MPI (mpirun -np 3 hello)?

```
int main(int argc, char **argv)
{
    int namelen, myid, numprocs;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    printf( "Process%d/%d; ", myid, numprocs);
    MPI_Finalize();
    printf( "Regards!" );
    return 0;
}
```

(0/3 Points)

- ☒ Process0/3;Process1/3;Process2/3;Regards!
- ☐ Process0/3;Process1/3;Process2/3;Regards!Regards!Regards! ✓
- ☐ Process2/3;Process1/3;Process0/3;Regards!Regards!Regards! ✓
- ☐ Process1/3;Process3/3;Process2/3;Regards!Regards!Regards!
- ☒ Process1/3;Process0/3;Process2/3;Regards!

1. Este posibil ca urmatorul cod MPI sa produca deadlock?

```
int main(int argc, char *argv[]) {
    // var declaration... init....
    int tag = 10;
    if (rank == 0) {
        dest = source = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    }
    else if (rank == 1){
        dest = source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    // ... finalizare
}
```

- (3/3 Points)
- ☐ DA
 - ☒ Nu ✓

```
#pragma omp parallel shared(chunk,a) private(i,tid) num_threads(
```

```
{
    tid = omp_get_thread_num();
    indx = indx + chunk * tid;
    for (i = indx; i < indx + chunk; i++)
        a[i] = tid + 1;
}
for (i = 0; i < n; ++i)    cout << a[i] << " ";
```

- ☐ 0 0 1 1 1 2 2 2 3 3 3
- ☒ 0 0 0 1 1 2 2 3 3 0 0 ✓
- ☐ 1 1 1 2 2 2 3 3 3 0 0

1. Un semafor care stocheaza procesele care asteapta intr-o multime, se numeste: *
(2/2 Points)

- ☐ Strong Semaphore (semafor puternic)
- ☒ Weak Semaphor (semafor slab) ✓
- ☐ Semafor binar

2. Livelock descrie situatia in care *
(-/2 Points)

- ☒ un grup de procese/threaduri nu progreseaza datorita faptului ca isi cedeaza reciproc executia
- ☐ un grup de procese/threaduri nu progreseaza datorita faptului ca isi blocheaza reciproc executia
- ☐ un grup de procese/threaduri nu progreseaza datorita faptului ca se nu se termina niciunul

3. Fata de Monitor, Semaforul este o structura de sincronizare *
(2/2 Points)

- ☐ de nivel mai inalt
- ☒ de nivel mai jos ✓
- ☐ de acelasi nivel

1.Ce este un monitor? Dati exemplu in Java.

Un monitor poate fi considerat un tip abstract de dată (poate fi implementat ca si o clasa) care constă din:

- un set permanent de variabile ce reprezintă resursa critică,
- un set de proceduri ce reprezintă operații asupra variabilelor și
- un corp (secvență de instrucțiuni).

Exemplu:

```
Object lock = new Object();
synchronized (lock) {
    // critical section
}
:
synchronized type m(args) {
    // body
}
• echivalent
type m(args) {
    synchronized (this) {
```

```

        // body
    }
}

```

- Corpul este apelat la lansarea 'programului' și produce valori inițiale pentru variabilele-monitor (cod de initializare).
- Apoi monitorul este accesat numai prin procedurile sale.

2. Faceti schita unu program mpi ce rezolva adunarea a doua matrici de nxn. Calculati costul, complezitatea timp, acceleratia si eficienta. Este solutia aleasa optima d.p.d.v. al costului? PAS PAS

3. Ce este granularitatea unui program? Cum este granularitatea aplicatiei "embarrassingly parallel programs"?(paralelizarea triviala)

Granularitatea("grain size") este un parametru calitativ care caracterizeaza atat – sistemele paralele cat si – aplicatiile paralele.

Granularitatea aplicatiei se defineste ca dimensiunea minima a unei unitati secventiale dintr-un program, exprimata in numar de instructiuni. – Prin unitate secventiala se intelege o parte programin care nu au loc operatii de sincronizare sau comunicare cu alte procese.

Granularitatea se referă la mărimea task-ului în comparație cu timpul necesar comunicației și sincronizării datelor. Granularitatea paralelizarii triviale poate fii coarse-grained sau fine-grained.

Fine-grain Parallelism: – Relatively small amounts of computational work are done between communication events – Low computation to communication ratio – Facilitates load balancing

– Implies high communication overhead and less opportunity for performance enhancement

Coarse-grain Parallelism: – Relatively large amounts of computational work are done between communication/synchronization events – High computation to communication ratio – Implies more opportunity for performance increase – Harder to load balance efficiently

Subiectul 4 de teorie:

1. Var cond

– O abstractizare care permite sincronizarea conditionala;

Operatii: wait; signal ; [broadcast]

– O variabila conditionala C este asociata cu o variabila de tip Lock – m

- Thread t apel wait =>

– suspenda t si il adauga in coada lui C + deblocheaza m (op atomica)

- Atunci cand t isi reia executia m se blocheaza

- Thread v apel signal =>

– se verifica daca este vreun thread care asteapta si il activeaza

Legatura cu monitor: – Variabile conditionale pot fi asociate cu lacatul unui monitor (monitor lock);

- Permit threadurilor sa astepte in interiorul unei sectiuni critice eliberand lacatul monitorului.

```

class CV {
    Semaphore s, x;
    Lock m;
    int waiters = 0;
    public CV(Lock m)
    { // Constructor
        this.m = m;
        s = new Semaphore();
        s.count = 0;
        s.limit = 1;
        x = new Semaphore();
        x.count = 1; x.limit = 1;
    } // x protejeaza accesul la variabila 'waiters'

```

2. Costul + o schita parca pentru un program cu n numere si procese nu mai stiu sigur.

Costul se defineste ca fiind produsul dintre timpul de executie ,si numarul maxim de procesoare care se folosesc: $C_p(n) = t_p(n) \cdot p$.

• O aplica= paralela este op4ma din punct de vedere al costului, daca valoarea acestuia este egala, sau este de acelasi ordin de marime cu =mpul celei mai bune variante secven=ale; • aplica=a este eficienta din punct de vedere al costului daca $C_p = O(t_1 \log p)$.

Costul unui sistem paralel (algoritm +sistem) • Cost = p x TP

SCHITA ^ - asa am gasit , nu stiu sigur daca este ok.

Adunare n numere cu p procesoare (ambele sunt puteri ale lui 2) • Fiecarui procesor dintre cele p ii sunt atribuite n / p procesoare virtuale. • Primii $\log n - \log p$ din cei $\log n$ pasi ai algoritmului original se simuleaza folosind p procesoare in $\Theta((n/p)(\log n - \log p)) = \Theta((n/p) \log(n/p))$ • Urmatorii $\log p$ pasi nu necesita nici par77onare (p noduri – p procesoare) • $T_p = \Theta((n/p) \log(n/p) + \log p)$ • $C = O(n \log n)$, • $T_s = \Theta(n) \Rightarrow$ Sistemul paralel nu este cost op=mal

3.race condition si zona critica

Race condition are loc atunci cand la executie exista interactiune intre threaduri/procese si rezultatul depinde de interleaving, pot fi extrem de greu de depistat.

• O sectiune de cod care conduce la race conditions se numeste critical section (sectiune critica).

```
public class Counter {  
    protected long count = 0;  
    public void add(long value){  
        this.count = this.count + value;  
    }  
}
```

Soluții: o atomicizarea zonei critice o dezactivarea preemptiei în zona critică o secvențializarea accesului la zona critică

Sub 2 teorie:

1. sablonul divide et impera, exemple, gradul de paralelism

Divide&impera este bine cunoscuta din dezvoltarea algoritmilor secventiali. O problema este impartita in doua sau mai multe subprobleme. Fiecare dintre aceste subprobleme este rezolvata independent si rezultatele lor sunt combinate pentru a se obtine rezultatul final.

Exemple: sortare prin interclasare:

Procedure interSort(A, n)

```
    if (n > 1) then  
        imparte(A, n, A0, n0, A1, n1);  
        in parallel  
            interSort(A0, n0),  
            interSort(A1, n1)  
        end in parallel  
        combina(A0, n0, A1, n1, A, n);  
    end if
```

cautare paralela

Function CautaParalel(A, n, x)

```
    if (n > m) then  
        imparte(A, n, A[0], n[0], A[1], n[1], A[2], n[2]);  
        for i = 0, 2 in parallel do  
            c[i] ← CautaParalel(A[i], n[i], x);  
        end for  
        if (c[0] != -1) then  
            CautaParalel ← c[0];  
        end if  
    end if
```

```

else
    if (c[1] != -1) then
        CautaParalel ← n[0] + c[1];
    else
        if (c[2] != -1) then
            CautaParalel ← n[0] + n[1] + c[2];
        else
            CautaParalel ← -1;
        end if
    end if
end if
else
    CautaParalel ← CautaSecvential(A, n, x);
end if

```

2. dedalocks pe threads, procese

• Deadlock – situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecare proces asteapta dupa o resursa care este retinuta de alt proces care la randul lui asteapta dupa alta resursa.

```

public class TreeNode {
    TreeNode parent = null;
    List children = new ArrayList();
    public synchronized void addChild(TreeNode child)
    { if(! this.children.contains(child))
      { this.children.add(child);
        child.setParentOnly(this);
      } }
    public synchronized void addChildOnly(TreeNode child)
    { if(!this.children.contains(child)
      { this.children.add(child); } }
    public synchronized void setParent(TreeNode parent){ this.parent = parent; parent.addChildOnly(this);
    } public synchronized void setParentOnly(TreeNode parent){ this.parent = parent; } }

```

3. wait notify notifyAll in java

Wait() suspenda threadul si deblocheaza operatia atomica

notify() deblocheaza un proces arbitrar.

Java "monitors" nu sunt starvation-free – notify() deblocheaza un proces arbitrar.

notifyAll() trezesc toate firele care așteaptă pe monitorul acestui obiect

• Apelurile metodelor notify() si notifyAll() nu se salveaza in cazul in care nici un thread nu asteapta atunci cand sunt apelate.

Asfel semnalul notify()se pierde.

Acest lucru poate conduce la situatii in care un thread asteapta nedefinit, pentru ca mesajul corespunzator de notificare se pierde.

Sub 3:

1. Semafoare

Semaforul este primitive de sincronizarea de nivel inalt. Inventata de E.W. Dijkstra in 1965 .Este caracterizat de o variabila count=v(s) (val semafor) si 2 operatii V(s)/up si P(s)/down.

– P(s) –este apelată de către procese care doresc să acceseze o regiune critică pt a obține acces.

• Efect: - incercarea obtinerii accesului procesului apelant la secțiunea critică si

decrementarea valorii.

- dacă v(s) <= 0 , procesul ce dorește execuția secțiunii critice așteaptă

– $V(s)$

- Efect : incrementarea valorii semaforului.
- se apelează la sfârșitul secțiunii critice și semnifică eliberarea acesteia pt. alte procese.

Semafor Binar • Valoarea semaforului poate lua doar valorile 0 și 1 Valoarea \Rightarrow poate fi de tip Boolean

Dacă semaforul se folosește fără a se menține o evidență a proceselor care așteaptă intrarea în secțiunea critică nu se poate asigura starvation-free

Un semafor 'slab' se poate defini ca o pereche $\{v(s), c(s)\}$ unde: $-v(s)$ este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese. $-c(s)$ o mulțime de așteptare la semafor - conține referințe la procesele care așteaptă la semaforul s . + Operațiile $P(s)/down$ și $V(s)/up$

Un semafor 'puternic' se poate defini ca o pereche $\{v(s), c(s)\}$ unde: $-v(s)$ este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese. $-c(s)$ o coadă de așteptare la semafor - conține referințe la procesele care așteaptă la semaforul s (FIFO). + Operațiile $P/down$ și V/up

2. Eficiența și alegerea la o problemă din aia trivială

Aproximarea numărului π Un calcul de tip Monte Carlo se realizează pentru aproximarea numărului π prin următoarea metodă: Se consideră un cerc de rază egală cu unitatea înscris într-un pătrat.

Complexitatea acestui algoritm este $O(n)$, unde n este numărul de puncte generate aleator de fiecare componentă. Varianta secvențială a acestui calcul are complexitatea $O(np)$, și prin urmare acest algoritm este un algoritm foarte eficient: $Ep(n) \approx 1$.

3. Scalabilitate

Principala proprietate a sistemelor cu memorie distribuită, care le avantajează față de cele cu memorie comună, este scalabilitatea Scalabilitate aplicativă: abilitatea unui program paralel să obțină o creștere de performanță proporțională cu numărul de procesoare și dimensiunea problemei.

Scalabilitatea măsura modul în care se schimbă performanța unui anumit algoritm în cazul în care sunt folosite mai multe elemente de procesare.

- Scalabilitatea unui sistem paralel este o măsură a capacității de a livra o accelerare cu o creștere liniară în funcție de numărul de procesoare folosite.

Sub6:

1. Distribuție date și distribuție funcțională

Există două strategii principale de partitionare: – descompunerea domeniului de date și – descompunerea funcțională.

În funcție de acestea putem considera aplicații paralele bazate pe: – descompunerea domeniului de date – paralelism de date, și – aplicații paralele bazate pe descompunerea funcțională.

Descompunerea domeniului de date • Este aplicabilă atunci când domeniul datelor este mare și regulat. Ideea centrală este de a divide domeniul de date, reprezentat de principalele structuri de date, în componente care pot fi manipulate independent. • Apoi se partitionează operațiile, de regulă prin asocierea calculelor cu datele asupra cărora se efectuează. • Astfel, se obține un număr de activități de calcul, definite de un număr de date și de operații.

Descompunerea funcțională este o tehnică de partitionare folosită atunci când aspectul dominant al problemei este funcția, sau algoritmul, mai degrabă decât operațiile asupra datelor. • Obiectivul este descompunerea calculelor în activități de calcul cât mai fine. • După crearea acestora se examinează cerințele asupra datelor. • Focalizarea asupra calculelor poate revela uneori o anumită structură a problemei, de unde oportunități de optimizare, care

nu sunt evidente numai din studiul datelor. • In plus, ea are un rol important ca si tehnica de structurare a programelor.

Exista mai multe tehnici de partitionare a datelor, care pot fi exprimate si formal prin functii definite pe multimea indicilor datelor de intrare cu valori in multimea indicilor de procese. • Cele mai folosite tehnici de partitionare sunt prin "taiere" si prin "incretire" care corespund distributiilor liniara si ciclica. Distributia liniara in curs este si cea –distributie bloc

2.acceleratie + legea lui Ambhdal

Acceleratia("speed-up"), notata cu Sp , este definita ca raportul dintre timpul de executie al celui mai bun algoritm serial cunoscut, executat pe un calculator monoprosesor si timpul de executie al programului paralel echivalent, executat pe un sistem de calccul paralel. Daca se noteaza cu ts timpul de executie al programului serial, iar tp timpul de executie corespunzator programului paralel, atunci: $Sp(n) = ts(n) / tp(n)$. (1.1) Numarul n reprezinta dimensiunea datelor de intrare, iar p numarul de procesoare folosite.

(Legea lui Amdahl) Fie $\alpha (0 \leq \alpha \leq 1)$ proportia operatiilor din algoritm care se executa secvential (fractia lui Amdahl). Atunci:

- Partea seriala a algoritmului se executa in timpul ats .
- Partea paralela a algoritmului se executa in timpul $(1-\alpha)ts / p$.
- Intregul algoritm se executa in timpul $tp = ats + (1-\alpha)ts / p$.
- Acceleratia relativa este $RSp = (1 + \alpha(p-1))^{-1}$ care nu poate depasi α^{-1} (Legea lui Amdhal)

3.client server vs peer to peer

Peer-to-peer se bazeaza pe sharingul de date si fisiere pe o retea de useri interconectati, reteaua e mica deci nu exista server in schimb fiecare user actioneaza ca si client si server In acelasi timp. Dezavantajul este securitatea deoarece oricine poate intra daca are parola si ca datele sunt mai instabile depinzand de fiecare membru al retelei in parte. Avantajul este uzul minim de resurse Client/server se bazeaza cu un server unde sunt stocate fisierele si parolele iar accesul la date este regulat de catre un administrator de retea, de aceea sunt mai sigure, dezavantajul este ca ele pot fi foarte scumpe ca resurse.

s5:

1.Bariere de sincronizare exemplu in mpi

O bariera de sincronizare este un mecanism de baza in sincronizarea globala. Este introdusa in punctul in care fiecare proces trebuie sa le astepte pe celelalte, iar executia se reia doar dupa ce toate procesele au atins bariera.

Exemplu:

MPI_Barrier

MPI_Barrier (comm)

MPI_BARRIER (comm,ierr)

2.sistemele flynn si ce tip de sistem crederi ca e CUDA?explicati .

Clasificarea Flynn Michael J. Flynn în 1966 • SISD: sistem cu un singur flux de instructiuni și un singur flux de date; • SIMD: sistem cu un singur flux de instructiuni și mai multe fluxuri de date; • MISD: sistem cu mai multe fluxuri de instructiuni și un singur flux de date; • MIMD: cu mai multe fluxuri de instructiuni și mai multe fluxuri de date.

Ce este CUDA? • Compute Unified Device Architecture" • o platforma de programare paralela-> • Arhitectura care foloseste GPU pt calcul general – permite cresterea performantei • Released by NVIDIA in 2007 • Model de programare – Bazat pe extensii C / C++ - pt a permite 'heterogeneous programming' – API pt gestionarea device-urilor, a memoriei etc.

CUDA foloseste SIMD pt ca poate sa scrie si sa citeasca din memorie direct , pe cand in GPU ai nevoie sa o uploadezi inainte de a fi accesata.

În al doilea rând, atât SIMD, cât și GPU-urile sunt rău la codul extrem de fragil, însă SIMD e mai puțin rău. Acest lucru se datorează faptului că GPU-urile grupează mai multe fire (un "warp") sub un singur dispecer de instrucțiuni

3. ce este granularitatea? exemplu aplicatie cu granularitatea ideala=1 & exemplu in mpi
Nu stiu la exemplu aplicatie cu granularitate ideala si mpi.

1.

```
#include <iostream>
#include <mpi.h>
#define MAX 20
int nprocs, myrank;
double a[MAX], b[MAX], c[MAX];
MPI_Status status;
//
//init MPI
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    int value = myrank + 10;
    int sum = 0;
    MPI_Recv(&sum, 1, MPI_INT, (myrank - 1 + nprocs) / nprocs, 10, MPI_COMM_WORLD, &status);
    sum += value;
    MPI_Send(&sum, 1, MPI_INT, (myrank + 1) % nprocs, 10, MPI_COMM_WORLD);
    if (myrank == 0) {
        printf("%d", sum);
    }
    MPI_Finalize();
    return 0;
}
```

Care din următoarele afirmatii sunt adevarate?

- Se executa corect si afisaza 30.
- Executia programului produce deadlock pentru ca procesul de la care primeste procesul 0 nu este bine definit.
- Executia programului produce deadlock pentru ca nici un proces nu poate sa trimit inainte se primeasca.

2. Care dintre urmatoarele afirmatii sunt adevarate ?

count INTEGER

blocked: CONTAINER

down

do

if count > 0 then


```

        count: = count -1
    else
        blocked.add(P) --P is the current process
        P.state:=blocked --block process P
    end
end
up
do
    if blocked.is_empty then
        cout:=count+1
    else
        Q:=blocked.remove--select some process Q
        Qstate:=ready -- unblock process Q
    end
end
end

```

1. aceasta varianta de implementare defineste un strong semaphore
2. aceasta varianta de implementare defineste un weak semaphore
3. aceasta varianta de implementare nu este "starvation free"
4. aceasta varianta de implementare este "starvation free"

Weak Semaphore = container, elementele sunt luate aleatoriu

Strong Semaphore = FIFO

Starvation – este posibila pt semafoarele de tip **weak semaphores**: Pentru ca procesul de selectie este de tip random

```

3. int main(int argc, char* argv[])
{
    int nprocs, myrank;
    MPI_Init(&argc, &argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Salutari de la %d",myrank);
    MPI_Finalize();
    printf("Program finalizat cu succes!");
    return 0;
}

```

Select one or more:

- 1.
- Salutari de la 0
- Salutari de la 1
- Salutari de la 2

programul finalizat cu succes
programul finalizat cu succes
programul finalizat cu succes

2.

Salutari de la 3
Salutari de la 1
Salutari de la 2
programul finalizat cu succes

3.

Salutari de la 2
Salutari de la 0
Salutari de la 1
programul finalizat cu succes
programul finalizat cu succes
programul finalizat cu succes

4.

Salutari de la 0
Salutari de la 1
Salutari de la 2
programul finalizat cu succes

5.

Salutari de la 3
Salutari de la 1
Salutari de la 2
programul finalizat cu succes
programul finalizat cu succes
programul finalizat cu succes

6.

Salutari de la 2
Salutari de la 0
Salutari de la 1
programul finalizat cu succes

4. Ce se poate intampla la executia programului urmator?

```
public class Main {  
    static Object l1 = new Object();  
    static Object l2 = new Object();  
    static int a = 4, b = 4;
```

```
    public static void main(String args[]) throws Exception{  
        T1 r1 = new T1(); T2 r2 = new T2();  
        Runnable r3 = new T1(); Runnable r4 = new T2();  
        ExecutorService pool = Executors.newFixedThreadPool( 1 );  
        pool.execute( r1 );  
        pool.execute( r2 );
```

```

        pool.execute( r3 );
        pool.execute( r4 );
        pool.shutdown();
        while ( !pool.awaitTermination(60,TimeUnit.SECONDS)) {

        }
        System.out.println("a=" + a + "; b="+ b);
    }

```

```

private static class T1 extends Thread {
    public void run() {
        synchronized (l1) {
            synchronized (l2) {
                int temp = a;
                a += b;
                b += temp;
            }
        }
    }
}

```

```

private static class T2 extends Thread {
    public void run() {
        synchronized (l2) {
            synchronized (l1) {
                a--;
                b--;
            }
        }
    }
}

```

Callable -> returneaza

Runnable -> NU

Select one or more:

1. se afiseaza : a=9; b=9
2. se afiseaza : a=13; b=13
3. se afiseaza : a=12; b=12
- 4.

nu poate aparea deadlock

5.

se afiseaza : a=14; b=14

5. Cate thread-uri vor fi create (cu exceptia thread Main) si care este rezultatul afisat de programul de mai jos?

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        AtomicNr a = new AtomicNr(5);
        for (int i = 0; i < 5; i++) {
            Thread t1 = new Thread()->{ a.Add(3); });
            Thread t2 = new Thread()->{ a.Add(2); });
            Thread t3 = new Thread()->{ a.Minus(1); });
            Thread t4 = new Thread()->{ a.Minus(1); });
            t1.start();
            t2.start();
            t3.start();
            t4.start();

            t1.join();
            t2.join();
            t3.join();
            t4.join();
        }
        System.out.println("a = " + a);
    }
};
```

```
class AtomicNr {
    private int nr;
    public AtomicNr(int nr){ this.nr = nr;}
    public void Add(int nr) { this.nr += nr;}
    public void Minus(int nr){ this.nr -= nr;}
}
```

```
@Override
public String toString() { return "" + this.nr;}
};
```

Select one or more:

1.

Nr threaduri: 5; a = 5

2.

Nr threaduri: 20; Valorile finale ale lui "a" pot fi diferite la fiecare rulare din cauza "data race"

3.

Nr threaduri: 0; a = 20

4.

Nr threaduri: 20; a = 15

5.

Nr threaduri: 20; a = 5

6. Pentru sablonul de proiectare paralela "Pipeline" sunt adevarate urmatoarele afirmatii:

5. pentru a avea o performanta cat mai buna este preferabil ca numarul de subtaskuri in care se descompune calculul sa fie cat mai mic

6. calculul se imparte in mai multe subtask-uri care se pot executa de catre unitati de procesare diferite

7. se obtine performanta prin paralelizare daca este nevoie de mai multe traversari ale pipeline-ului

8. pentru a obtine o performanta cat mai buna este preferabil ca impartirea pe subtaskurile sa fie cat mai echilibrata

7. Apare data-race la executia programului urmator?

```
static int sum=0;
```

```
static const int MAX=10000;
```

```
void f1(int a[], int s, int e) {  
    for(int i=s; i<e; i++) {  
        sum += a[i];  
    }  
}
```

```
}
```

```
int main() {
```

```
int a[MAX];
```

```
thread t1(f1, ref(a), 0, MAX/2);
```

```
thread t2(f1, ref(a), MAX/2, MAX);
```

```
t1.join(); t2.join();
```

```
cout<<sum<<endl;
```

```
return 0;
```

```
}
```

9. Adevărat

10. Fals

8. Care dintre urmatoarele afirmatii sunt adevarate?

Select one or more:

- 1. un monitor este definit de un set de proceduri
- 2. toate procedurile monitorului pot fi executate la un moment dat
- 3. un monitor poate fi accesat doar prin procedurile sale
- 4. o procedura a monitorului nu poate fi apelata simultan de catre 2 sau mai multe threaduri

Monitor

- Un monitor poate fi considerat un tip abstract de dată (poate fi implementat ca si o clasa) care constă din:
 - un set permanent de variabile ce reprezintă resursa critică,
 - un set de proceduri ce reprezintă operații asupra variabilelor și
 - un corp (secvență de instrucțiuni).
 - Corpul este apelat la lansarea 'programului' și produce valori inițiale pentru variabilele-monitor (cod de initializare).
 - Apoi monitorul este accesat numai prin procedurile sale.

9. Care este rezultatul executiei urmatorului program?

```
public class Main {
    static int value=0;
    static class MyThread extends Thread {
        Lock l; CyclicBarrier b;
        public MyThread(Lock l, CyclicBarrier b) {
            this.l = l; this.b = b;
        }
        public void run() {
            try {
                l.lock();
                value+=1;
                b.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            } finally {
                l.unlock();
            }
        }
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
```

```

    Lock l = new ReentrantLock();
    CyclicBarrier b = new CyclicBarrier(2);
    MyThread t1 = new MyThread(l, b );
    MyThread t2 = new MyThread(l, b );
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.print(value);
  }
}

```

Select one or more:

1. se termina si afiseaza 1
2. nu se termina
3. se termina si afiseaza 2

Explicatie: lock

10. Care dintre urmatoarele tipuri de comunicare MPI suspenda executia programului apelant pana cand comunicatia curenta este terminata?

Select one:

1. Asynchronous
2. Blocking
3. Nici una dintre variantele de mai sus
4. Nonblocking

11. Cate threaduri se folosesc la executia urmatorului kernel CUDA?

```

__global__ void VecAdd(float* A, float* B, float* C)
{
    ...
}
int main()
{
    int M= 16, N=8;
    ...
    VecAdd<<< M , N >>>(A, B, C);
    ...
}

```

Select one or more:

1. 128

2. 16

3. 8

12. Consideram executia urmatorului program MPI cu 3 procese.

```
int main(int argc, char *argv[]) {
    int nprocs, myrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    int value = myrank*10;
    int sum=0;
    MPI_Recv(&sum, 1, MPI_INT, (myrank-1+nprocs)/nprocs, 10, MPI_COMM_WORLD,
    &status);
    sum+=value;
    MPI_Send(&sum, 1, MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD);
    if (myrank ==0) {
        printf("%d", sum);
    }
    MPI_Finalize( );
    return 0;
}
```

Care dintre urmatoarele afirmatii sunt adevarate?

Select one or more:

1. se executa corect si afiseaza 30

2. executia programului produce deadlock pentru ca procesul de la care primeste procesul 0 nu este bine definit

3. executia programului produce deadlock pentru ca nici un proces nu poate sa trimita inainte sa primeasca

13. Consideram urmatorul program MPI care se executa cu 3 procese.

```
int main(int argc, char * argv[]) {
    int nprocs, myrank;
    int i;
    int * a, * b;
    MPI_Status status;
    MPI_Init( & argc, & argv);
    MPI_Comm_size(MPI_COMM_WORLD, & nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, & myrank);
    a = (int * ) malloc(nprocs * sizeof(int));
    b = (int * ) malloc(nprocs * nprocs * sizeof(int));
    for (int i = 0; i < nprocs; i++) {
        a[i] = nprocs * myrank + i;
```



```

}
if (myrank > 0) {
    MPI_Recv(b, nprocs * (myrank + 1), MPI_INT, (myrank - 1), 10,
    MPI_COMM_WORLD, & status);
}
MPI_Send(b, nprocs * (myrank + 1), MPI_INT, (myrank + 1) % nprocs, 10,
MPI_COMM_WORLD);
if (myrank == 0) {
    MPI_Recv(b, nprocs * nprocs, MPI_INT, (myrank - 1) daca era aici 2 era ok, 10,
    MPI_COMM_WORLD, &status);
}
MPI_Finalize();
return 0;
}

```

Intre ce perechi de procese se realizeaza comunicatia si in ce ordine se realizeaza comunicatiile

Select one or more:

1. (0->1), (1-2), (2->0) in ordine aleatorie
2. (2->1), (1->0), (0->2) in ordine aleatorie
3. (0->1) urmata de (1-2) urmata de (2->0)
4. (2->1), urmata de (1->0), urmata de (0->2)
5. comunicatiile nu sunt bine definite pentru ca nu se realizeaza corect perechile (sender, receiver)

14. La ce linie se creeaza/distrug thread-urile:

```

11. #include <stdio.h>
12. #include <omp.h>
13.
14. void main() {
15.     int i, k;
16.     int N = 3;
17.
18.     int A[3][3] = { { 1, 2, 3 }, { 5, 6, 7 }, { 8, 9, 10 } };
19.     int B[3][3] = { { 1, 2, 3 }, { 5, 6, 7 }, { 8, 9, 10 } };
20.     int C[3][3];
21.
22.     omp_set_num_threads(9);
23.
24.     #pragma omp parallel for private(i, k) shared(A, B, C, N) schedule(static)
25.     for (i = 0; i < N; i++) {
26.         for (k = 0; k < N; k++) {
27.             C[i][j] = (A[i][k] + B[i][k]);

```

```
28.     }
29. }
30. }
```

1. Creează: 17, distrug 18
2. Creează: 4, distrug 19
3. Creează: 14, distrug 20
4. Creează: 12, distrug 20

15. La ce linie se creeaza/distrug thread-urile:

```
31. #include <stdio.h>
32. #include "omp.h"
33.
34. void main() {
35.     int i, t, N = 12;
36.     int a[N], b[N], c[N];
37.
38.     for (i = 0; i < N; i++) a[i] = b[i] = 3;
39.
40.     omp_set_num_threads(3);
41.
42.     #pragma omp parallel shared(a, b, c) private(i, t) firstprivate(N)
43.     #pragma omp single
44.     t = omp_get_thread_num();
45.     #pragma omp sections
46. {
47.     #pragma omp section
48. {
49.         for (i = 0; i < N / 3; i++) {
50.             c[i] = a[i] + b[i] + t;
51.         }
52.     }
53.     #pragma omp section {
54.         for (i = N / 3; i < (N / 3) * 2; i++) {
55.             c[i] = a[i] + b[i] + t;
56.         }
57.     }
58.
59.     #pragma omp section
60. {
61.         for (i = (N / 3) * 2; i < N; i++) {
62.             c[i] = a[i] + b[i] + t;
```

63. }
 64. }
 65. }
 66. }

1. Creează: 10, distrug 36
2. Creează: 16, distrug 36
3. Creează: 4, distrug 34
4. Creează: 12, distrug 36

15.2. Care sunt variabilele shared, respectiv variabilele private, in regiunea paralela a codului de mai sus

67. Shared: a, b, c, i, t
 68. Shared: a, b, c / private: i, N, t
 69. Shared: a, b, c / private: i, t

16. Care sunt variabilele shared, respectiv variabilele private:

1. Shared: A, B, C, N / private: i, k
 2. Shared: C / private: i, k, A, B, N
 3. Shared: A, B, C / private: i, k, N

```
#include <stdio.h>
#include <omp.h>
void main(){
int i,k;
int N = 3;
int A[3][3] = {{1,2,3},{5,6,7},{8,9,10}};
int B[3][3] = {{1,2,3},{5,6,7},{8,9,10}};
int C[3][3] ;
omp_set_num_threads(9);
#pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)
for (i=0; i<N; i++) {
for (k=0; k<N; k++) {
C[i][i] = (A[i][k] + B[i][k]);
}
}
}
```

16.2. Cate thread-uri se vor crea:

- a. 9 + 1 main
- b. 3 + 1 main

c. 8 + 1 main

17. Se considera executia urmatoarei program MPI cu 2 procese

```
int main(int argc, char *argv[]) {
    int nprocs, myrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    int value = myrank*10;
    if (myrank ==0) MPI_Recv( &value, 1, MPI_INIT, 1, 10, MPI_COMM_WORLD,
    &status);
    if (myrank ==1) MPI_Send( &value, 1, MPI_INIT, 0, 10, MPI_COMM_WORLD);
    if (myrank ==0) printf("%d", value);
    MPI_Finalize( );
    return 0;
}
```

Input Parameters

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Select one or more:

1. programul nu se termina pentru ca nu sunt bine definite comunicatiile
2. programul se termina si afiseaza valoarea 0
3. programul se termina si afiseaza valoarea 10

18. Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un

program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore

binar? (Se ignora timpul de creare procese, distributie date, comunicare, iar timpul necesar operatiei

de adunare se considera egal cu 1.)

Select one or more:

	[complexitate-timp,	acceleratie,	eficienta,	cost]
1.	[1,	1024 ,	1,	1024]

2.	[10 ,	102.4 ,	0.1,	10240]
3.	[1 ,	102.4 ,	10,	102.4]
4.	[10 ,	102.4 ,	10.24,	1024]

Acceleratie = timpul celui mai bun algoritm secvential / timpul paralel

Eficienta = acceleratie / numar procese

Cost = cresc performanta cu un anumit factor, dar cate procese am adaugat totusi

19. Cate thread-uri se vor crea:

1. Cate core-uri exista pe CPU

2. 3 + 1 main

3. 15 + 1 main

4. 2 + 1 main

```
#include <stdio.h>
#include <omp.h>
void main() {
int i,j,k,t;
int N=4;
int A[4][4] = { {1,2,3,4},{5,6,7,8}, {8,9,10,11}, {1,1,1,1}};
int B[4][4] = { {1,2,3,4},{5,6,7,8}, {8,9,10,11}, {1,1,1,1}};
int C[4][4] = ;
omp_set_num_threads(3);
#pragma omp parallel shared(A,B,C) private(i,j,k,t) firstprivate(N) {
#pragma omp for schedule(dynamic)
for (i=0; i<N; i=i+1){
t = omp_get_thread_num();
for (j=0; j<N; j=j+1) {
C[i][j]=0.;
for(k=0; k<N; k=k+1) {
C[i][j] += A[i][k] * B[k][j] + t;
}
}
}
}
}
```

(A C E L E R A T I A am spus)

20. Aceleratia unui aplicatii paralele se defineste folosind urmatoarea formula:

(Se considera:

Ts = Complexitatea-timp a variantei secventiale

Tp= complexitatea-timp a variantei paralele

p=numarul de procesoare folosite pentru varianta paralela.

Select one or more:

1. T_s/T_p

2. $T_s/(p \cdot T_p)$

3. T_p/T_s

4. $p \cdot T_s/T_p$

}

21. Avem parte de data race in exemplul de mai jos?

1. Fals, deoarece fiecarui thread ii vor fi asociate task-uri independente astfel incat nu este posibila o suprapunere in calcule.

2. Adevarat, pentru ca paralelizarea for este dinamica daca nu se specifica explicit

3. Adevarat, pentru ca exista posibilitatea ca un thread sa modifice valoarea factorial[i-1] in timp ce alt thread o foloseste pentru actualizarea elementului factorial[i]

```
#pragma omp parallel for
```

```
for (i=1; i < 10; i++)
```

```
{
```

```
    factorial[i] = i * factorial[i-1];
```

```
}
```

22. Corespunzator clasificarii Flynn arhitecturile de tip cluster se incadreaza in clasa

Select one or more:

1.MISD

2.SIMD

3.MIMD

4.SISD

23.Un program paralel este optim din punct de vedere al costului daca:

Select one or more:

1. timpul paralel este de acelasi ordin de marime cu timpul secvential

2. timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential

3. acceleratia inmultita cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential

Cost = $p \times TP$

24.Overhead-ul in programele paralele se datoreaza:

Select one or more:

1. timpului necesar crearii threadurilor/proceselor

2. timpului de asteptare datorat sincronizarii

3. partitionarii dezechilibrate in taskuri
4. interactiunii interproces
5. timpului necesar distributiei de date
6. calcul in exces (repetat de fiecare proces/thread)

25. Consideram urmatorul program MPI care trebuie completat in zona specificata de comentariul "COD

de COMPLETAT". Cu care dintre variantele specificate rezultatul executiei cu 3

0 1 2 3 4 5 6 7 8

```
int main(int argc, char argv[]) {
    int nprocs, myrank;
    int i;
    int *a, *b;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_size(MPI_COMM_WORLD, &myrank);
    a = (int *) malloc( nprocs * sizeof(int));
    b = (int *) malloc( nprocs* nprocs * sizeof(int));
    for(int i=0; i < nprocs; i++) {
        a[i] = nprocs * myrank + i;
    }
    /*
    COD de COMPLETAT
    */
    if (myrank==0) {
        for(i=0; i<nprocs*nprocs; i++) {
            printf(" %d", b[i]);
        }
    }
    MPI_Finalize( );
    return 0;
}

70. MPI_Gather(a, nprocs, MPI_FLOAT, b, nprocs, MPI_FLOAT, 0 ,
    MPI_COMM_WORLD);

2. for (i =0; i < nprocs; i++) {
    b[i + nprocs * myrank] = a[i];
}

if (myrank > 0) {
    MPI_Recv(b , nprocs*(myrank+1), MPI_INT, (myrank-1),
    10, MPI_COMM_WORLD, &status);
```

```

}
MPI_Send(b, nprocs*(myrank+1), MPI_INT, (myrank+1) % nprocs, 10,
MPI_COMM_WORLD);
if (myrank==0) {
MPI_Recv(b, nprocs*nprocs, MPI_INT, (myrank-1), 10, MPI_COMM_WORLD,&status);
}
}

```

```

3. if (myrank > 0) {
    MPI_Send(a, nprocs, MPI_INT, 0, 10, MPI_COMM_WORLD);
}

    else {
        for (i = 1; i < nprocs; i++) {
            b[i] = a[i];
        }
        for (i = 1; i < nprocs; i++) {
            MPI_Recv(b + i * nprocs, nprocs, MPI_INT, i, 10, MPI_COMM_WORLD,&status);
        }
    }

```

26. Conform legii lui Amdahl, acceleratia este limitata de procentul(fractia) partii secventiale a unui program. Daca pentru un caz concret avem procentul partii secventiale egal cu 25% cat este acceleratia maxima care se poate obtine (cf legii lui Amdahl)?

1. 75
2. 25
3. 4

Legea lui Amdahl

- Afirma ca accelerarea procesarii depinde de raportul partii secventiale fata de cea paralelizabila:

seq = fractia calcului secvential; (e.g 20%=> seq=20/100)
 par = fractia calcului paralelizabil; (e.g 80%=> par=80/100)

Se considera **calculul serial** $T_s = \text{seq} + \text{par} = 1$ unitate

$$\text{Speedup} = 1/(\text{seq} + \text{par}/p)$$

$$\text{par} = (1 - \text{seq}), p = \# \text{ procesoare}$$

25 $\frac{1}{4}$ => 4
 20 $\frac{1}{5}$ => 5

27. Are programul de mai jos o executie determinista ?

1. Nu, pentru ca nu vom obtine acelasi rezultat de ori cate ori am rula programul contand ordinea de executie a thread-urilor.

2. Da, pentru ca vom obtine acelasi rezultat de ori cate ori am rula programul chiar daca programul se va executa paralel.

3. Da, pentru ca block-urile de tipul section vor fi executate secvential si nu in paralel.

```
#include "omp.h"
void main() {
    int i, t, N = 12;
    int a[N], b[N], c[N];
    for (i=0; i<N; i++) {
        a[i] = b[i] = 3;
    }
    omp_set_num_threads(3);
    #pragma omp parallel shared(a,b,c) private(i,t) firstprivate(N)
    #pragma omp single
    t = omp_get_thread_num();
    #pragma omp sections
    {
        #pragma omp section
        {
            for (i=0; i<N/3; i++) {
                c[i] = a[i] + b[i] + t;
            }
        }
        #pragma omp section
        {
            for (i=N/3; i<(N/3)*2; i++) {
                c[i] = a[i] + b[i] + t;
            }
        }
    }
    #pragma omp section
    {
        for (i=(N/3)*2; i<N; i++) {
            c[i] = a[i] + b[i] + t;
        }
    }
}
```

```
}
```

`single` and `critical` are two **very different** things. As you mentioned:

- `single` specifies that a section of code should be executed **by single thread** (not necessarily the master thread)
- `critical` specifies that code is executed **by one thread at a time**

So the former will be executed **only once** while the later will be executed **as many times as there are of threads**.

For example the following code

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical
    b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

will print

```
single: 1 -- critical: 4
```

28. Apelul pragma omp parallel for din exemplul de mai jos paralelizeaza executia ambelor structuri for?

1. Fals
2. Adevărat
3. Depinde de versiunea compilatorului folosită

```
#include <stdio.h>
#include "omp.h"
void main() {
    int i,k;
    int N = 3;
    int A[3][3] = {{1,2,3},{5,6,7},{8,9,10}};
    int B[3][3] = {{1,2,3},{5,6,7},{8,9,10}};
    int C[3][3] ;
    omp_set_num_threads(9);
    #pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)
    for (i=0; i<N; i++) {
        for (k=0; k<N; k++) {
            C[i][j] = (A[i][k] + B[i][k]);
        }
    }
}
```

29. Care dintre afirmatiile urmatoare sunt adevarate?

1. Partionarea prin descompunere functionala conduce in general la aplicatii cu scalabilitate mai buna decat partitionarea prin descompunerea domeniului de date.
2. Scalabilitatea unei aplicatii paralele este determinată de numarul de taskuri care se pot executa in paralel.

3. Daca numarul de taskuri care se pot executa in paralel creste liniar odata cu cresterea dimensiunii problemei atunci aplicatia are scalabilitate buna.

30. Pentru sablonul de proiectare paralela "Pipeline" sunt adevarate urmatoarele afirmatii:

1. pentru a avea o performanta cat mai buna este preferabil ca numarul de subtaskuri in care se descompune calculul sa fie cat mai mic
2. calculul se imparte in mai multe subtask-uri care se pot executa de catre unitati de procesare diferite
3. se obtine performanta prin paralelizare daca este nevoie de mai multe traversari ale pipeline-ului

4. pentru a obtine o performanta cat mai buna este preferabil ca impartirea pe subtaskurile sa fie cat mai echilibrata

31. Apare data-race la execuția programului următor?

```
public class Test {  
    static int value=0;  
  
    static class MyThread extends Thread{  
        public void run() {  
            value++;  
        }  
    }  
}  
  
public static void main(String[] args) throws InterruptedException {  
    MyThread t1 = new MyThread();  
    MyThread t2 = new MyThread();  
    t1.start(); t2.start();  
    t1.join(); t2.join();  
    System.out.print(value);  
}  
}
```

a) DA

b) NU

32. Consideram urmatoarea schita de implementarea pentru un semafor:

count: INTEGER

blocked: CONTAINER

down

do

if count > 0 then

count := count - 1

else

blocked.append(P) -- P is current process

```

        P.state := blocked -- blocked process P
    end
end
up
    do
    if blocked.is_empty then
    count := count + 1
    else
    Q := blocked.remove -- selected some process Q
    Q.state := ready-- unblock process Q
    end
end
end

```

Care dintre urmatoarele afirmatii sunt adevarate ?

1. aceasta varianta de implementare defineste un "strong-semaphor" (semafor puternic)
2. aceasta varianta de implementare defineste un "weak-semaphor" (semafor slab)
3. aceasta varianta de implementare nu este "starvation-free"
4. aceasta varianta de implementare este "starvation-free"

33. Se considera paralelizarea sortarii unui vector cu n elemente prin metoda "merge-sort" folosind

sablonul Divide&impera. În condițiile în care avem un număr nelimitat de procesoare, se poate ajunge la un anumit moment al execuției la un grad maxim de paralelizare egal cu:

1. $\log n$
2. $n / \log n$
3. n

34. Arhitecturile UMA sunt caracterizate de:

1. identificator unic pentru fiecare procesor
2. același timp de acces pentru orice locație de memorie

UMA = Uniform Memory Access

(U R M A T O R U L U I am zis)

35. Următorului program se va executa cu 3 procese. Ce valoare se va afișa?

```

int main(int argc, char *argv[]) {
    int nprocs, myrank;
    int i, value=0;
    int *a, *b;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0) {

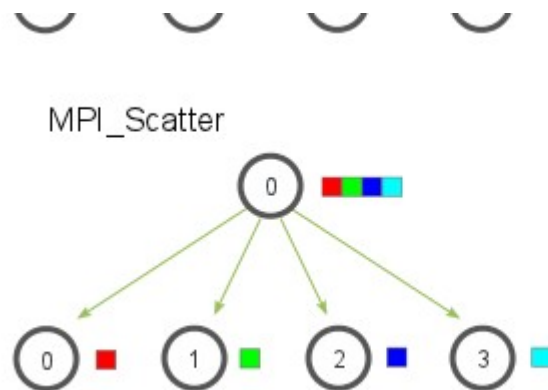
```

```

a = (int*) malloc(nprocs * sizeof(int));
for(int i=0;i<nprocs; i++) {
a[i]=i+1; // 1 2 3
}

}
b = (int *) malloc( sizeof(int));
MPI_Scatter(a, 1, MPI_INIT, b, 1, MPI_INT, 0 ,MPI_COMM_WORLD);
din a se trimite a[0], doar primul element pentru ca param nr 2 este 1( (count)
b[0] += myrank; // !! se mai adauga si myrank
MPI_Reduce(b, &value, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if( myrank == 0) {
printf("value = \"%d \n", value);
}
MPI_Finalize( );
return 0;
}

```



1. 9
2. 6
3. 12

Procesul 0 trimite a, adică $a + 0$, adică $a[0]$ proceselor 1 și 2, valoare memorata în b.
Pentru procesul 1 (rank 1) avem $b[0] = 1 + 1 = 2$
Pentru procesul 2 (rank 2) avem $b[0] = 1 + 2 = 3$
Pentru procesul 0 ramane $b = a = 1$
Se face reduce peste b-ul fiecarui proces, deci $1 + 2 + 3 = 6$

----- Part deux

PPD – QUIZ

1. Scalabilitatea este mai mare pentru sistemele:

a. **MPP (Massively Parallel Processing)**

b. SMP (Symmetric Processing, shared memory)

2. Latenta memoriei este:

a. rata de transfer a datelor din memorie către processor

b. timpul în care o data ajunge sa fie disponibilă la procesor după ce s-a inițiat cererea.

3. Asigurarea cache coherency determina pentru scalabilitate o:

a. Scădere

b. Creștere

c. Nu este legatura

4. Un calculator cu 1 procesor permite execuție paralelă?

a. Da

b. Nu

5. "Context Switch" este mai costisitor pentru:

a. Procese

b. Thread-uri

6. Thread1 executa {a=b+1; a=a+1} si Thread2 executa {b=b+1}. Apare "data race"? Required to answer. Single choice.

a. Da

b. Nu

Pentru ca în Thread2 se face write

7. Thread1 execută { c=a+1} și Thread2 executa {b=b+a;}. Apare "data race"?

a. Da

b. Nu

Pentru ca se face doar read

8. Într-o execuție deterministă poate să apară "race condition".

a. Fals

b. Adevărat

race condition = atunci când ordinea în care se modifica variabilele interne determina starea finală a sistemului

9. Granularitatea unei aplicații paralele este

a. Definite ca dimensiunea minima a unei unități secvențiale dintr-un program, exprimată în număr de instrucțiuni

b. Este determinate de numărul de taskuri rezultate prin descompunerea calculului

c. Se poate aproxima ca fiind raportul din timpul total de calcul și timpul total de comunicare

10. Granularitatea unei aplicații paralele este de dorit sa fie

a. Mica

b. Mare

11. Granularitatea unui sistem parallel este de dorit sa fie

a. Mica

b. Mare

12. Eficienta unui program parallel care face suma a 2 vectori de dimensiune n folosind p procese este:

- a. Maxim 1
- b. Minim 1
- c. Egala cu p

Eficiența = Acceleratia / p, iar accelerația poate să fie maxim p.
Deci, eficiența maximă poate să fie $p / p = 1$.

13. Costul unei aplicații paralele este optim dacă

- a. $C = O(T_s \cdot \log p)$
- b. $C = O(T_s)$
- c. $C = O(\Omega(T_s))$

14. Un semafor care stochează procesele care așteaptă într-o mulțime, se numește:

- a. Strong Semaphore (semafor puternic) - FIFO (coada)
- b. Weak Semaphore (semafor slab) - multime.
- c. Semafor binar

15. Livelock descrie situația în care:

- a. Un grup de procese/threaduri nu progresează datorită faptului că își cedează reciproc execuția
- b. Un grup de procese/threaduri nu progresează datorită faptului că își blochează reciproc execuția
- c. Un grup de procese/threaduri nu progresează datorită faptului că nu se termină niciunul

16. Fata de Monitor, Semaforul este o structura de sincronizare:

- a. De nivel înalt
- b. De nivel jos
- c. De același nivel



2. Care dintre urmatoarele variante pot fi rezultatul afisat al executiei urmatorului program MPI (mpirun -np 3 hello)?

```
int main(int argc, char **argv)
{
    int namelen, myid, numprocs;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf( "Process%d/%d; ", myid, numprocs);
    MPI_Finalize();
    printf( "Regards!");
    return 0;
}
```

(0/3 Points)

- ☒ Process0/3;Process1/3;Process2/3;Regards!
- ☐ Process0/3;Process1/3;Process2/3;Regards!Regards!Regards! ✓
- ☐ Process2/3;Process1/3;Process0/3;Regards!Regards!Regards! ✓
- ☐ Process1/3;Process3/3;Process2/3;Regards!Regards!Regards!
- ☒ Process1/3;Process0/3;Process2/3;Regards!

Send

Recv

Send

Recv => deadlock

1. Este posibil ca urmatorul cod MPI sa produca deadlock?

```
int main(int argc, char *argv[]) {  
    // var declaration... init...  
    int tag = 10;  
    if (rank == 0) {  
        dest = source = 1;  
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
    }  
    else if (rank == 1){  
        dest = source = 0;  
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
    }  
    // ... finalizare  
}
```

(3/3 Points)

☐ DA

☒ Nu ✓

1. Ce se va afisa in urma executiei codului:

```
int i, chunk = 2, indx = 3, tid;  
const int n=11;  
int a[n];  
for (i = 0; i < n; ++i) a[i] = 0;  
  
#pragma omp parallel shared(chunk,a) private(i,tid) num_threads(3) firstprivate(indx)  
{  
    tid = omp_get_thread_num();  
    indx = indx + chunk * tid;  
    for (i = indx; i < indx + chunk; i++)  
        a[i] = tid + 1;  
}  
for (i = 0; i < n; ++i)    cout << a[i] << " ";
```

(3/3 Points)

- ☐ 0 0 1 1 1 2 2 2 3 3 3
- ☒ 0 0 0 1 1 2 2 3 3 0 0
- ☐ 1 1 1 2 2 2 3 3 3 0 0

Tid = 0, 1, 2

Tid = 0

Indx = $3 + 2 * 0 = 3$

I = 3 -> $3+2=5-1=4 \Rightarrow a[3]=a[4]=1$

Tid = 1

Indx = $3 + 2 * 1 = 5$

I = 5 -> $5+2=7-1=6 \Rightarrow a[5]=a[6]=2$

Tid = 2

Indx = $3 + 2 * 2 = 7$

I = 7 $\rightarrow 7+2=9-1=8 \Rightarrow a[7]=a[8]=3$

-----Part Three-----

71. Care dintre afirmatiile urmatoare sunt adevarate?

d. Scalabilitatea unei aplicatii paralele este determinata de numarul de task-uri care se pot executa in paralel.

e. Daca numarul de task-uri care se pot executa in paralel creste liniar odata cu cresterea dimensiunii problemei atunci aplicatia are scalabilitate buna

f. Partitionarea prin descompunere functionala conduce in general la aplicatii cu scalabilitate mai buna decat partitionarea prin descompunerea domeniului de date.

2. Care din variantele de mai jos au aceleasi efect cu apelul functiei

omp_set_num_threads(12):

g. num_threads(12) ca si clauza intr-o directiva #pragma omp parallel

h. export OMP_NUM_THREADS = 11

i. omp_get_num_threads()

```

1  #include <stdio.h>
2  #include "omp.h"
3
4  void main() {
5      int i, t, N = 12;
6      int a[N], b[N], c[N];
7
8      for (i=0; i < N; i++) a[i] = b[i] = 3;
9
10     omp_set_num_threads(3);
11
12     #pragma omp parallel shared(a,b,c) private(i, t) firstprivate(N)
13     #pragma omp single
14     t = omp_get_thread_num();
15
16     #pragma omp sections
17     {
18         #pragma omp section
19         {
20             for (i=0; i < N/3; i++)
21                 c[i] = a[i] / b[i] + t;
22         }
23         #pragma omp section
24         {
25             for (i=N/3; i < (N/3)*2; i++) {
26                 c[i] = a[i] + b[i] + t;
27             }
28         }
29         #pragma omp section
30         {
31             for (i=(N/3)*2; i < N; i++) {
32                 c[i] = a[i] * b[i] + t;
33             }
34         }
35     }
36 }

```

Cate thread-uri se vor crea:

- ☐ 1. 2 + 1 main
- ☐ 2. Cate core-uri exista pe CPU
- ☐ 3. 11 + 1 main
- ☐ 4. 3 + 1 main

3. Raspuns: 1

13. Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore binar? (Se ignora timpul de creare procese, distributie date, comunicatie, iar timpul necesar operatiei de adunare se considera egal cu 1.)

Select one or more:

	[complexitate-timp,	acceleratie,	eficienta,	cost]
j.	[10 ,	102.4 ,	0.1,	10240]
k.	[10 ,	102.4 ,	10.24,	1024]
l.	[1 ,	1024 ,	1,	1024]
m.	[1 ,	102.4 ,	10,	102.4]

14. Corespunzator clasificarii Flynn arhitecturile de tip cluster se incadreaza in clasa

- n. SIMD
- o. MISD
- p. SISD
- q. MIMD

4.

```
#include <stdio.h>
```

```
#include "omp.h"
```

```
void main(){
```

```
int i,k;
```

```
int N=3;
```

```
int A[3][3]={ {1,2,3},{5,6,7},{8,9,10}};
```

```
int B[3][3]={ {1,2,3},{5,6,7},{8,9,10}};
```

```
int C[3][3];
```

```
omp_set_num_threads(9);
```

```
    #pragma omp parallel for private(i,k) shared (A,B,C,N) schedule(static)  
collapse(2)
```

```
    for(i=0; i <N;i++) {  
for(k=0;k<N; k++){  
C[i][k]=(A[i][k] + B[i][k]);  
    }  
    }
```

```
}
```

Care va fi schema de distribuire a iteratiilor între thread-urile create:

1.

Thread 0: i=0, k=0

Thread 1 : i=0; k=1

Thread 2 : i=0, k=2

Thread 3 : i=1, k=0

Thread 4 : i=1, k=1

...

Thread 8 : i=2, k=2

2.

Thread 0: i=0, k=0-2

Thread 1 : i=1; k=0-2

Thread 2 : i=2, k=0-2

Thread 3-8: standby

3. Ordinea de procesare nu este deterministă, astfel ca fiecare thread va prelua in mod aleator task-urile care la randul lor vor avea dimensiune diferită

5.Care varianta de definire pentru variabilele grid si block(de completat in locul comentariului) conduce la crearea unui numar de 1024 de threaduri CUDA pentru apelul functiei VecAdd? </p>

```
/***/ definire grid si block - de completat  
VecAdd << grid, block >>>(A,B,C);
```

Select one or more :

- 72. dim3 grid(16); dim3 block(256);
- 73. dim3 grid(4); dim3 block(256); 256x4=1024
- 74. dim3 grid(4); dim3 block(16,16); 16x16x4 =1024
- 75. dim3 grid(8,8); dim3 block(4,4); 8 x 8 x 4 x 4

Ex.

```
dim3 grid(256);                // defines a grid of 256 x 1 x 1 blocks  
dim3 block(512, 512);          // defines a block of 512 x 512 x 1 threads
```

10. Apelul pragma omp parallel for din exemplul de mai jos paralelizeaza executia la toate cele 3 structuri for?

```
void main() {  
    int i, j, k, t;  
        int N = 4;  
  
        int A[4][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {8, 9, 10, 11}, {1, 1, 1, 1} };  
        int B[4][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {8, 9, 10, 11}, {1, 1, 1, 1} };  
        int C[4][4];  
  
        #pragma omp parallel shared(A,B,C) private(i,j,k,t) firstprivate(N)  
        {  
            #pragma omp schedule(dynamic)  
            for (i = 0; i < N; i++) {  
                t = omp_get_thread_num();  
  
                for(j = 0; j < N; j++) {  
                    C[i][j] = 0;  
  
                    for(k = 0; k < N; k++) {  
                        C[i][j] += A[i][k] * B[k][j] + t;  
                    }  
                }  
            }  
        }  
}
```


}

Raspunsuri:

76. Adevarat

77. Depinde de versiunea de openMP folosita

78. Fals

11. Se considera executia urmatorului program MPI cu 2 procese:

```
int main (int argc, char *argv[]) {  
    int nprocs, myrank;  
    MPI_Status status;  
    MPI_Init(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
  
    int value = myrank*10;  
    if(myrank == 0) MPI_Recv (&value,1,MPI_INT, 0, 10, MPI_COMM_WORLD, &status);  
    if(myrank == 1) MPI_Send(&value,1,MPI_INT,1,10,MPI_COMM_WORLD);  
    if(myrank == 0) printf("%D", value);  
  
    MPI_FINALIZE();  
    return 0;  
}
```

Raspunsuri:

79. programul nu se termina pentru ca nu sunt bine definite comunicatiile

80. programul se termina si afiseaza valoarea 10

81. programul se termina si afiseaza valoarea 0

12. Care dintre urmatoarele afirmatii este adevarata

82. Ordinea executiei block-urilor de tipul **section** este determinata.

83. Fiecare block de tipul **section** este executat de un thread.

84. Daca sunt mai multe block-uri de tipul **section** decat thread-uri , exista riscul de a nu se procesa o parte dintre aceste block-uri.

Curs2 – quiz 1 – 6/6

1. Scalabilitatea este mai mare pentru sistemele
(2/2 Points)

- ☐ SMP
- ☒ MPP

✓

2. Latenta memoriei este

(2/2 Points)

- ☐ rata de transfer a datelor din memorie catre procesor
- ☒ timpul in care o data ajunge sa fie disponibila la procesor dupa ce s-a initiat cererea.

✓

3. Asigurarea cache coherency determina pentru scalabilitate o:
(2/2 Points)

- ☐ crestere
- ☒ scadere
- ☐ nu este legata

✓

Curs3 – quiz 2 – 6/10

1. Un calculator cu 1 procesor permite executie paralela? *
(2/2 Points)

- ☒ DA
- ☐ Nu

✓

2. "Context Switch" este mai costisitor pentru *
(0/2 Points)

- ☐ procese
- ☒ threaduri

✓

3. Thread1 executa $\{a=b+1; a=a+1\}$ si Thread2 executa $\{b=b+1\}$. Apare "data race"? *

☒ Da

☐ Nu



4. Thread1 executa $\{c=a+1\}$ si Thread2 executa $\{b=b+a\}$. Apare "data race"? *

☒ Da

☐ Nu



5. Intr-o executie determinista poate sa apara "race condition". *

☐ Adevarat

☒ Fals



1. Este posibil ca urmatorul cod MPI sa produca deadlock?

```
int main(int argc, char *argv[]) {  
    // var declaration... init....  
    int tag = 10;  
    if (rank == 0) {  
        dest = source = 1;  
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
    }  
    else if (rank == 1){  
        dest = source = 0;  
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);  
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);  
    }  
    // ... finalizare  
}
```

(3/3 Points)

- ☐ DA
- ☒ Nu

✓

3. Prin codul urmator se doreste transmiterea prin broadcast a valorii variabilei x setata in procesul 0, astfel incat fiecare proces sa afiseze valoarea 10. Este acest cod corect?

```
//init...  
int x;  
if (rank == 0){  
    x = 10;  
    MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);  
}  
printf("%d", x);  
//finalize *  
(2 Points)
```

- ☐ Da
- ☒ Nu

2. Care dintre urmatoarele variante pot fi rezultatul afisat al executiei urmatorului program MPI (mpirun -np 3 hello)?

```
int main(int argc, char **argv)
{
    int namelen, myid, numprocs;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf( "Process%d/%d; ", myid, numprocs);
    MPI_Finalize();
    printf( "Regards!");
    return 0;
}
```

*

(0/3 Points)

- ☒ Process0/3;Process1/3;Process2/3;Regards!
- ☐ Process0/3;Process1/3;Process2/3;Regards!Regards!Regards! ✓
- ☐ Process2/3;Process1/3;Process0/3;Regards!Regards!Regards! ✓
- ☐ Process1/3;Process3/3;Process2/3;Regards!Regards!Regards!
- ☐ Process1/3;Process0/3;Process2/3;Regards!


Curs6 ?/3

```
1. public class Counter {
    private long c = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc() {
        synchronized(lock1) {
            c++;
        }
    }
    public void dec() {
```

```

public void dec() {
    synchronized(lock2) {
        c--;
    }
}
(3 Points)

```

- ☒ codul  este incorect -poate sa apara race condition ✓
- ☐ codul este corect
- ☐ codul este incorect pentru blocurile synchronized nu sunt bine definite

R: Codul nu este corect- apare Race Condition
Sincronizare pe lockuri diferite, insa se modifica aceeaasi variabila

```

public class Counter {
    private long c = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc() {
        synchronized(lock1) {
            c++;
        }
    }
    public void dec() {
        synchronized(lock2) {
            c--;
        }
    }
}

```

1. Ce se va afisa in urma executiei codului:

```
int i, chunk = 2, indx = 3, tid;  
const int n=11;  
int a[n];  
for (i = 0; i < n; ++i) a[i] = 0;  
  
#pragma omp parallel shared(chunk,a) private(i,tid) num_threads(3) firstprivate(indx)  
{  
    tid = omp_get_thread_num();  
    indx = indx + chunk * tid;  
    for (i = indx; i < indx + chunk; i++)  
        a[i] = tid + 1;  
}  
for (i = 0; i < n; ++i)    cout << a[i] << " ";
```

(3/3 Points)

- ☐ 00111222333
- ☒ 00011223300
- ☐ 11122233300

Tid = 0, 1, 2

Tid = 0

Indx = $3 + 2 * 0 = 3$

$I = 3 \rightarrow 3+2=5-1=4 \Rightarrow a[3]=a[4]=1$

Tid = 1

Indx = $3 + 2 * 1 = 5$

$I = 5 \rightarrow 5+2=7-1=6 \Rightarrow a[5]=a[6]=2$

Tid = 2

Indx = $3 + 2 * 2 = 7$

$I = 7 \rightarrow 7+2=9-1=8 \Rightarrow a[7]=a[8]=3$

```

<pre>
public class Test {
    static int value=0;
    static class MyThread extends Thread{
        Lock l;    CyclicBarrier b;
        public MyThread(Lock l, CyclicBarrier b){
            this.l=l; this.b=b;
        }
        public void run(){
            try{
                l.lock();
                value+=1;
                b.await();
            } catch (InterruptedException|BrokenBarrierException e) {
                e.printStackTrace();
            }
            finally{ l.unlock();}
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Lock l = new ReentrantLock(); CyclicBarrier b = new CyclicBarrier(2);
        MyThread t1 = new MyThread(l, b); MyThread t2 = new MyThread(l, b);
        t1.start();    t2.start();
        t1.join();    t2.join();
        System.out.print(value);
    }
}
</pre>
{
    1) se termina si afiseaza 2
    2) se termina si afiseaza 1
    3) nu se termina
}

```

R. 3) nu se termina

Un thread intra, face loc, insa asteapta la bariera. Dar nu mai poate intra alt thread pentru ca nu s-a dat unlock. Deci programul nu se termina

Lock = excludere mutuala

Intra primul thread, se blocheaza la bariera, iar urmatorul nu poate intra la baeriera care asteapta 2, pentru ca primul nu a dat unlock.

Practic: fara CUDA si MPI

Multithreading – Java sau C++ (se poate OpenMP)

Threaduri, impartire corecta a threadurilor, sincronizari, asteptari conditionate

Internet: Putem accesa documentatia de la Java, C++, complet interzisa comunicarea (chatur, ..)

Acces la orice resurse, dar nu interactiune cu altcineva.

Corespunzator clasificarii Flynn arhitecturile de tip cluster se incadreaza in clasa

Select one or more:

- ☐ 1. SIMD
- ☐ 2. MISD
- ☐ 3. SISD
- ☐ 4. MIMD

Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore binar? (Se ignora timpul de creare procese, distributie date, comunicare, iar timpul necesar operatiei de adunare se considera egal cu 1.)

Select one or more:

- ☐ 1. [10 , 102.4 , 0.1, 10240]
- ☐ 2. [10 , 102.4 , 10.24, 1024]
- ☒ 3. [1 , 1024 , 1, 1024]
- ☐ 4. [1 , 102.4 , 10, 102.4]

Se considera executia urmatorului program MPI cu 2 procese:

```
1  int main(int argc, char *argv[] ) {
2      int nprocs, myrank;
3      MPI_Status status;
4      MPI_Init(&argc, &argv);
5      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
6      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7
8      int value = myrank*10;
9      if (myrank ==0) MPI_Recv( &value, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
10     if (myrank ==1) MPI_Send (&value, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
11     if (myrank ==0) printf("%d", value);
12     MPI_Finalize( );
13     return 0;
14 }
```

Select one or more:

- ☐ 1. programul nu se termina pentru ca nu sunt bine definite comunicatiile
- ☐ 2. programul se termina si afiseaza valoarea 10
- ☐ 3. programul se termina si afiseaza valoarea 0

Conform legii lui Amdahl acceleratia este limitata de procentul(fractia) partii secventiale a unui program. Daca pentru un caz concret avem procentul partii secventiale egal cu 25% cat este acceleratia maxima care se poate obtine (cf legii lui Amdahl)?

Select one or more:

- ☐ 1. 75
- ☒ 2. 4
- ☐ 3. 25

```

2  #include "omp.h"
3
4  void main() {
5      int i, j, k, t;
6      int N=4;
7
8      int A[4][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {8,9,10, 11}, {1, 1, 1, 1} };
9      int B[4][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {8,9,10, 11}, {1, 1, 1, 1} };
10     int C[4][4] ;
11
12     omp_set_num_threads(3);
13
14     #pragma omp parallel shared(A,B,C) private(i,j,k,t) firstprivate(N)
15     {
16         #pragma omp for schedule(dynamic)
17         for (i=0; i<N; i=i+1){
18             t = omp_get_thread_num();
19
20             for (j=0; j<N; j=j+1){
21                 C[i][j]=0.;
22
23                 for (k=0; k<N; k=k+1){
24                     C[i][j] += A[i][k] * B[k][j] + t;
25                 }
26             }
27         }
28     }
29 }

```

Apelul **pragma omp parallel for** din exemplul de mai sus paralelizeaza executia la toate cele 3 structuri **for** ?

- ☐ 1. Adevarat
- ☐ 2. Depinde de versiunea openMP folosita
- ☐ 3. Fals

Consideram urmatorul program MPI care se executa cu 3 procese

////////////////////////////////////

```
1  int main(int argc, char *argv[] ) {
2      int nprocs, myrank;
3      int i;
4      int *a, *b;
5      MPI_Status status;
6      MPI_Init(&argc, &argv);
7      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
8      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
9
10     a = (int *) malloc( nprocs * sizeof(int));
11     b = (int *) malloc( nprocs* nprocs * sizeof(int));
12     for(int i=0;i<nprocs; i++) a[i]=nprocs*myrank+i;
13
14     if (myrank>0) MPI_Recv(b , nprocs*(myrank+1), MPI_INT, (myrank-1), 10, MPI_COMM_WORLD, &status);
15     MPI_Send(b, nprocs*(myrank+1), MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD);
16     if (myrank==0) MPI_Recv(b , nprocs*nprocs, MPI_INT, (nprocs-1), 10, MPI_COMM_WORLD, &status);
17
18     MPI_Finalize( );
19     return 0;
20 }
```

////////////////////////////////////

Intre ce perechi de procese se realizeaza comunicatia si in ce ordine se realizeaza comunicatiile

Select one or more:

- ☐ 1. (2->1), (1->0), (0->2) in ordine aleatorie
- ☐ 2. (0->1) urmata de (1-2) urmata de (2->0)
- ☐ 3. (0->1), (1-2),(2->0) in ordine aleatorie
- ☐ 4. (2->1),urmata de (1->0),urmata de (0->2)
- ☐ 5. comunicatiile nu sunt bine definite pentru ca nu se realizeaza corect perechile (sender, receiver)

Consideram executia urmatoareului program MPI cu 3 procese.

////////////////////////////////////

```
1  int main(int argc, char *argv[] ) {
2      int nprocs, myrank;
3      MPI_Status status;
4      MPI_Init(&argc, &argv);
5      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
6      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
7
8      int value = myrank*10;
9      int sum=0;
10     MPI_Recv( &sum, 1, MPI_INT, (myrank-1+nprocs)/nprocs, 10, MPI_COMM_WORLD, &status);
11     sum+=value;
12     MPI_Send (&sum, 1, MPI_INT, (myrank+1)%nprocs, 10, MPI_COMM_WORLD);
13     if (myrank ==0)
14         printf("%d", sum);
15     MPI_Finalize( );
16     return 0;
17 }
```

////////////////////////////////////

Care dintre urmatoarele afirmatii sunt adevarate?

Select one or more:

- ☐ 1. se executa corect si afiseaza 30
- ☐ 2. executia programului produce deadlock pentru ca procesul de la care primeste procesul 0 nu este bine definit
- ☐ 3. executia programului produce deadlock pentru ca nici un proces nu poate sa trimita inainte sa primeasca

Browser tabs: Note, WhatsApp, Examen PPD 20 01 2021 [3], PPD_ALL.pdf, Examen PPD 2021.21 (page 12) X

Address bar: moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232977&cmid=31278&page=11

Page title: moodleubb

User profile: Raluca Oana Lenghel

Programare paralela si distribuita

Home / My courses / PPD / General / Examen PPD 20.01.21

Quiz navigation

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36				

Finish attempt ...

Time left 0:31:02

Question 12

Not yet answered

Marked out of 1.00

Flag question

Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficientei si costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare si un calcul de tip arbore bina? (Se ignora timpul de creare procese, distributie date, comunicatie, iar timpul necesar operatiilor de adunare se considera egal cu 1.)

Select one or more:

- ☐ 1. [10 , 1024 , 0.1, 10240]
- ☐ 2. [10 , 1024 , 10.24, 1024]
- ☐ 3. [1 , 1024 , 1, 1024]
- ☐ 4. [1 , 1024 , 10, 1024]

Next page

Announcements

Jump to...

You are logged in as Raluca Oana Lenghel (Log out)
PPD
Data retention summary

- ☐ 1. calculul se imparte in mai multe subtask-uri care se pot executa de catre unitati de procesare diferite
- ☐ 2. se obtine performanta prin paralelizare daca este nevoie de mai multe traversari ale pipeline-ului
- ☐ 3. pentru a obtine o performanta cat mai buna este preferabil ca impartirea pe subtaskurile sa fie cat mai echilibrata
- ☐ 4. pentru a avea o performanta cat mai buna este preferabil ca numarul de subtaskuri in care se descompune calculul sa fie cat mai mic

Programare paralela si distribuita

Home / My courses / PPD / General / Examen PPD 20.01.21

Quiz navigation

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36				

Finish attempt ...

Time left **0:24:07**

Question 17
Not yet answered
Marked out of 1.00
Flag question

Care varianta de definire: pentru variabilele grid si block(de completat in locul comentariului) conduce la crearea unui numar de 1024 de threaduri CUDA pentru apelul functiei VecAdd?</p>
<p><code>/**/* define grid si block - de completat
VecAdd <<< grid , block >>> {A, B, C};</code>

Select one or more:

- ☐ 1. dim3 grid{16}; dim3 block{256};
- ☐ 2. dim3 grid{4}; dim3 block{256};
- ☐ 3. dim3 gnd{4}; dim3 block{16,16};
- ☐ 4. dim3 grid{8, 8}; dim3 block{4, 4};

Announcements <input type="text" value="Jump to..."/> Next page

```

1  #include <stdio.h>
2  #include "omp.h"
3
4  void main() {
5      int i, k;
6      int N=3;
7
8      int A[3][3] = { {1, 2, 3}, {5, 6, 7}, {8,9,10} };
9      int B[3][3] = { {1, 2, 3}, {5, 6, 7}, {8,9,10} };
10     int C[3][3];
11
12     omp_set_num_threads(9);
13
14     #pragma omp parallel for private(i,k) shared(A, B, C, N) schedule(static) collapse(2)
15     for (i = 0; i < N; i++) {
16         for (k=0; k < N; k++) {
17             C[i][k] = (A[i][k] + B[i][k]);
18         }
19     }
20 }

```

Care va fi schema de distribuire a iteratiilor intre thread-urile create:

- ☐ 1. Thread 0: i = 0, k = 0
 Thread 1: i = 0, k = 1
 Thread 2: i = 0, k = 2
 Thread 3: i = 1, k = 0
 Thread 4: i = 1, k = 1

 Thread 8: i = 2, k = 2
- ☐ 2. Thread 0: i = 0, k = 0-2
 Thread 1: i = 1, k = 0-2
 Thread 2: i = 2, k = 0-2
 Thread 3-8: standby
- ☐ 3. Ordinea de procesare nu este determinista, astfel ca fiecare thread va prelua in mod aleator task-urile care la randul lor vor avea dimensiune definita.

moodleubb

Question 20
 Not yet answered
 Marked out of 1.00
 Flag question

Finish attempt ...
 Time left 0:17:09

```

1  #include <stdio.h>
2  #include "omp.h"
3
4  void main() {
5      int i, t, N = 32;
6      int a[N], b[N], c[N];
7
8      for (i=0; i < N; i++) a[i] = b[i] = 3;
9
10     omp_set_num_threads(3);
11
12     #pragma omp parallel shared(a,b,c) private(i, t) firstprivate(N)
13     #pragma omp single
14     t = omp_get_thread_num();
15
16     #pragma omp sections
17     {
18         #pragma omp section
19         {
20             for (i=0; i < N/3; i++)
21                 c[i] = a[i] / b[i] + t;
22         }
23         #pragma omp section
24         {
25             for (i=N/3; i < (N/3)*2; i++) {
26                 c[i] = a[i] + b[i] + t;
27             }
28         }
29         #pragma omp section
30         {
31             for (i=(N/3)*2; i < N; i++) {
32                 c[i] = a[i] * b[i] + t;
33             }
34         }
35     }
36 }

```

Cate thread-uri se vor crea:

☐ 1. 2 + 1 main

☐ 2. Cate core-uri exista pe CPU

☐ 3. 11 + 1 main

☐ 4. 3 + 1 main

10:58 AM
 1/20/2021

Care din variantele de mai jos au acelasi efect cu apelul functiei **omp_set_num_threads(12)**:

- ☐ 1. **num_threads(12)** ca si clauza intr-o directiva **#pragma omp parallel**
- ☐ 2. **export OMP_NUM_THREADS=11**
- ☐ 3. **omp_get_num_threads()**

Care dintre afirmatiile urmatoare sunt adevarate?

Select one or more:

- ☐ Scalabilitatea unei aplicatii paralele este determinata de numarul de taskuri care se pot executa in paralel.
- ☐ Daca numarul de taskuri care se pot executa in paralel creste liniar odata cu cresterea dimensiunii problemei atunci aplicatia are scalabilitate buna.
- ☐ Partionarea prin descompunere functionala conduce in general la aplicatii cu scalabilitate mai buna decat partitionarea prin descompunerea domeniului de date.