

PPD – QUIZ

1. Scalabilitatea este mai mare pentru sistemele:
 - a. MPP
 - b. SMP
 2. Latenta memoriei este .
 - a. rata de transfer a datelor din memorie catre processor
 - b. 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: Single choice.
 - a. Scadere
 - b. Crestere
 - c. Nu este legatura
 4. Un calculator cu 1 procesor permite executie paralela? Required to answer. Single choice.
 - a. Da
 - b. Nu
 5. "Context Switch" este mai costisitor pentru:
 - a. Procese
 - b. Threaduri
 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
 7. Thread1 executa {c=a+1} si Thread2 executa {b=b+a;}. Apare "data race"?
 - a. Da
 - b. Nu
 8. Intr-o executie determinista poate sa apara "race condition".
 - a. Fals
 - b. Adevarat
 9. Granularitatea unei aplicatii paralele este
 - a. Definita ca dimensiunea minima a unei unitati secentiale dintr-un program, exprimata in numar de instructiuni
 - b. Este determinata de numarul de taskuri rezultate prin descompunerea calculului
 - c. Se poate aproxima ca fiind raportul din timpul total de calcul si timpul total de comunicare
 10. Granularitatea unei aplicatii paralele este de dorit sa fie:
 - a. Mica
 - b. Mare Ca sa putem executa aplicatia mult mai eficient
 11. Granularitatea unui sistem paralel este de dorit sa fie:
 - a. Mica Ca sa putem executa cat mai multe aplicatii
 - b. Mare
 12. Eficiența unui program parallel care face suma a 2 vectori de dimensiune n folosind p procese este:
 - a. Maxim 1
- $$E = \text{best_serial} / (p * \text{best_parallel_pt_p})$$
- $$E = n / (p * (n/p)) \Rightarrow 1$$

- b. Minim 1
c. Egala cu p
13. Costul unei aplicatii paralele este optim daca
- $C=O(T_s * \log p)$
 - $C=O(T_s)$
 - $C=\Omega(T_s)$
14. Un semafor care stocheaza procesele care asteapta intr-o multime, se numeste:
- Strong Semaphore (semafor puternic)
 - Weak Semaphor (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
 - 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\n", 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

E si in alte subiecte cu enuntul complet

0 0 0 1 1 2 2 3 3 0 0 ✓

1 1 1 2 2 2 3 3 3 0 0

```

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 afirmații sunt adevărate?

Se executa corect si afisaza 30.

Executia programului produce deadlock pentru ca procesul de la care primescste 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 afirmații sunt adevărate ?

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”

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 (I1) {
        synchronized (I2) {
            int temp = a;
            a += b;
            b += temp;
        }
    }
}
}

private static class T2 extends Thread {
public void run() {
    synchronized (I2) {
        synchronized (I1) {
            a--;
            b--;
        }
    }
}
}

```

T1, T2, T1, T2
a = 4, b = 4
T1: a = 8; b=8
T2: 7, 7
T1: 14, 14
T2: 13, 13

7. 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

8.

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

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;
}
||||||||||||||||||||||||||||
```

Apare data race

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

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

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

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

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

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) era 3 daca 0 facea receive cum trebuie de la 2
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)

MPI_ANY_SOURCE = -2

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);

    #pragme 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]);
        }
    }
}

```

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

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
4. Creează: 12, distrug 36

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

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);

    #pragme 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]);
        }
    }
}
```

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

Cate thread-uri se vor crea:

Cate core-uri exista pe CPU

9 + 1 main

3 + 1 main

8 + 1 main

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_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

3. programul se termina si afiseaza valoarea 10

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

Complex timp: $\log_2(1024) \Rightarrow 10$

Accelartia: $TS / TP = 1024 / 10 = 102.4$

Eficienta: $acceleratia / p = 102.4 / 1024 = 0.1$

Cost: $TP * P = 10 * 1024 = 10240$

```
#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 există pe CPU
2. 3 + 1 main
3. 15 + 1 main
4. 2 + 1 main

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

Aceleratia unui aplicatie paralela 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:

1. T_s/T_p
2. $T_s/(p*T_p)$
3. T_p/T_s
4. $p*T_s/T_p$

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]
- 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
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]

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

1.
MISD
2.
SIMD
3.
MIMD
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
2. **timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential**
3. aceleratia inmultita cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential

Overhead-ul in programele parallele 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)

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_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;
}
```

1. Era bun daca era MPI_INT in loc de MPI_FLOAT

```
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);
```

E gresit si la recv de proces 0 dar si partea de send recv nu e ok

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];      Nu se populeaza b[0]
    for (i = 1; i < nprocs; i++)
        MPI_Recv(b + i * nprocs, nprocs, MPI_INT, i, 10, MPI_COMM_WORLD,
&status);
```

}

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

Select one or more:

- 1. 75
- 2. 25
- 3. 4**

```
#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 ? Prima varianta este corecta, a 2-a e gresita

- ✓ 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 <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);

    #pragme 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]);
        }
    }
}
```

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

- 1. Fals
- 2. Adevarat
- 3. Depinde de versiunea compilatorului folosita

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 (I1) {
            synchronized (I2) {
                int temp = a;
                a += b;
                b += temp;
            }
        }
    }
}

private static class T2 extends Thread {
    public void run() {
        synchronized (I2) {
            synchronized (I1) {
                a--;
                b--;
            }
        }
    }
}

```

Select one or more:

1. nu poate aparea 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

Care dintre afirmatiile urmatoare sunt adevarate?

Select one or more:

1. Partionarea prin descompunere functională conduce în general la aplicații cu scalabilitate mai bună decât partionarea prin descompunerea domeniului de date.
2. Scalabilitatea unei aplicații paralele este determinată de numărul de taskuri care se pot executa în paralel.
3. Dacă numărul de taskuri care se pot executa în paralel crește liniar odată cu creșterea dimensiunii problemei atunci aplicația are scalabilitate bună.

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

1. pentru a avea o performanță cat mai bună este preferabil ca numărul de subtaskuri în care se descompune calculul să fie cat mai mic
2. calculul se imparte în mai multe subtask-uri care se pot executa de către unități de procesare diferite
3. se obține performanță prin paralelizare dacă este nevoie de mai multe traversări ale pipeline-ului
4. pentru a obține o performanță cat mai bună este preferabil ca împărțirea pe subtaskurile să fie cat mai echilibrată

Cate thread-uri vor fi create (ce exceptia thr Main) și 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

```

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
- 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.app(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"

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 Momentul in care ajungem sa avem liste formate cu un singur elem
3. n

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

(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

```

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 afirmații sunt adevărate?

Se executa corect si afisaza 30.

Executia programului produce deadlock pentru ca procesul de la care primescste 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 afirmații sunt adevărate ? count INTEGER blocked:

COUNTAINER

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--;
    }
}
}
}
```

```
    }
}
}
}
```

7. 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

8.

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

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

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
False
```

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

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

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

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**
- 16
- 8

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**

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:

- (0->1), (1-2), (2->0) in ordine aleatorie
- (2->1), (1->0), (0->2) in ordine aleatorie
- 3. (0->1) urmata de (1-2) urmata de (2->0)
 - (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)**

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);

    #pragme omp parallel for private(i,k) shared (A,B,C,N) schedule(static)
    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
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
        {
            #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;
                }
            }
        }
    }
}

```

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

Care sunt variabilele shared, respectiv variabilele private, în 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);

    #pragme 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:

```

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

Cate thread-uri se vor crea:

- Cate core-uri exista pe CPU
- $9 + 1$ main
- $3 + 1$ main
- **$8 + 1$ main**

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_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:

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

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

Complexitate timp: $\log_2(1024) = 10$
Acceleratia: $TS / TP = 1024 / 10 = 102.4$
Eficienta: acceleratie / p = $102.4 / 1024 = 0.1$
Costul: $p * TP = 1024 * 10 = 10240$

```
#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
2. $3 + 1$ main
 - $15 + 1$ main

2 + 1 main

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

Aceleratia 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:

- 1. T_s/T_p**
- 2. $T_s/(p*T_p)$
 - T_p/T_s
 - $p*T_s/T_p$

```

#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 ?

- Fals, deoarece fiecarui thread ii vor fi asociate task-uri independente astfel incat nu este posibila o suprapunere in calcule.
- Adevarat, pentru ca paralelizarea for este dinamica daca nu se specifica explicit \
- Adevarat pentru ca fiecare thread acceseaza factorial[i-1], element ce poate fi modificat de alt thread

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

1. MISD
2. SIMD
- 3. MIMD**
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
- 2. timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential**
3. aceleratia inmultita cu numarul de procesoare este de acelasi ordin de marime cu timpul secvential

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
- interactiunii interproces**
- timpului necesar distributiei de date**
- calcul in exces (repetat de fiecare proces/thread)

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_size(MPI_COMM_WORLD, &myrank);

    • = (int *) malloc( nprocs * sizeof(int));
    • = (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;
}

Nu s rasp corecte
1.
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);
}
```

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

Select one or more:

- 75
- 25
- 3.4

```
#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
            {
                #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 e bun, 2 e rau

- ✓ 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 <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);

    #pragme 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?
```

1. Fals
2. Adevarat
3. Depinde de versiunea compilatorului folosita

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 (I2) { synchronized (I1) {
    a--; b--;
}
}
}
}
}
```

Select one or more:

1. nu poate aparea 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

Beziultatul executiei este nedeterminist.

Care dintre afirmațiile următoare sunt adevărate?

Select one or more:

- Partitionarea prin descompunere functională conduce în general la aplicații cu scalabilitate mai bună decât partitionarea prin descompunerea domeniului de date.
 - Scalabilitatea unei aplicații paralele este determinată de numărul de taskuri care se pot executa în paralel.

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

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);
}
}

```

- DA (conform Fisier final PPD)

b)nu

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

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

(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_INT, 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

- Scalabilitatea este mai mare pentru sistemele:
 - MPP
 - SMP
- Latenta memoriei este .
 - rata de transfer a datelor din memorie catre processor
 - timpul in care o data ajunge sa fie disponibila la procesor dupa ce s-a initiat cererea.
- Asigurarea cache coherency determina pentru scalabilitate o:. Single choice.
 - Scadere
 - Crestere
 - Nu este legatura
- Un calculator cu 1 procesor permite executie paralela? Required to answer. Single choice.
 - Da
 - Nu
- "Context Switch" este mai costisitor pentru:
 - Procese
 - Threaduri
- Thread1 executa { $a=b+1$; $a=a+1$ } si Thread2 executa { $b=b+1$ }. Apare "data race"? Required to answer. Single choice.
 - Da
 - Nu
- Thread1 executa { $c=a+1$ } si Thread2 executa { $b=b+a$ }. Apare "data race"?
 - Da
 - Nu
- Intr-o executie determinista poate sa apara "race condition".
 - Fals
 - Adevarat
- Granularitatea unei aplicatii paralele este
 - Definita ca dimensiunea minima a unei unitati secentiale 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
- Granularitatea unei aplicatii paralele este de droit sa fie
 - Mica
 - Mare

- Granularitatea unui sistem parallel este de droit sa fie
 - Mica
 - Mare
- Eficienta unui program parallel care face suma a 2 vectori de dimensiune n folosind p procese este:
 - Maxim 1
 - Minim 1
 - Egala cu p
- Costul unei aplicatii paralele este optim daca
 - $C=O(T_s * \log p)$
 - $C=O(T_s)$
 - $C=\Omega(\Theta(T_s))$
- Un semafor care stocheaza procesele care asteapta intr-o multime, se numeste:
 - Strong Semaphore (semafor puternic)
 - Weak Semaphor (semafor slab)
 - Semafor binar
- 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
- Fata de Monitor, Semaforul este o structura de sincronizare:
 - De nivel inalt
 - De nivel jos

De acelasi nivel

CS (D): PPD - Programare Paralela si Distribuita

Activity Out Teams Assignments Calendar Calls Files ... Apps Help

Q Search Close

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)

Process 0/3|Process 1/3|Process 2/3 Regards! ★

Process 0/3|Process 1/3|Process 2/3 Regards! Regards! ★

Process 2/3|Process 1/3|Process 0/3 Regards! Regards! Regards! ★

Process 1/3|Process 2/3|Process 0/3 Regards! Regards! Regards! ★

Process 1/3|Process 0/3|Process 2/3 Regards! ★

Go back to thank you page

10:54 AM 10/03/2020

CS (D): PPD - Programare Paralela si Distribuita

Activity Out Teams Assignments Calendar Calls Files ... Apps Help

Q Search Close

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

```
int main(int argc,char **argv) {
    // var declarations, 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);
    }
    // ... finalize
}
```

(3/3 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)
```

Activity Out Teams Assignments Calendar Calls Files ... Apps Help

Q Search Close

10:57 PM 10/03/2020

```
#pragma omp parallel shared(chunk,a) private(i,tid) num_threads(  
{  
    tid = omp_get_thread_num();  
    idx = idx + chunk * tid;  
    for (i = idx; i < idx + chunk; i++)  
        a[i] = tid + 1;  
}  
for (i = 0; i < n; ++i)      cout << a[i] << " ";
```

00111222333

00011223300 ✓★

11122233300

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

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

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

- un set permanent de variabile ce reprezinta resursa critica,
- un set de proceduri ce reprezinta operatii asupra variabilelor si
- un corp (secventa de instructiuni).

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, complexitatea 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 secentiale dintr-un program, exprimata in numar de instructiuni. – Prin unitate secentuala se intlege 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 fi 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 conditională;

Operatii: wait; signal ; [broadcast]

– O variabila conditională 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 aplicatie paralela este cea de tip paralel din punct de vedere al costului, daca valoarea acestuia este egala, sau este de acelasi ordin de marime cu cea mai buna varianta secvențială;
- Aplicatia este eficienta din punct de vedere al costului daca $C_p = O(t_1 \log p)$.

Costul unui sistem paralel (algoritm + sistem) • Cost = $p \times T_P$

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

Adunare n numere cu p procesoare (ambele sunt puteri ale lui 2) • Fiecare 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 paralelizare (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 optim

3. race condition si zona critica

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

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

```

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

```

Soluții: atomicizarea zonei critice o dezactivarea preemptionei 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 împartita 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];
    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. dedlocks pe threads, procese

- Deadlock – situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecarea 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.

Astfel semnalul notify() se pierde.

Acest lucru poate conduce la situatii in care un thread asteapta nedefinit, pentru ca mesajul corespondent 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 apelata 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 sectiunii critice așteaptă
- V(s)
 - Efect : incrementarea valorii semaforului.
 - se apelează la sfârșitul sectiunii critice și semnifică eliberarea acesteia pt. alte procese.

Semafor Binar • Valoarea semaforului poate lua doar valorile 0 si 1 Valoarea => poate fi de tip Boolean

Daca semaforul se foloseste fara a se mentine o evidenta a proceselor care asteapta intrarea in secțiunea critica 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 căruia valoare poate varia pe durata executiei diferitelor procese. -c(s) o multime de asteptare la semafor - conține referințe la procesele care așteaptă la semaforul s. + Operatiile P(s)/down si 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 căruia valoare poate varia pe durata executiei diferitelor procese. -c(s) o coadă de așteptare la semafor - conține referințe la procesele care așteaptă la semaforul s (FIFO). + Operatiile P/down si V/up

2. Eficiența și alea alea la o problema din viața trivială

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

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: $E_p(n) \approx 1$.

3. Scalabilitate

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

Scalabilitatea masoara modul in care se schimba performanta unui anumit algoritm in cazul in care sunt folosite mai multe elemente de procesare.

- Scalabilitatea unui sistem paralel este o masura a capacitatii de a livra o accelerare cu o crestere liniara in functie de numarul de procese folosite.

Sub6:

- **Distributie date si distributie functională**

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 mai fine. • Dupa 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. • În plus, ea are un rol important ca și tehnică de structurare a programelor.

Există mai multe tehnici de partitionare a datelor, care pot fi exprimate și formal prin funcții definite pe multimea indicilor datelor de intrare cu valori în multimea indicilor de procese. • Cele mai folosite tehnici de partitonare sunt prin "taiere" și prin "increștere" care corespund distribuțiilor liniare și ciclice. Distribuția liniară în curs este și cea –distribuție bloc

2.acceleratie + legea lui Ambhdal

Acceleratia ("speed-up"), notata cu Sp , este definită ca raportul dintre timpul de execuție al celui mai bun algoritm serial cunoscut, executat pe un calculator monoprocesor și timpul de execuție al programului paralel echivalent, executat pe un sistem de calcul paralel. Dacă se notează cu ts timpul de execuție al programului serial, iar tp timpul de execuție corespunzător programului paralel, atunci: $Sp(n) = ts(n)/tp(n)$.

(1.1) Numarul n reprezintă dimensiunea datelor de intrare, iar p numărul de procese folosite.

(Legea lui Amdahl) Fie α ($0 \leq \alpha \leq 1$) proporția operațiilor din algoritm care se executa sevențial (fractia lui Amdahl). Atunci:

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

3.client server vs peer to peer

Peer-to-peer se bazează pe sharingul de date și fisiere pe o rețea de utilizatori interconectați, rețeaua e mică deci nu există server în schimb fiecare utilizator acționează ca și client și server în același timp. Dezavantajul este securitatea deoarece oricine poate intra dacă are parola și că datele sunt mai instabile depinzând de fiecare membru al rețelei în parte. Avantajul este uzul minim de resurse

Client/server se bazează cu un server unde sunt stocate fisierile și parolile iar accesul la date este regulat de către un administrator de rețea, de aceea sunt mai sigure, dezavantajul este că ele pot fi foarte scumpe ca resurse.

s5:

1.Bariere de sincronizare exemplu in mpi

O bariera de sincronizare este un mecanism de baza în sincronizarea globală. Este introdusa în punctul în care fiecare proces trebuie să le aștepte pe celelalte, iar execuția se reia doar după 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 instrucțiuni și un singur flux de date; • SIMD: sistem cu un singur flux de instrucțiuni și mai multe fluxuri de date; • MISD: sistem cu mai multe fluxuri de instrucțiuni și un singur flux de date; • MIMD: cu mai multe fluxuri de instrucțiuni și mai multe fluxuri de date.

Ce este CUDA? • Compute Unified Device Architecture" • o platformă de programare paralelă-> • Arhitectura care foloseste GPU pt calcul general – permite creșterea 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 să scrie și să citească din memorie direct , pe cand în GPU ai nevoie să o uploadăz înainte de a fi accesată.

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 grupă mai multe fire (un "warp") sub un singur dispecer de instrucțiuni

3.ce este granularitatea? exemplu aplicatie cu granularitatea ideală=1 & exemplu in mpi

Nu stiu la exemplu aplicatie cu granularitate ideală și 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 afirmații sunt adevărate?

- 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 afirmații sunt adevărate ?

```

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

```

- 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

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:

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

- Adevărat**

- 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 execute 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 date (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:

- #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
- 3. Creează: 14, distrug 20**
4. Creează: 12, distrug 20

15. 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

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, în regiunea paralela a codului de mai sus

- Shared: a, b, c, i, t
- Shared: a, b, c / private: i, N, t
- 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][j] = (A[i][k] + B[i][k]);
}
}
}
```

16.2. Cate thread-uri se vor crea:

- 9 + 1 main
- 3 + 1 main
- 8 + 1 main

17. 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_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 – doar daca MPI_INIT e typo, altfel e var 1

TREBUIE INTREBAT

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, comunicatie, 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 secential / 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:

T_s = Complexitatea-timp a variantei secentiale

T_p= 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*T_p)
- 3. T_p/T_s
- 4. p*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 paraleлизarea 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 secential
- 2. timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secential

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

$$\text{Cost} = p \times TP$$

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 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);

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 secentiale a unui program. Daca pentru un caz concret avem procentul partii secentiale 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 secentiale fata de cea paraleizabila:

seq = fractia calcului secential; (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

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

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

25 ¼ => 4

20 ½ => 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 secential 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;  
}  
}  
} DA
```

}

`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);
    #pragme 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][k] = (A[i][k] + B[i][k]);
        }
    }
}
```

29. Care dintre afirmatiile urmatoare sunt adevarate?

1. Partitionarea prin descompunere functională conduce în general la aplicații cu scalabilitate mai bună decât partitionarea prin descompunerea domeniului de date.
2. Scalabilitatea unei aplicații paralele este determinată de numărul de taskuri care se pot executa în paralel.
3. Dacă numărul de taskuri care se pot executa în paralel crește liniar odată cu creșterea dimensiunii problemei atunci aplicația are scalabilitate bună.

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

1. pentru a avea o performanță cat mai bună este preferabil ca numărul de subtaskuri în care se descompune calculul să fie cat mai mic
- 2. calculul se imparte în mai multe subtask-uri care se pot executa de către unități de procesare diferite**
3. se obține performanță prin paralelizare dacă este nevoie de mai multe traversări ale pipeline-ului
- 4. pentru a obține o performanță cat mai bună este preferabil ca împărțirea pe subtaskurile să fie cat mai echilibrată**

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.app(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 ?

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 execuție la un grad maxim de paralelizare egal cu:

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

34. Arhitecturile UMA sunt caracterizate de:

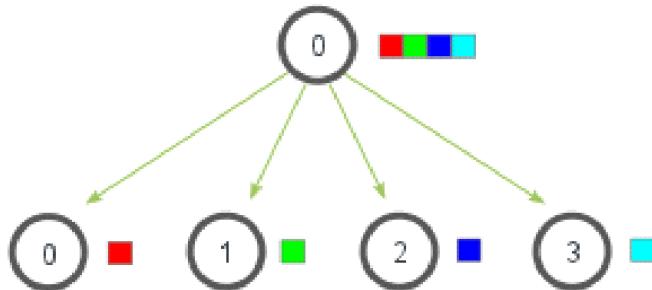
1. identificator unic pentru fiecare procesor
 2. același timp de acces pentru orice locație de memorie
- UMA = Uniform Memory Access

(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 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; // 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 trimită 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;  
}
```

MPI_Scatter



1. 9

2. 6

3. 12

Procesul 0 trimite a, adică $a + 0$, adică $a[0]$ proceselor 1 și 2, valoare memorată 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 date ajunge să fie disponibilă la procesor după ce s-a inițiat cererea.

3. Asigurarea cache coherency determină pentru scalabilitate o:

- a. Scădere
- b. Creștere
- c. Nu este legătura

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 determină 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 approxima ca fiind raportul din timpul total de calcul și timpul total de comunicare

10. Granularitatea unei aplicații paralele este de dorit să fie

a. Mica

b. Mare

11. Granularitatea unui sistem parallel este de dorit să fie

a. Mica

b. Mare

12. Eficiența 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 acceleratia poate să fie maxim p.

Deci, eficiența maxima poate să fie p / p = 1.

13. Costul unei aplicații paralele este optim dacă

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

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

- a. Strong Semaphore (semafor puternic) - FIFO (coada)
- b. Weak Semaphor (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 ca nu se termina 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)

- 001112223333
- 00011223300 
- 11122233300

Tid = 0, 1, 2

Tid = 0

Idx = $3 + 2 * 0 = 3$

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

Tid = 1

Idx = $3 + 2 * 1 = 5$

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

Tid = 2

Idx = $3 + 2 * 2 = 7$

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

Part Three

- **Care dintre afirmatiile urmatoare sunt adevarate?**
- Scalabilitatea unei aplicatii paralele este determinata de numarul de task-uri care se pot executa in paralel.
- Daca numarul de task-uri care se pot executa in paralel creste liniar odata cu cresterea dimensiunii problemei atunci aplicatia are scalabilitate buna
- 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)`:

- `num_threads(12)` ca si clauza intr-o directiva `#pragma omp parallel` – trb intrebata la examen
- `export OMP_NUM_THREADS = 11` – e ciudat ca e 11, ar trb sa fie 12
- `omp_get_num_threads()` – nu are treaba


```

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 există 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] |
|---|---------------------|--------------|------------|---------|
| • | [10 , | 102.4 , | 0.1, | 10240] |
| • | [10 , | 102.4 , | 10.24, | 1024] |
| • | [1 , | 1024 , | 1, | 1024] |
| • | [1 , | 102.4 , | 10, | 102.4] |

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

- SIMD
- MISD
- SISD
- 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 în mod aleator task-urile care la randul lor vor avea dimensiune diferită

5.Care varianta de definire pentru variabilele grid si block(de completat în locul comentariului) conduce la crearea unui număr 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 :

- dim3 grid(16); dim3 block(256);
- dim3 grid(4); dim3 block(256); 256x4=1024
- dim3 grid(4); dim3 block(16,16); 16x16x4 =1024
- 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 for 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:

- Adevarat
- Depinde de versiunea de openMP folosita
- **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:

- programul nu se termina pentru ca nu sunt bine definite comunicatiile
- programul se termina si afiseaza valoarea 10
- programul se termina si afiseaza valoarea 0

12. Care dintre urmatoarele afirmatii este adevarata

- Ordinea executiei block-urilor de tipul **section** este determinista.
- Fiecare block de tipul **section** este executat de un thread.
- 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 legatura



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"? *

(2/2 Points)

Da

Nu



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

(0/2 Points)

Da

Nu



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

(2/2 Points)

Adevarat

Fals



Curs5 – quiz MPI 3 5/8

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 * 4
(2 Points)
```

Da

Nu



MPI_Bcast ar fi trebuit sa fie executata de fiecare process in parte pentru a trimite(in cazul lui 0) / sa primeasca de la 0 valoarea lui x

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 incorrect - poate sa apara race condition  
- codul este corect:
- codul este incorrect pentru ca blocurile synchronized nu sunt bine definite

R: Codul nu este corect- apare Race Condition
Sincronizare pe lockuri diferite, insa se modifica aceeasi 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--;
    }
}

```

Curs 9 – OpenMP – 3/3

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 -> 3+2=5-1=4 => a[3]=a[4]=1

Tid = 1
Idx = 3 + 2 * 1 = 5
I = 5 -> 5+2=7-1=6 => a[5]=a[6]=2

Tid = 2
Idx = 3 + 2 * 2 = 7
I = 7 -> 7+2=9-1= 8 => 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 (chaturi, ...)

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, comunicatie, iar timpul necesar operatiunii 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 secentiale a unui program. Daca pentru un caz concret avem procentul partii secentiale 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  -----
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)

Aici e interpretabil raspunsul pentru
ca nu stim la ce se refera perechea
sender si receiver, ca si indexi e ok,
altfel ca si ce se trimite / ce se
primeste nu e ok – trb intrebat la
examen!!!

Consideram executia urmatorului 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 

Programare paralela si distribuita

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

Quiz navigation

Question 12

Not yet answered

Marked out of 1.00

Flag question

Ce valori corespund evaluarii teoretice a complexitatii-timp, acceleratiei, eficienței și costului pentru un program care face suma a 1024 de numere folosind 1024 de procesoare și un calcul de tip arbore binar? (Se ignora timpul de creare procese, distribuire date, comunicație, iar timpul necesar operatiilor de adunare se consideră egal cu 1.)

Select one or more:

1. {10, 1024, 0.1, 10240} ✓

2. [10, 1024, 1024, 1024]

3. [1, 1024, 1, 1024]

4. [1, 1024, 10, 1024]

Finish attempt ...

Time left 0:31:02

Next page

You are logged in as Raluca.Oana.Lenghel (Log out)

PPD

Data retention summary

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

Select one or more:

1. calculul se imparte în mai multe subtask-uri care se pot executa de către unități de procesare diferite

2. se obține performanța prin paralelizare dacă este nevoie de mai multe traversări ale pipeline-ului

3. pentru a obține o performanță cât mai bună este preferabil ca împărțirea pe subtaskurile să fie cât mai echilibrată

4. pentru a avea o performanță cât mai bună este preferabil ca numărul de subtaskuri în care se descompune calculul să fie cât mai mic

Examen PPD 20.01.2021 | 30 | Examen PPD 20.01.21 (page 17 of 16) | PPD

moodleubb

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 0:24:07

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>

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

Select one or more:

1. dim3 grid(16); dim3 block(256); ✓

2. dim3 grid(4); dim3 block(256); ✓

3. dim3 grid(4); dim3 block(16,16); ✓

4. dim3 grid(8, 8); dim3 block(4, 4); ✓

Next page

You are logged in as Raluca.Oana.Lenghel (Log out)

PPD

Data retention summary

Announcements

Jump to...

10:51 AM 1/29/2021

```

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

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 = 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-un exista pe CPU
- 3. 11 + 1 main
- 4. 3 + 1 main

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 partionarea prin descompunerea domeniului de date.

1. Un semafor este un semafor puternic (strong semaphore) daca

Are asociata o multime in care se stocheaza threadurile/procesele care asteapta

Are asociata o coada (FIFO) in care se stocheaza threadurile/procesele care asteapta ✓

Nu are asociata nici o colectie in care sa se stocheaza threadurile/procesele care asteapta

✓ Correct 4/4 Points

2. Variabilele conditionale se pot defini:

in cadrul unui monitor ✓

independent de existenta unui monitor dar conectate la un lacat (lock/mutex) ✓

independent de monitoare si lacate

✓ Correct 3/3 Points

3. Un monitor este un mecanism de sincronizare cu nivel de abstractizare

mai inalt decat nivelul de abstractizare
mai jos decat

✓ Correct 3/3 Points

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] << " ";
```

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

Idx = 3 + 2 * 0 = 3

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

Tid = 1

Idx = 3 + 2 * 1 = 5

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

Tid = 2

Idx = 3 + 2 * 2 = 7

I = 7 -> 7+2=9-1=8 => a[7]=a[8]=3

SCALABILITATE

✓ Correct 2/2 Points

1. Scalabilitatea este mai mare pentru sistemele

SMP

MPP ✓

✓ Correct 2/2 Points

2. Latenta memoriei este

rata de transfer a datelor din memorie catre procesor

timpul in care o data ajunge sa fie disponibila la procesor dupa ce s-a initializat cererea. ✓

✗ Incorrect 0/2 Points

3. Asigurarea cache coherency determina pentru scalabilitate o:

crestere

scadere ✓

nu este legatura ✗

MPI

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
```

DA

Nu

2. Cu ce urmare se va afisa la executie urmatorul 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,&numproc
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
printf( "Process%d/%d", myid, numprocs);
MPI_Finalize();
printf( "Regards!" );
return 0;
}
```

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!

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
```

MPI_Bcast ar fi trebuit sa fie executata de fiecare process in parte pentru a trimite(in cazul lui 0) / sa primeasca de la 0 valoarea lui x

Da

Nu

CONCURENTA GENERAL

1. Un calculator cu 1 procesor permite executie paralela?

*

DA ✓

Nu

2. "Context Switch" este mai costisitor pentru *

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 ✓

I. Care dintre urmatoarele afirmatii este adevarata:

1. sections este o directiva care nu determina executia in paralel a unui block de cod
2. nu este necesara gruparea codului in block-uri de tipul section pentru a indica ce dorim sa fie executat in paralel in cadrul unui block de tipul sections
3. Exista o bariera de sincronizare implicita la sfarsitul block-ului de tipul sections, astfel executia programului principal ramane suspendata pana cand toate threadurile termina de procesat taskurile asociate in cadrul acestui block.

II. Care va fi schema de distribuire a iteratiilor intre threadurile create:

1. Thread 0: p = 0-2
Thread 1: p = 3-5
Thread 2: p = 6-9
2. Thread 0: p = 0
Thread 1: p = 1 ??
Thread 2: p = 2
Thread 3: p = 3
Thread 4: p = 4
3. Thread 8: p = 8
4. Ordinea de procesare nu este determinista, pentru fiecare thread va prelua in mod aleator taskurile create conform schedule(dynamic).

III.Ce se poate intampla la executia programului urmator?

Cate thread-uri se vor crea

```
//////////  
public class Main {  
    static int numar = 2;  
  
    public static void main(String args[]) throws Exception{  
        ThrCall task1 = new ThrCall(numar,3);  
        ThrCall task2 = new ThrCall(numar,5);  
  
        ExecutorService pool = Executors.newFixedThreadPool(2);  
        Future<Integer> future1 = pool.submit(task1);  
        Future<Integer> future2 = pool.submit(task2);  
        pool.shutdown();
```

```

        Integer result1 = future1.get();
        Integer result2 = future2.get();
        System.out.println("rez1 = " + result1 + "; rez2 = " + result2);
    }
}

class ThrCall implements Callable<Integer> {
    int a, b;
    public ThrCall(int a, int b){
        this.a=a;
        this.b=b;
    }

    @Override
    public Integer call() throws Exception {
        int p = 1;
        for (int i = 0; i < b; i++) {
            p *= a;
        }
        return p;
    }
}
///////////////////////////////////////////////////

```

Select one or more:

1. nu apare “data-race”
2. nu poate aparea deadlock
3. apare “data-race”
4. poate aparea deadlock
5. afiseaza: rez1 = 8; rez2 = 32

IV.Ce se poate întâmpla la executia programului urmator?

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

```

mutex myMutex1, myMutex2;
void foo1(int n) {
    myMutex1.lock();
    for (int i = 10 * (n - 1); i < 10 * n; i++) {
        cout << " " << i << " ";
    }
    myMutex1.unlock();
}

```

```

void foo2(int n) {
    myMutex2.lock();
    for (int i = 10 * (n - 1); i < 10 * n; i++) {
        cout << " " << i << " ";
    }
    myMutex2.unlock();
}
int main()
{
    thread t1(foo1, 1);
    thread t2(foo2, 2);
    thread t3(foo1, 3);
    thread t4(foo2, 4);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}

```

//////////

Select one or more:

1. Poate aparea deadlock
- 2. Nu poate aparea deadlock**
3. Afiseaza in ordine numerele de la 0 la 39
- 4. Afiseaza aleator numerele din intervalul [0, 39]**
5. Afiseaza grupuri de cate 10 numere (0...9; 10...19; 20..29; 30..39); in interiorul grupului numerele sunt ordonate, iar afisarea grupurilor este aleatorie

V.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:

- 1. ---dim3 grid(8, 8); dim3 block(4, 4);**
- 2. ---dim3 grid(4); dim3 block(16,16);**
- 3. ---dim3 grid(4); dim3 block(256);**
4. dim3 grid(16); dim3 block(256);

VI.

Care dintre urmatoarele afirmații este adevarata?

1. Granularitatea unei aplicații paralele este determinată de numărul de taskuri rezultante prin descompunerea calculului
2. Granularitatea unei aplicații paralele este definită ca dimensiunea minima a unei unități secventiale dintr-un program, exprimată în număr de instrucții
3. Granularitatea unei aplicații paralele se poate approxima ca fiind raportul din timpul total de calcul și timpul total de comunicare

VII.

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

Apelul pragma omp parallel for din exemplul de mai sus paralelizează execuția la toate cele 3 structuri for ?

1. Depinde de versiunea openMP folosită
2. Adevarat

3. Fals

VIII.

```
public class Main {  
    static int numar = 1;  
    public static void main(String args[]) throws Exception{  
        ThrCall task1 = new ThrCall( 3 );  
        ThrCall task2 = new ThrCall( 4 );  
        ExecutorService pool = Executors.newFixedThreadPool( 2 );  
        Future<Integer> future1 = pool.submit( task1 );  
        Future<Integer> future2 = pool.submit( task2 );  
        pool.shutdown();  
        Integer result1 = future1.get();  
        Integer result2 = future2.get();  
        System.out.println( "rez1 = " + result1 + "; rez2 = " + result2 );  
    }  
}
```

```
static class ThrCall implements Callable<Integer> {  
    int n;  
    public ThrCall( int n )  
    {  
        this.n=n;  
    }  
    @Override public Integer call() throws Exception {  
        for (int i = 0; i < n; i++)  
        {  
            numar *= numar;  
        }  
        return numar;  
    }  
}
```

- 1. nu poate aparea deadlock ✓
- 2. nu poate aparea data race ✗
- 3. rez1=1 rez2=2
- 4. poate aparea data race ✓
- 5. poate aparea deadlock

Raspunsuri finale:

1

4

IX."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. omp_get_num_threads()
3. export OMP_NUM_THREADS=11"

X.

```
#include <stdio.h>
#include "omp.h"
void main(){
int i, k, p, j;
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,p,j) shared(A, B, C, N) schedule(dynamic)
for(p=0;p<N*N;p++){
    i=p/N;
    k=p%N;
    j=omp_get_thread_num();
    C[i][k]=A[i][k]+B[i][k]*j;
}
}
```

Poate aparea data race in exemplul de mai sus ?

1. Fals, desi rezultatul este nedeterminist nu exista data race.
2. Adevarat, deoarece variabila C este shared.
3. Adevarat, deoarece rezultatul poate sa fie diferit de la o rulare la alta.
4. Determinista

XI.

```
#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]);
    }
}

```

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

1. Fals
- 2. Adevarat**
3. Depinde de versiunea compilatorului folosit

XII. Care este rezultatul urmatorului program?

```

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

    const int MAX_MESSAGE_LENGTH=50;
    MPI_Request *request=new MPI_Request[nprocs];

    char *message=new char[MAX_MESSAGE_LENGTH];
    char *message_global=new char[nprocs* MAX_MESSAGE_LENGTH];

    sprint(message, "Hello from %d!!!", myrank);
    if(myrank!=0){
        MPI_Isend(message, strlen(message)+1, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
        &request[myrank]);
    }
    else{
        strcat(message_global, message);
        char message_recv[MAX_MESSAGE_LENGTH];;
        for(int i=1;u<nprocs;i++){
            if(MPI_SUCCESS==MPI_Recv(message_recv, MAX_MESSAGE_LENGTH,
            MPI_CHAR, I, tag, MPI_COMM_WORLD, &status)){
                strcat(message_global, message_recv);
            }
        }
        printf(message_global);
    }
    MPI_Finalize();
}

```

```
    return 0;  
}
```

Select one or more:

- 1. Hello from 0!!!Hello from 1!!!Hello from 2!!!
- 2. Hello from 0!!!Hello from 2!!!Hello from 1!!!
- 3. Hello from 1!!!Hello from 0!!!Hello from 2!!!

XIII. 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;  
}
```

Select one or more:

- 1. Programul nu se termina pentru ca nu sunt bine definite comunicatiile
- 2. Programul se termina si afisaza valoarea 0
- 3. Programul se termina si afisaza valoarea 10

XIV.

```
Void foo1(int n) {
    myMutex1.lock();
    myMutex2.lock();

    for(int i=10*(n-1);i<10*n;i++){
        cout<<" "<<i<<" ";
    }

    myMutex1.unlock();
    myMutex2.unlock();
}

void foo2(int n){
    myMutex1.lock();
    myMutex2.lock();

    for(int i=10*(n-1);i<10*n;i++){
        cout<<" "<<i<<" ";
    }

    myMutex1.unlock();
    myMutex2.unlock();
}

int main(){
    thread t1(foo1, 1);
    thread t2(foo2, 2);
    thread t3(foo1, 3);
    thread t4(foo2, 4);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

Select one or more:

1. Afisaza in ordine numerele de la 0 la 39
2. Afisaza aleator numerele din intervalul [0, 39]
3. Poate aparea deadlock
4. Afisaza grupuri de cate 10 numere {0...9; 10...19; 20...29; 30...39;} in interiorul grupului
numerele sunt ordonate, iar afisarea grupurilor este aleatoare
5. Nu poate aparea deadlock

XV.

...

```
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(l1){
            synchronized(l2){
                a--;
                b--;
            }
        }
    }
}
```

Select one or more:

1. Rezultatul exercitiului este nedeterminist ✓
2. Se afiseaza: a=4; b=4
3. Se afiseaza: a=20; b=20
4. Apare eroare pentru ca nu se accepta nested synchronized
5. Nu poate aparea deadlock ✓

XVI.

```
#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 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 taskurile care la randul lor vor avea dimensiune diferita

```

1 #include <stdio.h>
2 #include "omp.h"
3
4 void main() {
5     int i, k, p, j;
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, p, j) shared(A, B, C, N) schedule(dynamic)
15    for (p = 0; p < N * N; p++) {
16        i = p / N;
17        k = p % N;
18
19        j = omp_get_thread_num();
20
21        C[i][k] = A[i][k] + B[i][k] * j;
22    }
23}

```

La ce linie se creeaza/distrug thread-urile:

- 1. Creează: 19, distrug 21
- 2. Creează: 14, distrug 23
- 3. Creează: 4, distrug 23
- 4. Creează: 12, distrug 23

La ce linie se creeaza/distrug thread-urile:

```

1. public class Main {
2.     static Object l1 = new Object();
3.     static Object l2 = new Object();
4.     static int a = 4, b = 4;
5.
6.     public static void main(String args[]) throws Exception{
7.         T1 r1 = new T1();           T2 r2 = new T2();
8.         Runnable r3 = new T1(); Runnable r4 = new T2();
9.
10.        ExecutorService pool = Executors.newFixedThreadPool(4);
11.        pool.execute(r1); pool.execute(r2); pool.execute(r3); pool.execute(r4);
12.        pool.shutdown();
13.        while ( !pool.awaitTermination(60,TimeUnit.SECONDS)){ }
14.
15.        System.out.println("a=" + a + "; b=" + b);
16.    }
17.
18.    private static class T1 extends Thread {
19.        public void run() {
20.            synchronized (l1) {
21.                synchronized (l2) {
22.                    int temp = a;
23.                    a += b;
24.                    b += temp;
25.                }
26.            }
27.        }
28.    }
29.
30.    private static class T2 extends Thread {
31.        public void run() {
32.            synchronized (l1) {
33.                synchronized (l2) {
34.                    a--;
35.                    b--;
36.                }
37.            }
38.        }
39.    }
40. }

```

//////////

Select one or more:

- apare eroare pentru ca nu se accepta nested synchronized newFixedThreadPool(4)
- rezultatul executiei este nedeterminist ✓
- se afiseaza : a=20; b=20
- se afiseaza : a=4; b=4
- nu poate aparea deadlock ✓

WhatsApp (4) Crywank - Tomorrow Is Near! std:mutex - cppreference.com Messenger

← → C web.whatapp.com

FileList :: Login Resursele bibliograf... Anul 3 Informatica... Facultatea de Mate... Acces la resursele, c... Recent updates | G... Letterboxd • Social... Sci-Hub | Screening... State of AI Report 2...

Paralela si structura de date

0 / General / Examen PPD 20.01.21

Question 2 Not yet answered Marked out of 1.00 Flag question

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

Pentru sablonul de proiectare paralela "Pipeline" sunt adevarate urmatoarele afirmatii - bune 3, 4

Select one or more:

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

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

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

Next page

Type here to search

screenrec 11:50 AM ENG 2/16/2021

WhatsApp (4) Crywank - Tomorrow Is Presenting... Give control Stop presenting

← → C web.whatsapp.com

FileList :: Login Resursele bibliograf... Anul 3 Informatica... Facultatea de Mate... Acces la resursele, c... Recent updates | G... Letterboxd • Social... Sci-Hub | Screening... State of AI Report 2...

re paralela si distribuita

PPD / General / Examen PPD 20.01.21

Question 1
Not yet answered
Marked out of 1.00

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

Select one or more:

1. aceleratia inmultita cu numarul de procesoare este de acelasi ordin de marime cu timpul secential

2. timpul paralel este de acelasi ordin de marime cu timpul secential

3. timpul paralel inmultit cu numarul de procesoare este de acelasi ordin de marime cu timpul secential ✓

Announcements Jump to...

Type here to search screenrec

Question 34
Not yet answered
Marked out of 1.00

Se considera paraleлизarea sortării unui vector cu n elemente prin metoda "merge-s

In condițiile în care avem un număr nelimitat de procesoare, se poate ajunge la un

Select one or more:

$n / \log n$

n ✓

$\log n$

Question 33
Not yet answered
Marked out of 1.00

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

Select one or more:

1. [1, 1024, 1, 1024]

2. [10, 102.4, 10.24, 1024]

3. [10, 102.4, 0.1, 1024] ✓

4. [1, 102.4, 10, 102.4]

```

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 }
```

Are programul de mai sus o executie determinista ?

- 1. Da, pentru ca block-urile de tipul **section** vor fi executate secential si nu in paralel.
- 2. Nu, pentru ca nu vom obtine acelasi rezultat de ori cate ori am rula programul contand ordinea de executie a thread-urilor.
- 3. Da, pentru ca vom obtine acelasi rezultat de ori cate ori am rula programul chiar daca programul se va executa in paralel.

WhatsApp x +

← → C 🔒 web.whatsapp.com

FileList :: Login Resursele bibliograf... Anul 3 Informatica... Facultatea de Mate... Acces la resursele, c... Recent updates | G... Letterboxd • Social... Sci-Hu...

Courses / PPD / General / Examen PPD 20.01.21

Question 25
Not yet answered
Marked out of 1.00
Flag question

Overhead-ul in programele paralele se datoreaza:

Select one or more:

1. timpului necesar distributiei de date

2. interactiunii interproces

3. calcul in exces (repetat de fiecare proces/thread)

4. partitionarii dezechilibrate in taskuri

5. timpului necesar crearii threadurilor/proceselor

6. timpului de asteptare datorat sincronizarii

Type here to search

Question 22
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>

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

Select one or more:

1. dim3 grid(8, 8); dim3 block(4, 4);

2. dim3 grid(4); dim3 block(256);

3. dim3 grid(4); dim3 block(16,16);

4. dim3 grid(16); dim3 block(256);

WhatsApp

web.whatsapp.com

FileList :: Login Resursele bibliografice Anul 3 Informatica... Facultatea de Mat...

ubbduj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=20

Bogdan-Mihai Ognan

22 }
1. MPI_Gather(a, nprocs, MPI_FLOAT, b, nprocs, MPI_FLOAT, 0, MPI_COMM_WORLD);
2. 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);
}
3. 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, (nprocs-1), 10, MPI_COMM_WORLD, &status);

3.0
0.12
3.45
6.7

Paul yesterday Type a message

ROU 10:00 ROP 20.01.2

Type here to search

Urmatorului program se va executa cu 3 procese. Ce valoare se va afisa?

```
//////////  
1 int main(int argc, char *argv[] ) {  
2     int nprocs, myrank, mpi_err;  
3     int i, value=0;  
4     int *a, *b;  
5     MPI_Init(&argc, &argv);  
6     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  
7     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
8     if (myrank == 0) {  
9         a = (int *) malloc(nprocs * sizeof(int));  
10        for(int i=0;i<nprocs; i++) a[i]=i+1;  
11    }  
12    b = (int *) malloc( sizeof(int));  
13    MPI_Scatter(a, 1, MPI_INT, b, 1, MPI_INT, 0 ,MPI_COMM_WORLD);  
14    b[0] += myrank;  
15    MPI_Reduce(b, &value, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);  
16    if( myrank == 0) {  
17        printf("value = %d \n", value);  
18    MPI_Finalize();  
19    return 0;  
20 }
```

//////////

Select one or more:

1. 6

2. 12

 3. 9

WhatsApp x +

← → C web.whatsapp.com

FileList :: Login Resursele bibliograf... Anul 3 Informatica... Facultatea de Mate... Acces la resursele, c... Recent updates | G... Letterboxd • Social... Sci-Hub | Screening... State of AI R...

re paralela si distribuita

PPD / General / Examen PPD 20.01.21

Question 18
Not yet answered
Marked out of 1.00
Flag question

Care dintre urmatoarele afirmatii sunt adevarate?

Select one or more:

1. un monitor este definit de un set de proceduri ✓

2. un monitor poate fi accesat doar prin procedurile sale ✓

3. o procedura a monitorului nu poate fi apelata simultan de catre 2 sau mai multe threaduri ✓

4. toate procedurile monitorului pot fi executate la un moment dat

Type here to search

FileList :: Login Resursele bibliograf... Anul 3 Informatica... Facultatea de Mate... Acces la resursele, c... Recent updates | G... Letterboxd • Social... Sci-Hub | Screening... State of AI Report 2... screenrec

paralela si distribuita

/ General / Examen PPD 20.01.21

Question 17
Not yet answered
Marked out of 1.00
Flag question

Care dintre afirmatiile urmatoare sunt adevarate?

Select one or more:

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

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

Scalabilitatea unei aplicatii paralele este determinata de numarul de taskuri care se pot executa in paralel.

Next page

Announcements Jump to...

Course: Programare paralela si distribuita | Examen PPD 20.01.21 (page 16) | Examen PPD 20.01.21 | +
https://moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=15

moodleubb

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				

Question 16
Not yet answered
Marked out of 1.00
 Flag question

```
#pragma omp parallel for
for (i=1; i < 10; i++)
{
    factorial[i] = i * factorial[i-1];
}
```

Aveam parte de data race in exemplul de mai sus ?

1. Adevarat, pentru ca paraleлизarea for este dinamica daca nu se specifica explicit

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

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]`

Course: Programare paralela si distribuita | Examen PPD 20.01.21 (page 14) | Examen PPD 20.01.21 | +
https://moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=13

moodleubb

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				

Question 14
Not yet answered
Marked out of 1.00
 Flag question

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

Select one or more:

1. 75

2. 25

3. 4 ✓

[Announcements](#)

Course: Programare paralela si... X Examen PPD 20.01.21 (page 13) X Examen PPD 20.01.2021 | J x +

← → ⌂ ⌂ https://moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=12

moodleubb

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 1:09:27

Question 13

Not yet
answered
Marked out of
1.00

Flag question

```

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)
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 }
```

Cate thread-uri se vor crea:

✓ ✓ 8 + 1 main

9 + 1 main

Cate core-uri exista pe CPU

3 + 1 main

Course: Programare paralela si... X Examen PPD 20.01.21 (page 12) X Examen PPD 20.01.2021 | J x +

← → ⌂ ⌂ https://moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=11

moodleubb

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 1:09:55

Question 12

Not yet
answered
Marked out of
1.00

Flag question

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 0
3. programul se termina si afiseaza valoarea 10

Next page

WhatsApp | (4) Crywank - Tomorrow Is Nearly | std::mutex - cppreference.com | Teo îți-a trimis un mesaj | +

[←](#) [→](#) [⟳](#) [🔒](#) web.whatsapp.com

FileList :: Login | Resursele bibliograf... | Anul 3 Informatica... | Facultatea de Mate... | Acces la resursele, c... | Recent updates | G... | Letterboxd • Social... | Sci-Hub | Screening... | S...

Raluca Lenghel
13/02/2021 at 11:37

Course: Programare paralela si... | Examen PPD 20.01.21 (page 11 of 10) | Examen PPD 20.01.2021 | +

https://moodle.cs.ubbcluj.ro/mod/quiz/attempt.php?attempt=232955&cmid=3127&page=10

moodleubb

Finish attempt ...
Time left 1:10:10

```

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

```

```

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

```

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

1. Depinde de versiunea openMP folosita

2. Adevarat ✓

3. Fals ✓

[Clear my choice](#)

Type here to search

17 of 50

screenrec

```

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), urmata de (1->0), urmata de (0->2)

2. comunicatiile nu sunt bine definite pentru ca nu se realizeaza corect perechile (sender, receiver) ✓

3. (0->1) urmata de (1->2) urmata de (2->0)

4. (0->1), (1->2), (2->0) in ordine aleatorie

5. (2->1), (1->0), (0->2) in ordine aleatorie

AtomicNr

```
1. public class Main {
2.     public static void main(String[] args) throws InterruptedException {
3.         AtomicNr a = new AtomicNr(5);
4.
5.         for (int i = 0; i < 5; i++) {
6.             Thread t1 = new Thread(()->{ a.Add(3); });
7.             Thread t2 = new Thread(()->{ a.Add(2); });
8.             Thread t3 = new Thread(()->{ a.Minus(1); });
9.             Thread t4 = new Thread(()->{ a.Minus(1); });
10.
11.            t1.start(); t2.start(); t3.start(); t4.start();
12.            t1.join(); t2.join(); t3.join(); t4.join();
13.        }
14.        System.out.println("a = " + a);
15.    }
16. }
17.
18. class AtomicNr{
19.     private int nr;
20.     public AtomicNr(int nr){ this.nr = nr;}
21.
22.     public synchronized void Add(int nr) { this.nr += nr;}
23.     public synchronized void Minus(int nr){ this.nr -= nr;}
24.
25.     @Override
26.     public String toString() { return "" + this.nr;}
27. }
```

Select one or more:

1. Nr threaduri: 20; Valorile finale ale lui "a" pot fi diferite la fiecare rulare pentru ca programarea paralela este nedeterminista

2. Nr threaduri: 0; a = 20

3. Nr threaduri: 5; a = 5

4. Nr threaduri: 20; a = 20 ✓

5. Nr threaduri: 4; a = 20

```
1. mutex myMutex1, myMutex2;
2. void foo1(int n) {
3.     myMutex1.lock();
4.     for (int i = 10 * (n - 1); i < 10 * n; i++) {
5.         cout << " " << i << " ";
6.     }
7.     myMutex1.unlock();
8. }
9. void foo2(int n) {
10.    myMutex2.lock();
11.    for (int i = 10 * (n - 1); i < 10 * n; i++) {
12.        cout << " " << i << " ";
13.    }
14.    myMutex2.unlock();
15. }
16.
17. int main0
18. {
19.     thread t1(foo1, 1);
20.     thread t2(foo2, 2);
21.     thread t3(foo1, 3);
22.     thread t4(foo2, 4);
23.     t1.join(); t2.join(); t3.join(); t4.join();
24.     return 0;
25. }
```

```
14.     myMutex2.unlock();
15. }
16.
17. int main0
18. {
19.     thread t1(foo1, 1);
20.     thread t2(foo2, 2);
21.     thread t3(foo1, 3);
22.     thread t4(foo2, 4);
23.     t1.join(); t2.join(); t3.join(); t4.join();
24.     return 0;
25. }
```

Select one or more:

Poate aparea deadlock

Nu poate aparea deadlock ✓

Afiseaza in ordine numerele de la 0 la 39

Afiseaza aleator numerele din intervalul [0, 39] ✓

Afiseaza grupuri de cate 10 numere (0...9; 10...19; 20...29; 30...39); in interiorul grupului numerele sunt ordonate, iar afisarea grupurilor este aleatorie

Subiecte scris PPD din Word

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

Schita programului MPI este:

Initializari de MPI (Init, Status)

Fiecare process o sa aiba de prelucrat n^2 / p elemente din fiecare matrice.

O sa trimitem fiecarui process urmatoarele date:

- Dimensiunea unui array
- Cele 2 array-uri

Dupa ce fiecare process termina calculele, o sa trimita catre procesul 0 urmatoarele date:

- Dimensiunea array-ului rezultat
- Array-ul rezultat

Dupa ce 0 primeste toate rezultatele o sa afiseze matricea finala.

Finalize-ul din MPI.

Metrici:

Complexitatea timp: $O(n^2 / p)$

Acceleratia: $TS/TP = n^2 / (n^2 / p) = p$

Eficienta: acceleratie / p = p / p = 1

Costul: $TP * p = (n^2 / p) * p = n^2$

Este costul optim: n^2 este de acelasi ordin cu $O(n^2)$ => Adevarat => Solutia aleasa este optima d.p.d.v al costului

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

Granularitatea este o metrica de performanta pentru programele paralele. Aceasta este definita ca dimensiunea minima a unei unitati secentiale dintr-un program, exprimata in numar de instructiuni. Totodata, granularitatea unui algoritm poate fi aproximata ca fiind raportul dintre timpul total de calcul si timpul total de comunicare. O aplicatie de tipul "embarrassingly parallel programs" are o granularitate mare deoarece threadurile nu trebuie sa comunice intre ele.

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

Costul este o metrica de performanta ce masoara efortul necesar in obtinerea unei viteze de calcul mai mare. Aceasta este definit cu ajutorul numarului de procese si al timpului parallel de executie, iar formula este:

$$C = p * TP$$

P – nr procese

TP – timpul parallel de executie

Totodata, acesta poate sa fie optim sau efficient.

- Optim: costul este optim daca acesta este de ordinul timpului secvential

$$C = O(TS)$$

- Eficient: costul este efficient daca

$$C = O(TS * \log(p))$$

Adunarea a n numere cu p procese:

$$PP\ p = n$$

$$TP = \log(n)$$

$$\Rightarrow C = n * \log(n)$$

\Rightarrow Optim: $n * \log(n)$ nu e de ordinul $O(n)$ \Rightarrow nu e optim

\Rightarrow Eficient: $n * \log(n)$ de ordinul $O(n * \log(n))$ \Rightarrow e efficient

4. Race condition si zona critica

Zona critica a unui program reprezinta partea in care un thread actualizeaza o variabila. Fenomenul de “race condition” apare atunci cand rezultatul final poate fi influentat de ordinea in care se executa threadurile. Totodata avem 2 tipuri de race condition anume critical(ordinea in care se modifica variabilele e relevanta) si non-critical(ordinea in care se modifica variabilele nu este relevanta). In functie de problema, rezolvarea “race condition”-ului poate utiliza metode de sincronizare.

Ex:

```
Public class Test {  
    Private Integer a = 0;  
  
    Public synchronized void add() { // in acest caz am folosit synchronized pentru a  
    rezolva “race condition”-ul  
  
        a++; // aceasta este zona critica a programului nostru  
  
    }  
}
```

5. Deadlocks pe threads/procese

Deadlock-ul apare in momentul in care unul sau mai multe thread-uri / procese se blocheaza reciproc lucrul ce duce la o asteptare infinita. Acest fenomen apare din cauza dependentelor circulare in asteptarea unor resurse. Pentru a identifica deadlock-ul, putem realiza graful de dependente si in cazul in care avem un ciclu inseamna ca se produce deadlock-ul. Pentru preventirea acestora, se stabileste o ordine fixa pentru ordinea de blocare si deblocare a resurselor.

Ex:

T1 si T2 au 2 variabile commune a si b

T1	T2
Lock(a);	lock(b);
A++;	b++;
Lock(b);	lock(a);

6. Wait(), Notify(), NotifyAll() in Java

Aceste metode sunt specifice monitorului din Java. Fiecare obiect are un monitor asociat acesta fiind un mechanism de sincronizare.

Wait() - ii spune thread-ului ce a accesat obiectul current sa astepte dupa notificare si deblocheaza lock-ul asociat obiectului

Notify() – trezeste un thread ales aleator

NotifyAll() – trezeste toate thread-urile ce asteptau notificarea

Ex:

```
Public synchronized void add() {  
    This.list.add(1);  
    This.notifyAll(); // aici un thread notifica celelalte thread-uri ca a fost adaugat un  
element in lista  
}  
Public synchronized Integer remove() {  
    While ( this.list.isEmpty() ) {  
        If (done)  
            Return Null;  
        This.wait(); // in cazul in care lista este vida si nu putem extrage un element  
thread-ul ar trebui sa astepte  
    }  
    Return this.list.remove(0);  
}
```

7. Eficienta + exemplu

Eficienta este o metrica de performanta pentru un algoritm parallel. Aceasta masoara gradul de folosire al fiecarui process. Formula este:

$$E = \text{acceleratie} / p = TS / (TP * p)$$

$$\text{Accelartia} = \text{acceleratia algoritmului} = TS / TP$$

TS = timpul secential

TP = timpul paralel

P = numarul de procese

Adunarea a 2 vectori cu n elemente si p procesoare

$$TS = n$$

$$TP = n / p$$

$$\text{Acceleratia} = n / (n / p) = p$$

Eficienta = $p / p = 1 \Rightarrow$ Eficienta maxima => procesoarele sunt folosite la capacitate maxima

8. Scalabilitatea

Scalabilitatea este o metrica de performanta pentru un program parallel. Aceasta reprezinta evolutia in timp a unui program caruia ii creste complexitatea si numarul de procesoare.

Ex:

Adunarea a 2 vectori cu n elemente si p procesoare

TS = n

TP = n / p

Acceleratia = $n / (n / p) = p$

Eficienta = $p / p = 1 \Rightarrow$ Eficienta maxima => procesoarele sunt folosite la capacitate maxima

Daca eficienta este 1 putem spune ca acest program este scalabil.

9. Distributie date + distributie functionala

Există 2 strategii principale de partitionare:

- Descompunerea domeniului de date:

In aceasta strategie, domeniul de date este impartit pe baza structurilor de date prezente in cadrul problemei noastre. Astfel, se obtin mai multe componente ce pot fi manipulate independent. Aceasta strategie se preteaza in momentul in care domeniul de date este foarte mare.

- Descompunerea functionala:

Aceasta strategie este folosita in momentul in care accentul cade asupra algoritmului ce necesita mai multe calcule. Acestea sunt impartite ulterior in calcule mai fine.

10. Acceleratie + legea lui Amdahl

Acceleratia este o metrica de performanta pentru un program parallel. Aceasta masoara performanta aplicatiei dupa ce au fost adaugate resurse suplimentare. Acceleratia este egala cu raportul dintre timpul secvential si parallel.

$$\text{Acc} = \text{TS} / \text{TP}$$

Legea lui Amdahl sustine ca cresterea acceleratiei duce la modificarea raportului dintre partea secventiala si partea paralela.

Formula este:

$$1 / (\text{seq} + (\text{par} / p)) \Rightarrow p / ((p - 1) * \text{seq} + 1)$$

Seq + par = 1

Seq = procentul secvential

Par = procentul parallel

P = nr procese

In cazul in care p -> infinit => legea lui Amdahl devine $1 / \text{seq}$

11. Client-server vs peer-to-peer

Arhitectura client server are 2 componente mari, client si server. Clientul trimite cereri catre server, iar server-ul ii raspunde ulterior. Exista 2 tipuri de server:

- Stateless server – starea unei sesiuni e gestionata de catre client
- Statefull server – starea unei sesiuni e gestionata de catre server, aceasta fiind asociata id-ului clientului respectiv

Arhitectura peer-to-peer este destul de asemanatoare cu client-server, insa diferenta majora este ca fiecare nod din aceasta arhitectura poate sa fie atat client cat si server, in functie de volumul de cereri. Ex:Torrent

12. Bariera de sincronizare + exemplu

Bariera este un mechanism de sincronizare folosit atat la nivel de thread-uri, cat si la nivel de procese. Aceasta permite trecerea a n thread-uri / procese in acelasi timp. In momentul in care un thread ajunge la o bariera si da wait la aceasta, se face o verificare daca numarul de thread-uri ce asteapta la bariera este egal cu n sau nu, in cazul in care este egal, toate threadurile ce erau in starea de wait sunt notificate.

Exemplu:

Presupunem ca vrem sa afisam o lista ce are toate valorile egale cu dublul acestora si dorim sa folosim 2 thread-uri pentru a eficientiza acest proces de calcul si un al 3-lea pentru a afisa rezultatul. Sa avem nevoie de o bariera cu $n = 3$. Cele 2 thread-uri responsabile cu calculul sa dea wait in momentul in care isi termina sarcina, iar al 3-lea thread sa dea wait de la inceput, asteptandu-le pe celelalte 2 thread-uri sa termine calculul. Dupa ce bariera le da drumul, primele 2 thread-uri isi termina executia, iar cel al 3-lea isi termina executia dupa ce face afisarea.

13. Sisteme Flynn si ce tip de system este CUDA?

Sistemele Flynn sunt:

- SISD – Single Instruction Stream Single Data stream
- SIMD – Single Instruction Stream Multiple Data streams
- MISD – Multiple Single Instruction Stream Single Data stream
- MIMD - Multiple Single Instruction Stream Multiple Data streams

Majoritatea procesoarelor de azi sunt de tipul MIMD.

CUDA nu se incadreaza perfect in sistemele Flynn, dar cea mai apropiata este SIMD deoarece kernel-ul este executat de toate threadurile, fiecare putand executa independent instructiunile din kernel.

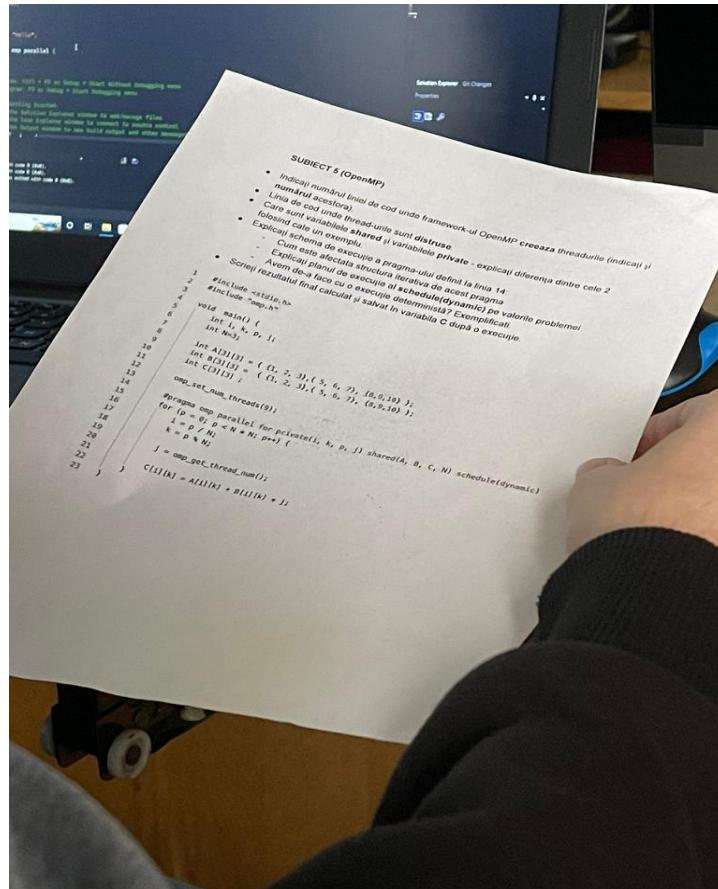
14. Ce este CUDA?

CUDA este o tehnologie dezvoltata de NVIDIA in anul 2007 ce a reusit sa foloseasca GPU-ul pentru a efectua calcule paralele. GPU-ul fata de CPU are mult mai multe nuclee dar nu la fel de puternice. Codul ce se executa pe GPU se numeste kernel. Arhitectura CUDA se bazeaza pe blocuri, fiecare bloc avand un numar de thread-uri allocate. Pentru a comunica cu GPU-ul, in C++ s-au introdus anumite "declaratii", anume:

- `_device_` - vizibil pe GPU, rulabil doar pe GPU
- `_host_` - vizibil pe CPU, rulabil doar pe CPU
- `_global_` - vizibil pe CPU, rulabil doar pe GPU

Totodata, pentru a putea efectua calcule pe placa video, este necesara copierea datelor din memoria RAM in memoria VRAM.

Subiecte Scris PPD



1. Threadurile se creeaza la linia 14 si sunt in numar de 9(8+1 main).
2. Threadurile se distrug la linia 23.
3. Variabilele private sunt i, j, k si p, iar variabilele shared sunt A, B, C si N. Diferenta dintre cele 2 tipuri este ca variabilele private sunt vizibile doar la nivel thread, fiind diferite pentru fiecare thread, iar variabilele shared sunt partajate intre toate threadurile.
4. In acest pragma se incercă paralelizarea for-ului folosindu-ne de un schedule dynamic. Acest tip de schedule nu este chiar balansat in ideea ca workload-ul nu este distribuit uniform. Pragma for distribuie unui thread un indice in cazul nostru p, reprezentand sarcina pe care trebuie sa o rezolve. In cazul de fata nu putem spune ca programul este determinant pentru ca pentru un indice p, valoarea lui j poate sa fie diferita.
5. 1, 2, 3
5, 6, 7
8, 9, 10

Subiecte de la curs:

1. Explicati conceptul de semafor – explicatii si exemplificare:

Semaforul este un mechanism de sincronizare de nivel inalt(dar nu cel mai inalt). Permite intrarea simultana a maxim n procese intr-o zona critica si in momentul in care un process paraseste zona critica, semaforul o sa selecteze urmatorul process. Exista 2 tipuri de semafoare:

- Semafoare Weak
- Semafoare Strong

Diferenta dintre cele 2 este ca procesele asteapta intr-o multime in cazul semafoarelor Weak, iar in cazul semafoarelor Strong procesele asteapta intr-o coada ce functioneaza dupa principiul FIFO acest lucru prevenind starvation-ul. Totodata, semaforul are 2 metode:

- Down – se apeaza in momentul in care un process vine la semafor si aici avem 2 cazuri:
 - Mai este loc in zona critica => procesul decrementeaza variabila ce tine numarul de procese ce acceseaza in acel moment zona critica
 - Nu mai este loc in zona critica => procesul este pus intr-o multime / coada de asteptare
- Up – se apeleaza in momentul in care un process paraseste zona critica, moment in care contorul intern al semaforului este incrementat.

Exemplu:

Am putea folosi un semafor pentru un server ce poate gestiona maxim n cereri in acelasi timp. Codul ar arata in urmatorul fel:

Client-ul face un request

```
// incercă să vădă dacă server-ul îl poate răspunde la request  
Semaphore.down();
```

```
... // se executa request-ul
```

```
// spune faptul că s-a terminat de executat request-ul  
Semaphore.up();
```

2. Explicati conceptul de monitor – explicatii si exemplificare

Monitorul este un mechanism de sincronizare high level. Acesta este format din attribute si metode. Totodata, metodele unui monitor nu pot fi accesate simultan, iar singura modalitate de accesa monitorul este utilizarea metodelor aferente acestuia. Cele 2 operatii specifice acestui mechanism de sincronizare sunt:

- Wait – pentru a ii spune proceseului current sa astepte
- Notify – pentru a le spune proceselor faptul ca, un alt proces a parasit zona critica

Exemplu:

O sa ne folosim de monitorul afferent obiectelor din Java. Presupunem ca avem un producer ce adauga numere intr-o lista, numere ce trebuie afisate pe ecran de catre consumeri. Implementarea ar arata asa:

```
Public class ListaPartajata() {  
  
    Private List<Integer> lista;  
  
    Private bool done;  
  
    Public ListaPartajata() {  
  
        This.lista = new ArrayList<>();  
  
        This.done = false;  
    }  
  
    Public synchronized void setDone() { // le comunicam thread-urilor ca s-a terminat  
    citirea  
  
        This.done = true;  
  
        This.notifyAll();  
    }  
  
    public synchronized void add(Integer elem) { // in momentul in care producer-ul  
    adauga un elemnt, ii notifica pe consumers de acest lucru  
  
        this.lista.add(elem);  
  
        this.notifyAll();  
    }  
}
```

Public synchronized Integer remove() { // consumerii incearca sa preia un element din lista

```
    While (this.list.size().equals(0)) {  
        If ( this.done) {  
            Return Null;  
        }  
        This.wait();  
    }  
    Return this.list.remove(0);  
}
```

Avem un producer ce adauga elemente intr-o lista si mai multi consumers ce incearca sa dea remove la un element pe care sa-l afiseze ulterior. In acest caz consumerii nu pot sa astepte la nesfarsit ca producer-ul sa adauge un element asa ca o sa faca wait(), iar in momentul in care producer-ul o sa adauge unul sau mai multe elemente, o sa faca notifyAll().

3. Explicati conceptul de variabila conditională – explicatii si exemplificare

Variabila conditională este un mechanism de sincronizare ce rezolva problema busy-waiting. Aceasta are o coada in care stocheaza threadurile si 3 metode:

- Wait – ii spune threadului current sa astepte ca nu a fost indeplinita conditia
- Notify – deblocheaza un thread din coada
- NotifyAll – deblocheaza toate threadurile din coada

Exemplu:

```
// se poate folosi si exemplul de mai sus
```

Avem un producer ce adauga elemente intr-o lista si mai multi consumers ce incearca sa dea remove la un element pe care sa-l afiseze ulterior. In acest caz consumerii nu pot sa astepte la nesfarsit ca producer-ul sa adauge un element asa ca o sa faca wait(), iar in momentul in care producer-ul o sa adauge unul sau mai multe elemente o sa faca notifyAll().

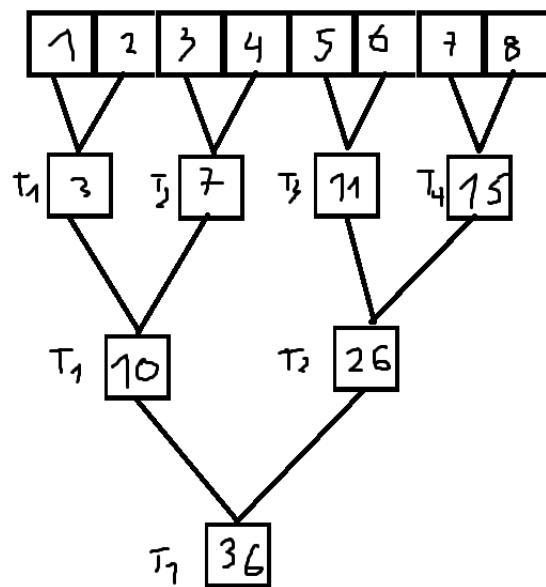
4. Divide&impera – sablon de programare paralela - explicatii si exemplificare cu evaluare complexitate si acceleratie + DOP

Dupa cum ii spune si in nume, rolul acestui algoritm este de a imparti problema in probleme mai mici ce pot fi rezolvate independent, iar rezultatul este construit pe baza rezultatelor problemelor mai mici.

Ca si exemplu am ales adunarea a n elemente. Sa zicem ca avem urmatoarele numere:
 $1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \Rightarrow \text{suma} = 36$

DOP reprezinta numarul de procese ce se poate efectua in parallel.

$$\text{DOP} = n / 2$$



Complexitate:

Secventiala: $O(n)$

Paralela: $O(\log(n))$

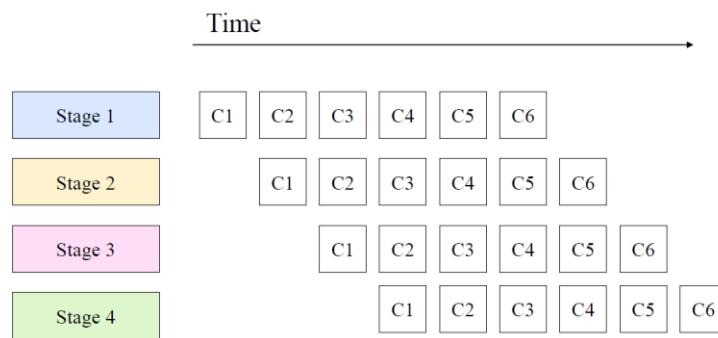
Acceleratie:

$$Sp = acc = TS / TP = n / \log(n)$$

5. Pipeline – sablon de programare paralela - explicatii si exemplificare cu evaluare complexitate si acceleratie

Pipeline-ul este un sablon de programare paralela ce este format din mai multe subtask-uri independente(adica se poate executa in parallel cu alte subtask-uri) si echilibrate(asa ar fi ideal).

Exemplu:



6. Descompunere geometrica (cazul unidimensional si bidimensional) - sablon de programare paralela - explicatii si exemplificare cu evaluare

Acest sablon a fost utilizat in cadrul laboratorului si presupune impartirea balansata a workload-ului intre procese astfel incat diferența dintre maximul workload-ului si minimul workload-ului sa fie 1.

Avem 2 cazuri:

- Cazul unidimensional:

Presupunem un array cu n elemente si p procese. Fiecare process o sa primeasca n/p elemente iar in cazul in care impartirea nu este exacta, impartim restul unul cate unul la threaduri.

Ex: $[1, 2, 3, 4, 5]$ $n = 5$; $p = 3$

$$\text{Cat} = 5 / 3 = 1$$

$$\text{Rest} = 5 \% 3 = 2$$

$$\Rightarrow T1 = [1, 2]; T2 = [3, 4]; T3 = [5]$$

- Cazul bidimensional:

Presupunem o matrice $N \times M$ si p procese. Fiecare process o sa primeasca $N * M / p$ elemente iar in cazul in care impartirea nu este exacta, impartim restul unul cate unul la threaduri.

Ex: $[[1, 2], [1, 2], [1, 2], [1, 2]]$ $N = 4$ $M = 2$; $p = 3$

$$\text{Cat} = 8 / 3 = 2$$

$$\text{Rest} = 8 \% 3 = 2$$

$$\Rightarrow T1 = [1, 2, 1]; T2 = [2, 1, 2]; T3 = [1, 2]$$

Dupa cum se poate observa abordarea este una similara cu cazul unidimensional. Se recomanda liniarizarea matricii si pentru a calcula

indicele liniei si indicele coloanei ne putem folosi de index-ul elementului current.

Matrice liniarizata: [1, 2, 1, 2, 1, 2, 1, 2]

Luam elementul de pe pozitia 3, adica numarul 2.

Linia = poz / M = 3 / 2 = 1

Col = poz % M = 3 % 2 = 1

7. Matrici de evaluare teoretica a performantei programelor paralele exemplificari Legea lui Amdahl si a lui Gustafson

Metricile de evaluare a performantei programelor paralele sunt:

- Acceleratia – masoara raportul dintre timpul serial si timpul parallel
- Eficienta – reprezinta raportul dintre acceleratie si numarul de procese
- Scalabilitatea – prezinta evolutia unui program ce isi marea atat complexitatea, cat si resursele disponibile in timp
- Costul – suma timpului pe care il petrece fiecare processor in rezolvarea problemei – formula $P * TP$
- Complexitatea – 2 tipuri, complexitatea de timp ce masoara timpul de executie pentru algoritm si complexitatea de spatiu ce masoara memoria folosita

Legea lui Amdahl:

- accelerarea procesarii depinde de raportul dintre partea sequentiala si partea paralela
- formula:

$$Acc = 1 / (RS + RP / p)$$

RS – raportul sesequential

RP – raportul parallel

Legea lui Gustafson:

- Cu cat dimensiunea problemei creste, se micsoreaza partea seriala in procent
- Formula:

$$Acc = seq + p * par$$

seq – raportul sesequential

par – raportul paralel