Laboratorul 1 - Introducere în OpenMP

Laboratorul 1 - Introducere în OpenMP

Ce învățăm la APP?

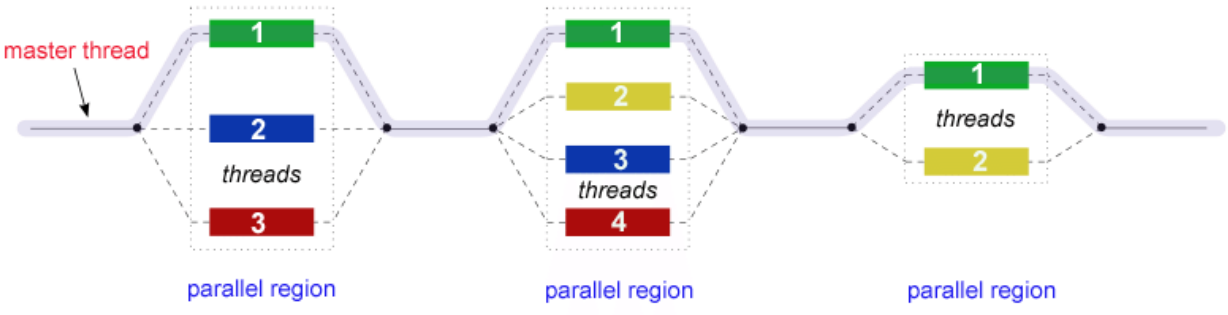
La APP vom învăța cum să analizăm o problemă și o soluție a acesteia (mai precis, un program secvențial) și cum am putea să îmbunătățim soluțiile la problema respectivă (adică cum am putea să eficientizăm prin paralelizare soluțiile problemei).

În cadrul laboratoarelor de APP vom studia despre programarea paralelă (OpenMP, pthreads), despre programarea distribuită (MPI) și despre analiza performanțelor unui program (profiling).

Ce este OpenMP?

OpenMP reprezintă un API prin care putem paraleliza programe secvențiale scrise în C/C++. Acesta este un API high-level, în sensul că programatorul are o varietate de tool-uri și de opțiuni la dispoziția sa, ele putând fi folosite cu mare ușurință.

OpenMP este creat pe baza modelului fork-join, unde avem un thread principal (master), din care se creează alte thread-uri (fork, echivalent cu `pthread_create` din pthreads), care, împreună cu thread-ul master, execută task-uri în paralel, în cadrul unor zone numite regiuni paralele. După ce task-urile respective sunt terminate, thread-urile forked “revin” în thread-ul principal (join).



Includere și compilare

Pentru a putea folosi OpenMP în cod, trebuie inclusă biblioteca `omp.h` în cod: `#include <omp.h>` .

Pentru compilare, este necesar un flag, care diferă în funcție de compilator:

- gcc: `gcc helloworld.c -o helloworld -fopenmp`
- SunStudio (pe fep): `cc -xopenmp helloworld.c -o helloworld`

Sintaxa OpenMP

Pentru OpenMP, se folosesc directive de compilare de tip pragma pentru a marca blocuri de cod paralelizate și pentru a folosi elemente de sincronizare.

Tipar: `#pragma omp numele_directivei [clause, ...]`

Exemplu: `#pragma omp parallel default(shared) private(beta, pi)`

Paralelizarea secvențelor de cod

Pentru ca o bucată de cod să fie executată de mai multe thread-uri, folosim directiva `#pragma omp parallel` prin care marcăm faptul că acea zonă de cod este executată în paralel de mai multe thread-uri.

```
#pragma omp parallel
{
        // cod paralelizat
}
```

Exemplu de folosire:

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

int main(int argc, char** argv) {
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("Hello world from thread number %d\n", tid);
    }

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("Frumos in anul 4, zice thread-ul %d\n", tid);
    }
    return 0;
}
```

De asemenea, putem avea regiuni paralele imbricate:

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

int main(int argc, char** argv) {
    #pragma omp parallel
    {
        printf("Parallel region thread %d\n", omp_get_thread_num());

        #pragma omp parallel
        {
            printf("Nested parallel region thread %d\n", omp_get_thread_num());
        }
    }
    return 0;
}
```

Funcții utile

Pentru setarea numărului de thread-uri din cadrul programului paralelizat, putem să facem în două moduri:

- din linia de comandă: `export OMP_NUM_THREADS=8`
- din cod: `omp_set_num_threads(8)`

Dacă dorim să măsurăm timpul de execuție al unei secvențe de cod paralelizat, putem folosi `omp_get_wtime()`. Exemplu de folosire:

```
t1 = omp_get_wtime();
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    printf("Hello world from thread number %d\n", tid);
}
t2 = omp_get_wtime();
printf("Total execution time = %lf\n", (t2 - t1));
```

Alte funcții utile:

- numărul de threads: `omp_get_num_threads()`
- id-ul thread-ului curent: `omp_get_thread_num()`
- numărul de procesoare: `omp_get_num_procs()`

Vizibilitatea variabilelor

Variabilele în cadrul blocurilor paralele pot fi:

- globale - văzute și partajate de toate thread-urile
- private - fiecare thread are variabilele sale private care nu sunt vizibile către alte thread-uri

În acest caz avem două clauze pentru context:

- `SHARED` - variabilă partajată între thread-uri (exemplu: `SHARED(c)`)
- `PRIVATE` - variabilă văzută doar de thread-ul respectiv în blocul paralelizat (exemplu: `PRIVATE(a, b)`)

Exemplu:

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

int main(int argc, char** argv) {
    int a = 6, b = 9, c = 10;

    #pragma omp parallel private(a,b) shared(c)
    {
        // privates set the scope of variables
        a = 1, b = 2, c = a + b; // cu private(a, b), aceste valori (la a si b) vor
        int tid = omp_get_thread_num();
        printf("In parallel block, in thread no %d: %d %d %d\n", tid, a, b, c); //
    }
    printf("%d %d %d\n", a, b, c); // printing 6 9 3

    #pragma omp parallel shared(c)
    {
        a = 1, b = 2, c = a + b;
        int tid = omp_get_thread_num();
        printf("In parallel block, in thread no %d: %d %d %d\n", tid, a, b, c); //
    }
    printf("%d %d %d\n", a, b, c); // printing 1 2 3

    #pragma omp parallel
    {
        a = 1, b = 2, c = a + b;
        int tid = omp_get_thread_num();
        printf("In parallel block, in thread no %d: %d %d %d\n", tid, a, b, c); //
    }
    printf("%d %d %d\n", a, b, c); // printing 1 2 3

    return 0;
}
```

Paralelizarea buclelor

În OpenMP putem paraleliza buclele de tip for folosind directiva `#pragma omp for` în cadrul unei zone paralele. În acest fel, iterațiile din for sunt împărțite egal thread-urilor, fiecare thread având iterațiile sale din cadrul buclei for.

Paralelizarea buclelor poate fi eficientizată folosind directiva `SCHEDULE`, despre care vom discuta în laboratorul 2.

Exemplu de folosire:

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

int main(int argc, char** argv) {
    int i, x[20];
    #pragma omp parallel private(i) shared(x)
    {
        #pragma omp for
        for (i = 0; i < 20; i++) {
            x[i] = i;
            printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
        }
    }

    printf("\n");

    // o alta forma, aceeasi functionalitate
    #pragma omp parallel for private(i) shared(x)
    for (i = 0; i < 20; i++) {
        x[i] = i;
        printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
    }

    return 0;
}
```

Elemente de sincronizare

În OpenMP avem la dispoziție elemente de sincronizare, prin care putem să ne asigurăm faptul că soluția paralelizată funcționează corect, fără probleme în ceea ce privește rezultatele incorecte sau deadlocks.

Mutex

Pentru zonele critice, unde avem operații de read-write, folosim directiva `#pragma omp critical`, care reprezintă un mutex, echivalentul lui `pthread_mutex_t` din pthreads, care asigură faptul că un singur thread accesează zona critică la un moment dat, thread-ul deținând lock-ul pe zona critică în momentul respectiv, și că celelalte thread-uri care nu au intrat încă în zona critică așteaptă eliberarea lock-ului de către thread-ul aflat în zona critică în acel moment.

Exemplu de folosire:

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

int main (int argc, char** argv) {
    int thread_id, sum = 0;
    #pragma omp parallel private(thread_id) shared(sum)
    {
        thread_id = omp_get_thread_num();
        #pragma omp critical
        sum += thread_id;
    }
    printf("%d",sum);

    return 0;
}
```

Barieră

Un alt element de sincronizare reprezintă bariera, care asigură faptul că niciun thread gestionat de barieră nu trece mai departe de aceasta decât atunci cand toate thread-urile gestionate de barieră au ajuns la punctul unde se află bariera.

În OpenMP, pentru barieră avem directiva `#pragma omp barrier`, echivalent cu `pthread_barrier_t` din pthreads.

Exemplu de folosire:

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

int main (int argc, char** argv) {
    #pragma omp parallel
    {
        printf("First print by %d\n", omp_get_thread_num());
        #pragma omp barrier
        printf("Second print by %d\n", omp_get_thread_num());
    }

    return 0;
}
```

Reduction

`reduction` este o directivă folosită pentru operații de tip reduce / fold pe arrays / colecții sau simple însumări / înmulțiri în cadrul unui loop. Mai precis, elementele dintr-un array sau indecșii unui loop sunt "acumulați" într-o singură variabilă, cu ajutorul unei operații, al cărui semn este precizat.

Tipar: `reduction(operator_operatie:variabila_in_care_se_acumuleaza)`

Exemplu de reduction: `reduction(+:sum)`, unde se însumează elementele unui array în variabila `sum`

Exemplu de folosire de reduction:

```
int sum = 0;

#pragma omp parallel for reduction(+:sum) private(i)
for (i = 1; i <= num_steps; i++) {
    sum += i;
}
```

Exerciții

1) (2 puncte) Descărcați scheletul de laborator și demo-ul pentru laborator. Rulați exemplele din demo.

2) (8 puncte) Paralelizați însumarea elementelor unui array, încât suma să fie corectă, folosind fișierul `array_sum.c` din schelet, unde este implementată suma serială a elementelor dintr-un array.

Resurse utile

- http://jakascorner.com/blog/
- https://ppc.cs.aalto.fi/

Laboratorul 2 - Paralelizarea buclelor în OpenMP

Laboratorul 2 - Paralelizarea buclelor în OpenMP

Loop scheduling

În OpenMP, când o structură de tip for este paralelizată fiecărui thread îi revine un număr egal de iterații din cadrul acelui for. Uneori, se întâmplă ca iterațiile să fie echilibrate între ele în ceea ce privește workload-ul, alteori nu. Când workload-ul nu este echilibrat între thread-uri, pot apărea probleme în ceea ce privește performanțele programului.

Pentru a preveni situații când thread-urile au volume diferite de workload, există conceptul de scheduling în OpenMP. Pentru scheduling se folosește directiva `schedule(tip_de_schedule, chunk_size)` , unde se precizează tipul de schedule (static, dynamic, guided, auto) și dimensiunea unui chunk (acest lucru este opțional, în cazul în care nu se specifică acesta are o valoare default).

Tipar de scheduling:

```
#pragma omp parallel for schedule(tip_de_schedule, chunk_size)
for (int i = 0; i < 100; i++) {
    // do stuff
}
```

Static scheduling

În cadrul static scheduling, iterațiile unui for împărțite în chunks, de dimensiune `chunk_size` , și distribuite thread-urilor în ordine circulară. Dacă `chunk_size` nu este precizat, acesta va fi egal cu `numărul_de_iterații_for / numărul_de_thread-uri`

Exemple - folosim 4 thread-uri:

1) `chunk_size` = 2

```
#pragma omp parallel for private(i) schedule(static, 2)
for (i = 0; i < 16; i++) {
    printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
}
```

În acest caz, distribuția iterațiilor pe thread-uri va fi în felul următor:

- thread-ul 0: 0, 1 (chunk 1), 8, 9 (chunk 5)
- thread-ul 1: 2, 3 (chunk 2), 10, 11 (chunk 6)
- thread-ul 2: 4, 5 (chunk 3), 12, 13 (chunk 7)
- thread-ul 3: 6, 7 (chunk 4), 14, 15 (chunk 8)

2) `chunk_size` = 4 (dacă nu l-am preciza, default ar fi 4 în acest caz - 16 iterații și 4 threads, deci 16 / 4)

```
#pragma omp parallel for private(i) schedule(static, 4)
for (i = 0; i < 16; i++) {
    printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
}
```

În acest caz, distribuția iterațiilor pe thread-uri va fi în felul următor:

- thread-ul 0: 0, 1, 2, 3 (chunk 1)
- thread-ul 1: 4, 5, 6, 7 (chunk 2)
- thread-ul 2: 8, 9, 10, 11 (chunk 3)
- thread-ul 3: 12, 13, 14, 15 (chunk 4)

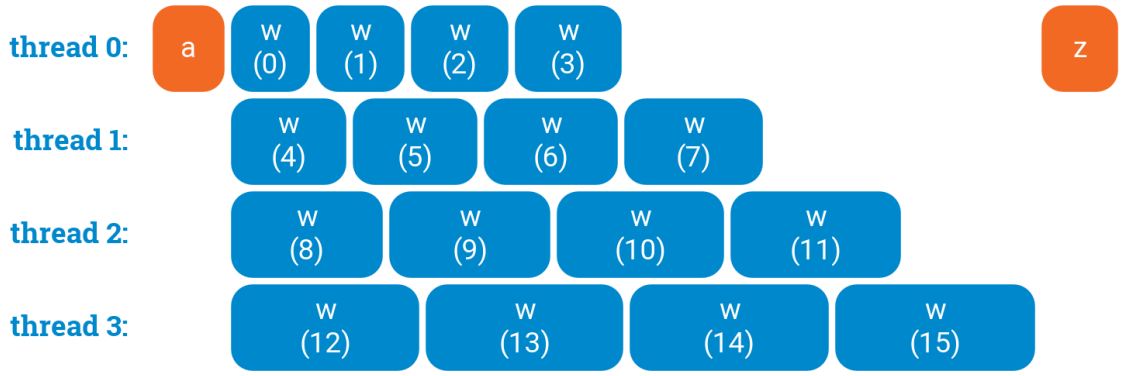
Static schedule este util de folosit atunci când se știe că există echilibru între iterațiile din chunks pentru un anumit `chunk_size`.

Cum poate influența valoarea `chunk_size` performanțele

Să luam următorul exemplu de cod (rulând cu 4 threads), unde `w(int i)` este o funcție a cărei complexitate este influențată de parametrul de intrare i:

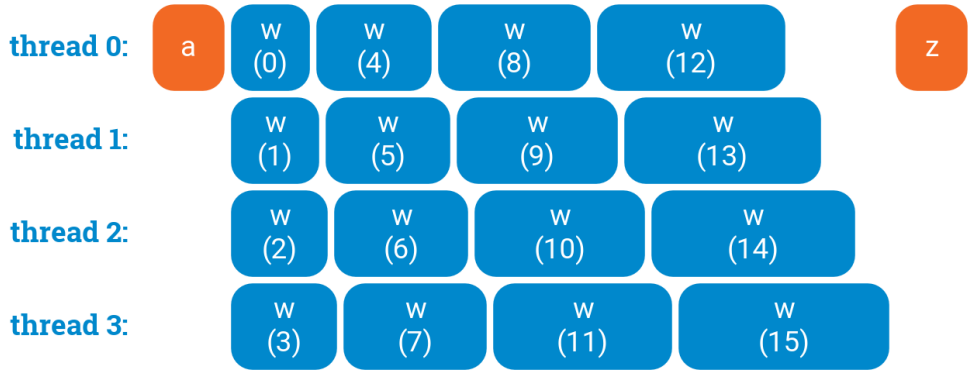
```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```

Grafic, fără niciun tip de scheduling, performanța este ilustrată în felul următor:



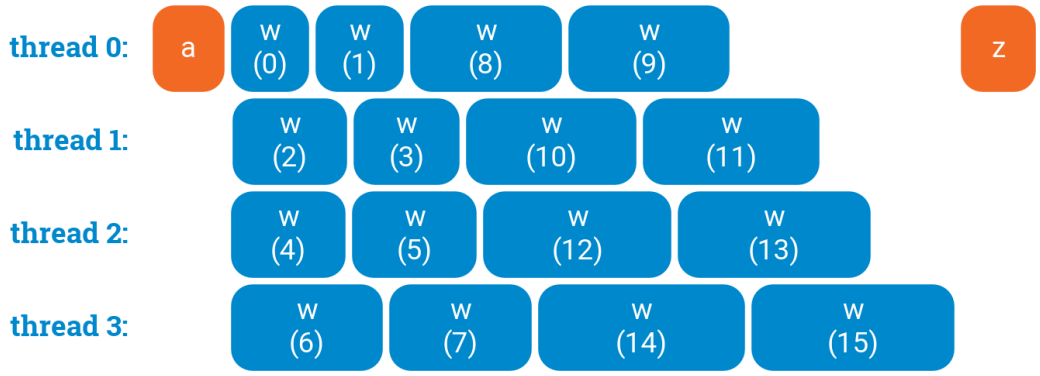
Aici se poate observa un dezechilibru în ceea ce privește workload-ul între thread-uri, thread-ul 3 având un workload dublu față de thread-ul 0.

Dacă setăm `chunk_size` cu 1, performanța este ilustrată în felul următor:



În acest caz, thread-urile au un workload echilibrat, datorită distribuirii uniforme a iterațiilor din for, așadar avem o performanță bună.

Dacă setăm `chunk_size` cu 2, performanța este ilustrată în felul următor:



În acest caz, se observă un dezechilibru între thread-uri din punctul de vedere al workload-ului, implicând o performanță mai proastă decât în cazul când `chunk_size` este setat cu 1.

Dynamic scheduling

În cadrul dynamic scheduling, iterațiile sunt împărțite în chunks de dimensiune chunk_size, ca la static scheduling, însă diferența față de static scheduling este că iterațiile nu sunt distribuite într-o anumită ordine către thread-uri, așa cum se întâmplă la static schedule. Dacă nu se precizează valoarea pentru `chunk_size`, atunci va avea valoarea default de 1.

Dynamic schedule este folosit atunci când avem iterații total debalansate în ceea ce privește workload-ul (în timp ce la static schedule putem intui un pattern de workload între iterații - acest lucru puteți observa în demo, în fișierul `static_schedule.c`).

Exemplu:

```
#pragma omp parallel for private(i) schedule(dynamic, 2)
for (i = 0; i < 16; i++) {
    w(i);
    printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
}
```

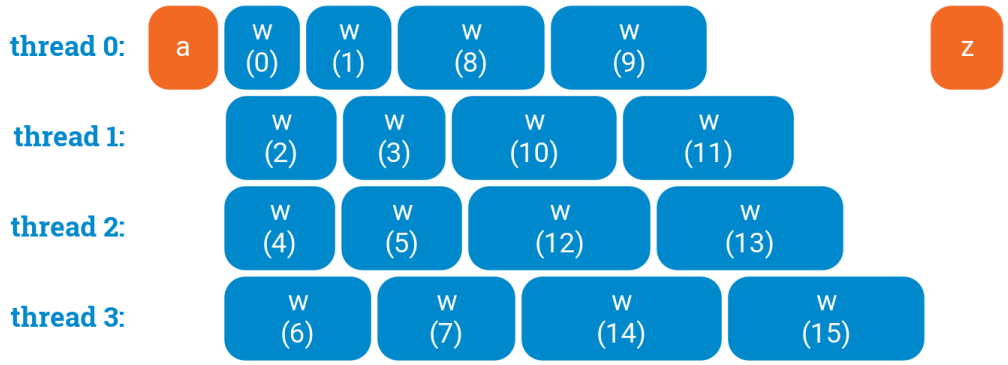
unde `w(i)` este o funcție ce are performanță aleatoare, care poate să nu depindă de parametrul de intrare.

Spre deosebire de static scheduling, la dynamic scheduling există comunicare între threads după fiecare iterație, cu scopul de a construi o împărțire a iterațiilor între thread-uri, pentru a avea workload-urile între thread-uri cât mai balansate, fapt ce duce la overhead.

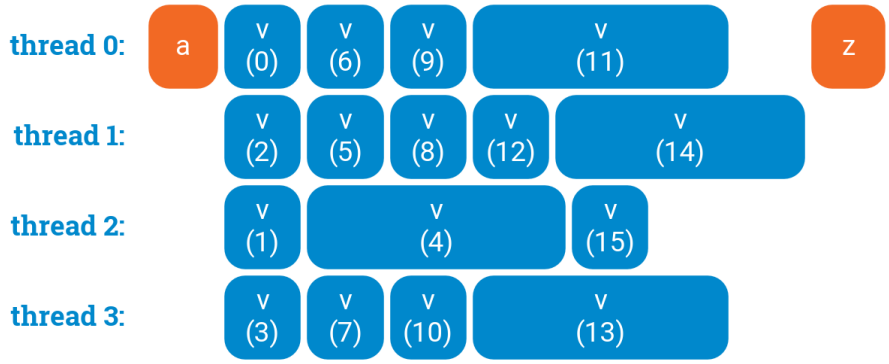
După fiecare rulare, împărțirea iterațiilor către thread-uri se schimbă.

Un exemplu grafic ar fi următorul:

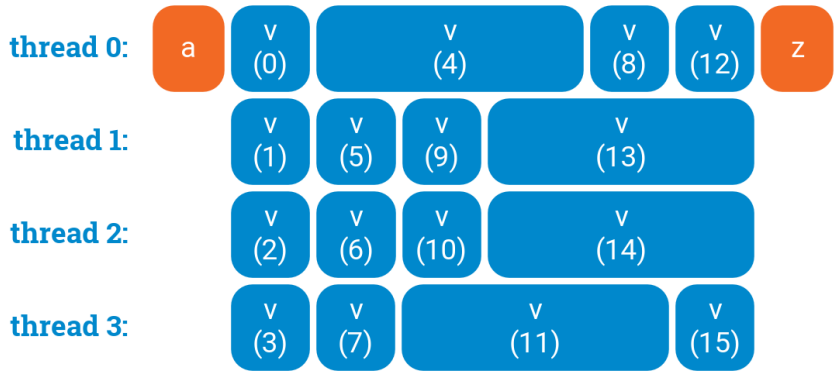
- aici nu folosim niciun tip de schedule:



- aici folosim dynamic schedule și se poate observa că nu există distribuție uniformă a iterațiilor către thread-uri:



De reținut faptul că dynamic schedule poate să nu fie mereu optim și este posibil ca static schedule să fie o soluție mai bună, cum ar fi în acest exemplu (static cu `chunk_size = 1`):



Guided scheduling

Guided scheduling se aseamănă cu dynamic schedule, în sensul că avem o împărțire pe chunks și distribuire neuniformă a iterațiilor.

Diferența față de dynamic schedule constă în dimensiunea chunk-urilor. Dimensiunea unui chunk este proporțională cu numărul de iterații neasignate în acel moment thread-ului împărțit la numărul de threads, la început un chunk putând avea dimensiunea `nr_iterații / nr_threads`, ca pe parcurs să scadă dimensiunea acestuia. De exemplu la Valoarea la chunk_size (dacă nu avem, default este 1) reprezintă dimensiunea minimă pe care o poate avea un chunk (se poate ca ultimele chunk să aibă o dimensiune mai mică decât dimensiunea dată unui chunk).

Exemplu:

```
#pragma omp parallel for private(i) schedule(guided, 2)
for (i = 0; i < 16; i++) {
    w(i);
    printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
}
```

Guided schedule este folosit când între iterații există un dezechilibru major în ceea ce privește workload-ul.

Auto scheduling

La auto scheduling, tipul de scheduling (static, dynamic, guided) este determinat la compilare și/sau la runtime.

Exemplu:

```
#pragma omp parallel for private(i) schedule(auto)
for (i = 0; i < 16; i++) {
    w(i);
    printf("iteration no. %d | thread no. %d\n", i, omp_get_thread_num());
}
```

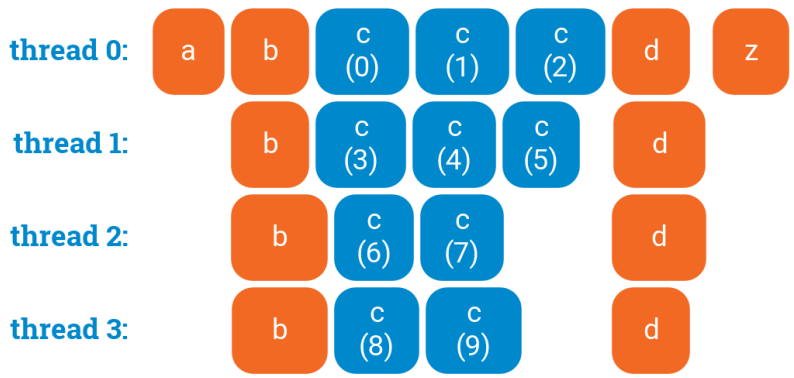
nowait

Atunci când paralelizăm un for, există o barieră după fiecare for paralelizat, unde se așteaptă ca toate thread-urile din for-ul paralelizat să ajungă în același punct în același timp, ca apoi să-și continue execuția.

Exemplu:

```
#pragma omp parallel
{
    #pragma omp for nowait private(i)
    for (i = 0; i < 16; i++) {
        c(i);
    }
    d();
}
```

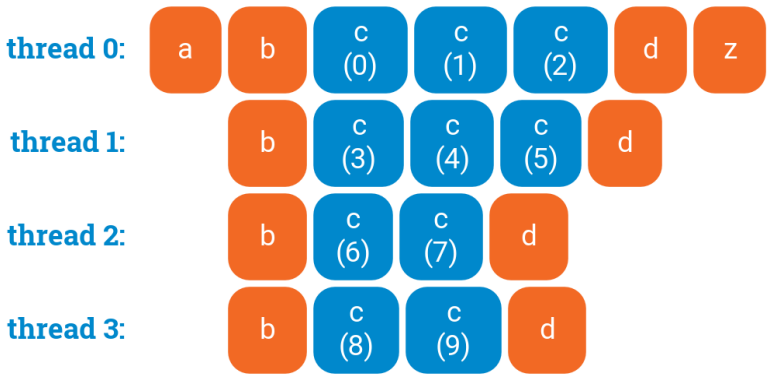
În exemplul de mai sus, thread-urile din for-ul paralelizat trebuie să ajungă în același punct (să se sincronizeze), ca apoi să execute funcția `d()`.



Pentru a elimina această barieră / sincronizare, putem folosi directiva `nowait`, prin care thread-urile nu mai așteaptă ca fiecare să ajungă în același punct (să fie sincronizate încât să fie în același punct în același timp), astfel un thread, după ce trece de for-ul paralelizat, trece imediat la execuția următoarelor instrucțiunilor, fără să mai aștepte după celelalte thread-uri.

Exemplu:

```
#pragma omp parallel
{
    #pragma omp for nowait private(i)
    for (i = 0; i < 16; i++) {
        c(i);
    }
    d();
}
```



Reduction - recapitulare

`reduction` este o directivă folosită pentru operații de tip reduce / fold pe arrays / colecții sau simple însumări / înmulțiri în cadrul unui loop. Mai precis, elementele dintr-un array sau indecșii unui loop sunt "acumulați" într-o singură variabilă, cu ajutorul unei operații, al cărui semn este precizat.

Tipar: `reduction(operator_operatie:variabila_in_care_se_acumuleaza)`

Exemplu de reduction: `reduction(+:sum)`, unde se însumează elementele unui array în variabila `sum`

Exemplu de folosire de reduction:

```
int sum = 0;

#pragma omp parallel for reduction(+:sum) private(i)
for (i = 1; i <= num_steps; i++) {
    sum += i;
}
```

Exerciții

1) (1 punct) Rulați demo-urile legate de schedule și modificați valorile la `chunk_size`, unde să faceți observații legate de performanțe. De asemenea, puteți schimba și tipul de schedule, pentru a observa eventuale schimbări în privința performanței.

2) (3 puncte) Paralelizați fișierul `atan.c` din schelet folosind reduction.

3) (3 puncte) Rulați programul din fișierul `schedule.c` din schelet de mai multe ori și schimbați tipurile de schedule (cu tot cu `chunk_size`) și observați performanțele.

4) (3 puncte) Generați fișiere folosind scriptul gen_files.sh și paralelizați programul din fișierul `count_letters.cpp` din schelet.

Resurse utile

- http://jakascorner.com/blog/
- https://ppc.cs.aalto.fi/
- **dynamic scheduling vs. guided scheduling**

Laboratorul 3 - Advanced OpenMP

Laboratorul 3 - Advanced OpenMP

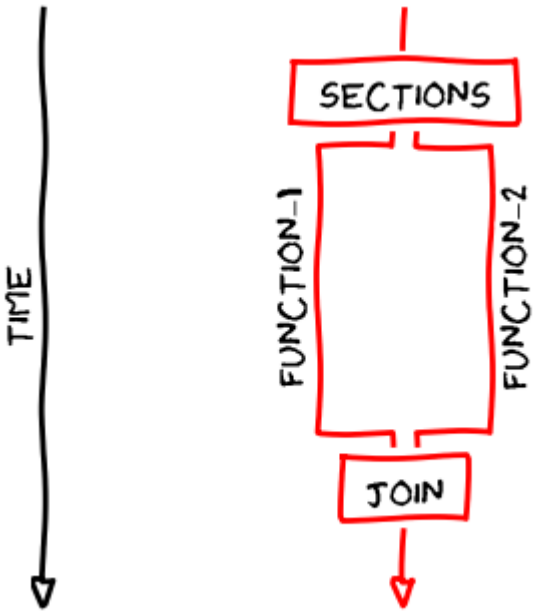
Sections

Uneori dorim să distribuim ca thread-uri diferite să execute task-uri diferite în același timp. În această privință ne vine de ajutor conceptul de sections, prin care două sau mai multe thread-uri execută două sau mai multe sections corespunzătoare acestora (adică thread-urilor, fiecare thread cu un section).

În OpenMP se folosește directiva `sections` pentru a marca o zonă din cod în care distribuim task-urile diferite (sections) thread-urilor (fiecare thread cu câte un section). Sintaxa în OpenMP este următoarea:

```
#pragma omp parallel
{
    // se marchează blocul de sections
    #pragma omp sections
    {
        #pragma omp section
        {
            // section executat de thread-ul X
        }

        #pragma omp section
        {
            // section executat de thread-ul Y
        }

        #pragma omp section
        {
            // section executat de thread-ul Z
        }
    }

    #pragma omp sections
    {
        #pragma omp section
        {
            // section executat de thread-ul X
        }
```

```
        #pragma omp section
        {
            // section executat de thread-ul Y
        }
    }
}

#pragma omp parallel sections
{
    #pragma omp section
    {
        // section executat de thread-ul X
    }

    #pragma omp section
    {
        // section executat de thread-ul Y
    }
}
```



Single

Dacă dorim ca o secvență de cod (dintr-o bucată de cod paralelizat) să fie executat doar de un singur thread, folosim directiva SINGLE. Aceasta este folosită, de regulă, în operații I/O.

Exemplu:

```
#pragma omp parallel
{
```

```
    #pragma omp single
    {
        // cod executat de un singur thread
    }
}
```

Master

Directiva MASTER este o particularizare a directivei SINGLE, unde codul din zona paralelizată este executat de thread-ul master (cel cu id-ul 0).

```
#pragma omp parallel
{
    #pragma omp master
    {
        // cod executat de un singur thread
    }
}
```

Construcții de sincronizare

Mutex

Pentru zonele critice, unde avem operații de read-write, folosim directiva `#pragma omp critical`, care reprezintă un mutex, echivalentul lui `pthread_mutex_t` din pthreads, care asigură faptul că un singur thread accesează zona critică la un moment dat, thread-ul deținând lock-ul pe zona critică în momentul respectiv, și că celelalte thread-uri care nu au intrat încă în zona critică așteaptă eliberarea lock-ului de către thread-ul aflat în zona critică în acel moment.

Exemplu de folosire:

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

int main (int argc, char** argv) {
    int thread_id, sum = 0;
    #pragma omp parallel private(thread_id) shared(sum)
    {
        thread_id = omp_get_thread_num();
        #pragma omp critical
        sum += thread_id;
    }
    printf("%d",sum);

    return 0;
}
```

Barieră

Un alt element de sincronizare reprezintă bariera, care asigură faptul că niciun thread gestionat de barieră nu trece mai departe de aceasta decât atunci cand toate thread-urile gestionate de barieră au ajuns la punctul unde se află bariera.

În OpenMP, pentru barieră avem directiva `#pragma omp barrier` , echivalent cu `pthread_barrier_t` din pthreads.

Exemplu de folosire:

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

int main (int argc, char** argv) {
    #pragma omp parallel
    {
        printf("First print by %d\n", omp_get_thread_num());
        #pragma omp barrier
        printf("Second print by %d\n", omp_get_thread_num());
    }

    return 0;
}
```

Reduction

`reduction` este o directivă folosită pentru operații de tip reduce / fold pe arrays / colecții sau simple însumări / înmulțiri în cadrul unui loop. Mai precis, elementele dintr-un array sau indecșii unui loop sunt “acumulați” într-o singură variabilă, cu ajutorul unei operații, al cărui semn este precizat.

Tipar: `reduction(operator_operatie:variabila_in_care_se_acumuleaza)`

Exemplu de reduction: `reduction(+:sum)` , unde se însumează elementele unui array în variabila `sum`

Exemplu de folosire de reduction:

```
int sum = 0;

#pragma omp parallel for reduction(+:sum) private(i)
for (i = 1; i <= num_steps; i++) {
    sum += i;
}
```

Atomic

Directiva ATOMIC permite executarea unor instrucțiuni în mod atomic, instrucțiuni care provoacă race conditions între thread-uri, problemă pe care această directivă o rezolvă.

Exemplu de folosire:

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

int main (int argc, char** argv) {
    int thread_id, sum = 0;
    #pragma omp parallel private(thread_id) shared(sum)
    {
        thread_id = omp_get_thread_num();
        #pragma omp atomic
        sum += thread_id;
    }
    printf("%d",sum);

    return 0;
}
```

Ordered

Directiva ORDERED este folosit în for-uri cu scopul de a distribui în ordine iterațiile către thread-uri.

Exemplu:

```
#pragma omp parallel for ordered private(i)
for (i = 0; i < 10; i++) {
    printf("** iteration %d thread no. %d\n", i, omp_get_thread_num());
}
```

Afișare:

```
** iteration 9 thread no. 7
** iteration 5 thread no. 3
** iteration 6 thread no. 4
** iteration 7 thread no. 5
** iteration 4 thread no. 2
** iteration 2 thread no. 1
** iteration 3 thread no. 1
** iteration 0 thread no. 0
** iteration 1 thread no. 0
** iteration 8 thread no. 6
```

Clauze legate de vizibilitatea variabilelor

- `SHARED`
- `PRIVATE` - variabilă cu valoarea vizibilă doar în blocul paralel (diferă de THREADPRIVATE)
- `DEFAULT`
- `REDUCTION`
- `NONE`
- `THREADPRIVATE` - fiecare thread are propriile sale copii ale unor variabile
- `FIRSTPRIVATE` - folosit pentru ca variabilele `THREADPRIVATE` să aibă valorile, inițial, din exterior (dinainte)
- `LASTPRIVATE` - invers FIRSTPRIVATE, ultima valoare asignată unei variabile `THREADPRIVATE` e vizibilă după blocul paralelizat
- `COPYPRIVATE` - folosit în blocurile `SINGLE`, pentru a face vizibilă valoarea atribuită unei variabile într-un bloc `SINGLE` pentru toate thread-urile
- `COPYIN` - asignarea unei variabile `THREADPRIVATE` este vizibilă tuturor thread-urilor

Tasks (opțional)

Task-urile în OpenMP reprezintă un concept prin care putem să avem thread pools pentru paralelizarea de soluții ale căror dimensiune nu o știm (echivalent cu `ExecutorService` din Java). Un task este executat la un moment dat de către un thread din thread pool.

Pentru crearea unui task se folosește directiva `TASK`:

```
#pragma omp task [clause1 [[,] clause2, ...]]
```

Pentru sincronizarea task-urilor (în sensul să așteptăm toate rezultatele task-urilor, în stilul barierei), se folosește directiva `TASKWAIT` (exemplu de folosire în exemplul Fibonacci de mai jos).

În privința variabilelor dintr-un task, aici avem trei variante de variabile:

- `shared` - toate task-urile au acces la aceeași adresă a unei variabile, o modificare asupra variabilei din partea unui task va fi vizibilă către toate task-urile (uneori putem avea potențial de erori în acest caz).
- `firstprivate` - fiecare task va avea o copie a unei variabile inițializate cu o valoare înainte de crearea task-ului respectiv.
- `private` - aici putem să avem variabile care nu sunt inițializate înainte de crearea task-ului și care să fie inițializate în cadrul task-ului.

```
void f () {
    double x1 = 1.0;
    double x2 = 2.0;
    #pragma omp parallel firstprivate(x2)
    {
        double x3 = 3.0; // private to each implicit task due to scope
        #pragma omp task
        {
```

```
            double x4 = 4.0; // private due to scope
            // x1 : shared ( shared by all implicit tasks )
            // x2 : firstprivate ( due to “firstprivate(x2)” )
            // x3 : firstprivate ( not shared by all implicit tasks )
        }
    }
}
```

Pentru paralelizarea unor probleme recursive (Fibonacci, parcurgeri, etc.), task-urile reprezintă o soluție optimă în acest caz. De asemenea, putem crea task-uri în cadrul unui task părinte.

Exemplu:

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

int fib(int n) {
    int i, j;
    printf("n = %d | Thread id = %d\n", n, omp_get_thread_num());

    if (n < 2) {
        return n;
    }

    #pragma omp task shared(i)
    i = fib(n - 1);

    #pragma omp task shared(j)
    j = fib(n - 2);

    // se așteaptă să se termine task-urile de mai sus
    #pragma omp taskwait
    return i + j;
}


int main() {
    int n = 10;
    omp_set_num_threads(4);

    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf ("fib(%d) = %d\n", n, fib(n));
    }
}
```

Se pot observa asemănări între tasks și sections în ceea ce privește modul de folosire (se pot folosi sections în cadrul problemelor recursive), însă diferențele dintre aceastea sunt următoarele:

- la sections instrucțiunile sunt executate imediat când thread-ul asociat acelui section ajunge în section-ul respectiv, când la tasks instrucțiunile pot fi executate după ce thread-ul asociat task-ului trece de task-ul respectiv
- la sections putem avea overhead și load balancing slab

Exerciții

1) (10 puncte) Paralelizați fișierul main.c din schelet, unde se citește un fișier, unde pe prima linie se află numărul de elemente pentru un array și pe următoarea linie se află array-ul respectiv, se face suma numerelor (aici faceți în trei moduri, separat, cu reduction, cu atomic și cu critical, unde veți măsura timpii de execuție - hint, folosiți directiva master ca un singur thread să facă măsurătorile), iar la final, cu ajutorul sections, scrieți timpii de execuție în trei fișiere (este deja implementată funcția de scriere în fișier).

Hint: o să aveți nevoie de barieră la citire și înainte de scrierea în fișiere.

De probă, încercați să puneți `ORDERED` la for-urile paralelizate, pentru a vedea cum este afectată performanța.

2) (opțional) Paralelizați folosind task-uri codul din `tree.c` (folosiți task-uri în funcțiile `preorder` și `height` - la ultima trebuie să folosiți `taskwait`).

Resurse

- **Cursuri de OpenMP, MPI, CUDA - COSC462**

Laboratorul 4 - MPI

Laboratorul 4 - MPI

Despre MPI

MPI (Message Passing Interface) reprezintă un standard pentru comunicarea prin mesaje în cadrul programării distribuite, elaborat de MPI Forum, și are la bază modelul proceselor comunicante prin mesaje.

Un proces reprezintă un program aflat în execuție și se poate defini ca o unitate de bază care poate executa una sau mai multe sarcini în cadrul unui sistem de operare. Spre deosebire de thread-uri, un proces are propriul său spațiu de adrese (propria zonă de memorie) și acesta poate avea, în cadrul său, mai multe thread-uri în execuție, care partajează resursele procesului.

Compilare și rulare

În cadrul lucrului în C/C++, MPI reprezintă o bibliotecă, care are funcționalitățile implementate într-un header numit `mpi.h`. Pentru compilare, la MPI există un compilator specific:

- `mpicc`, pentru lucrul în C
- `mpic++`, pentru lucrul în C++

În ambele limbaje, pentru rularea unui program MPI folosim comanda `mpirun`, împreună cu parametrul `-np`, unde precizăm numărul de procese care rulează în cadrul programului distribuit.

Exemplu:

- compilare:
  - C: `mpicc hello.c -o hello`
  - C++: `mpic++ hello.cpp -o hello`
- rulare: `mpirun -np 4 hello` - rulare cu 4 procese

Dacă încercați să rulați comanda mpirun cu un număr de procese mai mare decât numărul de core-uri fizice disponibile pe procesorul vostru, este posibil să primiți o eroare cum ca nu aveți destule sloturi libere. Puteți elimina acea eroare adăugând parametrul `--oversubscribe` atunci când rulați mpirun.

Instalare MPI

Pentru a lucra cu MPI, trebuie să instalați biblioteca pentru MPI pe Linux, folosind următoarea comandă:

```
sudo apt install openmpi-bin openmpi-common openmpi-doc libopenmpi-dev
```

Implementarea unui program distribuit în MPI

Exemplu de program MPI - Hello World:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define  MASTER 0

int main (int argc, char *argv[]) {
    int numtasks, rank, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(hostname, &len);
    if (rank == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
    else
        printf("WORKER: Rank: %d\n",rank);

    MPI_Finalize();
}
```

Un comunicator (`MPI_Comm`) reprezintă un grup de procese care comunică între ele. `MPI_COMM_WORLD` reprezintă comunicatorul default, din care fac parte toate procesele.

Funcții:

- `MPI_Init` - se inițializează programul MPI, mai precis se creează contextul în cadrul căruia rulează procesele. Argumentele din linie de comandă sunt pasate către contextul de rulare a proceselor.
- `MPI_Comm_size` - funcție care determină numărul de procese (numtasks) care rulează în cadrul comunicatorului (de regulă `MPI_COMM_WORLD`)
- `MPI_Comm_rank` - funcție care determină identificatorul (rangul) procesului curent în cadrul comunicatorului.
- `MPI_Get_processor_name` - determină numele procesorului
- `MPI_Finalize` - declanșează terminarea programului MPI

În cadrul schimbului de date între procese, este necesar mereu să precizăm tipul acestora. În MPI, se folosește enum-ul `MPI_Datatype` , care se mapează cu tipurile de date din C/C++, după cum puteți vedea în tabelul de mai jos: | `MPI_Datatype` | Echivalentul din C/C++ | | ————- | ———— | | `MPI_INT` | `int` | | `MPI_LONG` | `long` | | `MPI_CHAR` | `char` | | `MPI_FLOAT` | `float` | | `MPI_DOUBLE` | `double` |

Funcții de transmitere a datelor

MPI_Send

`MPI_Send` reprezintă funcția prin care un proces trimite date către un alt proces. Semnătura funcției este următoarea:

```
int MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag
```

, unde:

- `data (↓)` - reprezintă datele trimise de la procesul sursă către procesul destinație
- `count (↓)` - dimensiunea datelor transmise
- `datatype (↓)` - tipul datelor transmise
- `destination (↓)` - rangul / identificatorului procesului destinație, către care se trimit datele
- `tag (↓)` - identificator al mesajului
- `communicator (↓)` - comunicatorul în cadrul căruia se face trimiterea datelor între cele două procese

MPI_Send este o funcție blocantă. Mai precis, programul se blochează până când bufferul dat ca prim parametru poate fi refolosit, chiar dacă nu se execută acțiunea de primire a mesajului transmis de procesul curent (`MPI_Recv`)

Dacă apare cazul în care procesul P1 trimite date (`MPI_Send`) la procesul P2, iar P2 nu are suficient loc în buffer-ul de recepție (buffer-ul nu are suficient loc liber sau este plin) atunci P1 se va bloca.

MPI_Recv

`MPI_Recv` reprezintă funcția prin care un proces primește date de la un alt proces. Semnătura funcției este următoarea:

```
int MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI
```

, unde:

- `data (↑)` - reprezintă datele primite de la procesul sursă de către procesul destinație
- `count (↓)` - dimensiunea datelor primite
- `datatype (↓)` - tipul datelor primite
- `source (↓)` - rangul / identificatorului procesului sursă, care trimite datele
- `tag (↓)` - identificator al mesajului
- `communicator (↓)` - comunicatorul în cadrul căruia se face trimiterea datelor între cele două procese
- `status` - conține date despre mesajul primit, `MPI_Status` fiind o structură ce conține informații despre mesajul primit (sursa, tag-ul mesajului, dimensiunea mesajului). Dacă nu dorim să ne folosim de datele despre mesajul primit, punem `MPI_STATUS_IGNORE` , prin care se ignoră status-ul mesajului.

În situația în care procesul P apelează funcția de `MPI_Recv()` , el se va bloca până va primi toate datele asteptate, astfel că dacă nu va primi nimic sau ceea ce primește este insuficient, P va rămâne blocat. Adică `MPI_Recv()` se termină doar în momentul în care buffer-ul a fost umplut cu datele așteptate.

Structura `MPI_Status` include următoarele câmpuri:

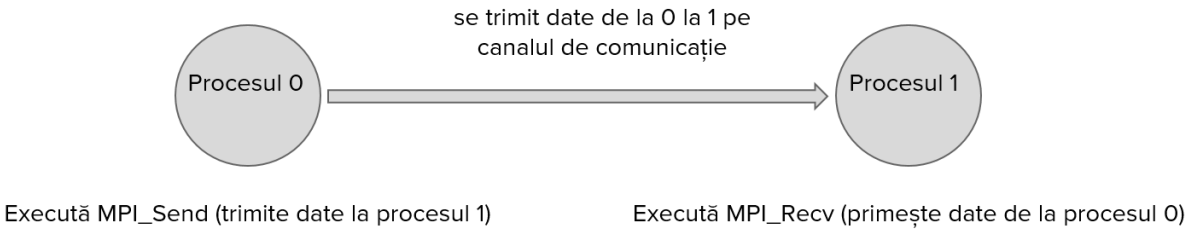
- `int count` - dimensiunea datelor primite
- `int MPI_SOURCE` - identificatorul procesului sursă, care a trimis datele
- `int MPI_TAG` - tag-ul mesajului primit MPI_Recv este o funcție blocantă, mai precis programul se poate bloca până când se execută acțiunea de trimitere a mesajului către procesul sursă.

Un exemplu de program în care un proces trimite un mesaj către un alt proces:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int  numtasks, rank, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // Total number of processes.
    MPI_Comm_rank(MPI_COMM_WORLD,&rank); // The current process ID / Rank.
    MPI_Get_processor_name(hostname, &len);

    srand(42);
    int random_num = rand();
    printf("Before send: process with rank %d has the number %d.\n", rank,
            random_num);

    if (rank == 0) {
        MPI_Send(&random_num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        MPI_Recv(&random_num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process with rank %d, received %d with tag %d.\n",
                rank, random_num, status.MPI_TAG);
    }

    printf("After send: process with rank %d has the number %d.\n", rank,
            random_num);

    MPI_Finalize();

}
```

Când un proces X trimite un mesaj către un proces Y, tag-ul T al mesajului din `MPI_Send` , executat de procesul X, trebuie să fie același cu tag-ul mesajului din `MPI_Recv` , executat de procesul Y, deoarece procesul Y așteaptă un mesaj care are tag-ul T, altfel, dacă sunt tag-uri diferite, programul se va bloca.

O ilustrație a modului cum funcționează împreună funcțiile MPI_Send și MPI_Recv:



Mai jos aveți un exemplu în care un proces trimite un întreg array de 100 de elemente către un alt proces:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int numtasks, rank, len;
    int size = 100;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    int arr[size];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(hostname, &len);

    srand(42);
    if (rank == 0) {
        for (int i = 0; i < size; i++) {
            arr[i] = i;
        }

        printf("Process with rank [%d] has the following array:\n", rank);
        for (int i = 0; i < size; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");

        MPI_Send(&arr, size, MPI_INT, 1, 1, MPI_COMM_WORLD);
        printf("Process with rank [%d] sent the array.\n", rank);
    } else {
        MPI_Status status;
        MPI_Recv(&arr, size, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        printf("Process with rank [%d], received array with tag %d.\n",
                rank, status.MPI_TAG);

        printf("Process with rank [%d] has the following array:\n", rank);
        for (int i = 0; i < size; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }
```

```
    MPI_Finalize();

}
```

MPI_Bcast

`MPI_Bcast` reprezintă o funcție prin care un proces trimite un mesaj către toate procesele din comunicator (message broadcast), inclusiv lui însuși.

În cadrul implementării `MPI_Bcast` sunt executate acțiunile de trimitere și de recepționare de mesaje, așadar nu trebuie să apelați `MPI_Recv`.

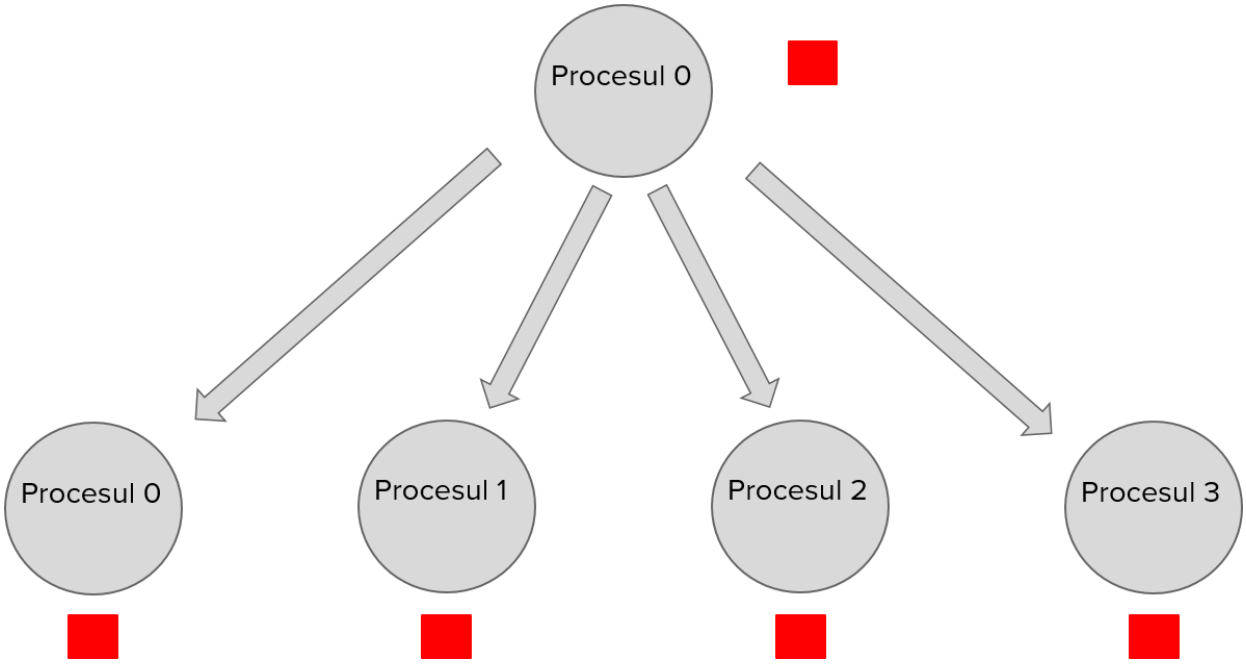
Semnătura funcției este următoarea:

```
int MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm comm
```

, unde:

- `data (↓ + ↑)` - reprezintă datele care sunt transmise către toate procesele. Acest parametru este de tip input pentru procesul cu identificatorul root și este de tip output pentru restul proceselor.
- `count (↓)` - dimensiunea datelor trimise
- `datatype (↓)` - tipul datelor trimise
- `root (↓)` - rangul / identificatorului procesului sursă, care trimite datele către toate procesele din comunicator, inclusiv lui însuși
- `tag (↓)` - identificator al mesajului
- `communicator (↓)` - comunicatorul în cadrul căruia se face trimiterea datelor către toate procesele din cadrul acestuia

O ilustrație care arată cum funcționează `MPI_Bcast` aveți mai jos:



MPI_Scatter

`MPI_Scatter` este o funcție prin care un proces împarte un array pe bucăți egale ca dimensiuni, unde fiecare bucată revine, în ordine, fiecărui proces, și le trimite tuturor proceselor din comunicator, inclusiv lui însuși.

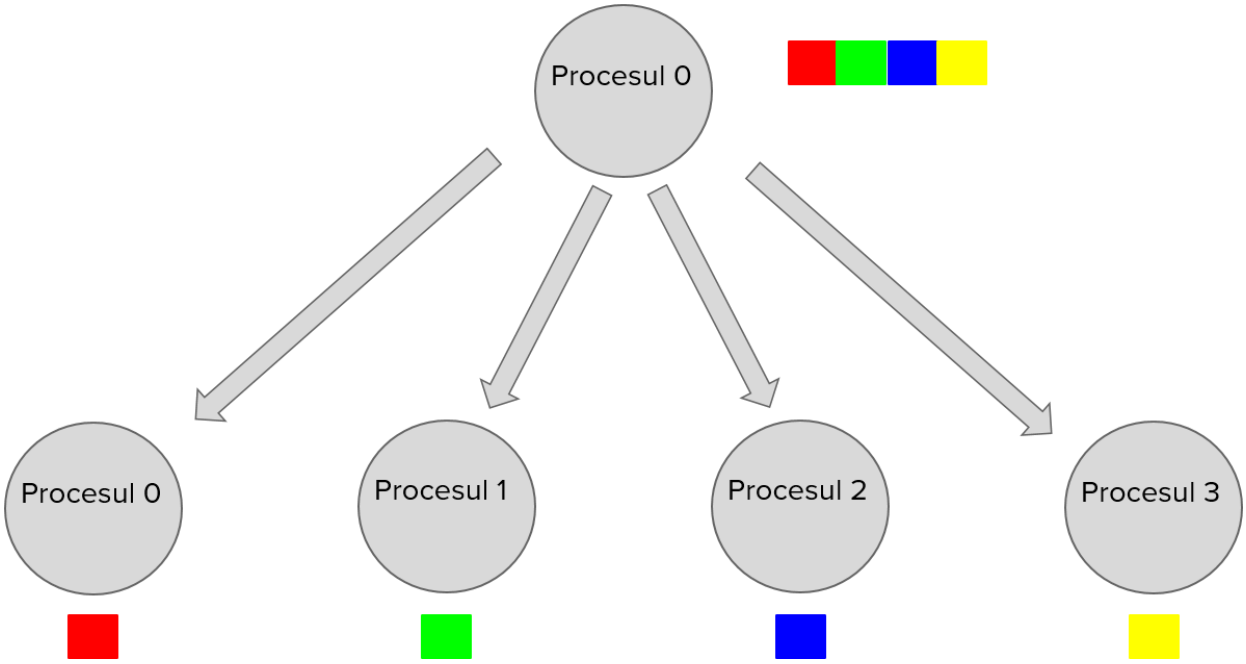
Semnătura funcției este următoarea:

```
int MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void*
```

, unde:

- send_data (↓) - reprezintă datele care sunt împărțite și trimise către procesele din comunicator
- send_count (↓) - reprezintă dimensiunea bucății care revine fiecărui proces (de regulă se pune ca fiind dimensiunea_totală / număr_de_procese).
- send_datatype (↓) - tipul datelor trimise către procese
- recv_data (↑) - reprezintă datele care sunt primite și stocate de către procese
- recv_count (↓) - dimensiunea datelor primite (de regulă dimensiunea_totală / număr_de_procese)
- recv_datatype (↓) - tipul datelor primite de către procese (de regulă este același cu send_datatype)
- root (↓) - identificatorul procesului care împarte datele și care le trimite către procesele din comunicator, inclusiv lui însuși
- communicator (↓) - comunicatorul din care fac parte procesele (de regulă `MPI_COMM_WORLD`)

O ilustrație a modului cum funcționează `MPI_Scatter` :



MPI_Gather

`MPI_Gather` este o funcție care reprezintă inversul lui `MPI_Scatter` , în sensul că un proces primește elemente de la fiecare proces din comunicator, inclusiv de la el însuși, și le unifică într-o singură colecție.

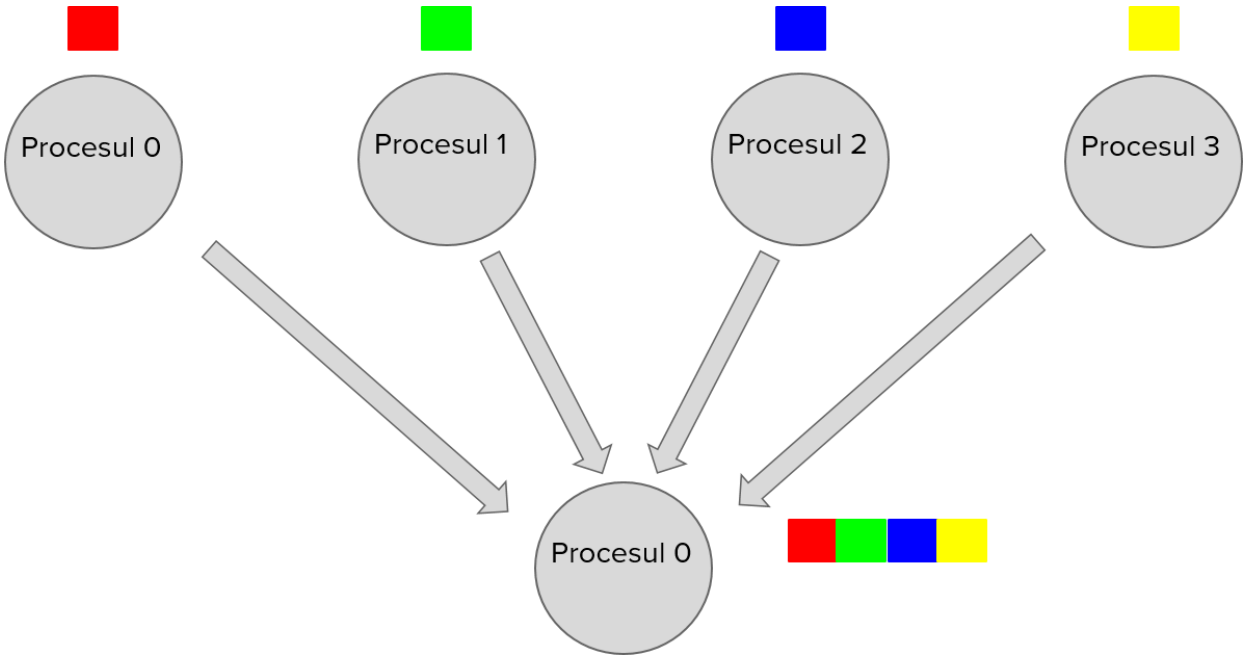
Semnătura funcției este următoarea:

```
int MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* r
```

, unde:

- send_data (↓) - reprezintă datele care trimise de fiecare proces către procesul cu id-ul root
- send_count (↓) - reprezintă dimensiunea bucății trimisă de fiecare proces (de regulă se pune ca fiind dimensiunea_totală / număr_de_procese).
- send_datatype (↓) - tipul datelor trimise de către procese
- recv_data (↑) - reprezintă datele care sunt primite și stocate de către procesul root
- recv_count (↓) - dimensiunea datelor primite (de regulă dimensiunea_totală / număr_de_procese)
- recv_datatype (↓) - tipul datelor primite de către procesul root (de regulă este același cu send_datatype)
- root (↓) - identificatorul procesului care primește datele (inclusiv de la el însuși)
- communicator (↓) - comunicatorul din care fac parte procesele (de regulă `MPI_COMM_WORLD`)

O ilustrare a modului cum funcționează `MPI_Gather` :



Mai jos aveți un exemplu de MPI_Scatter folosit împreună cu MPI_Gather:

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

#define ROOT 0
#define CHUNK_SIZE 5 // numarul de elemente per proces

int main (int argc, char **argv) {
    int rank, proc, a;

    int* arr;
    int* process_arr;
    int* result_arr;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc);

    if (rank == ROOT) {
        arr = malloc (CHUNK_SIZE * proc * sizeof(int));
        for (int i = 0; i < proc * CHUNK_SIZE; ++i) {
            arr[i] = 0;
        }
    }

    process_arr = malloc (CHUNK_SIZE * sizeof(int));
    MPI_Scatter(arr, CHUNK_SIZE, MPI_INT, process_arr, CHUNK_SIZE, MPI_INT, ROOT, M

    for (int i = 0; i < CHUNK_SIZE; i++) {
        printf("Before: rank [%d] - value = %d\n", rank, process_arr[i]);
```

```
        process_arr[i] = i;
        printf("After: rank [%d] - value = %d\n", rank, process_arr[i]);
    }

    if (rank == ROOT) {
        result_arr = malloc (CHUNK_SIZE * proc * sizeof(int));
    }

    MPI_Gather(process_arr, CHUNK_SIZE, MPI_INT, result_arr, CHUNK_SIZE, MPI_INT, R

    if (rank == ROOT) {
        for (int i = 0; i < CHUNK_SIZE * proc; i++) {
            printf("%d ", result_arr[i]);
        }
        printf("\n");
    }

    if (rank == ROOT) {
        free(arr);
        free(result_arr);
    }

    free(process_arr);

    MPI_Finalize();
    return 0;
}
```

Alte funcții

- Funcții nonblocante: `MPI_Irecv`, `MPI_Isend`, `MPI_Ibcast`, `MPI_Igather`, `MPI_Iscatter` etc.
- Funcții sincrone: `MPI_Ssend`, `MPI_Issend`
- `MPI_Bsend` - send cu buffer
- `MPI_Barrier` - barieră
- `MPI_Reduce` - operație distribuită de reduce pe arrays

Exerciții

1) (1 punct) Rulați exemplele de cod din folder-ul de demo din cadrul laboratorului.

2) (3 puncte) Scrieți un program ce adună un vector de elemente folosind MPI (pentru ușurință considerați că numărul de elemente e divizibil cu numărul de procese), folosind doar `MPI_Send` și `MPI_Recv`. Fiecare proces va calcula o suma intermediară, procesul master fiind cel care va calcula suma finală.

3) (3 puncte) Extindeți programul de calcul al sumei unui vector prin adăugarea unui coeficient la suma finală, fiecare proces va calcula suma parțială * coeficient.

```
sum = sum * coeficient
```

Folosiți `MPI_Bcast` pentru a propaga valoarea coeficientului introdus de la tastatură.

4) (3 puncte) Modificați programul de calcul al sumei unui vector astfel încât să folosiți `MPI_Scatter` și `MPI_Gather` pentru transferul informației (vector parțial și suma parțială).

Resurse

- **Laborator introductiv MPI - APD**

Laboratorul 5 - pthreads

Laboratorul 5 - pthreads

Despre pthreads

pthreads reprezintă o bibliotecă din C/C++, nativă Linux, prin care se pot implementa programe multithreaded.

Spre deosebire de OpenMP, pthreads este low-level și oferă o mai mare flexibilitate în ceea ce privește sincronizarea thread-urilor și distribuirea task-urilor către thread-uri.

Implementarea unui program paralel în pthreads

Includere și compilare

Pentru a putea folosi pthreads, este necesar să includem în program biblioteca `pthread.h`. De asemenea la compilare este necesar să includem flag-ul `-lpthread`:

```
gcc -o program program.c -lpthread
./program
```

Crearea și terminarea thread-urilor

În pthreads, avem un thread principal, pe care rulează funcția main. Din thread-ul principal se pot crea thread-uri noi, care vor executa task-uri în paralel.

Pentru a crea thread-uri în pthreads, folosim funcția `pthread_create`:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*thread_fu
```

unde:

- thread - thread-ul pe care vrem să-l pornim
- attr - atributele unui thread (NULL - atribute default)
- thread_function - funcția pe care să o execute thread-ul

- arg - parametrul trimis la funcția executată de thread (dacă vrem să trimitem mai mulți parametri, îi împachetăm într-un struct

Exemplu de funcție pe care o execută un thread:

```
void *f(void *arg) {
        // do stuff
        // aici putem să întoarcem un rezultat, dacă este cazul
    pthread_exit(NULL); // termină un thread - mereu apelat la finalul unei funcții
}
```

Pentru terminarea thread-urilor, care vor fi “lipite înapoi” în thread-ul principal, folosim funcția `pthread_join` , care așteaptă terminarea thread-urilor:

```
int pthread_join(pthread_t thread, void **retval);
```

unde:

- thread - thread-ul pe care îl așteptăm să termine
- retval - valoarea de retur a funcției executate de thread (poate fi NULL)

Exemplu de program scris folosind pthreads:

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

#define NUM_THREADS 2

void *f(void *arg)
{
    long id = *(long*) arg;
    printf("Hello World din thread-ul %ld!\n", id);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int r;
    long id;
    void *status;
    long arguments[NUM_THREADS];
```

```
    for (id = 0; id < NUM_THREADS; id++) {
        arguments[id] = id;
        r = pthread_create(&threads[id], NULL, f, (void *) &arguments[id]);

        if (r) {
            printf("Eroare la crearea thread-ului %ld\n", id);
            exit(-1);
        }
    }

    for (id = 0; id < NUM_THREADS; id++) {
        r = pthread_join(threads[id], &status);

        if (r) {
            printf("Eroare la asteptarea thread-ului %ld\n", id);
            exit(-1);
        }
    }

    return 0;
}
```

În caz că dorim să trimitem mai mulți parametri funcției executate de threads, facem un struct, în care incapsulăm datele, și îl trimitem ca parametru al funcției executate de threads.

Exemplu:

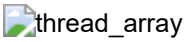
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 8

struct pair {
    int first, second;
};

void *f(void *arg)
{
    struct pair info = *(struct pair*) arg;
    printf("First = %d; second = %d\n", info.first, info.second);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int r;
    long id;
```

```
    void *status;
    struct pair arguments[NUM_THREADS];

    for (id = 0; id < NUM_THREADS; id++) {
        arguments[id].first = id;
        arguments[id].second = id * 2;
        r = pthread_create(&threads[id], NULL, f, (void *) &arguments[id]);

        if (r) {
            printf("Eroare la crearea thread-ului %ld\n", id);
            exit(-1);
        }
    }

    for (id = 0; id < NUM_THREADS; id++) {
        r = pthread_join(threads[id], &status);

        if (r) {
            printf("Eroare la asteptarea thread-ului %ld\n", id);
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

Când dorim să paralelizăm operații efectuate pe arrays, fiecărui thread îi va reveni o bucată din array, pe acea bucată executând funcția atribuită lui (thread-ului).

Formula de împărțire:

- `start_index = id * (double) n / p`
- `end_index = min(n, (id + 1) * (double) n / p))` , unde id = id-ul thread-ului, n = dimensiunea array-ului, p = numărul de threads


thread_array

Elemente de sincronizare

Mutex

Un mutex (mutual exclusion) este folosit pentru a delimita și pentru a proteja o zonă critică, unde au loc, de regulă, operații de citire și de scriere. Un singur thread intră în zona critică (o rezervă pentru el - lock), unde se execută instrucțiuni, iar celelalte thread-uri așteaptă ca thread-ul curent să termine de executat instrucțiunile din zona critică. După ce thread-ul curent termină de executat instrucțiuni în zona critică, aceasta o eliberează (unlock) și următorul thread urmează aceiași pași.

Pentru zonele critice în pthreads folosim `pthread_mutex_t` , care reprezintă un mutex, care asigură faptul că un singur thread accesează zona critică la un moment dat, thread-ul deținând lock-ul pe zona critică în

momentul respectiv, și că celelalte thread-uri care nu au intrat încă în zona critică așteaptă eliberarea lock-ului de către thread-ul aflat în zona critică în acel moment.

Funcții pentru mutex:

- crearea unui mutex: `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- lock pe mutex: `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- unlock pe mutex: `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- distrugerea unui mutex: `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

Exemplu de folosire:

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

#define NUM_THREADS 2

int a = 0;
pthread_mutex_t mutex;

void *f(void *arg)
{
    // facem lock pe mutex
        pthread_mutex_lock(&mutex);
    // zona critica
        a += 2;
    // facem unlock pe mutex
        pthread_mutex_unlock(&mutex);

        pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
        int i;
        void *status;
        pthread_t threads[NUM_THREADS];
        int arguments[NUM_THREADS];

    // cream mutexul
        pthread_mutex_init(&mutex, NULL);

        for (i = 0; i < NUM_THREADS; i++) {
                arguments[i] = i;
                pthread_create(&threads[i], NULL, f, &arguments[i]);
        }

        for (i = 0; i < NUM_THREADS; i++) {
                pthread_join(threads[i], &status);
        }
```

```
    // distrugem mutex-ul
        pthread_mutex_destroy(&mutex);

        printf("a = %d\n", a);

        return 0;
}
```

Barieră

Bariera este folosită atunci când dorim să sincronizăm thread-urile încât să ajungă (să se sincronizeze) în același punct. Mai concret, ea asigură faptul că niciun thread, gestionat de barieră, nu trece mai departe de zona în care aceasta este amplasată decât atunci când toate thread-urile gestionate de barieră ajung în aceeași zonă. În pthreads folosim structura `pthread_barrier_t` pentru barieră.

Funcții:

- crearea unei bariere: `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count);`
- așteptarea thread-urilor la barieră: `int pthread_barrier_wait(pthread_barrier_t *barrier);`
- distrugerea unei bariere: `int pthread_barrier_destroy(pthread_barrier_t *barrier);`

Exemplu de folosire:

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

#define NUM_THREADS 8

pthread_barrier_t barrier;

void *f(void *arg)
{
    int index = *(int *) arg;

    printf("Before barrier - thread %d\n", index);
    pthread_barrier_wait(&barrier);
    printf("After barrier - thread %d\n", index);

        pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
        int i;
        void *status;
        pthread_t threads[NUM_THREADS];
        int arguments[NUM_THREADS];
```

```
        pthread_barrier_init(&barrier, NULL, NUM_THREADS);

        for (i = 0; i < NUM_THREADS; i++) {
                arguments[i] = i;
                pthread_create(&threads[i], NULL, f, &arguments[i]);
        }

        for (i = 0; i < NUM_THREADS; i++) {
                pthread_join(threads[i], &status);
        }
        pthread_barrier_destroy(&barrier);
        return 0;
}
```

Semafor

Un semafor este un element de sincronizare, care reprezintă o generalizare a mutex-ului. Semaforul are un contor care este incrementat la intrarea unui thread în zona de cod critică și care e decrementat când thread-ul respectiv iese din zona critică (contorul nu poate fi negativ în pthreads).

Pentru a folosi semafoare în pthreads ne folosim de structura `sem_t`, pentru care trebuie să includem biblioteca `semaphore.h`.

Semafoarele POSIX sunt de două tipuri:

- cu nume - sincronizare între procese diferite
- fără nume - sincronizare între thread-urile din cadrul aceluiași proces

Funcții:

- `int sem_init(sem_t *sem, int pshared, unsigned int value);` - inițiere semafor
- `int sem_destroy(sem_t *sem);` - distrugere semafor
- `int sem_post(sem_t *sem);` - acquire
- `int sem_wait(sem_t *sem);` - release

Exemplu folosire - producer - consumer:

```
#define _REENTRANT    1

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

#include <pthread.h>
#include <semaphore.h>

#define NUM_THREADS    50
#define CONSUMER        0
#define PRODUCER        1
```

```
#define BUF_LEN         3

pthread_mutex_t mutex;
sem_t full_sem;     // semafor contor al elementelor pline
sem_t empty_sem;    // semafor contor al elementelor goale

char buffer[BUF_LEN];
int buf_cnt = 0;

void my_pthread_sleep (int millis) {
    struct timeval timeout;

    timeout.tv_sec = millis / 1000;
    timeout.tv_usec = (millis % 1000) * 1000;

    select (0, NULL, NULL, NULL, &timeout);
}

void *producer_func (void *arg) {
    sem_wait (&empty_sem);

    pthread_mutex_lock (&mutex);

    buffer[buf_cnt] = 'a';
    buf_cnt++;
    printf ("Produs un element.\n");

        pthread_mutex_unlock (&mutex);

    my_pthread_sleep (rand () % 1000);

    sem_post (&full_sem);

    return NULL;
}

void *consumer_func (void *arg) {
    sem_wait (&full_sem);

    pthread_mutex_lock (&mutex);

    buf_cnt--;
    char elem = buffer[buf_cnt];
    printf ("Consumat un element: %c\n", elem);

    pthread_mutex_unlock (&mutex);

    my_pthread_sleep (rand () % 1000);

    sem_post (&empty_sem);

    return NULL;
}
```

```
int main () {
    int i;
    int type;
    pthread_t tid_v[NUM_THREADS];

    pthread_mutex_init (&mutex, NULL);

    sem_init (&full_sem, 0, 0);
    sem_init (&empty_sem, 0, 3);

    srand (time (NULL));
    for (i = 0; i < NUM_THREADS; i++) {
        type = rand () % 2;
        if (type == CONSUMER) {
                pthread_create (&tid_v[i], NULL, consumer_func, NULL);
                } else {
            pthread_create (&tid_v[i], NULL, producer_func, NULL);
        }
        }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join (tid_v[i], NULL);
        }

        pthread_mutex_destroy(&mutex);

    return 0;
}
```

Variabile condiție

Variabilele condiție reprezintă o structură de sincronizare, care au asociat un mutex, și ele au un sistem de notificare a thread-urilor, astfel încât un thread să fie blocat până când apare o notificare de la alt thread. Pentru a putea folosi variabile condiție în pthreads ne folosim de structura `pthread_cond_t`.

Variabilele condiție sunt folosite pentru a bloca thread-ul curent (mutexul și semaforul blochează celelalte thread-uri). Acestea permit unui thread să se blocheze până când o condiție devine adevărată, moment când condiția este semnalată de thread că a devenit adevărată și thread-ul / thread-urile blocate de condiție își reiau activitatea o variabilă condiție va avea mereu un mutex pentru a avea race condition, care apare când un thread 0 se pregătește să aștepte la variabila condiție și un thread 1 semnalează condiția înainte ca thread-ul 0 să se blocheze

Funcții:

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);` - inițializare variabilă condiție
- `int pthread_cond_destroy(pthread_cond_t *cond);` - distrugere variabilă condiție
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;` - inițializare statică a unei variabile condiție (atribute default, nu e nevoie de distrugere / eliberare)

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);` - blocarea unui thread care așteaptă după o variabilă condiție
- `int pthread_cond_signal(pthread_cond_t *cond);` - deblocarea unui thread
- `int pthread_cond_broadcast(pthread_cond_t *cond);` - deblocarea tuturor thread-urilor blocate

Exemplu folosire - producer - consumer:

```
#define _REENTRANT    1

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

#include <pthread.h>

#define NUM_THREADS    50
#define CONSUMER        0
#define PRODUCER        1

#define BUF_LEN         3

pthread_mutex_t mutex; // folosit pentru incrementarea si decrementarea marimii buf
pthread_cond_t full_cond; // cand buffer-ul este gol
pthread_cond_t empty_cond; // cand buffer-ul este plin

char buffer[BUF_LEN];
int buf_cnt = 0;

void my_pthread_sleep(int millis) {
        struct timeval timeout;

        timeout.tv_sec = millis / 1000;
        timeout.tv_usec = (millis % 1000) * 1000;

        select (0, NULL, NULL, NULL, &timeout);
}

void *producer_func (void *arg) {
        pthread_mutex_lock (&mutex);

        // cat timp buffer-ul este plin, producatorul asteapta
        while (buf_cnt == BUF_LEN) {
                pthread_cond_wait (&full_cond, &mutex);
        }

        buffer[buf_cnt] = 'a';
        buf_cnt++;
        printf ("Produs un element.\n");

        pthread_cond_signal (&empty_cond);
        my_pthread_sleep (rand () % 1000);

        pthread_mutex_unlock (&mutex);
```

```
        return NULL;
}

void *consumer_func (void *arg) {
        pthread_mutex_lock (&mutex);

        // cat timp buffer-ul este gol, consumatorul asteapta
        while (buf_cnt == 0) {
                pthread_cond_wait (&empty_cond, &mutex);
        }

        buf_cnt--;
        char elem = buffer[buf_cnt];
        printf ("Consumat un element: %c\n", elem);

        pthread_cond_signal (&full_cond);
        my_pthread_sleep (rand () % 1000);

        pthread_mutex_unlock (&mutex);

        return NULL;
}

int main() {
        int i;
        int type;
        pthread_t tid_v[NUM_THREADS];

        pthread_mutex_init (&mutex, NULL);
        pthread_cond_init (&full_cond, NULL);
        pthread_cond_init (&empty_cond, NULL);

        srand (time (NULL));
        for (i = 0; i < NUM_THREADS; i++) {
                type = rand () % 2;
                if (type == CONSUMER) {
                        pthread_create (&tid_v[i], NULL, consumer_func, NULL);
                } else {
                        pthread_create (&tid_v[i], NULL, producer_func, NULL);
                }
        }

        for (i = 0; i < NUM_THREADS; i++) {
                pthread_join (tid_v[i], NULL);
        }

        pthread_mutex_destroy(&mutex);

        return 0;
}
```

Modele de programare

Boss worker

Modelul boss-worker este o versiune generalizată a modelului producer-consumer, unde avem un thread cu rolul de boss / master, care produce task-uri, și restul thread-urilor au rol de worker, ele având rolul de a executa task-urile produse de thread-ul boss.

Task-urile sunt puse într-o coadă de către thread-ul boss și preluate de către thread-urile worker, care le execută.

În general avem un singur thread boss, dar putem avea mai multe thread-uri de tip boss.

Probleme ce pot apărea:

- performanța când sunt multe task-uri executate prin refolosirea thread-urilor
- resursele limitate pentru execuția thread-urilor
- probleme de sincronizare

Boss worker se poate implementa folosind:

- două variabile condiționale
  - prima folosită de thread-urile worker să aștepte când coada este goală. Dacă nu e goală, workers semnalează thread-urilor boss și iau din coadă
  - a doua folosită de thread-urile boss să aștepte când coada este plină. Dacă nu e plină, boss semnalează thread-urilor worker și pune în coadă.

- un mutex - protejează coada, variabilele condiționale și contoarele
- o coadă - sincronizare când se pun / preiau date de către thread-uri
  - dimensiunea cozii poate să fie:
    - statică - sunt necesare două variabile condiție (una pentru boss, alta pentru workers)
    - dinamică - o necesară o variabilă condiție pentru workers (când coada e goală), dar are un dezavantaj pentru boss, mai precis acesta poate să fie supraîncărcat cu task-uri

- trei contoare:
  - numărul de task-uri în coadă
  - numărul de thread-uri worker în așteptare
  - numărul de thread-uri boss în așteptare

Thread-urile acționează într-o buclă continuă în ceea ce privește crearea și execuția task-urilor și ele nu își termină execuția când coada este goală, astfel trebuie să avem o modalitate de terminare a execuției thread-urilor boss și workers.

Putem face terminarea thread-urilor în două moduri:

- un task special de tip ”exit” pentru thread-urile worker, care se vor opri când primesc acest task (ordinea task-urilor să fie FIFO)
- coada să aibă un flag de exit (ea trebuie să fie goală), setat de către thread-ul boss. Thread-urile worker se opresc când văd că coada este goală și flag-ul de exit setat pe true, iar apoi thread-ul boss se oprește

Work crew

Work crew reprezintă un model de programare paralelă, unde avem un thread principal și N - 1 thread-uri, care sunt controlate de către thread-ul principal, care le creează. Cele N - 1 thread-uri sunt thread-uri de tip worker, care execută task-uri distribuite de către thread-ul principal, pe care, de asemenea, le execută, după ce thread-urile worker au terminat execuția lor.

Probleme de sincronizare - opțional

Producer - consumer

Problema se referă la două thread-uri: producător și consumator. Producătorul inserează date într-un buffer, iar consumatorul extrage date din acel buffer. Buffer-ul are o dimensiune prestabilită, astfel că:

- producătorul nu poate insera date dacă buffer-ul este plin
- consumatorul nu poate extrage date dacă buffer-ul este gol
- producătorul și consumatorul nu pot acționa simultan asupra buffer-ului

O implementare corectă a problemei presupune asigurarea faptului că nu vor exista situații de deadlock, adică situații în care cele două thread-uri așteaptă unul după celălalt, neexitând posibilitatea de a se debloca.

Această problemă se poate rezolva în mai multe moduri (rezolvările sunt mai sus, în cadrul textului laboratorului):

- folosind semafoare
- folosind variabile condiție

Pseudocod varianta cu semafoare:

```
T[] buffer = new T[k];
semaphore gol(k);
semaphore plin(0);
mutex mutex;

producer(int id) {
    T v;
    while (true) {
        v = produce();
                gol.acquire();

                mutex.lock();
                buf.add(v);
                mutex.unlock();

                plin.release();
    }
}

consumer(int id) {
```

```
    T v;
    while (true) {
        plin.acquire();

                mutex.lock();
                v = buf.poll();
                mutex.unlock();

                gol.release();
                consume(v);
    }
}
```

Problema cititorilor și a scriitorilor (Readers - Writers)

Avem o zonă de memorie asupra căreia au loc mai multe acțiuni de citire și de scriere. Această zonă de memorie este partajată de mai multe thread-uri, care sunt de două tipuri: cititori (care execută acțiuni de citire din zona de memorie) și scriitori (care execută acțiuni de scriere în zona de memorie).

În această privință avem niște constrângeri:

- un scriitor poate scrie în zona de memorie doar dacă nu avem cititori care citesc din zona respectivă în același timp și dacă nu avem alt scriitor care scrie în același timp în aceeași zonă de memorie.
- un cititor poate să citească în zona de memorie doar dacă nu există un scriitor care scrie în zona de memorie în același timp, însă putem să avem mai mulți cititori care citesc în paralel în același timp din aceeași zonă de memorie.

Pentru această problemă avem două soluții:

- folosind excludere mutuală, cu prioritate pe cititori
- folosind sincronizare condiționată, cu prioritate pe scriitori

Soluția cu excludere mutuală

Folosind această soluție, un cititor nu va aștepta ca ceilalți cititori să termine de citit zona de memorie, chiar dacă avem un scriitor care așteaptă. Un scriitor poate să aștepte foarte mult, în caz că sunt foarte mulți scriitor, fapt ce poate duce la un fenomen numit writer’s starvation.

De asemenea, nu poate să intre un scriitor cât timp există deja un scriitor care scrie în zona de memorie partajată.

Pseudocod:

```
// numărul de cititori care citesc simultan din resursa comună
int readers = 0;

// mutex (sau semafor) folosit pentru a modifica numărul de cititori
mutex mutexNumberOfReaders; // sau semaphore mutexNumberOfReaders(1);
```

```
// semafor (sau mutex) folosit pentru protejarea resursei comune
semaphore readWrite(1); // sau mutex readWrite

reader (int id) {
    while (true)
        mutexNumberOfReaders.lock();
                readers = readers + 1;
                        // dacă e primul cititor, atunci rezervăm zona de memorie î
                if (readers == 1) {
                        readWrite.acquire();
                };
                mutexNumberOfReaders.unlock();

                        // citește din resursa comună;

                mutexNumberOfReaders.lock();
                readers = readers - 1;
                        // dacă e ultimul cititor, eliberăm zona de de memorie din
                if (readers == 0) {
                        readWrite.release();
                }
                mutexNumberOfReaders.unlock();
    }
}

writer (int id) {
    while (true) {
        // intră scriitorul în resursa comună
        readWrite.acquire();

        // scrie în resursa comună;

        // scriitorul eliberează resursa
        readWrite.release();
    }
}
```

Soluția cu sincronizare condiționată

Folosind această soluție, niciun cititor nu va intra în zona de memorie partajată cât timp există un scriitor care scrie în zona de memorie. De asemenea, nu poate să intre alt scriitor cât timp există un scriitor care se află în zona de memorie partajată.

```
// cititori care citesc din zona de memorie
int readers = 0;
// scriitori care scriu în zona de memorie
// (va fi doar unul, nu pot fi mai mulți scriitori care scriu simultan)
int writers = 0;

int waiting_readers = 0; // cititori care așteaptă să intre în zona de memorie
```

```
int waiting_writers = 0; // scriitori care așteaptă să intre în zona de memorie

// semafor folosit pentru a pune scriitori în așteptare, dacă avem un scriitor
// sau unul sau mai mulți cititori în zona de memorie (zona critică)
semaphore sem_writer(0);

// semafor folosit pentru a pune cititori în așteptare dacă avem un scriitor care s
// sau dacă avem scriitori în așteptare (deoarece ei au prioritate față de cititori
semaphore sem_reader(0);

// semafor folosit pe post de mutex pentru protejarea zonei de memorie (zona critic
semaphore enter(1);

reader (int id) {
    while(true) {
        enter.acquire();

        // dacă avem cel puțin un scriitor care scrie în resursa comună
        // sau dacă avem un scriitor în așteptare, cititorul așteaptă
                if (writers > 0 || waiting_writers > 0) {
                        waiting_readers++;
                        enter.release();
                        sem_reader.acquire();
                }

                        readers++;
                if (waiting_readers > 0) {
                                // a venit încă un cititor în resursa comună,
                                // ieșind din starea de așteptare

                                waiting_readers--;
                        sem_reader.release();
                } else if (waiting_readers == 0) {
                        enter.release();
                }

                // citește din zona partajată
                enter.acquire();
                        readers--;

                        if (readers == 0 && waiting_writers > 0) {
                        waiting_writers--;
                        sem_writer.release();
                } else if (readers > 0 || waiting_writers == 0) {
                        enter.release();
                }
    }
}

writer (int id) {
    while(true) {
        enter.acquire();

                if (readers > 0 || writers > 0) {
```

```
                        waiting_writers++;
                        enter.release();
                        sem_writer.acquire();
                }

                writers++;

                enter.release();

        // scrie în zona partajată

        enter.acquire();

                writers--;
                if (waiting_readers > 0 && waiting_writers == 0) {
                        waiting_readers--;
                        sem_reader.release();
                } else if (waiting_writers > 0) {
                        waiting_writers--;
                        sem_writer.release();
                } else if (waiting_readers == 0 && waiting_writers == 0) {
                        enter.release();
                }
    }
}
```

Problema filosofilor

Problema se referă la mai mulți filozofi (thread-uri) așezați la o masă circulară. Pe masă se află N farfurii și N tacâmuri, astfel încât fiecare filozof are un tacâm în stânga și unul în dreapta lui. În timp ce stau la masă, filozofii pot face două acțiuni: mănâncă sau se gândesc. Pentru a mânca, un filozof are nevoie de două tacâmuri (pe care le poate folosi doar dacă nu sunt luate de către vecinii săi).

Rezolvarea trebuie să aibă în vedere dezvoltarea unui algoritm prin care să nu se ajungă la un deadlock (situația în care fiecare filozof ține câte un tacâm în mână și așteaptă ca vecinul să elibereze celălalt tacâm de care are nevoie).

Ca soluție, avem în felul următor: vom avea N lock-uri (având în vedere că avem N thread-uri), fiecare filosof va folosi câte două lock-uri. Pentru a evita deadlock-ul, totul va funcționa în felul următor:

- fiecare din primele N - 1 thread-uri va face lock mai întâi pe lock pe lock[i], apoi pe lock[i + 1], apoi execută o acțiune, apoi face release pe lock[i], apoi pe lock[i + 1].
- al N-lea thread va face lock mai întâi pe lock[0], apoi pe lock[N - 1] (deci invers față de restul thread-urilor), execută o acțiune, apoi face release pe lock[0], apoi pe lock[N - 1].

Pseudocod:

```
Lock[] locks = new Lock[N];

philosopher(int id) {
        while (true) {
                if (id != N - 1) {
                        locks[id].lock();
                        locks[id + 1].lock();
                        // eat
                        locks[id].release();
                        locks[id + 1].release();
                        // think
                } else {
                        locks[0].lock();
                        locks[N - 1].lock();
                        // eat
                        locks[0].release();
                        locks[N - 1].release();
                        // think
                }
        }
}
```

Problema bărbierului

Avem următoarea situație: avem o frizerie cu un bărbier (un thread), un scaun de bărbier, N scaune de așteptare și M clienți (M thread-uri).

La această problemă avem următoarele constrângeri:

- bărbierul doarme atunci când nu sunt clienți
- când vine un client, acesta fie trezește bărbierul, fie așteaptă dacă bărbierul este ocupat
- dacă toate scaunele sunt ocupate, clientul pleacă

```
int freeChairs = N;
semaphore clients(0);
semaphore barber_ready(0);
semaphore chairs(1); // sau mutex

barber() {
    while(true) {
        clients.acquire(); // se caută client; dacă există, el este chemat

                chairs.acquire(); // are client, un scaun este eliberat, modificăm

                freeChairs++; // scaun eliberat

                barber_ready.release(); // bărbierul e gata să tundă
                chairs.release(); // freeChairs modificat
```

```
                // tunde bărbierul
    }
}

client(int id) {
    while(true) {
        chairs.acquire(); // vine un client și caută un scaun liber
        if (freeChairs > 0) {
                        freeChairs--; // clientul a găsit scaun

                        clients.release(); // bărbierul știe că s-a ocupat un scaun

                        chairs.release(); // freeChairs modificat

                        barber_ready.acquire(); // clientul își așteaptă rândul la
                } else {
                        // nu sunt scaune libere
                        chairs.release();
                        // clientul pleacă netuns
                }
    }
}
```

Exerciții

1) (2.5 puncte) Implementați bariera folosind variabile condiție.

2) (2.5 puncte) Implementați bariera folosind semafoare.

3) (2.5 puncte) Rezolvați boss worker pe baza scheletului.

4) (2.5 puncte) Folosind work crew, implementați însumarea elementelor dintr-o matrice, cu împărțirea matricei pe linii / coloane, fiecare thread cu liniile / coloanele sale, pe baza scheletului.

5) (opțional) Rezolvați problema bărbierului pe baza scheletului.

6) (opțional) Rezolvați problema Readers-Writers pe baza scheletului.

7) (opțional) Rezolvați problema filosofilor pe baza scheletului.