1.Care din următoarele afirmatii sunt adevarate?

#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;

}

a.Se executa corect si afisaza 30.

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

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

2.Care dintre urmatoarele afirmatii sunt adevarate ?​

count INTEGER

blocked: CONTAINER

**down**

**do**

if count > 0 then

count: = count -1

**else**

blocked.add(P) --P is the current process

P.state:=blocked --block process P

end

end

**up**

**do**

if blocked.is\_empty ​**then**

cout:=count+1

**else**

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

Qstate:=ready -- unblock process Q

**end**

**end**

a.aceasta varianta de implementare defineste un strong semaphore

b**.**aceasta varianta de implementare defineste un weak semaphore

c.aceasta varianta de implementare nu este “starvation free”

d.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. Rezultatul executiei:

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;

}

a.

Salutari de la 0

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

programul finalizat cu succes

programul finalizat cu succes

b.

Salutari de la 3

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

c.

Salutari de la 2

Salutari de la 0

Salutari de la 1

programul finalizat cu succes

programul finalizat cu succes

programul finalizat cu succes

d.

Salutari de la 0

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

e.

Salutari de la 3

Salutari de la 1

Salutari de la 2

programul finalizat cu succes

programul finalizat cu succes

programul finalizat cu succes

f.

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--;

}

}

}

}

}

a. se afiseaza : a=9; b=9

b. se afiseaza : a=13; b=13

c. se afiseaza : a=12; b=12

d. nu poate aparea deadlock

e. se afiseaza : a=14; b=14

5. Cate thread-uri vor fi create (cu 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;}

};

a. Nr threaduri: 5; a = 5

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

c. Nr threaduri: 0; a = 20

d. Nr threaduri: 20; a = 15

e. Nr threaduri: 20; a = 5

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

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

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

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

d. 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? -> DA

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;

}

8. Care dintre urmatoarele afirmatii sunt adevarate?

a. un monitor este definit de un set de proceduri

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

c. un monitor poate fi accesat doar prin procedurile sale

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

Text

Description automatically generated

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);

}

}

a. se termina si afiseaza 1

b. nu se termina

c. se termina si afiseaza 2

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:

a. 128

b.16

c.8

12. Consideram urmatorul program MPI care se executa cu 3 procese. Intre ce perechi de procese se realizeaza comunicatia si in ce ordine se realizeaza comunicatiile

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;

}

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

b(2->1), (1->0), (0->2) in ordine aleatorie

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

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

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

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

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) -> aici se creeaza

for (i=0; i<N; i++) {

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

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

}

}

} -> aici se distruge

a. Creează: 17, distrug 18

b. Creează: 4, distrug 19

c. Creează: 14, distrug 20

d. Creează: 12, distrug 20

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

#include <stdio.h>

#include "omp.h"

void main() {

int i, t;

const int 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)-> aici se creeaza

#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;

}

}

}

}-> aici se distrug

a. Creează: 10, distrug 36

b. Creează: 16, distrug 36

c. Creează: 4, distrug 34

d. Creează: 12, distrug 36

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

a.Shared: a, b, c, i, t

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

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

16. Care sunt variabilele shared, respectiv variabilele private:

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]);

}

}

}

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

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

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

17. Cate thread-uri se vor crea:

a. 9 + 1 main

b. 3 + 1 main

c. 8 + 1 main

18. 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, 0, 10, MPI\_COMM\_WORLD);

if (myrank ==0)

printf("%d", value);

MPI\_Finalize( );

return 0;

}

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

b.programul se termina si afiseaza valoarea 0

c.programul se termina si afiseaza valoarea 10 -> daca la receive era cu 1 in loc de 0 atunci era asta

19. 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.)

a. [ 1 , 1024 , 1, 1024 ]

b. [ 10 , 102.4 , 0.1, 10240]

c. [ 1 , 102.4 , 10, 102.4 ]

d. [ 10 , 102.4 , 10.24, 1024 ]

20. Cate thread-uri se vor crea:

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;

}

}

}

}

}

1. Cate core-uri exista pe CPU
2. 3 + 1 main
3. 15 + 1 main
4. 2 + 1 main

21. Acceleratia unui aplicatii paralele se defineste folosind urmatoarea formula:

(Se considera: Ts = Complexitatea-timp a variantei secventiale

Tp= complexitatea-timp a variantei paralele

p=numarul de procesoare folosite pentru varianta paralela.

a. Ts/Tp

b. Ts/(p\*Tp)

c.Tp/Ts

d.p\*Ts/Tp

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

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

{

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

}

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

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

c.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]

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

1. MISD

2. SIMD

3. MIMD

4. SISD

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

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

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

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

25. Overhead-ul in programele paralele se datoreaza:

a. timpului necesar crearii threadurlor/proceselor

b. timpului de asteptare datorat sincronizarii

c. partitionarii dezechilibrate in taskuri

d.interactiunii interproces

e.timpului necesar distributiei de date

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

26. 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;

}

a.MPI\_Gather(a, nprocs, MPI\_FLOAT, b, nprocs,MPI\_FLOAT, 0 , MPI\_COMM\_WORLD);

b. 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);

c. 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);

}

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

a.75

b.25

c.4

28. Are programul de mai jos o executie determinista ?

#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;

}

}

}

}

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

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

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

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

#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]);

}

}

}

(X) a. Fals

(XX) b. Adevarat

c. Depinde de versiunea compilatorului folosita

30. Care dintre afirmatiile urmatoare sunt adevarate?

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

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

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

31. 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);

}

}

1. DA (conform Fisier final PPD)
2. NU

32. Arhitecturile UMA sunt caracterizate de:

a. identificator unic pentru fiecare procesor

b. acelasi timp de acces pentru orice locatie de memorie

33. Urmatorul 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;

}

(X) a. 9

(XX) b. 6

c. 12

35. Scalabilitatea este mai mare pentru sistemele:

a. MPP (Massively Parallel Processing)

b. SMP (Symmetric Processing, shared memory)

36. Latenta memoriei este:

a. rata de transfer a datelor din memorie către processor

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

37. Asigurarea cache coherency determina pentru scalabilitate o:

a. Scădere

b. Creștere

c. Nu este legatura

38. Un calculator cu 1 procesor permite execuție paralelă?

a. Da

b. Nu

39. “Context Switch” este mai costisitor pentru:

a. Procese

b. Thread-uri

40. Thread1 executa {a=b+1; a=a+1} si Thread2 executa {b=b+1}. Apare "data race"?

a. Da

b. Nu

Pentru ca în Thread2 se face write

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

a. Da

b. Nu

Pentru ca se face doar read

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

a. Fals

b. Adevărat

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

43. Granularitatea unei aplicații paralele este

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

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

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

44. Granularitatea unei aplicații paralele este de dorit sa fie

a. Mica

b. Mare

45. Granularitatea unui sistem parallel este de dorit sa fie

a. Mica

b. Mare

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

a. Maxim 1

b. Minim 1

c. Egala cu p

Eficienta = Acceleratia / p, iar acceleratia poate sa fie maxim p.

Deci, eficienta maxima poate sa fie p / p = 1.

47. Costul unei aplicații paralele este optim dacă

a. C=O(T\_s\*log p)

b. C=O(T\_s)

c. C=Omega(T\_s)

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

a. Strong Semaphone (semafor puternic) - FIFO (coada)

b. Weak Semaphor (semafor slab) - multime.

c. Semafor binar

49. 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

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

a. De nivel înalt

b. De nivel jos

c. De același nivel

Graphical user interface, application

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

Text, letter

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

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

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

– un set permanent de variabile ce reprezintă resursa critică,

– un set de proceduri ce reprezintă operaţii asupra variabilelor şi

– un corp (secvenţă de instrucţiuni).

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’ şi produce valori iniţiale pentru variabilele-monitor (cod de initializare).

• Apoi monitorul este accesat numai prin procedurile sale.

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

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

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

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

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

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

– Implies high communication overhead and less opportunity for performance enhancement

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

58. Var cond

– O abstractizare care permite sincronizarea conditionala;

Operatii: wait; signal ; [broadcast]

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

• Thread t apel wait =>

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

• Atunci cand t isi reia executia m se blocheaza

• Thread v apel signal =>

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

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

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

class CV {

Semaphore s, x;

Lock m;

int waiters = 0;

public CV(Lock m)

{ // Constructor

this.m = m;

s = new Semaphore();

s.count = 0;

s.limit = 1;

x = new Semaphore();

x.count = 1; x.limit = 1;

} // x protejeaza accesul la variabila ‘waiters’

59. 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: Cp(n) = tp(n) · p.

• O aplica=e paralela este op4ma din punct de vedere al costului, daca valoarea acestuia este egala, sau este de acelasi ordin de marime cu =mpul celei mai bune variante secven=ale; • aplica=a este eficienta din punct de vedere al costului daca Cp = O(t1 log p).

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

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

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

60.race condition si zona critica

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

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

public class Counter {

protected long count = 0;

public void add(long value){

this.count = this.count + value;

}

}

Soluţii: o atomicizarea zonei critice o dezactivarea preempţiei în zona critică o secvenţializarea accesului la zona critică

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

Divide&impera este bine cunoscuta din dezvoltarea algoritmilor secventiali. O problema este ımpartita ın 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

62. dedalocks 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; } }

63. 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 notifiAll() nu se salveaza in cazul in care nici un thread nu asteapta atunci cand sunt apelate.

Asfel semnalul notify()se pierde.

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

64. Semafoare

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

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

• Efect: - incercarea obtinerii accesului procesului apelant la secţiunea critică si decrementarea valorii.

- dacă v(s) <= 0 , procesul ce doreşte execuţia sectiunii critice aşteaptă

– V(s)

• Efect : incrementarea valorii semaforului.

• se apelează la sfârşitul secţiunii critice şi 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 sectiunea critica nu se poate asigura starvation-free

Un semafor ‘slab’ se poate defini ca o pereche {v(s),c(s)} unde: -v(s) este valorea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuţiei diferitelor procese. -c(s) o multime de asteptare la semafor - conţine referinţe la procesele care aşteaptă 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 valorea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuţiei diferitelor procese. -c(s) o coadă de aşteptare la semafor - conţine referinţe la procesele care aşteaptă la semaforul s (FIFO). + Operatiile P/down si V/up

65. Eficienta si alea alea la o problema din aia trivial

Aproximarea numarului π Un calcul de tip Monte Carlo se realizeaza pentru aproximarea numarului π prin urmatoarea metoda: Se considera un cerc de raza egala cu unitatea ınscris ıntr-un patrat.

Complexitatea acestui algoritm este O(n), unde n este numarul de puncte generate aleator de fiecare componenta. Varianta secventiala a acestui calcul are complexitatea O(np) ¸si prin urmare acest algoritm este un algoritm foarte eficient: Ep(n) ≈ 1.

66. Scalabilitate

Principala proprietate a sistemelor cu memorie distribuita, care le avantajeaza fata de cele cu memorie comuna, este scalabilitatea Scalabilitate aplica+e: abilitatea unui program paralel sa ob+na o crestere de performanta propor+onal cu numarul de procesoare si dimensiunea problemei.

Scalabilitatea masoara modul ın care se schimba performanta unui anumit algoritm ın cazul ın 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 procesoare folosite.

67. Distributie date si distributie functionala

Exista doua strategii principale de partitionare: – descompunerea domeniului de date si – descompunerea functionala.

In functie de acestea putem considera aplicatii paralele bazate pe: – descompunerea domeniului de date – paralelism de date, si – aplicatii paralele bazate pe descompunerea functionala.

Descompunerea domeniului de date • Este aplicabila atunci cand domeniul datelor este mare si regulat. Ideea centrala este de a divide domeniul de date, reprezentat de principalele structuri de date, ın componente care pot fi manipulate independent. • Apoi se partitioneaza operatiile, de regula prin asocierea calculelor cu datele asupra carora se efectueaza. • Astfel, se obtine un numar de activitati de calcul, definite de un numar de date si de operatii.

Descopunerea functionala este o tehnica de partitionare folosita atunci cand aspectul dominant al problemei este functia, sau algoritmul, mai degraba decat operatiile asupra datelor. • Obiectivul este descompunerea calculelor ın activitati de calcul cat mai fine. • Dupa crearea acestora se examineaza cerintele asupra datelor. • Focalizarea asupra calculelor poate revela uneori o anumita structura a problemei, de unde oportunitati de optimizare, care nu sunt evidente numai din studiul datelor. • In plus, ea are un rol important ca si tehnica de structurare a programelor.

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

68.acceleratie + legea lui Ambhdal

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

(Legea lui Amdahl) Fie α(0 ≤ α ≤ 1) proportia operatiilor din algoritm care se executa secvential (fractia lui Amdahl). Atunci:

- Partea seriala a algoritmului se executa ın timpul αts.

- Partea paralela a algoritmului se executa ın timpul (1−α)ts p .

- Intregul algoritm se executa ın timpul tp = αts + (1−α)ts p .

- Acceleratia relativa este RSp = (α+ 1−α p ) −1 care nu poate depasi α −1 (Legea lui Amdhal)

69.client server vs peer to peer

Peer-to-peer se bazeaza pe sharingul de date si fisiere pe o retea de useri interconectati, reteaua e mica deci nu exista server in schimb fiecare user actioneaza ca si client si server In acelasi timp. Dezavantajul este securitatea deoarece oricine poate intra daca are parola si ca datele sunt mai instabile depinzand de fiecare membru al retelei in parte. Avantajul este uzul minim de resurse

Client/server se bazeaza cu un server unde sunt stocate fisierele si parolele iar accesul la date este regulat de catre un administrator de retea, de aceea sunt mai sigure, dezavantajul este ca ele pot fi foarte scumpe ca resurse.

70.Bariere de sincronizare exemplu in mpi

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

Exemplu:

MPI\_Barrier

MPI\_Barrier (comm)

MPI\_BARRIER (comm,ierr)

71.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 instrucţiuni şi un singur flux de date; • SIMD: sistem cu un singur flux de instrucţiuni şi mai multe fluxuri de date; • MISD: sistem cu mai multe fluxuri de instrucţiuni şi un singur flux de date; • MIMD: cu mai multe fluxuri de instrucţiuni şi mai multe fluxuri de date.

Ce este CUDA? • Compute Unified Device Architecture” • o pla7orma de programare paralela-> • Arhitectura care foloseste GPU pt calcul general – permite cresterea performantei • Released by NVIDIA in 2007 • Model de programare – Bazat pe extensii C / C++ - pt a permite ‘heterogeneous programming’ – API pt gesQonarea device-urilor, a memoriei etc.

CUDA foloseste SIMD pt ca poate sa scrie si sa citeasca din memorie direct , pe cand in GPU ai nevoie sa o uploadez inainte de a fi accesata.   
În al doilea rând, atât SIMD, cât și GPU-urile sunt rău la codul extrem de fragil, însă SIMD e mai putin rău. Acest lucru se datorează faptului că GPU-urile grupă mai multe fire (un "warp") sub un singur dispecer de instrucțiuni

72.ce este granularitatea? exemplu aplicatie cu granularitatea ideala=1 & exemplu in mpi

Nu stiu la exemplu aplicatie cu granularitate ideala si mpi.

Graphical user interface, text, application

Description automatically generated

73. Care din variantele de mai jos au aceleasi efect cu apelul functiei omp\_set\_num\_threads(12):

a.num\_threads(12) ca si clauza intr-o directiva #pragma omp parallel

b.export OMP\_NUM\_THREADS = 11

c.omp\_get\_num\_threads()

Graphical user interface

Description automatically generated with medium confidence  
**3. Raspuns:** 1

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

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]);

}

}

}

a.

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

b.

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

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

75.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);

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

b.dim3 grid(4); dim3 block(256); 256x4=1024

c.dim3 grid(4); dim3 block(16,16); 16x16x4 =1024

d.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

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

void main() {

int i, j, k, t;

int N = 4;

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

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

int C[4][4];

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

{

#pragma omp schedule(dynamic)

for (i = 0; i < N; i++) {

t = omp\_get\_thread\_num();

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

C[i][j] = 0;

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

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

}

}

}

}

}

a.Adevarat

b.Depinde de versiunea de openMP folosita

c.Fals

77. Care dintre urmatoarele afirmatii este adevarata

a.Ordinea executiei block-urilor de tipul **section** este determinista.

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

c.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.

Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application, email

Description automatically generated

80.

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--;

}

}

}

R: Codul nu este incorrect- apare Race Condition

Text

Description automatically generated

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.

Text

Description automatically generated with medium confidenceA picture containing graphical user interface

Description automatically generatedText

Description automatically generated with low confidenceGraphical user interface, text, application

Description automatically generatedA screenshot of a computer

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated