l

5. Pthreads; sincronizări cu mutex, cond, barrier

Contents

5.1. Principalele tipuri de date şi funcţii de lucru cu threaduri

Tabelul următor prezintă principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

Fişere header	`<pthread.h>`
Specificare biblioteci	`-pthread`

Tipuri de date	`pthread_t`
	`pthread_mutex_t`
	`pthread_cond_t`
	`pthread_barrier_t`
	`pthread_rwlock_t`
	`sem_t`

Funcţii de creare thread şi aşteptare terminare	`pthread_create`
	`pthread_join`
	`pthread_exit`

Variabile mutex	`pthread_mutex_init`
	`pthread_mutex_lock`
	`pthread_mutex_unlock`
	`pthread_mutex_destroy`

Variabile condiţionale	`pthread_cond_init`
	`pthread_cond_wait`
	`pthread_cond_signal`
	`pthread_cond_broadcast`
	`pthread_cond_destroy`

Bariere	`pthread_barrier_init`
	`pthread_barrier_destroy`
	`pthread_barrier_wait`

Variabile reader/writer	`pthread_rwlock_init`
	`pthread_rwlock_wrlock`
	`pthread_rwlock_rdlock`
	`pthread_rwlock_unlock`
	`pthread_rwlock_destroy`

Semafoare	`sem_init`
	`sem_wait`
	`sem_post`
	`sem_destroy`

Funcţia de descriere a acţiunii threadului	`void* work(void* a)`

5.2. Capitalizarea cuvintelor dintr-o listă de fişiere text

Dorim sa transformăm un fişier text într-un alt fişier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapă cu literă mare. Ne propunem să prelucrăm simultan mai multe astfel de fisiere. Programul primeste numele a **n** fisiere de intrare, iar fisierele de iesire vor primi acelasi nume la care i se adauga terminatia .CAPIT. Programul va crea câte un thread pentru fiecare pereche de fişiere. Sursa `capitalizari.c` este:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000
pthread_t tid[100];
// pthread_t tid; // Vezi comentariul de la sfârşitul sursei
void* ucap(void* numei) {
    printf("Threadul: %ld ...> %s\n", pthread_self(), (char*)numei);
    FILE *fi, *fo;
    char linie[MAXLINIE], numeo[100], *p;
    strcpy(numeo, (char*)numei);
    strcat(numeo, ".CAPIT");
    fi = fopen((char*)numei, "r");
    fo = fopen(numeo, "w");
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]); // Cuvant incepe in coloana 0
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (p == NULL) break;
            p++;
            if (*p == '\n') break;
            *p = toupper(*p);
        }
        fprintf(fo, "%s", linie);
    }
    fclose(fo);
    fclose(fi);
    printf("Terminat threadul: %ld ...> %s\n", pthread_self(), (char*)numei);
}
int main(int argc, char* argv[]) {
    int i;
    for (i=1; argv[i]; i++) {
        pthread_create(&tid[i], NULL, ucap, (void*)argv[i]);
        // pthread_create(&tid, NULL, ucap, (void*)argv[i]); // Vezi comentariul
de la sfârşitul sursei
        printf("Creat threadul: %ld ...> %s\n", tid[i], argv[i]);
    }
    for (i=1; argv[i]; i++) pthread_join(tid[i], NULL);
    // for (i=1; argv[i]; i++) pthread_join(tid, NULL); // Vezi comentariul de
la sfârşitul sursei
    printf("Terminat toate threadurile\n");
}
```

Compilarea se face: `gcc -pthread capitalizari.c`, iar rularea `./a.out f1 f2 . . .`

In sursa de mai sus sunt trei linii comentariu, în care, în loc de `tid[i]` se foloseşte `tid`. Este vorba de a folosi o singură variabilă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare

thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, atunci funcţiile `pthread_join` vor aştepta numai după ultimul thread!

5.3. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia go

```
//Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2
...
package main
import (
    "bufio"
    "fmt"
    "io"
    "os"
    "strings"
    "bytes"
)
func ucap(numei string, finish chan<- struct{}) {
    print("     Capitalizare: "+numei+"\n")
    f,_ := os.Open(numei)
    fi := bufio.NewReader(f)
    fo,_ := os.Create(numei+".CAPIT")
    for {
        linie,err := fi.ReadString('\n')
        if err == io.EOF { break }
        r := 0
        b := byte(linie[0])
        u := string(bytes.ToUpper([]byte{b}))
        linie = u+linie[1:]
        for {
            i := strings.IndexByte(linie[r:], ' ')
            if i == -1 { break }
            i = r + i + 1
            r = i + 1
            if r >= len(linie) { break }
            b = byte(linie[i])
            u = string(bytes.ToUpper([]byte{b}))
            linie = linie[:i]+u+linie[(i+1):]
        }
        fmt.Fprint(fo, linie)
    }
    f.Close()
    fo.Close()
    print("     Terminat capitalizare "+numei+"\n")
    finish <- struct{}{}
}
func main() {
    finish := make(chan struct{})
    for i :=1; i < len(os.Args); i++ { go ucap(os.Args[i], finish) }
    for i :=1; i < len(os.Args); i++ { <- finish }
    print("Terminat toate capitalizarile\n");
}
```

5.4. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia python

```
# Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2
...
import sys
import os
import threading
```

```
def ucap(numei):
    print("     Capitalizare: "+numei+"\n")
    fi = open(numei, "r")
    fo = open(numei+".CAPIT", "w")
    while True:
        linie = fi.readline()
        if linie == "": break
        r = 0
        linie = linie[0].upper()+linie[1:]
        while True:
            i = linie[r:].find(" ")
            if i == -1: break
            i = r + i + 1
            r = i + 1
            if r >= len(linie): break
            linie = linie[:i]+linie[i].upper()+linie[(i+1):]
        fo.write(linie)
    fi.close()
    fo.close()
    print("     Terminat capitalizare "+numei+"\n")
def main():
    t = []
    for i in range(1, len(sys.argv)):
        t.append(threading.Thread(target=ucap, args=(sys.argv[i],)))
        t[i-1].start()
    for i in range(1, len(sys.argv)):
        t[i-1].join()
    print("Terminat toate capitalizarile\n")
main()
```

5.5. De ce sunt necesare variabilele mutex?

Să construim un program care are o variabilă globală **count** şi 1000 de threaduri. Fiecare thread incrementează de câte 1000 de ori variabila **count**. In acest scop să considerăm programul următor:

```
#include <pthread.h>
#include <stdio.h>
int count = 0;
pthread_t tid[1000];
//pthread_mutex_t exclusiv;
void* inc(void* nume) {
    for (int i = 0; i < 1000; i++) {
        //pthread_mutex_lock(&exclusiv);
        int temp = count; temp++; count = temp;
        //pthread_mutex_unlock(&exclusiv);
    }
}
int main(int argc, char* argv[]) {
    int i;
    //pthread_mutex_init(&exclusiv, NULL);
    for (i=0; i < 1000; i++)
        pthread_create(&tid[i], NULL, inc, NULL);
    for (i=0; i < 1000; i++) pthread_join(tid[i], NULL);
    //pthread_mutex_destroy(&exclusiv);
    printf("count=%d\n", count);
}
```

Evident, funcţia inc a threadului se poate scrie: **count += 1000**. In mod intenţionat am "prelungit" execuţia în funcţia threadului. Rezultatele unor execuţii repetate ale programului de mai sus sunt:

```
florin@ubuntu:~/pthreads$ gcc -pthread inc.c
florin@ubuntu:~/pthreads$ ./a.out
count=997000
```

```
florin@ubuntu:~/pthreads$ ./a.out
count=997000
florin@ubuntu:~/pthreads$ ./a.out
count=980000
florin@ubuntu:~/pthreads$ ./a.out
count=1000000
florin@ubuntu:~/pthreads$ ./a.out
count=991000
florin@ubuntu:~/pthreads$ ./a.out
count=996000
florin@ubuntu:~/pthreads$ ./a.out
count=980000
florin@ubuntu:~/pthreads$ ./a.out
count=1000000
florin@ubuntu:~/pthreads$ ./a.out
count=997000
florin@ubuntu:~/pthreads$ ./a.out
count=995000
florin@ubuntu:~/pthreads$
```

De ce acest comportament ciudat? Pentru că execuţia threadurilor se face în paralel, deci este posibil ca cele trei instrucţiuni ale corpului for din funcţia **inc** să se înterpătrundă, aşa încât două threaduri să memoreze succesiv valoarea lui **count** independent de reţinerea ei în variabila locală **temp**. Astfel, se pierde efectul unuia dintre threaduri.

Pentru a asigura derularea corectă, se impune utilizarea unui mutex:

Se declară variabila globală

```
pthread_mutex_t exclusiv
```

Corpul ciclului se va înlocui cu:

```
pthread_mutex_lock(&exclusiv);
int temp = count; temp++; count = temp;
pthread_mutex_unlock(&exclusiv);
```

5.6. Câte perechi de argumente au suma număr par?

La linia de comandă se dau **n** perechi de argumente despre care se presupune ca sunt numere întregi si pozitive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: Se va crea câte un thread pentru fiecare pereche. Trei variabile globale, cu rol de contoare, vor număra fiecare categorie de pereche.

Important: deoarece threadurile pot incrementa simultan unul dintre contoare, este necesar să se asigure accesul exclusiv la aceste contoare. De aceea, vom folosi o variabilă mutex care va proteja acest acces.

Intrebare: este corect sau nu să fie protejat fiecare contor printr-o variabilă mutex proprie? In exemplul nostru le protejăm simultan pe toate trei cu acelaşi mutex.

Sursa programului este:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXLINIE 1000
typedef struct {char*n1; char*n2;} PERECHE;
pthread_t tid[100];
PERECHE pereche[100];
```

```
// PERECHE pereche; // Vezi comentariul de la sfârşitul sursei
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
int pare = 0, impare = 0, nenum = 0;
void* tippereche(void* pereche) {
    int n1 = atoi(((PERECHE*)pereche)->n1);
    int n2 = atoi(((PERECHE*)pereche)->n2);
    if (n1 == 0 || n2 == 0) {
        pthread_mutex_lock(&mut);
        nenum++;
        pthread_mutex_unlock(&mut);
    } else if ((n1 + n2) % 2 == 0) {
        pthread_mutex_lock(&mut);
        pare++;
        pthread_mutex_unlock(&mut);
    } else {
        pthread_mutex_lock(&mut);
        impare++;
        pthread_mutex_unlock(&mut);
    }
}
int main(int argc, char* argv[]) {
    int i, p, n = (argc-1)/2;
    for (i = 1, p = 0; p < n; i += 2, p++) {
        pereche[p].n1 = argv[i];
        pereche[p].n2 = argv[i+1];
        pthread_create(&tid[p], NULL, tippereche, (void*)&pereche[p]);
        // !! Apelul urmator nu este corect, Vezi explicatia la sfârşitul sursei
        // pthread_create(&tid[p], NULL, tippereche, (void*)&pereche);
    }
    for (i=0; i < n; i++) pthread_join(tid[i], NULL);
    printf("perechi=%d pare=%d impare=%d nenum=%d\n",n,pare,impare,nenum);
}
```

In sursa de mai sus există un comentariu unde în loc de `pereche[p]` se foloseşte `pereche`. Este vorba de a folosi o aceaşi variabilă simplă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, se comite una dintre cele mai dese erori de logică în lucrul cu threaduri: Dacă se foloseşte o singură variabilă care transmite parametrul de intrare pentru toate apelurile `ucap` ale tuturor threadurilor, este posibil ca `(PERECHE*)pereche)->n1` sau `(PERECHE*)pereche)->n2` să nu preia intrările pregătite, ci să preia valori pregătite pentru threadul următor!

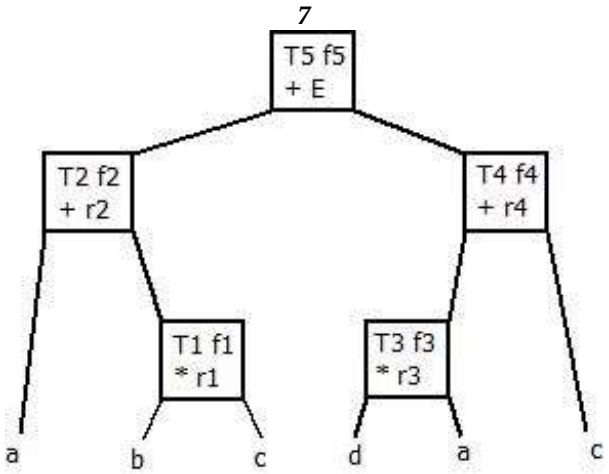
5.7. Evaluarea expresie aritmetică operator / thread şi paralelizare maximă

Se dă expresia aritmetică

`E = ( a + b * c ) + ( d * a + c )`

unde `a, b c, d` sunt numere întregi. Se cere evaluarea acestei expresii executând fiecare operaţie într-un thread separat şi cu lansarea a câtor mai multe threaduri în acelaşi timp.

Pentru rezolvare, vom folosi cinci threaduri `T1, T2, T3, T4, T5`, variabilele intermediare `r1, r3, r3, r4, E` şi funcţiile de thread `f1, f2, f3, f4, f5`. Arborele din figura următoare ilustrează evaluarea acestei expresii:



In cele cinci pătrate sunt ilustrate cele cinci threaduri care contribuie la rezolvarea problemei. Pentru fiecare thread se indică numele threadului, funcţia care îi determină funcţionarea, operaţia executată şi variabila unde se depune rezultatul operaţiei.

Sursa programului este dată mai jos:

```
# include <stdio.h>
# include <pthread.h>
pthread_t t[6];
int a=1, b=2, c=3, d=4, r1, r2, r3, r4, E;
void * f1 (void * x) {r1 = b * c;}
void * f2 (void * x) {r2 = a + r1;}
void * f3 (void * x) {r3 = d * a;}
void * f4 (void * x) {r4 = r3 + c;}
void * f5 (void * x) {E = r2 + r4;}
int main() {

    // O prima varianta de creare / join
    pthread_create(&t[1], NULL, f1, NULL);
    pthread_create(&t[2], NULL, f2, NULL);
    pthread_create(&t[3], NULL, f3, NULL);
    pthread_create(&t[4], NULL, f4, NULL);
    pthread_create(&t[5], NULL, f5, NULL);
    pthread_join(t[1], NULL);
    pthread_join(t[2], NULL);
    pthread_join(t[3], NULL);
    pthread_join(t[4], NULL);
    pthread_join(t[5], NULL);
    //Sfarsit varianta 1 - rezultat E=0 sau alte valori aiurea!
/*
    // Varianta corecta:
    // T2 lansat dupa ce se termina T1 si T4 lansat dupa ce se termina T3
    // T5 lansat numai dupa ce se termina T2 si T4
    pthread_create(&t[1], NULL, f1, NULL);
    pthread_create(&t[3], NULL, f3, NULL);
    pthread_join(t[1], NULL);
    pthread_join(t[3], NULL);
    pthread_create(&t[2], NULL, f2, NULL);
    pthread_create(&t[4], NULL, f4, NULL);
    pthread_join(t[2], NULL);
    pthread_join(t[4], NULL);
    pthread_create(&t[5], NULL, f5, NULL);
    pthread_join(t[5], NULL);
*/
    printf("E = %d\n", E);
}
```

Dacă se rulează prima variantă, în care nici un thread nu aşteaptă după altul, rezultatul este 0! Asta pentru faptul că threadurile care încarcă variabilele `r1 - r4` nu apucă să o facă şi threadul `T5` preia din ele valorile 0!

Varianta corectă este aceea în care `T2` şi `T4` aşteaptă să se termine mai întâi `T1` şi `T2`, apoi `T5` aşteaptă să se termine mai întâi `T2` şi `T4`. Rezultatul va fi, evident, 14.

5.8. De ce sunt necesare variabilele condiţionale?

In exemplul de mai sus unele threaduri au fost nevoite să aştepte terminarea altora pentru a îşi termina activitatea lor.
Sa ne imaginăm situaţia că un thread **`asteaptaeveniment`** NU poate trece de un punct, să zicem **A**, al execuţiei până când un alt thread **`produceeveniment`** ajunge cu execuţia într-un punct, să zicem **B**. (In literatură fenomenul se numeşte **rendezvouz**). Să schematizăm acest fenomen prin următorul program cu două threaduri, în care evenimentul de rendezvouz este schimbarea valorii unei variabile globale **eveniment**. Să urmărim această schemă:

```
#include <pthread.h>
#include <stdio.h>
int eveniment = 0;
// - - -
pthread_t tid[2];
void* asteaptaeveniment(void* nume) {
    // - - - Calcule - - -
    // A
    printf("A asteapta evenimentul\n");
    while (eveniment == 0) {
        ;
    }
    printf("A a primit evenimentul\n");
    // - - - alte calcule - - -
}
void* produceeveniment(void* nume) {
    // - - - Calcule - - -
    // B
    printf("B va produce evenimentul\n");
    eveniment = 1;
    printf("B a produs evenimentul\n");
    // - - - alte calcule - - -
}
int main(int argc, char* argv[]) {
    pthread_create(&tid[1], NULL, asteaptaeveniment, NULL);
    pthread_create(&tid[0], NULL, produceeveniment, NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

A asteapta evenimentul	B va produce evenimentul
B va produce evenimentul	B a produs evenimentul
B a produs evenimentul	A asteapta evenimentul
A a primit evenimentul	A a primit evenimentul

Marea problemă este aşteptarea evenimentului: **`while (eveniment == 0){ ; };`**
In modul de mai sus avem de-a face cu o **aşteptare activă**, threadul care aşteaptă ocupă procesorul cu execuţia lui **while**! Este de dorit ca până la apariţia evenimentului, threadul care aşteaptă **să nu ocupe procesorul**! (Din păcate, soluţia cu semnale, pe care am descris-o la procese NU MERGE, deoarece threadurile producător şi de aşteptare sunt în acelaşi proces.).

Rezolvarea se face printr-o variabilă condiţională. Pentru aceasta se declară o variabilă condiţională şi o variabilă mutex asociată ei:

```
pthread_cond_t variabila = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutexasociat = PTHREAD_MUTEX_INITIALIZER;
```

In locul constantelor **`PTHREAD*INITIALIZER`** se pot folosi funcţiile de iniţializare **`pthread*init`**.

Secvenţa de provocare a evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

```
pthread_mutex_lock(&mutexasociat);

eveniment = 1;
pthread_cond_signal(&variabila);

pthread_mutex_unlock(&mutexasociat);
```

```
pthread_mutex_lock(&mutexasociat);
printf("B va produce evenimentul\n");
eveniment = 1;
pthread_cond_signal(&variabila);
printf("B a produs evenimentul\n");
pthread_mutex_unlock(&mutexasociat);
```

Secvenţa de cedare a procesorului pe durata aşteptării evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

```
pthread_mutex_lock(&mutexasociat);

while (eveniment == 0) {
    pthread_cond_wait(&variabila,
                      &mutexasociat);
}

pthread_mutex_unlock(&mutexasociat);
```

```
pthread_mutex_lock(&mutexasociat);
printf("A asteapta evenimentul\n");
while (eveniment == 0) {
    pthread_cond_wait(&variabila,
                      &mutexasociat);
}
printf("A a primit evenimentul\n");
pthread_mutex_unlock(&mutexasociat);
```

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

```
B va produce evenimentul
B a produs evenimentul
A asteapta evenimentul
A a primit evenimentul
```

```
A asteapta evenimentul
B va produce evenimentul
B a produs evenimentul
A a primit evenimentul
```

Exemplul următor foloseşte trei threaduri care accesează un contor: unul anunţă când ajunge la valoarea 12, iar celelalte două contorul de câte şapte ori. De remarcat posibilitatea ca un thread să oprească celelalte două, dar cu grijă la stadiul contextului la oprire - pot rămâne mutex-uri blocate!

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int contor = 0;
pthread_mutex_t mutcontor = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condcontor = PTHREAD_COND_INITIALIZER;
pthread_t th[3];

void* incContor (void *id) {
    int i;
    printf ("START incContor %d\n", *(int*)id);
    for (i = 0; i < 7; i++) {
        sleep(random() % 3);
        pthread_mutex_lock (&mutcontor);
        contor++;
        printf("incContor: thread %d contor vechi %d contor nou %d\n",
*(int*)id, contor-1, contor);
        if (contor == 12) pthread_cond_signal (&condcontor);
        pthread_mutex_unlock (&mutcontor);
   }
```

```
        printf ("STOP incContor %d\n", *(int*)id);
    }

    void* verifContor (void *id) {
        printf ("START verifContor\n");
        pthread_mutex_lock (&mutcontor);
        while (contor <= 12) {
            pthread_cond_wait (&condcontor, &mutcontor);
            printf ("verifContor: thread %d contor %d\n", *(int*)id, contor);
            break;
        }
        //pthread_cancel(th[1]); // cu grija!
        //pthread_cancel(th[2]); // cu grija!
        pthread_mutex_unlock (&mutcontor);
        printf ("STOP verifContor\n");
    }

    int main () {
        int id[3] = { 0, 1, 2 };
        //creaaza cele 3 thread-uri
        pthread_create (&th[0], NULL, verifContor, &id[0]);
        pthread_create (&th[1], NULL, incContor, &id[1]);
        pthread_create (&th[2], NULL, incContor, &id[2]);
        //asteapta terminarea thread-urilor
        pthread_join (th[0], NULL);
        pthread_join (th[1], NULL);
        pthread_join (th[2], NULL);
        return 0;
    }
```

5.9. Bariera - exemplu

Pthread_barrier este o construcție care permite ca mai multe threaduri independente să „aștepte” în spatele unei „bariere” până când toate cele care aşteaptă ajung la barieră. Bariera este opțională în standardul POSIX, aşa că unele sisteme de operare nu o includ. Iată un exemplu simplu de utilizare:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#define THREAD_COUNT 4
pthread_barrier_t mybarrier;
void* threadFn(void *id_ptr) {
    int thread_id = *(int*)id_ptr;
    int wait_sec = 1 + rand() % 5;
    printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
    sleep(wait_sec);
    printf("thread %d: I'm ready...\n", thread_id);
    pthread_barrier_wait(&mybarrier);
    printf("thread %d: going!\n", thread_id);
    return NULL;
}
int main() {
    int i;
    pthread_t ids[THREAD_COUNT];
    int short_ids[THREAD_COUNT];
    srand(time(NULL));
    pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);
    for (i=0; i < THREAD_COUNT; i++) {
        short_ids[i] = i;
        pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
    }
```

```
    printf("main() is ready.\n");
    pthread_barrier_wait(&mybarrier);
    printf("main() is going!\n");
    for (i=0; i < THREAD_COUNT; i++) pthread_join(ids[i], NULL);
    pthread_barrier_destroy(&mybarrier);
    return 0;
}
```

5.10. Adunarea în paralel a n numere

Vom da, ca exemplu de utilizare a thread-urilor, evaluarea în paralel a sumei mai multor numere întregi. Evident, operaţia de adunare a *n* numere, chiar dacă *n* este relativ mare, nu impune cu necesitate însumarea lor în paralel. O facem totuşi, deoarece reprezintă un exemplu elocvent de calcul paralel, în care esenţa este reprezentată de organizarea prelucrării paralele, aceeaşi şi pentru calcule mult mai complicate.

Presupunem că se dă un număr natural *n* şi un vector *a* având componentele întregi *a[0] a[1] ... a[n-1]*. Ne propunem să calculăm, folosind cât mai multe thread-uri, deci un paralelism cât mai consistent, suma acestor numere. Modelul de paralelism pe care ni-l propunem este ilustrat mai jos, pentru *m* = 8:

Mai întâi sunt calculate, în paralel, următoarele patru adunări:
a[0] = a[0] + a[1]; a[2] = a[2] + a[3]; a[4] = a[4] + a[5]; a[6] = a[6] + a[7];

După ce primele două adunări, respectiv ultimele două adunări s-au terminat, se mai execută în paralel încă două adunări:
a[0] = a[0] + a[2]; a[4] = a[4] + a[6];

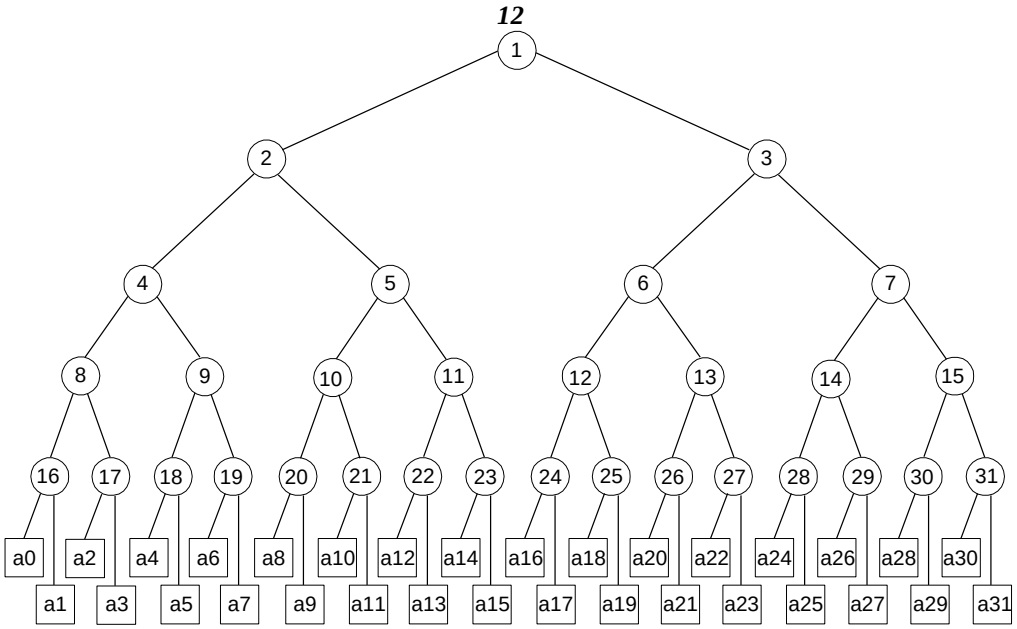
In sfârşit, la terminarea acestora, se va executa:
a[0] = a[0] + a[4];

Operaţiile de adunare se desfăşoară în paralel, având grijă ca fiecare adunare să se efectueze numai după ce operanzii au primit deja valori în adunările care trebuie să se desfăşoare înaintea celei curente. Este deci necesară o operaţie de *sincronizare* între iteraţii. Considerând că fiecare operaţie de adunare se execută într-o unitate de timp, din cauza paralelismului s-au consumat doar 3 unităţi de timp în loc de 7 unităţi de timp care s-ar fi consumat în abordarea secvenţială. In calcule s-au folosit 7 thread-uri, din care maximum 4 s-au executat în paralel.

Să considerăm acum problema pentru *n* numere şi să implementăm soluţia, cu intenţia de a folosi un număr maxim de thread-uri. Mai întâi extindem setul de numere până la *m* elemente, unde *m* este cea mai mică putere *l* a lui 2 mai mare sau egală cu *n*, adică: $2^{l-1} < n <= 2^l = m$, unde *l = partea întreagă superioară a lui* $log_2 n$. Elementele *a[n], . . ., a[m-1]* vor primi valoarea 0. Determinarea valorii lui **m** (cea mai mică putere a lui 2 mai mare sau egală cu **n**) se face uşor prin:

```
for (m = 1; n > m; m *= 2);
```

Pentru organizarea calculelor în regim multithreading, este convenabil să adoptăm o schemă arborescentă de numerotare a thread-urilor, ilustrată în figura de mai jos pentru 32 de numere.



Frunzele acestui arbore, reprezentate în pătrăţele, reprezintă **operanzii** de adunat. Nodurile interioare, reprezentate în cerculeţe, reprezintă **threadurile** care efectuează adunările.

Un thread oarecare **i** are doi fii. Threadurile de pe ultimul nivel (penultimul nivel al arborelui) au ca fii câte doi operanzi din tabloul de însumat. Celelalte le vom numi **threaduri interioare** şi au câte două **threaduri fii**, numerotate **2*i** şi **2*i+1**. Fiecare thread interior îşi va face propria operaţie de adunare numai după ce cei doi fii ai săi îşi vor termina adunările lor.

Fiecare thread **i** face o adunare de forma *a[s] = a[s] + a[d]*. In cele ce urmează vom determina indicii **s** şi **d** în funcţie de **i**. Vom numi **threaduri fraţi** threadurile ce se află pe acelaşi nivel, numerotaţi în ordinea crescătoare a vârstei lor. In cazul nostru, 2 şi 3 sunt fraţi cu 2 cel mai mic, 4, 5, 6 şi 7 sunt fraţi cu 4 cel mai mic, 8, 9, . . ., 15 sunt fraţi cu 8 cel mai mic, 16, 17, . . ., 31 sunt fraţi cu 16 cel mai mic ş.a.m.d. Fraţii cei mici de pe fiecare nivel au ca număr o putere a lui doi.

Este uşor de observat că fraţii de pe acelaşi nivel au acelaşi număr de operanzi: dacă **m** este numărul total de operanzi (putere a lui 2), **i** este numărul unui thread de pe un anumit nivel, iar **j** este numărul fratelui cel mic al acestuia, atunci fraţii de pe acest nivel au fiecare câte **m / j** operanzi. Indicele **s** al primului operand al threadului **i** este egal cu suma numărului de operanzi ai fraţilor lui mai mici, iar pentru **d** se mai adaugă jumătate din numărul de operanzi ai threadului, adică **d = s + m / j / 2**.

Determinarea numărului **j** al fratelui cel mic înseamnă găsirea celei mai mari puteri a lui 2 care este mai mică sau egală cu **i** şi ea se determină uşor prin secvenţa:

```
for (j = m; j > i; j /= 2);
```

Fiecare thread care are fii trebuie să aştepte mai întâi terminarea acestora şi apoi să îşi facă calculele proprii. Funcţia **pthread_join** aşteaptă terminarea unui thread. Din păcate, dacă threadul aşteptat nu a început încă, această funcţie va considera că threadul s-a terminat!

Soluţionarea testării începerii înaintea aşteptării terminării se poate face în mai multe moduri:

- Să se pornească threadurile în maniera "bottom up" în arborele asociat. Astfel este sigur că pornesc mai întâi fii şi apoi părinţii. Există însă riscul, foarte puţin probabil, ca unul din threadurile pornite să prindă procesorul înaintea unuia din fii, ceea ce ar conduce la rezultate inconsistente. De aici necesitatea testării pornirii fiilor înaintea testării terminării lor.
- Să se asocieze câte un mutex la fiecare thread. Iniţial fiecare astfel de mutex va fi blocat. La momentul pornirii unui thread acesta îşi va debloca mutexul lui. Când un părinte vrea să aştepte după fii lui, mai întâi va bloca mutexurile acestora. După ce va reuşi blocarea celor două mutexuri va putea să facă join ca să aştepte terminarea acestora. Pe parcursul prelucrărilor statutul acestor

mutexuri nu influenţează alte threaduri. La terminare, toate mutexurile vor fi distruse. In acest mod, threadurile se pot pornii în orice ordine şi se vor aştepta corespounzător.

- Iniţial la fiecare id de thread se atribuie valoarea -1. După pornire, acest id îşi va schimba valoarea. Astfel, pornirea se va aştepta "activ" sub forma **`while (tid[st] == -1);`**o variantă care reduce ocuparea procesorului prin aşteptare activă este ca să se testeze terminarea din când în când (polling), sub forma: **`while (tid[st] == -1) sleep(1);`**
- Să se pornească doar primul thread, iar dacă este nevoie fiecare thread să îşi pornească cele două threaduri fii.

Cu aceste precizări, sursa programului de adunare multithreading este:

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

int n, m; // n = numarul de operanzi; m = min {2^k >= n}
int* a; // valoarea 1 pentru pana la n-1,  0 de la n la m-1
pthread_t* tid; // id-urile threadurilor; -1 thread nepornit
int* pi; // Aici vor fi indicii threadurilor
pthread_mutex_t* off; // Mutexuri de inceput a threadurilor. La pornire fiecare
thread si-o deblocheaza
pthread_mutex_t print = PTHREAD_MUTEX_INITIALIZER; // Printare exclusiva

// Rutina thread-ului nr i de adunare
void* aduna(void* pi) {
    int i, j, sa, da, st = 0, dr = 0, k;
    i = *(int*)pi; // Retine numarul threadului
    pthread_mutex_unlock(&off[i]);
    if (i < m / 2) {
        st = 2 * i; // Retine fiul stang
        dr = st + 1; // Retine fiul drept
        pthread_mutex_lock(&off[st]); // Asteapta sa inceapa fiul stang
        pthread_mutex_lock(&off[dr]); // Asteapta sa inceapa fiul drept
        //while (tid[st] == -1); // Cu tid initiali -1 se poate asteapta si asa
        //while (tid[st] == -1) sleep(1); // Sau poate poate asa!
        //while (tid[dr] == -1);
        //while (tid[dr] == -1) sleep(1);
        pthread_join(tid[st], NULL); // Asteapta sa se termine fiul stang
        pthread_join(tid[dr], NULL); // Asteapta sa se termine fiul drept
    }
    for (j = m; j > i; j /= 2); // Determina fratele cel mic
    for (k = j, sa = 0; k < i; k++) sa += m / j; // operand stang
    da = sa + m / j / 2; // operand drept
    a[sa] += a[da]; // Face adunarea proppriu-zisa
    pthread_mutex_lock(&print);// Asigura printare exclusiva
    printf("Thread %d:\ta[%d] += a[%d]", i, sa, da);
    if (st > 0) printf("\t(dupa fii %d %d)\n", st, dr); else printf("\n");
    pthread_mutex_unlock(&print);
}

// Functia main, in care se creeaza si lanseaza thread-urile
int main(int argc, char* argv[]) {
    n = atoi(argv[1]); // Numarul de numere de adunat
    for (m = 1; n > m; m *= 2); // m = min {2^k >= n}
    int i;
    a = (int*) malloc(m*sizeof(int)); // Spatiu pentru intregii de adunat
    pi = (int*) malloc(m*sizeof(int)); // Spatiu pentru indicii threadurilor
    tid = (pthread_t*) malloc(m*sizeof(pthread_t)); // id-threads
    off = (pthread_mutex_t*) malloc(m*sizeof(pthread_mutex_t)); // Spatiu
mutexuri de pornire
    for (i = 0; i < n; i++) a[i] = 1; // Aduna numarul 1 de n ori
    for (i = n; i < m; i++) a[i] = 0; // Completeaza cu 0 pana la m
    for (i = 1; i < m; i++) {
        pthread_mutex_init(&off[i], NULL);
        pthread_mutex_lock(&off[i]);     // Marcheaza threadurile nepornite
```

```
            //tid[i] = -1; // Se pot marca si asa threadurile nepornite si nu se mai
    foloseste tabloul de mutexuri.
        }
        for (i = 1; i < m; i++) pi[i] = i;
        for (i = 1; i < m; i++)
            // De ce folosim mai jos &pi[i] in loc de &i? vezi un exemplu precedent!
            pthread_create(&tid[i], NULL, aduna, (void*)(&pi[i])); // Threadul i
        pthread_join(tid[1], NULL); // Asteapta dupa primul thread
        printf("Terminat adunarile pentru n = %d. Total: %d\n", n, a[0]);
        free(a); // Eliberaza tabloul de numere
        free(pi); // Elibereaza tabloul de indici de threaduri
        free(tid); // Elibereaza tabloul de id-uri de threaduri
        for (i = 1; i < m; i++) pthread_mutex_destroy(&off[i]);
        free(off); // Elibereaza tabloul de id-uri de threaduri
    }
```

De exemplu, rularea adunării a 15 numere va putea da o ieşire sub forma:

```
Thread 8: a[0] += a[1]
Thread 9: a[2] += a[3]
Thread 4: a[0] += a[2]  (dupa fii 8 9)
Thread 12:    a[8] += a[9]
Thread 15:    a[14] += a[15]
Thread 10:    a[4] += a[5]
Thread 14:    a[12] += a[13]
Thread 11:    a[6] += a[7]
Thread 7: a[12] += a[14](dupa fii 14 15)
Thread 13:    a[10] += a[11]
Thread 5: a[4] += a[6]  (dupa fii 10 11)
Thread 6: a[8] += a[10] (dupa fii 12 13)
Thread 2: a[0] += a[4]  (dupa fii 4 5)
Thread 3: a[8] += a[12] (dupa fii 6 7)
Thread 1: a[0] += a[8]  (dupa fii 2 3)
Terminat adunarile pentru n = 15. Total: 15
```

5.11. Jocul Sudoku rezolvat cu threaduri

5.11.1. Datele de intrare şi cele 6 surse

Sudoku.c

```
/*
 Se rezolva foposind pthreads un joc de Sudoku. La intrarea standard se da jocul sub
forma:
 ABCDE
 -----
 ----D
 --E--
 ---B-
 A----
 aabbb
 aaccb
 acceb
 dceee
 dddde
 Prima linia, alfabetul de intrare, cu n caractere distincte
 Urmatoarele n linii, matricea initiala a jocului
 Urmatoarele n linii sunt marcate cele n grupuri de cate n celule.
 */
```

```
# include "Sudoku.h"
main() {
    int i, j;
    char c;
    Globale(stdin);
    // Fixeaza multimile maximale pentru fiecare linie, coloana si grup.
    IMPUR *im = Impur('\xff');
    im->curent = 0;
    for (i = 0; i < gl.n; i++) {
        for (j = 0; j < gl.n; j++) {
            c = gl.t[i][j];
            if (c == gl.gol) continue;
            im->table[i][j] = c;
            deleteChr(im->ml[i], c);
            deleteChr(im->mc[j], c);
            deleteChr(im->mg[gl.ig[i][j]], c);
        }
    }

    printf("Alfabet: %s\n\nHarta grupurilor de celule: \n\n", gl.alfabet);
    for (i = 0; i < gl.n; i++) {
        for (j = 0; j < gl.n; j++)
        printf("%c", gl.sg[gl.ig[i][j]]);
        printf("\n");
    }
    printf("\n");
    prinT(im, "Problema initiala"); // Tipareste enuntul initial
    gl.listIm = im;

    pthread_create(&(im->descriptorThread),  NULL,  (void*)ThrSud,  im);//  Lanseaza
primul thread

    pthread_mutex_lock(&gl.mutex4cond);
    // Se asteapta terminarea tuturor partialelor active.
    while (gl.partiale > 0)
    pthread_cond_wait(&gl.cond, &gl.mutex4cond);
    pthread_mutex_unlock(&gl.mutex4cond);

    printf("Partiale %d, esecuri %d, solutii %d\n",
            gl.curent+1, gl.esecuri, gl.solutii);
}

# include "Globale.c"
# include "ChrSet.c"
# include "Impur.c"
# include "ThrSud.c"
```

Sudoku.h

```
# define MAX 64
# define TRUE 1
# define FALSE 0

# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <pthread.h>

typedef struct {
    int curent; // Numarul partialului curent
    int parinte; // Numarul partialului parinte
    char **table; // Pointer la tabel
    char **ml, **mc, **mg; // Pointer la multimile asociate
    void *pred, *succ; // Pointeri in lista dublu inlantuita
    pthread_t descriptorThread;
} IMPUR;
```

```
typedef struct {
    // Date constante pe durata rezolvarii jocului
    char gol; // Cu acest caracter se indica spatiile goale.
    char alfabet[MAX]; // Alfabetul jocului
    int n; // Dimensiunea jocului
    int dm; // Dimensiunea unei multimi in intregi
    int ig[MAX][MAX]; // Indicii grupului de celule
    char sg[MAX]; // Alfabetul de marcare a celulelor
    char M_010_[8];
    char M_101_[8];

    // Date la care se are acces sincronizat
    int esecuri; // Numar curent de esecuri
    int solutii; // Numar curent de solutii
    int curent; // Numarul ultimului partial creat
    int partiale; // Numar de partiale active sau de pornit
    IMPUR *listIm; // Lista spatiilor impure inca active
    char **t; // Pointer de serviciu

    // Pentru sincronizarea threadurilor
    pthread_mutex_t mutex;
    pthread_mutex_t mutex4cond;
    pthread_cond_t cond;

} GLOBALE;

GLOBALE gl;

char* ChrSet(char octet);
void deleteChr(char *multime, char c);
int containsChr(char *multime, char c);
char getChr(char *multime, int i);
void intersectionChr(char *multime, char *s);
void unionChr(char *multime, char *s);
void str(char *multime, char *s);
int len(char *multime);
void Globale(FILE *iN);
void prinT(IMPUR* im, char *s);
IMPUR* Impur(char octet);
void ThrSud(IMPUR *im);
void delListIm(IMPUR *im);
```

Globale.c

```
void Globale(FILE *iN) {
    int i, j, k;
    char sp[MAX], s[MAX], c, *p;
    gl.gol = '-';
    gl.esecuri = 0;
    gl.solutii = 0;
    gl.curent = 0;
    gl.partiale = 1;
    gl.listIm = NULL;
    pthread_mutex_init(&gl.mutex, NULL);
    pthread_mutex_init(&gl.mutex4cond, NULL);
    pthread_cond_init(&gl.cond, NULL);
    memset(sp, ' ', MAX); // Un sir de spatii (pt diverse servicii)
    sp[MAX - 1] = 0;
    c = 1;
    for (i = 0; i < 8; i++) {
        gl.M_010_[i] = c << i;
        gl.M_101_[i] = ~gl.M_010_[i];
    }
    // Citeste alfabetul
    fgets(gl.alfabet, MAX, iN);
    for (i = strlen(gl.alfabet) - 1; isspace(gl.alfabet[i]); i--)
        ;
```

```
    gl.alfabet[i + 1] = 0;
    for (i = 0; isspace(gl.alfabet[i]); i++)
        ;
    strcpy(&gl.alfabet[i], gl.alfabet);
    gl.n = strlen(gl.alfabet); // Fixeaza constanta n a jocului
    gl.dm = (gl.n + 7) / 8;
    gl.t = (char**) malloc(gl.n * sizeof(char*));

    // Tabelul initial
    // Depune valorile citite, gol la caractere incorecte
    for (i = 0; i < gl.n; i++) {
        strcpy(s, sp);
        fgets(s, MAX, iN);
        s[strlen(s)] = ' ';
        s[gl.n] = 0;
        gl.t[i] = (char*) malloc(gl.n * sizeof(char));
        for (j = 0; j < gl.n; j++) {
            c = s[j];
            gl.t[i][j] = c;
            if (strchr(gl.alfabet, c) != NULL)
                continue;
            gl.t[i][j] = gl.gol;
        }
    }
    // Celulele din acelasi grup de celule se definesc punand acelasi caracter
    // pentru toate celulele care definesc grupul
.    memset(gl.sg, 0, gl.n + 1); // Alfabetul de marcare a grupurilor de celule
    k = -1;
    for (i = 0; i < gl.n; i++) {
        strcpy(s, sp);
        fgets(s, MAX, iN);
        s[strlen(s)] = ' ';
        s[gl.n] = 0;
        for (j = 0; j < gl.n; j++) {
            c = s[j];
            p = strchr(gl.sg, c);
            if (p == NULL) {
                k++;
                gl.sg[k] = c;
                gl.ig[i][j] = k;
            } else {
                gl.ig[i][j] = p - gl.sg;
            }
        }
    }
}

void prinT(IMPUR* im, char *s) {
    int i, j;
    char p[MAX * 2 + 2];
    // Tipareste, in regim sincronizat, continutul unui obiect impur.
    printf("Partial %d, parinte %d, total %d, active %d. %s\n", im->curent,
            im->parinte, gl.curent, gl.partiale, s);
    // Tipareste tabelul.
    printf("\n");
    for (i = 0; i < gl.n; i++) {
        for (j = 0; j < gl.n; j++)
            printf("%c", im->table[i][j]);
        printf("\n");
    }
    // Harta grupurilor de celule si toate multimile asociate fiecarei pozitii.
    printf("\n");
    for (i = 0; i < gl.n; i++) {
        str(im->ml[i], p);
        printf("ml[%d]= %s\n", i, p);
    }
    printf("\n");
    for (i = 0; i < gl.n; i++) {
```

```
        str(im->mc[i], p);
        printf("mc[%d]= %s\n", i, p);
    }
    printf("\n");
    for (i = 0; i < gl.n; i++) {
        str(im->mg[i], p);
        printf("mg[%c]= %s\n", gl.sg[i], p);
    }
    printf("\n");
    fflush (stdout);
}
```

ChrSet.c

```
// Operatii cu multimi de caractere necesare la sudoku.
char* ChrSet(char octet) {
    char* multime = (char*) malloc(gl.dm * sizeof(char));
    memset(multime, octet, gl.dm);
    return multime;
}
void deleteChr(char *multime, char c) {
    int k = strchr(gl.alfabet, c) - gl.alfabet;
    multime[k / 8] &= gl.M_101_[k % 8];
}
int containsChr(char *multime, char c) {
    char *p = strchr(gl.alfabet, c);
    if (p == NULL) return FALSE;
    int k = p - gl.alfabet;
    if ((multime[k / 8] & gl.M_010_[k % 8]) == 0) return FALSE;
    return TRUE;
}
char getChr(char *multime, int i) {
    for (int j = 0, k = 0; j < gl.n; j++) {
    if ((multime[j / 8] & gl.M_010_[j % 8]) != 0) {
        if (k == i)
            return gl.alfabet[j];
        k++;
        }
    }
    return gl.gol;
}
void intersectionChr(char *multime, char *s) {
    for (int i = 0; i < gl.dm; i++) multime[i] &= s[i];
}
void unionChr(char *multime, char *s) {
    for (int i = 0; i < gl.dm; i++) multime[i] |= s[i];
}
void str(char *multime, char *s) {
    int i, k;
    s[0] = '[';
    for (i = 0, k = 1; i < gl.n; i++) {
        if ((multime[i / 8] & gl.M_010_[i % 8]) == 0) continue;
        s[k] = gl.alfabet[i];
        s[k + 1] = ',';
        k += 2;
    }
    if (k > 1) k--;
    s[k] = ']';
    s[k + 1] = 0;
}
int len(char *multime) {
    int j, k;
    for (j = 0, k = 0; j < gl.n; j++) {
        if ((multime[j / 8] & gl.M_010_[j % 8]) == 0) continue;
        k++;
    }
    return k;
```

```
}
```

Impur.c

```
IMPUR* Impur(char octet) {
    IMPUR *im;
    int i, j;
    im = (IMPUR*) malloc(sizeof(IMPUR)); // Spatiu pentru IMPUR
    im->curent = -1; // Numarul partialului
    im->parinte = -1; // Numarul partialului parinte
    im->pred = NULL;
    im->succ = NULL;
    im->table = (char **) malloc(gl.n * sizeof(char *)); // Adrese linii
    im->ml = (char**) malloc(gl.n * sizeof(char*)); // Adrese ml
    im->mc = (char**) malloc(gl.n * sizeof(char*)); // Adrese mc
    im->mg = (char**) malloc(gl.n * sizeof(char*)); // Adrese mg
    for (i = 0; i < gl.n; i++) {
        im->table[i] = (char *) malloc(gl.n * sizeof(char));
        memset(im->table[i], gl.gol, gl.n); // Initializare linie
        im->ml[i] = ChrSet(octet); // Multimile initiale
        im->mc[i] = ChrSet(octet);
        im->mg[i] = ChrSet(octet);
    }
    return im;
}
```

ThrSud.c

```
void ThrSud(IMPUR *im) {
    if (im == NULL || gl.listIm == NULL) {
// Sectiune critica
        pthread_mutex_lock(&gl.mutex);
        gl.partiale--;
        if (gl.partiale == 0)
            pthread_cond_signal(&gl.cond);
        pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica
        return;
    }
    IMPUR *newIm;
    char *minim, *temp;
    int i, j, k, iMin, jMin;
    char s[MAX], c, p[MAX];
    while (TRUE) {
        // Determina celula libera cu minimul posibilitatilor de completare:
        // Daca nu exista grupuri de celule libere ==> SOLUTIE.
        // Daca exista o celula libera cu 0 posibilitati ==> ESEC.
        // Completeaza toate grupurile pentru care exista fix o posibilitate.
        // Se iese din ciclu daca toate celulele libere exista >= 2 posibilitati.
        minim = ChrSet('\xff');
        iMin = -1;
        jMin = -1;
        for (i = 0; i < gl.n; i++) {
            for (j = 0; j < gl.n; j++) {
                if (im->table[i][j] != gl.gol)
                    continue;
                // Intersectia multimilor celulei (i,j)
                temp = ChrSet('\x00');
                unionChr(temp, im->ml[i]);
                intersectionChr(temp, im->mc[j]);
                intersectionChr(temp, im->mg[gl.ig[i][j]]);
                if (len(temp) < len(minim)) {
                    // Retine intersectia de lungime minima
                    iMin = i;
                    jMin = j;
                    free(minim);
```

```
                    minim = temp;
                } else {
                    free(temp);
                }
            }
        }
        if (iMin == -1 && im->table[0][0] == gl.gol) {
            // Caz limita tabelul vid
            iMin = 0;
            jMin = 0;
        }
        // Exista solutie? Daca da, se semnaleaza si se termina partialul.
        if (iMin == -1) {
            free(minim);
// Sectiune critica
            pthread_mutex_lock(&gl.mutex);
            gl.solutii++;
            sprintf(s, "SOLUTIA nr: %d", gl.solutii);
            prinT(im, s);
            gl.partiale--;
            delListIm(im);
            if (gl.partiale == 0)
                pthread_cond_signal(&gl.cond);
            pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica
            return;
        }
        // Este esec? Daca da, se semnaleaza si se termina partialul.
        if (len(minim) == 0) {
            free(minim);
// Sectiune critica
            pthread_mutex_lock(&gl.mutex);
            gl.esecuri++;
            sprintf(s, "ESEC nr: %d", gl.esecuri);
            prinT(im, s);
            gl.partiale--;
            delListIm(im);
            if (gl.partiale == 0)
                pthread_cond_signal(&gl.cond);
            pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica
            return;
        }
        if (len(minim) == 1) {
            // Completare unica. Dupa fixarea valorii, se reiau controalele in tabel.
            c = getChr(minim, 0);
            im->table[iMin][jMin] = c;
            deleteChr(im->ml[iMin], c);
            deleteChr(im->mc[jMin], c);
            deleteChr(im->mg[gl.ig[iMin][jMin]], c);
            continue; // Cauta din nou in tabel dupa completare
        }
        // Este obligatorie ramificarea: toate celulele libere au >= 2 posibilitati.
        break;
    }
    // Celula (iMin,jMin) poate contine oricare valoare din minim.
    // Se creaza minim.size() noi partialuri, punand cu valorile din minim.
    // Dupa crearea noilor partialuri, partialul curent se termina.
// Sectiune critica
    pthread_mutex_lock(&gl.mutex);
    gl.partiale += len(minim);
    pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica
    for (k = 0; k < len(minim); k++) {
        c = getChr(minim, k);
        // c este valoarea curenta care urmeaza a fi completata.
        // Creaza un obiect impur pentru un nou partial.
        newIm = Impur('\x00');
```

```
        // Copiaza in noul obiect tabelul si multimile partialului curent.
        for (i = 0; i < gl.n; i++) {
            unionChr(newIm->ml[i], im->ml[i]);
            unionChr(newIm->mc[i], im->mc[i]);
            unionChr(newIm->mg[i], im->mg[i]);
            for (j = 0; j < gl.n; j++)
                newIm->table[i][j] = im->table[i][j];
        }
        // Fixeaza valoarea c in tabel si o sterge din multimile asociate.
        newIm->table[iMin][jMin] = c;
        deleteChr(newIm->ml[iMin], c);
        deleteChr(newIm->mc[jMin], c);
        deleteChr(newIm->mg[gl.ig[iMin][jMin]], c);
        // Creaza un nou partial.
        newIm->parinte = im->curent;
// Sectiune critica
        pthread_mutex_lock(&gl.mutex);
        gl.curent++;
        newIm->curent = gl.curent;
        newIm->succ = gl.listIm;
        gl.listIm->pred = newIm;
        gl.listIm = newIm;
        pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica
        pthread_create(&(newIm->descriptorThread), NULL, (void*) ThrSud, newIm);
    }
    str(minim, p);
    sprintf(s, "Ramificare in (%d,%d) cu %s", iMin, jMin, p);
    free(minim);
// Sectiune critica
    pthread_mutex_lock(&gl.mutex);
    prinT(im, s);
    gl.partiale--;
    delListIm(im);
    if (gl.partiale == 0)
        pthread_cond_signal(&gl.cond);
    pthread_mutex_unlock(&gl.mutex);
// Sfarsit sectiune critica

// S-au creat cele len(minim) partialuri si se termina partialul curent.
}

void delListIm(IMPUR *im) {
    int i;
    if (im == NULL)
        return;

    // Elimina im din lista simplu inlantuita
    if (im->pred == NULL && im->succ == NULL)
        gl.listIm = NULL;
    else if (im->pred == NULL)
        gl.listIm = (IMPUR*) im->succ;
    else if (im->succ == NULL)
        ((IMPUR*) im->pred)->succ = im->succ;
    else {
        ((IMPUR*) im->succ)->pred = im->pred;
        ((IMPUR*) im->pred)->succ = im->succ;
    }

    // Elibereaza spatiile coloanelor
    for (i = 0; i < gl.n; i++) {
        free(im->table[i]);
        free(im->ml[i]);
        free(im->mc[i]);
        free(im->mg[i]);
    }

    // Elibereaza capetele de coloane
```

```
    free(im->table);
    free(im->ml);
    free(im->mc);
    free(im->mg);

    // Elimina spatiul ocupat de im
    free(im);
}
```

5.11.2. Un exemplu de dimensiune 5

Cu fişierul de intrare **1** avem ieşirea **1.out**

1

```
ABCDE
-----
----D
--E--
---B-
A----
aabbb
aaccb
acceb
dceee
dddde
```

1.out

```
Alfabet: ABCDE

Harta grupurilor de celule:

aabbb
aaccb
acceb
dceee
dddde

Partial 0, parinte -1, total 0, active 1. Problema initiala

-----
----D
--E--
---B-
A----

ml[0]= [A,B,C,D,E]
ml[1]= [A,B,C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= [A,C,D,E]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C,E]
mg[c]= [A,B,C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 0, parinte -1, total 2, active 3. Ramificare in (1,3) cu [A,C]

-----
----D
--E--
---B-
A----

ml[0]= [A,B,C,D,E]
ml[1]= [A,B,C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]
```

```
mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= [A,C,D,E]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C,E]
mg[c]= [A,B,C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 2, parinte 0, total 4, active 4. Ramificare in (0,3) cu [A,E]

-----
---CD
--E--
---B-
A----

ml[0]= [A,B,C,D,E]
ml[1]= [A,B,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= [A,D,E]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C,E]
mg[c]= [A,B,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 1, parinte 0, total 6, active 5. Ramificare in (0,3) cu [C,E]

-----
---AD
--E--
---B-
A----

ml[0]= [A,B,C,D,E]
ml[1]= [B,C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= [C,D,E]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C,E]
mg[c]= [B,C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 4, parinte 2, total 8, active 6. Ramificare in (1,0) cu [B,E]

---E-
---CD
--EA-
---B-
A--D-

ml[0]= [A,B,C,D]
ml[1]= [A,B,E]
ml[2]= [B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= []
mc[4]= [A,B,C,E]
```

```
mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C]
mg[c]= [A,B,D]
mg[e]= [C,D,E]
mg[d]= [B,C,E]

Partial 3, parinte 2, total 8, active 5. ESEC nr: 1

--CAE
E-BCD
CAEDB
---B-
A--EC

ml[0]= [B,D]
ml[1]= [A]
ml[2]= []
ml[3]= [A,C,D,E]
ml[4]= [B,D]

mc[0]= [B,D]
mc[1]= [B,C,D,E]
mc[2]= [A,D]
mc[3]= []
mc[4]= [A]

mg[a]= [A,B,D]
mg[b]= []
mg[c]= [D]
mg[e]= [A,E]
mg[d]= [B,C,D]

Partial 6, parinte 1, total 10, active 6. Ramificare in (1,2) cu [B,C]

---E-
---AD
--E--
---B-
A----

ml[0]= [A,B,C,D]
ml[1]= [B,C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,B,C,D]
mc[3]= [C,D]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C]
mg[c]= [B,C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 8, parinte 4, total 10, active 5. ESEC nr: 2

---E-
EB-CD
--EA-
CADB-
A--D-

ml[0]= [A,B,C,D]
ml[1]= [A]
ml[2]= [B,C,D]
ml[3]= [E]
ml[4]= [B,C,E]

mc[0]= [B,D]
mc[1]= [C,D,E]
mc[2]= [A,B,C]
mc[3]= []
mc[4]= [A,B,C,E]

mg[a]= [A,C,D]
mg[b]= [A,B,C]
mg[c]= [B,D]
mg[e]= [C,E]
mg[d]= [B,E]

Partial 5, parinte 1, total 10, active 4. ESEC nr: 3
```

```
DABCE
-ECAD
-BED-
--AB-
A--EC

ml[0]= []
ml[1]= [B]
ml[2]= [A,C]
ml[3]= [C,D,E]
ml[4]= [B,D]

mc[0]= [B,C,E]
mc[1]= [C,D]
mc[2]= [D]
mc[3]= []
mc[4]= [A,B]

mg[a]= [B,C]
mg[b]= [A]
mg[c]= [D]
mg[e]= [E]
mg[d]= [B,C,D]

Partial 7, parinte 4, total 10, active 3. ESEC nr: 4

-CBEA
BEACD
-BEAC
-D-B-
A--D-

ml[0]= [D]
ml[1]= []
ml[2]= [D]
ml[3]= [A,C,E]
ml[4]= [B,C,E]

mc[0]= [C,D,E]
mc[1]= [A]
mc[2]= [C,D]
mc[3]= []
mc[4]= [B,E]

mg[a]= [A,D]
mg[b]= []
mg[c]= []
mg[e]= [C,D,E]
mg[d]= [B,C,E]

Partial 10, parinte 6, total 10, active 2. ESEC nr: 5

DABEC
BECAD
CBED-
-DAB-
AC---

ml[0]= []
ml[1]= []
ml[2]= [A]
ml[3]= [C,E]
ml[4]= [B,D,E]

mc[0]= [E]
mc[1]= []
mc[2]= [D]
mc[3]= [C]
mc[4]= [A,B,E]

mg[a]= []
mg[b]= [A]
mg[c]= []
mg[e]= [C,E]
mg[d]= [B,D,E]

Partial 9, parinte 6, total 12, active 3. Ramificare in (0,2) cu [A,C]

---E-
--BAD
--E--
---B-
A----
```

```
ml[0]= [A,B,C,D]
ml[1]= [C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [A,C,D]
mc[3]= [C,D]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [A,B,C]
mg[c]= [C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 12, parinte 9, total 12, active 2. SOLUTIA nr: 1

DACEB
CEBAD
BCEDA
EDABC
ABDCE

ml[0]= []
ml[1]= []
ml[2]= []
ml[3]= []
ml[4]= []

mc[0]= []
mc[1]= []
mc[2]= []
mc[3]= []
mc[4]= []

mg[a]= []
mg[b]= []
mg[c]= []
mg[e]= []
mg[d]= []

Partial 11, parinte 9, total 14, active 3. Ramificare in (0,4) cu [B,C]

--AE-
--BAD
--E--
---B-
A----

ml[0]= [B,C,D]
ml[1]= [C,E]
ml[2]= [A,B,C,D]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,D,E]
mc[2]= [C,D]
mc[3]= [C,D]
mc[4]= [A,B,C,E]

mg[a]= [A,B,C,D,E]
mg[b]= [B,C]
mg[c]= [C,D]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 14, parinte 11, total 16, active 4. Ramificare in (0,0) cu [B,D]

--AEC
--BAD
--E-B
---BA
A---E

ml[0]= [B,D]
ml[1]= [C,E]
ml[2]= [A,C,D]
ml[3]= [C,D,E]
ml[4]= [B,C,D]

mc[0]= [B,C,D,E]
```

```
mc[1]= [A,B,C,D,E]
mc[2]= [C,D]
mc[3]= [C,D]
mc[4]= []

mg[a]= [A,B,C,D,E]
mg[b]= []
mg[c]= [C,D]
mg[e]= [C,D]
mg[d]= [B,C,D,E]

Partial 13, parinte 11, total 16, active 3. ESEC nr: 6

--AEB
--BAD
-DE-C
---B-
A----

ml[0]= [C,D]
ml[1]= [C,E]
ml[2]= [A,B]
ml[3]= [A,C,D,E]
ml[4]= [B,C,D,E]

mc[0]= [B,C,D,E]
mc[1]= [A,B,C,E]
mc[2]= [C,D]
mc[3]= [C,D]
mc[4]= [A,E]

mg[a]= [A,B,C,D,E]
mg[b]= []
mg[c]= [C]
mg[e]= [A,C,D,E]
mg[d]= [B,C,D,E]

Partial 16, parinte 14, total 16, active 2. ESEC nr: 7

DBAEC
E-BAD
C-E-B
---BA
A---E

ml[0]= []
ml[1]= [C]
ml[2]= [A,D]
ml[3]= [C,D,E]
ml[4]= [B,C,D]

mc[0]= [B]
mc[1]= [A,C,D,E]
mc[2]= [C,D]
mc[3]= [C,D]
mc[4]= []

mg[a]= [A]
mg[b]= []
mg[c]= [C,D]
mg[e]= [C,D]
mg[d]= [B,C,D,E]

Partial 15, parinte 14, total 16, active 1. ESEC nr: 8

BDAEC
--BAD
C-E-B
---BA
A---E

ml[0]= []
ml[1]= [C,E]
ml[2]= [A,D]
ml[3]= [C,D,E]
ml[4]= [B,C,D]

mc[0]= [D,E]
mc[1]= [A,B,C,E]
mc[2]= [C,D]
mc[3]= [C,D]
mc[4]= []

mg[a]= [A,E]
mg[b]= []
```

```
mg[c]= [C,D]
mg[e]= [C,D]
mg[d]= [B,C,D,E]

Partiale 17, esecuri 8, solutii 1
```

5.11.3. Un exemplu de dimensiune 9

Prezentăm intrarea şi câteva ieşiri

2

```
ACDEILORX
-R---E-I-
--X--O-D-
-X--DA---
--R-I---C
CLE-O-RA-
----C-X--
---XA--C-
-A-R--O--
-C-I---R-
aaaabbbbc
aaaabbbcc
addebbccc
dddeeefcc
gdddefffc
ggdeeefff
ggghheffi
gghhhiiii
ghhhhiiii
```

2.out

```
Alfabet: ACDEILORX

Harta grupurilor de celule:

aaaabbbbc
aaaabbbcc
addebbccc
dddeeefcc
gdddefffc
ggdeeefff
ggghheffi
gghhhiiii
ghhhhiiii

Partial 0, parinte -1, total 0, active 1. Problema initiala

-R---E-I-
--X--O-D-
-X--DA---
--R-I---C
CLE-O-RA-
----C-X--
---XA--C-
-A-R--O--
-C-I---R-

ml[0]= [A,C,D,L,O,X]
ml[1]= [A,C,E,I,L,R]
ml[2]= [C,E,I,L,O,R]
ml[3]= [A,D,E,L,O,X]
ml[4]= [D,I,X]
ml[5]= [A,D,E,I,L,O,R]
ml[6]= [D,E,I,L,O,R]
ml[7]= [C,D,E,I,L,X]
ml[8]= [A,D,E,L,O,X]

mc[0]= [A,D,E,I,L,O,R,X]
mc[1]= [D,E,I,O]
mc[2]= [A,C,D,I,L,O]
mc[3]= [A,C,D,E,L,O]
mc[4]= [E,L,R,X]
mc[5]= [C,D,I,L,R,X]
mc[6]= [A,C,D,E,I,L]
mc[7]= [E,L,O,X]
mc[8]= [A,D,E,I,L,O,R,X]

mg[a]= [A,C,D,E,I,L,O]
```

```
mg[b]= [C,L,R,X]
mg[c]= [A,E,I,L,O,R,X]
mg[d]= [A,C,D,I,O]
mg[e]= [A,D,E,L,R,X]
mg[f]= [D,E,I,L,O]
mg[g]= [D,E,I,L,O,R,X]
mg[h]= [D,E,L,O]
mg[i]= [A,C,D,E,I,L,X]

Partial 0, parinte -1, total 2, active 3. Ramificare in (0,4) cu [L,X]

-R---E-I-
--X--O-D-
-XC-DA---
AOR-I---C
CLEDOIRAX
--I-C-X--
---XA--C-
-A-R--O--
-C-I---R-

ml[0]= [A,C,D,L,O,X]
ml[1]= [A,C,E,I,L,R]
ml[2]= [E,I,L,O,R]
ml[3]= [D,E,L,X]
ml[4]= []
ml[5]= [A,D,E,L,O,R]
ml[6]= [D,E,I,L,O,R]
ml[7]= [C,D,E,I,L,X]
ml[8]= [A,D,E,L,O,X]

mc[0]= [D,E,I,L,O,R,X]
mc[1]= [D,E,I]
mc[2]= [A,D,L,O]
mc[3]= [A,C,E,L,O]
mc[4]= [E,L,R,X]
mc[5]= [C,D,L,R,X]
mc[6]= [A,C,D,E,I,L]
mc[7]= [E,L,O,X]
mc[8]= [A,D,E,I,L,O,R]

mg[a]= [A,C,D,E,I,L,O]
mg[b]= [C,L,R,X]
mg[c]= [A,E,I,L,O,R]
mg[d]= []
mg[e]= [A,D,E,L,R,X]
mg[f]= [D,E,L,O]
mg[g]= [D,E,I,L,O,R,X]
mg[h]= [D,E,L,O]
mg[i]= [A,C,D,E,I,L,X]

Partial 14, parinte 5, total 20, active 10. Ramificare in (1,1) cu [E,I]

DROCXELIA
--X-ROCD-
-XC-DA---
AOR-I---C
CLEDOIRAX
--I-C-X--
---XA--C-
-A-R--O--
-C-I---R-

ml[0]= []
ml[1]= [A,E,I,L]
ml[2]= [E,I,L,O,R]
ml[3]= [D,E,L,X]
ml[4]= []
ml[5]= [A,D,E,L,O,R]
ml[6]= [D,E,I,L,O,R]
ml[7]= [C,D,E,I,L,X]
ml[8]= [A,D,E,L,O,X]

mc[0]= [E,I,L,O,R,X]
mc[1]= [D,E,I]
mc[2]= [A,D,L]
mc[3]= [A,E,L,O]
mc[4]= [E,L]
mc[5]= [C,D,L,R,X]
mc[6]= [A,D,E,I]
mc[7]= [E,L,O,X]
mc[8]= [D,E,I,L,O,R]

mg[a]= [A,E,I,L]
mg[b]= []
```

```
mg[c]= [E,I,L,O,R]
mg[d]= []
mg[e]= [A,D,E,L,R,X]
mg[f]= [D,E,L,O]
mg[g]= [D,E,I,L,O,R,X]
mg[h]= [D,E,L,O]
mg[i]= [A,C,D,E,I,L,X]

Partial 11, parinte 7, total 20, active 9. SOLUTIA nr: 1

DRAOXECIL
EIXCROLDA
LXCEDAIOR
AORLIXDEC
CLEDOIRAX
REIACDXLO
ODLXARECI
IADRLCOXE
XCOIELARD

ml[0]= []
ml[1]= []
ml[2]= []
ml[3]= []
ml[4]= []
ml[5]= []
ml[6]= []
ml[7]= []
ml[8]= []

mc[0]= []
mc[1]= []
mc[2]= []
mc[3]= []
mc[4]= []
mc[5]= []
mc[6]= []
mc[7]= []
mc[8]= []

mg[a]= []
mg[b]= []
mg[c]= []
mg[d]= []
mg[e]= []
mg[f]= []
mg[g]= []
mg[h]= []
mg[i]= []

Partial 9, parinte 8, total 22, active 9. ESEC nr: 3

DRAOXECIL
IEXCROLDA
LXCEDAIOR
AORLIXDEC
CLEDOIRAX
-DIACRXL-
-I-XADEC-
-A-R--O--
-C-I---R-

ml[0]= []
ml[1]= []
ml[2]= []
ml[3]= []
ml[4]= []
ml[5]= [E,O]
ml[6]= [L,O,R]
ml[7]= [C,D,E,I,L,X]
ml[8]= [A,D,E,L,O,X]

mc[0]= [E,O,R,X]
mc[1]= []
mc[2]= [D,L,O]
mc[3]= []
mc[4]= [E,L]
mc[5]= [C,L]
mc[6]= [A]
mc[7]= [X]
mc[8]= [D,E,I,O]

mg[a]= []
mg[b]= []
mg[c]= []
```

```
mg[d]= []
mg[e]= []
mg[f]= [O]
mg[g]= [E,L,O,R,X]
mg[h]= [D,E,L,O]
mg[i]= [A,C,D,E,I,L,X]

Partiale 31, esecuri 15, solutii 1
```

5.11.4. Un exemplu clasic, dimensiune 9 si grupuri 3X3

3

```
123456789
700040090
380200004
005006000
120030000
050000080
000080079
000100400
800007031
070090006
aaabbbccc
aaabbbccc
aaabbbccc
dddeeefff
dddeeefff
dddeeefff
ggghhhiii
ggghhhiii
ggghhhiii
```

3.out

```
Alfabet: 123456789

Harta grupurilor de celule:

aaabbbccc
aaabbbccc
aaabbbccc
dddeeefff
dddeeefff
dddeeefff
ggghhhiii
ggghhhiii
ggghhhiii

Partial 0, parinte -1, total 0, active 1. Problema initiala

7---4--9-
38-2----4
--5--6---
12--3----
-5-----8-
----8--79
---1--4--
8----7-31
-7--9---6

ml[0]= [1,2,3,5,6,8]
ml[1]= [1,5,6,7,9]
ml[2]= [1,2,3,4,7,8,9]
ml[3]= [4,5,6,7,8,9]
ml[4]= [1,2,3,4,6,7,9]
ml[5]= [1,2,3,4,5,6]
ml[6]= [2,3,5,6,7,8,9]
ml[7]= [2,4,5,6,9]
ml[8]= [1,2,3,4,5,8]

mc[0]= [2,4,5,6,9]
mc[1]= [1,3,4,6,9]
mc[2]= [1,2,3,4,6,7,8,9]
mc[3]= [3,4,5,6,7,8,9]
mc[4]= [1,2,5,6,7]
mc[5]= [1,2,3,4,5,8,9]
mc[6]= [1,2,3,5,6,7,8,9]
mc[7]= [1,2,4,5,6]
```

```
mc[8]= [2,3,5,7,8]

mg[a]= [1,2,4,6,9]
mg[b]= [1,3,5,7,8,9]
mg[c]= [1,2,3,5,6,7,8]
mg[d]= [3,4,6,7,8,9]
mg[e]= [1,2,4,5,6,7,9]
mg[f]= [1,2,3,4,5,6]
mg[g]= [1,2,3,4,5,6,9]
mg[h]= [2,3,4,5,6,8]
mg[i]= [2,5,7,8,9]

Partial 2, parinte 0, total 4, active 4. Ramificare in (0,2) cu [1,2]

76--4--9-
38-2----4
--5--6---
128739645
-5-----8-
----8--79
---1--4--
8----7-31
-7--9---6

ml[0]= [1,2,3,5,8]
ml[1]= [1,5,6,7,9]
ml[2]= [1,2,3,4,7,8,9]
ml[3]= []
ml[4]= [1,2,3,4,6,7,9]
ml[5]= [1,2,3,4,5,6]
ml[6]= [2,3,5,6,7,8,9]
ml[7]= [2,4,5,6,9]
ml[8]= [1,2,3,4,5,8]

mc[0]= [2,4,5,6,9]
mc[1]= [1,3,4,9]
mc[2]= [1,2,3,4,6,7,9]
mc[3]= [3,4,5,6,8,9]
mc[4]= [1,2,5,6,7]
mc[5]= [1,2,3,4,5,8]
mc[6]= [1,2,3,5,7,8,9]
mc[7]= [1,2,5,6]
mc[8]= [2,3,7,8]

mg[a]= [1,2,4,9]
mg[b]= [1,3,5,7,8,9]
mg[c]= [1,2,3,5,6,7,8]
mg[d]= [3,4,6,7,9]
mg[e]= [1,2,4,5,6]
mg[f]= [1,2,3]
mg[g]= [1,2,3,4,5,6,9]
mg[h]= [2,3,4,5,6,8]
mg[i]= [2,5,7,8,9]

Partial 3, parinte 2, total 12, active 7. ESEC nr: 1

761-4--9-
3892----4
245-76-1-
128739645
-5-----8-
-3--8--79
-9-1--4--
8----7-31
-7--9---6

ml[0]= [2,3,5,8]
ml[1]= [1,5,6,7]
ml[2]= [3,8,9]
ml[3]= []
ml[4]= [1,2,3,4,6,7,9]
ml[5]= [1,2,4,5,6]
ml[6]= [2,3,5,6,7,8]
ml[7]= [2,4,5,6,9]
ml[8]= [1,2,3,4,5,8]

mc[0]= [4,5,6,9]
mc[1]= [1]
mc[2]= [2,3,4,6,7]
mc[3]= [3,4,5,6,8,9]
mc[4]= [1,2,5,6]
mc[5]= [1,2,3,4,5,8]
mc[6]= [1,2,3,5,7,8,9]
mc[7]= [2,5,6]
mc[8]= [2,3,7,8]
```

```
mg[a]= []
mg[b]= [1,3,5,8,9]
mg[c]= [2,3,5,6,7,8]
mg[d]= [4,6,7,9]
mg[e]= [1,2,4,5,6]
mg[f]= [1,2,3]
mg[g]= [1,2,3,4,5,6]
mg[h]= [2,3,4,5,6,8]
mg[i]= [2,5,7,8,9]

Partiale 61, esecuri 30, solutii 1
```