ACADÈMIASOL

estudis universitaris

ENGINYERIES

ADE i ECONOMIA

ARQUITECTURES

CIÈNCIES

BIOCIÈNCIES

CIÈNCIES SOCIALS

Des de 1975 formant universitaris!!

EL NOSTRE OBJECTIU: **EL TEU APROVAT**

www.academiasol.com - www.facebook.com/academiasol

ENGINYERIA INFORMÀTICA - PAR

Pack teoria Final

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

PARAL·LELISME

TEMA 1

Paral·lelisme vs Concurrència

Definicions:

- Parallel Computing: Redueix l'execució d'un programa en 1/P, on P és el número de processadors.
- Throughput Computing: Maximitza el número de programes executats per unitat de temps. N/P, on N és el número de programes i P és el número de processadors.
- Concurrència: És quan s'executen varis fils d'execució (threads) en un sol processador (no és simultani).
- Paral·lelisme: És l'ús simultani de múltiples processadors per executar un programa. El programa es divideix en parts discretes per ser resoltes concurrentment.

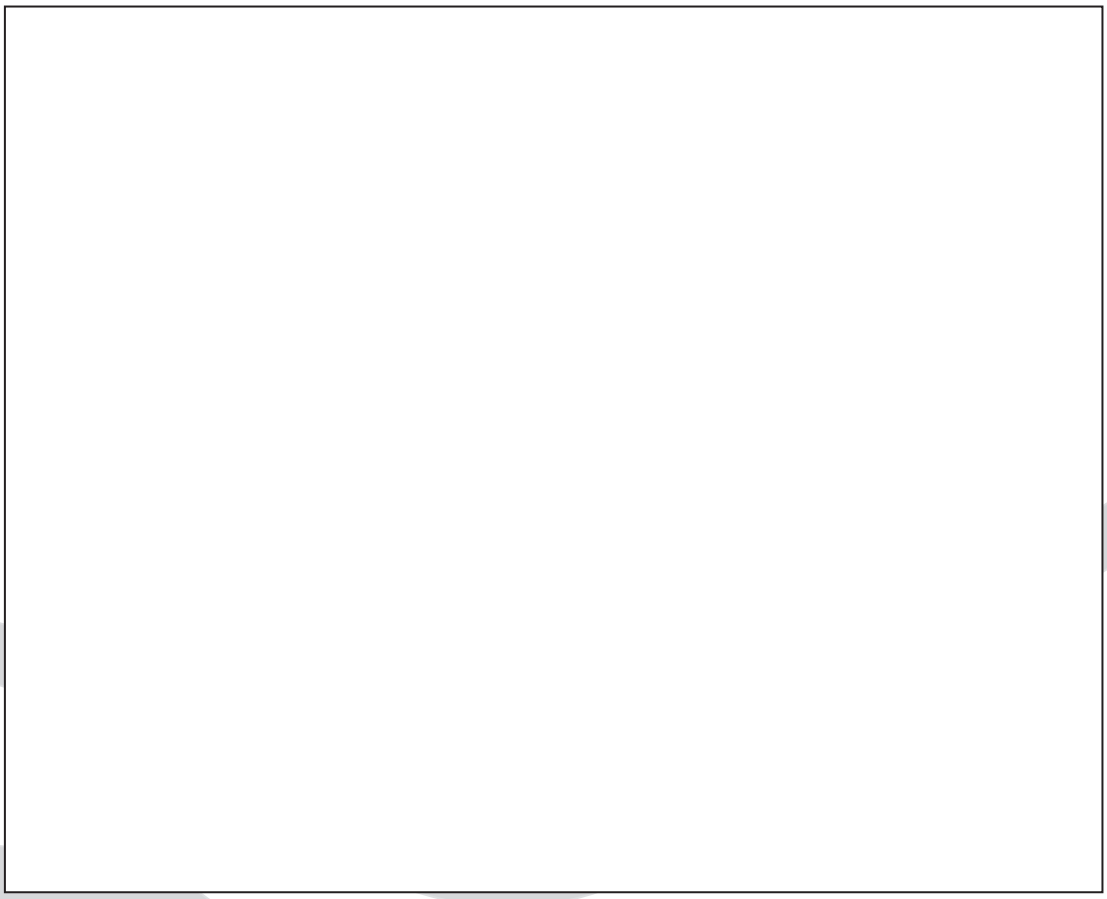
Problemes quan paral·lelitzes:

- Data Race, *Condició de carrera*:
  - Explicació: Depenent de l'ordre en que s'executin les instruccions el resultat pot variar.
  - Exemple: Dos usuaris modificant una mateixa variable.
  - Solució: Exclusió mútua.
- Starvation, *Inanició*:
  - Explicació: Es quan un procés es queda bloquejat indefinidament a l'espera que s'alliberi un recurs, es desbloqui un semàfor, etc.
  - Exemple: Una persona entra com a usuari a un servei d'una entitat financera (la qual només deixa que estigui loguejat un usuari per compte en un mateix instant de temps) i una altra persona vol entra al mateix servei amb el mateix usuari, doncs haurà d'esperar indefinidament fins que l'altre persona acabi.
  - Solució: -
- Dead Lock, *Interbloqueig* o *Abraçada mortal*:
  - Explicació: És una situació on dues o més accions s'esperen mútuament, incapaces de seguir fins que les altres no acabin, i, per tant, cap d'elles no acaba mai.
  - Exemple:

	Procés 1	Procés 2
1	lock(sem1)	
2		lock(sem2)
3		lock(sem1)
4	lock(sem2)	

- Solució: Fer un ordre en el bloqueig:

Exemple de solució:



- LiveLock, *Bloqueig actiu*.
  - Explicació: es un deadlock però canviant contínuament d'estat.
  - Exemple: Dos persones es troben en un passadís en direcció oposada i al voler ser amables es queden bloquejats per moure's cap al mateix costat tota l'estona.
  - Solució: Que el més prioritari ha de passar primer o usar un algoritme aleatori.

Punts a tenir en compte que suma temps del paral·lelisme:

- El balanç de feina és equitatiu?
- Hi ha dependencies entre els processos?
- Hi ha espera per comunicació?
- S'ha de sumar el temps de creació i comunicació entre processos.

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

TEMA 2

Tipus de descomposició del còmput:

- Descomposició per dades (dividint un vector o una matriu)
- Descomposició per tasques (funcions o iteracions de bucle)

Definicions i fórmules:

- **T_1** → Suma total del temps de tota l'aplicació si es fes de forma seqüencial.
- **T_∞** → Camí crític de l'execució, és a dir, si es paral·lelitzar l'aplicació quin serà el temps del camí més llarg.
- **Parallel = T_1/T_∞** → Aquesta fórmula ens dóna el paral·lelisme potencial de l'aplicació (és independent del número de processos).
- **Parallel Slackness = $(T_1/T_\infty)/P_{min}$** → Pmin és el número mínim de processos necessaris per aconseguir el paral·lelisme.
- **$T_p \geq T_1/P$ and $T_p \geq T_\infty$** → És el temps que triga en executar tot el codi amb P processos.
- **$S_p = T_1/T_p$ (Speedup)** → És la relativa reducció de l'execució quan utilitzes P processos respecte l'execució seqüencial.
- **$Eff_p = T1/(Tp*P) = Sp/P$ (Eficiència)** → És una mesura de la fracció de temps durant el qual un element de l'aplicació s'utilitza útilment.
- **Llei d'Amdahl:** La millora de rendiment d'un programa paral·lelitzat esta donat per la fracció de temps que aquest programa treballa en paral·lel.

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

Strong vs Weak Scaling

- Strong Scaling: es quan amb l'augment del número de processadors repercuteix en una reducció de temps de l'execució, per a una dimensió de problema constant.
- Weak Scaling: es quan amb l'augment del número de processadors repercuteix en un augment de la mida del problema, mantenint el temps per processador constant.

Mètrica del mig-rendiment

- $N_{1/2}$ = mida de l'entrada que aconsegueix una $Eff_p = 0.5$, per una p donada.
- Per una $N_{1/2}$ més gran el problema és més difícil de paral·lelitzar eficientment.

Exercici

Donat el diagrama temporal i el graf dirigit següent calcula:

T_1 =

T_∞ =

Parallel =

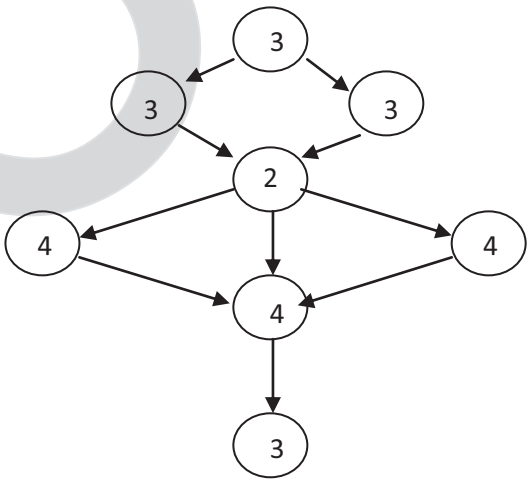
T_p =

Sp_3 =

Eff_p =

Sp_ideal o Sp_∞ =

CPU 1	3	3	2	4	3
CPU 2		3		4	
CPU 3				4	



Compartició de dades

Seguirem el següent model simple per calcular el temps de transmissió de dades entre processos:
- Inicialització (Start up): temps gastat (ts) en preparar l'accés a les dades remotes.
- Accés remot (Remote access): temps gastat en accedir i transferir dades. Número de bytes (m) pel temps de transferir cada byte (tw).

$$T_{acces} = T_s + m \cdot T_w$$

Estratègies de descomposició de dades d'una matriu a calcular de mida NxN

Per files o columnes:
- Cada processador processa segments de N*N/P elements de la matriu.
- Útil quan hi ha dependències en el càlcul sobre un eix:
    - Files: quan les dependències són horitzontals.
    - Columnes: quan les dependències són verticals.

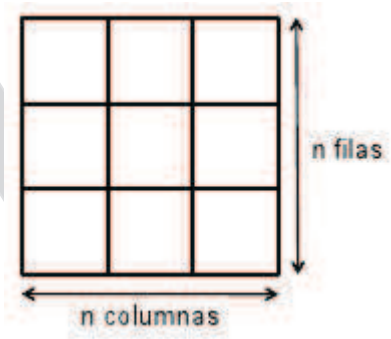
Per blocs:
- Cada processador processa segments de N*N/P elements de la matriu.
- Cada segment esta dividit en blocs de B columnes.
- Útil quan hi ha dependències de càlcul sobre els dos eixos.

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

Exercici:

Escriu el Tp de recorre una Matriu M de mida NxN que actualitza una altre Matriu B amb la mateixa mida seguint el codi següent amb P processadors:

```
for ( int i = 0; i < N; ++i ) {
        for ( int j = 0; j < N; ++j ) {
                int  aux = M[ i ][ j ]*100 + M[ i-1 ][ j ];
                B[ i ][ j ] = aux;
        }
}
```



//Suposem que el cost de dintre del bucle es tc

Hi ha dependències?

Com dividiries la taula per files, per columnes o per blocs?

Stencil algorithm

Es un algoritme que actualitza cada element de la matriu utilitzant els quatre veïns directes horitzontals i verticals (és a dir, amb l'element de la dreta, esquerra, a dalt i el de baix).

Suposant aquest algoritme:

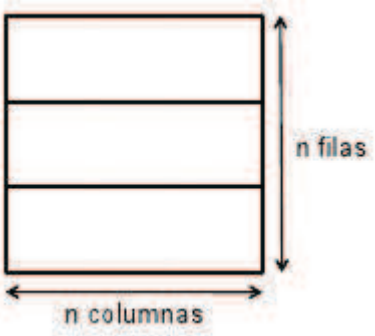
```
for ( int i = 0; i < N; ++i ) {
        for ( int j = 0; j < N; ++j ) {
                int aux = M[ i+1 ][ j ] + M[ i-1 ][ j ] + M[ i ][ j - 1] + M[ i ][ j + 1 ];
                M[ i ][ j ] = aux;
        }
}
```

Tenim que comprovar les dependències que té cada element abans de ser calculat!

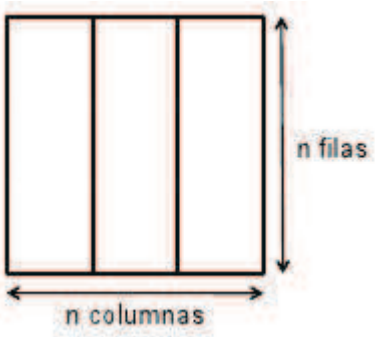
Si tenim dependències tan verticals com horitzontals:

- Si dividim per files la dependència vertical ens retardarà.
- Si dividim per columnes la dependència horitzontal ens retardarà.
- SOLUCIÓ: Dividir per blocs.

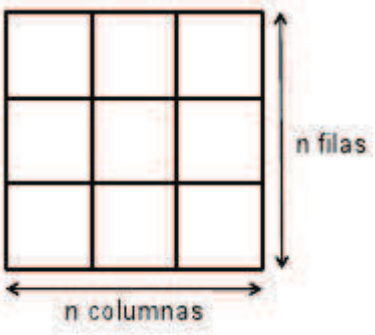
FILES:



COLUMNES:



BLOCS:

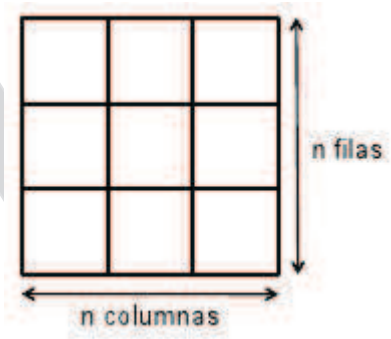


Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

Exercici:

Escriu el Tp de recorre una Matriu M de mida NxN seguint el codi següent amb P processadors:

```
for ( int i = 0; i < N; ++i ) {
        for ( int j = 0; j < N; ++j ) {
                int aux = M[ i ][ j ]*100 + M[ i-1 ][ j ] - M[ i ][ j - 1];
                M[ i ][ j ] = aux;
        }
}
```



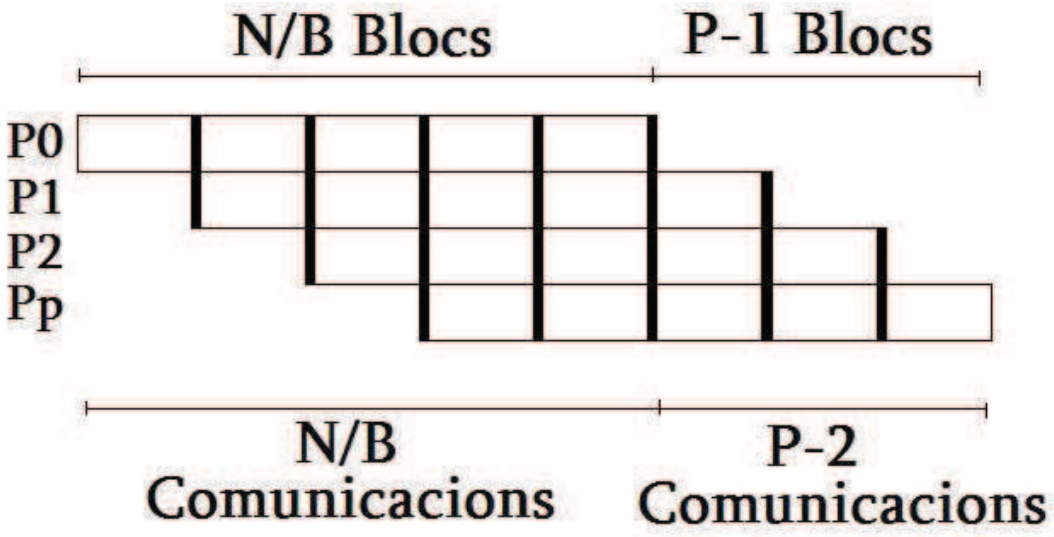
//Suposem que el cost de dintre del bucle es tc

Hi ha dependències?

Com dividiries la taula per files, per columnes o per blocs?

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

BASIC STENCIL BLOCK
RECORDAR!!!



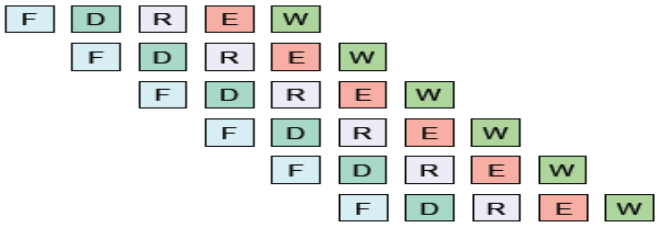
TEMA 3

Introducció a arquitectures paral·leles

Historia dels processadors:

- **Pipeling:**

L'execució d'una instrucció es subdivideix en diferents estadis.



*Extret d'apunts de Paral·lelisme FIB (UPC) diapositiva 3 Tema 3

En veritat no es tant perfecta, perquè hi ha esperes, i fa que algunes etapes hagin d'esperar a la finalització de l'anterior.

- **ILP** (Instruction-Level Parallelism):

Uniprocessadors superescalars, que permeten que permeten més múltiples instruccions per cicle.

-**TLP** (Thread-Level Parallelism):

Uniprocessadors multi-threads, que poden executar varis fils d'execució alhora.
S'aprofita més la unitat funcional
- Problemes: Augmenta els Miss de caché

-**DLP** (Data-Level Parallelism):

Arquitectura SIMD: una instrucció simple s'executa sobre múltiples dades (sumes cada component d'un vector per exemple).

2	4	3	7	3

\+ + + + +

1	1	1	1	1

La suma dels dos vectors es fa en una sola instrucció i en paral·lel!

3	4	4	8	4

Jerarquia de Memòria

- Introducció a la memòria cache als processadors.
  - Avantatges de l'ús de la jerarquia de memòria:
  - Principi de Localitat.
  - Principi Temporal.

- Sempre ens emportarem de memòria principal o cache una *línia* o *bloc.*

- Els accessos en un nivell de la jerarquia memòria:
  - Hit, la dada apareix en una de les línees d'aquest nivell.
  - Miss, la dada no apareix en cap línea i s'ha d'anar a buscar al següent nivell.
  - Temps mig d'accés: $T = t_h \cdot h + t_m \cdot m$

- Polítiques:
  - Mapping.
  - Substitueix: quan un nivell esta ple (segons algoritme Random, FIFO, etc).
  - Escriptura: Quan s'actualitza una dada en una escriptura? (quan és hit: write through/copy back, on miss: allocate/no allocate).

Shared-Memory Multiprocessor (SMP)

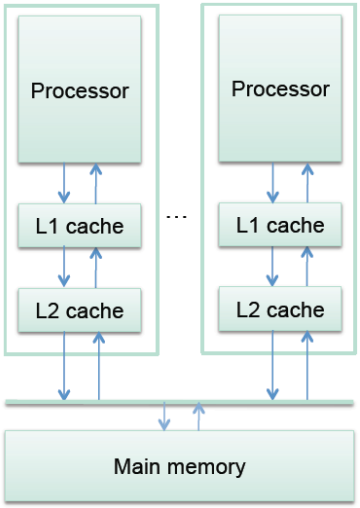
- Dos o més processadors amb una mateixa memòria, els dos estan a la mateixa distància de memòria, per tant, triguen el mateix en accedir.

↓

- Problema: Masses accessos a memòria (a la vegada).
- Solució: L'ús de memòries cache o memòria multibank.

↓

- Problema: No hi ha coherència ni consistència amb les dades.
- Solució: Snooping-based.



*Extret d'apunts de Paral·lelisme FIB (UPC) diapositiva 17 Tema 3

Snoop-Based

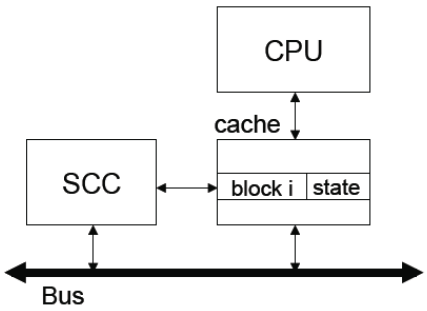
Les transaccions dintre del bus que comunica amb la memòria és visible per tots els processadors.
Els processadors poden tafanejar (snoop) sobre el que passa en el bus i prendre partit quan veuen un esdeveniment.

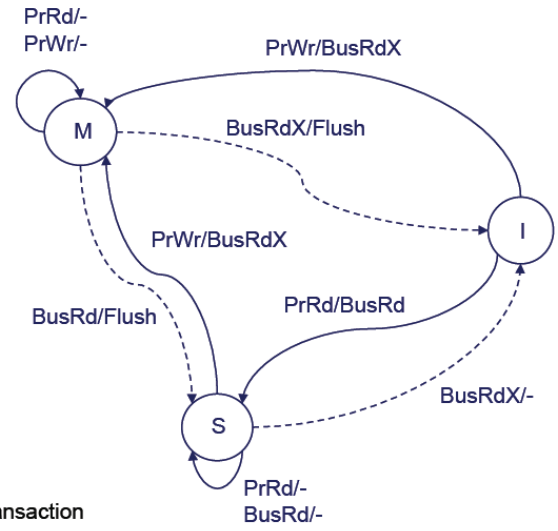
Intentar minimitzar l'intercanvi (Sharing)

- True Sharing: Si una dada és sobreescrita freqüentment és possible que causi un coll d'ampolla a l'hora d'accedirla.
- False Sharing: Es quan varis processos intenten escriure en variables diferents (no hi ha problema de dependència) però aquestes variables resideixen en una mateixa línea de cache. Això activa el mecanisme de coherència quan no és necessari.
- Coherència: L'escriptura a una localització concreta ha de ser en ordre, és a dir, si dos CPUs en un SMP treballen sobre la mateixa dada, quin és el valor que han de veure les dues CPUs de la mateixa dada? El mateix!
- Consistència: L'escriptura a diferents localitzacions es faran en l'ordre que tingui sentit, segons el codi font i les regles de consistència.

Graf d'estats amb Snoop-Based

- Estats:
  - Dirty or Modified (M)
  - Shared (S)
  - Invalidate (I)
- CPU Events:
  - PrRd (Lectura d'un procés)
  - PRWr (Escriptura d'un procés)
- Transaccions dintre del bus:
  - BusRd: pregunta per copiar sense intent de modificar.
  - BusRdX: pregunta per copiar amb intent de modificar (Invalida)
  - Flush: Actualitza a memòria





Notation: Command (form processor or bus) / Bus transaction

*Extret d'apunts de Paral·lelisme FIB (UPC) diapositiva 23 Tema 3

Exercici:

Suposa 3 processadors en Snoop-Based. Escriuen i llegeixen el mateix valor de memòria.

- r1 significa que el processador 1 llegeix.
- w3 significa que el processador 3 llegeix

L'accès a memòria és el següent:

r1, r2, w3, r2, w1,w2,r3,r2,r1

Dieu les transaccions del bus i els estats del processador sobre aquesta dada:

	CPU1		CPU2		CPU3	
R1						
R2						
W3						
R2						
W1						
W2						
R3						
R2						
R1						

TEMA 4/5

Introducció

La segona part de l'assignatura de PAR aplicarem el que s'ha aprés en els primers temes per tal de dissenyar correctament algoritmes paral·lels.

Fem un recordatori dels problemes que tenen els algoritmes paral·lelitzem:

- Data Race, condició de carrera.
- Starvation, inanició.
- Dead Lock, abraçada mortal.
- Live Lock, bloqueig actiu.
- Balança de feina equitativa.
- Dependències entre tasques.
- Esperes per comunicació.
- False i True Sharing
- Etc.

Objectius

Sobre una especificació que resol un problema:

- Identificar tasques (parts de feina que es poden executar concurrentment).
- Descompondre les dades del problema (l'entrada, la sortida o les dues).
- Trobar les dependencies entre tasques.

Les dos formes que veurem per descompondre la feina:

- En tasques: Dividim la feina total computacional en petits fragments de feina.
- En dades: Semblant a les tasques però aplicat a les dades del problema.

Paral·lelitzar amb memòria compartida: OpenMP

- OpenMP és una API per la programació multiprocés amb memòria compartida. Es basa en el model **fork-join** (UNIX) on una feina gran es divideix en fils o threads (fork) amb un pes menor, per després, recollir els seus resultats i reduir-los en un sol (join).
- Treballarem OpenMP en C/C++.
- NOTA: la memòria és compartida, és a dir, sinó privatitzem les variables globals (compartides entre els threads) tots podran accedir i modificar aquestes variables amb els clars problemes de condició de carrera.

Sintaxis bàsica d'OpenMP:

pragma omp <directiva> [clàusula [, ...] ...]

Directives

- **parallel**: Aquesta directiva indica la part del codi que s'executarà amb varis fils.
  - # pragma omp parallel
- **for:** Igual que el parallel però aplicat als bucles for.
  - # pragma omp parallel for
- **single:** La part de codi que defineix aquesta directiva l'executarà un únic thread, no necessàriament el thread pare (malgrat estiguem dintre d'una estructura on s'executen varis threads)
  - # pragma omp single
- **critical:** En la secció de codi definida pel critical només un thread pot estar executant alhora, és a dir, és una forma d'exclusió.
  - # pragma omp critical
- **atomic:** En la secció de codi només ha de ser executada només per un thread (com el critical) però només es modifica una posició de memòria. Per exemple: *++cont;* o *x = x + y%10;*.
  - # pragma omp atomic
- **flush:** Exporta el valor que hagin modificat un altre thread d'una variable .compartida.
  - # pragma omp flush
- **barrier:** Els fils d'execució no poden passar d'aquest punt fins que tots els threads hagin arribat.

- # pragma omp barrier
  Nota: el *parallel* i el *for* tenen un barrier implícit al final.

- **task**: Llança una tasca a fer, és a dir, el segment de codi ho farà el primer thread que estigui disponible. Ens els tasks les variables per defecte són firstprivate.
  - # pragma omp task

- **taskwait:** Espera que acabin totes les tasques llançades (semblant al barrier).
  - # pragma omp taskwait

Funcions

Estableix el número de threads que es crearan en la secció paral·lela:

- void omp_set_num_threads(int NUM_THREADS);

Retorna el número de threads en la regió paral·lela actual:

- int omp_get_num_threads();

Retorna l'identificador del thread actual:

- int omp_get_thread_num();

Retorna el número de processadors de la màquina:

- int omp_get_num_procs();

Retorna un número de segons des de un punt arbitrari.

- double omp_get_wtime();

Clàusules

- num_threads(expression)
  - S'estableix el número de threads a la regió paral·lela indicats a l'expressió.
- if (expression)
  - Si ens interessa condicionar que la regió es faci en paral·lel o no, si l'expressió és avaluada a fals només s'executarà amb un thread.
- shared(var-list)
  - Es declara les variables de var-list compartida per tots els threads, és a dir, tots accedeixen a la mateixa posició de memòria de la variable.
    És necessari sincronització.
    Per defecte, les variables estan implícitament compartides.

- private(expression)
  - Quan en el constructor definim una variable private, es crea una variable nova amb el mateix nom a cada thread inicialment sense valor. No es necessària la sincronització.
- firstprivate(expression)
  - El mateix que private però el valor de la variable esta inicialitzat amb el valor que tenia la variable inicialment.
    No es necessària la sincronització.
- reduction(operator:list)
  - Redueix l'acumulació de tots els threads en una variable.
    Els operadors poden ser: +,-,*,|,||.&,&&,^.
    Es crea una variable privada a cada thread i al final el compilador s'assegura que totes les acumulacions parcials es redueixen en la variable final (segons l'operador indicat).
- schedule(schedule-kind, chunk) -> *va amb la directiva parallel for!*
  - Determina quines iteracions del bucle for són executades per cada thread.
  - static: És la repartició per defecte, es reparteixen les i iteracions entre els num_threads en seccions de N/num_threads iteracions on N és el número de iteracions totals.
  - static, C: Es reparteixen les i iteracions entre els num_threads en seccions de C iteracions.
  - dynamic, C: Es reparteixen C iteracions a cada thread, quan un thread acaba el càlcul agafa C iteracions més i així fins acabar les N iteracions (si no s'especifica la C és 1).
  - guided, C: És semblant al dynamic, però el número de iteracions es va disminuint exponencialment. Si no s'especifica la C és 1.
- nowait
  - Treu el barrier implícit després de fer la directiva parallel (anar amb compte dependències de feina alhora d'usar el nowait!).

Locks

OpenMP també prové funcions per fer exclusió mútua de baix nivell:

- omp_init_lock(&omp_lock_t);
- omp_set_lock(&omp_lock_t);
- omp_unset_lock(&omp_lock_t);
- omp_test_lock(&omp_lock_t);
- omp_destroy_lock(&omp_lock_t);

Fragment de codi paral·lel amb OpenMP:

```
#include “omp.h”
#include <stdio.h>
#include <stdlib.h>

void main( )
{
        ...
        omp_set_num_threads(2);
        #pragma omp parallel
        {
                id = omp_get_thread_num();
                printf(“Sóc el thread número %d\n”,id);
        }
        #pragma omp barrier
        …
}
```

Critical vs Atomic:

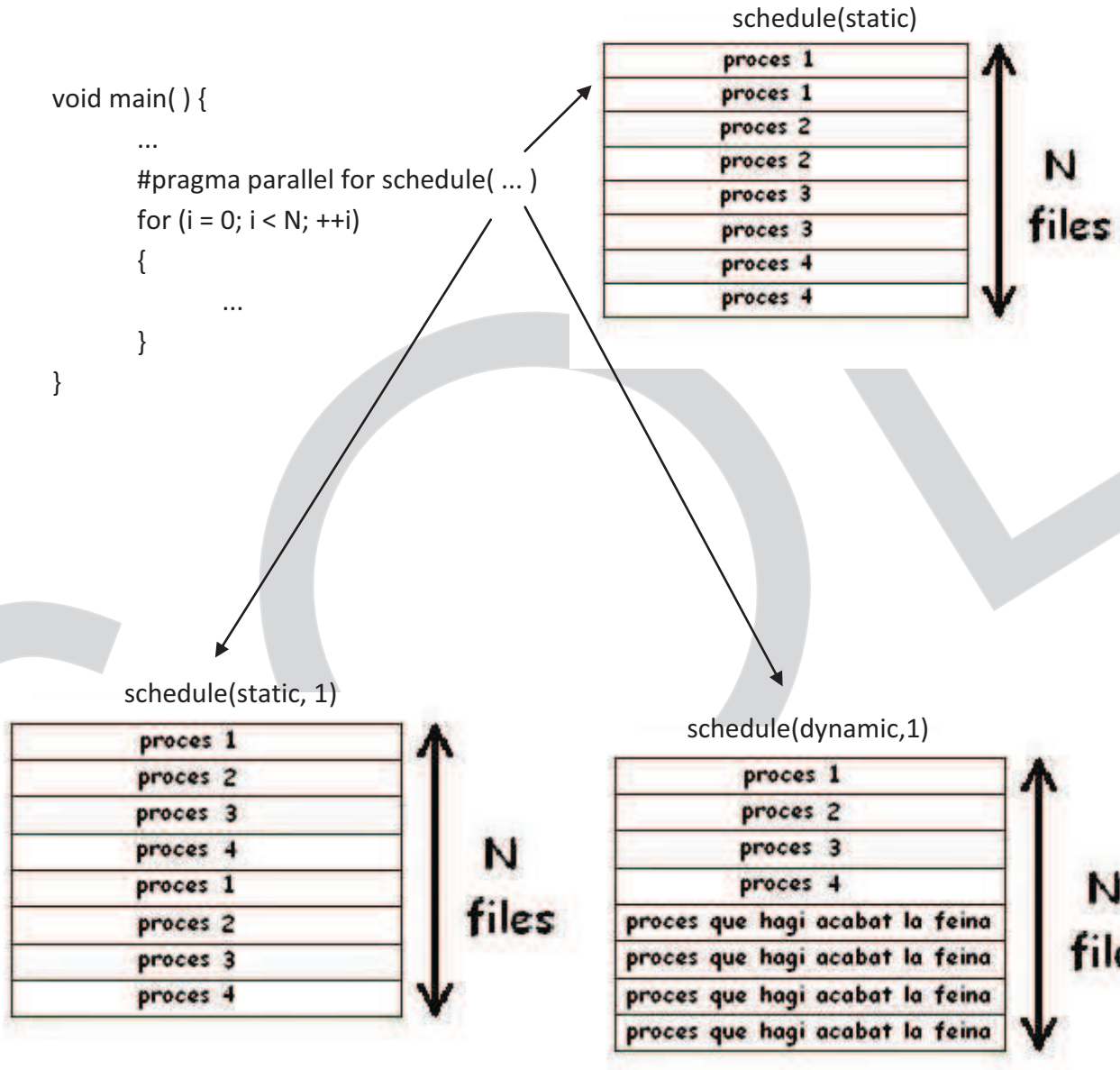
```
void main( )
{
        ...
        #pragma omp parallel
        {
                ...
                #pragma omp critical
                x += altresOperacions( );
                ...
        }
        ...
}
```

```
void main( )
{
        ...
        #pragma omp parallel
        {
                ...
                #pragma omp atomic
                ++x;
                ...
        }
        ...
}
```

Sabino de Arana, 56-58 bxs. · BCN · Tel. 93 490 82 42
Trias i Giró, 15-17 · BCN · Tel. 93 203 34 59
Cartoixa de Vallparadís, 14 · TERRASSA · Tel. 93 784 53 23
www.academiasol.com · www.facebook.com/academiasol

Fragment amb parallel for:

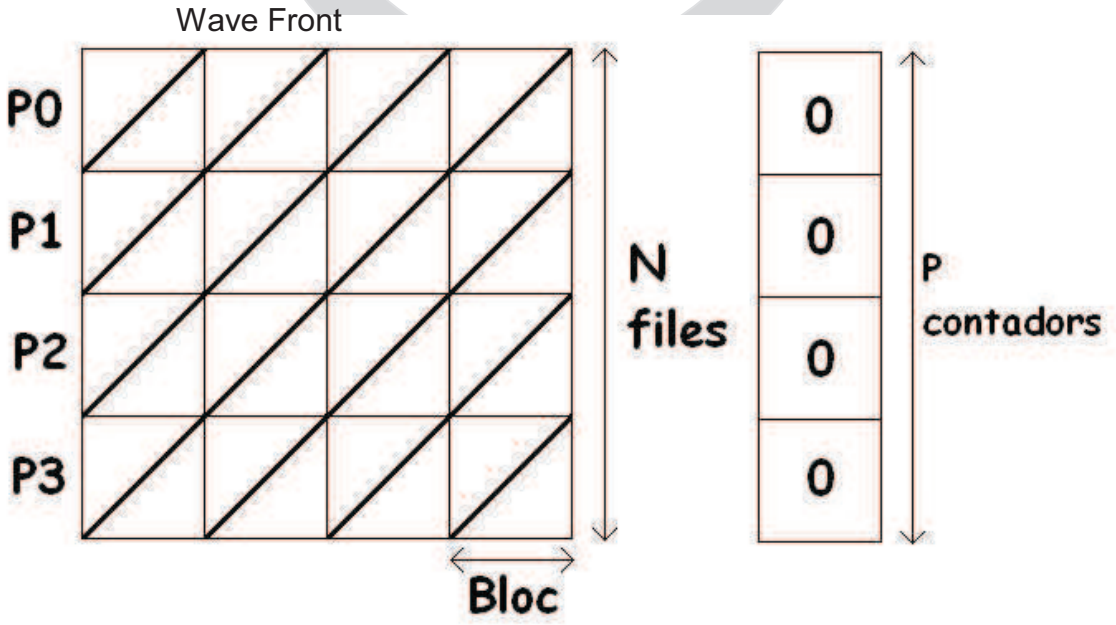
```
void main( ) {
        ...
        #pragma parallel for schedule( ... )
        for (i = 0; i < N; ++i)
        {
                ...
        }
}
```



Fragment omp amb flush:

```
int blocksfinished[NUMTHREADS];
#pragma omp parallel private(j) num_threads(NUMTHREADS)
{
        #pragma omp for
        for (i=0; i<NUMTHREADS; i++) {
                for (j=0; j<n;j++) {
                        if (i > 0) {
                                while (blocksfinished[i-1]<=j) {
                                        #pragma omp flush
                                }
                        }
                        foo(i,j);
                        blocksfinished[i]++;
                        #pragma omp flush
                }
        }

}
```



Divide & Conquer

```
...
#define CUTOFF 4

int divide_conquer(int esq, int der, int x, int vec[], int altura)
{
        if (izq < der)
        {
                int med = (esq+dre)/2;
                int cont, cont1, cont2;
                if (altura < CUTOFF)
                {
                        #pragma omp task
                                cont1 = divide_conquer(0,med,x,vec,altura+1);
                        #pragma omp task
                                cont2 = divide_conquer(med, vec.size(),x,vec,altura+1);
                        #pragma omp taskwait
                                cont = cont1 + cont2;
                {
                else
                {
                        cont1 = divide_conquer(0,med,x,vec,altura+1);
                        cont2 = divide_conquer(med, vec.size(),x,vec,altura+1);
                        cont = cont1 + cont2;
                }
                if (x == vec[med]) ++cont;
        {
        return 0;
}

void main( )
{
        ...
        #pragma omp parallel
        #pragma omp single
        divide_conquer(0,vec.size(),x,vec,0);
        #pragma omp barrier //Hace falta????
        ...
}
```

TEMA 6

Paral·lelitzar amb memòria distribuïda: MPI

MPI ("Message Passing Interface", Interfície d'enviament de missatges) és un estàndard que defineix la sintaxis i la semàntica de les funcions contingudes a la biblioteca d'enviament de missatges dissenyada per a ser utilitzada en programes que explotin els múltiples processadors d'una màquina.

La principal característica es que no precisa de memòria compartida.

El procés que envia, el que rep i el missatge són els trets destacats en l'enviament d'un missatge en MPI.

Esquema bàsic d'un programa amb MPI:

```
#include "mpi.h"
int main(int argc, char * argv [ ])
{
        int nproc, myrank;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        //CODI A EXECUTAR
        MPI_Finalize( );
}
```

Sempre es comença inicialitzant el programa amb MPI_Init per indicar que treballem amb pas de missatges. Després inicialitzem les variables d'entorn (número de processos i el rank d'aquests).

Indica que finalitza la zona de memòria distribuïda amb pas de missatges.

Característiques en els Missatges en MPI

Podem destacar:

- **Bloquejant**: L'execució s'atura fins que no s'ha finalitzat el missatge.
- **No bloquejant**: L'execució es pot seguir malgrat el missatge no hagi acabat.
- **Punt a punt**: Comunicació entre dos processos.
- **Comunicacions Col·lectives**: Comunicació de un a molts, de molts a un i de molts a molts.
- **Tipus de dades en el pas de missatges**:
  - MPI_CHAR
  - MPI_SHORT
  - MPI_INT
  - MPI_LONG
  - MPI_UNSIGNED_CHAR
  - MPI_UNSIGNED_SHORT
  - MPI_UNSIGNED
  - MPI_UNSIGNED_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - MPI_LONG_DOUBLE
  - MPI_BYTE
  - MPI_PACKED

Tipus de Missatges en MPI

MPI_SEND

```
int MPI_Send ( void* buf,                → Adreça del buffer on estan les
                                           dades del missatge a enviar.
               int count,                → Número de dades del buffer.
               MPI_Datatype datatype,    → Tipus de dades que hi ha en el
                                           buffer.
               int destination,          →Identificador del procés que
                                           rebrà el missatge.
               int tag,                  → Etiqueta del missatge.
               MPI_Comm com );           → És el comunicador de MPI que
                                           usarem en el pas de missatges.
```

- És un missatge bloquejant que envia una adreça d'una variable, el número de valors que ha de recollir a partir de la posició de memòria donada, de quin tipus són i amb una etiqueta per titular el missatge.

- Aquest missatge es rep per les comandes MPI_Recv o MPI_IRecv.

MPI_RECV

int MPI_Recv (void* buf, → Adreça del buffer on estan les dades del missatge que rep.
int count, → Número de dades del buffer.
MPI_Datatype datatype, → Tipus de dades que hi ha en el buffer.
int source, →Identificador del procés que envia el missatge.
int tag, → Etiqueta del missatge.
MPI_Comm com, → És el comunicador de MPI que usarem en el pas de missatges.
MPI_Status* status); → Característiques que puguem necessitar del missatge rebut (es veurà més endavant).

- És un missatge bloquejant que rep una adreça d'una variable, el número de valors que ha de recollir a partir de la posició de memòria donada, de quin tipus són, amb una etiqueta per titular el missatge i una variable status amb les característiques del missatge rebut que ens puguin faltar.
- Aquest missatge es rep missatges de MPI_Send o MPI_ISend.

MPI_SENDRECV

int MPI_Recv (void* sendbuf, → Adreça del buffer on estan les dades del missatge que s'envia.
int sendcount, → Número de dades del buffer.
MPI_Datatype sendtype, → Tipus de dades que hi ha en el buffer.
int dest, →Identificador del procés que rep el missatge.
int sendtag, → Etiqueta del missatge.
void* recvbuf, → Adreça del buffer on estan les dades del missatge que rep.
int recvcount, → Número de dades del buffer.
MPI_Datatype recvtype, → Tipus de dades que hi ha en el buffer.
int source, →Identificador del procés que envia el missatge.

int recvtag,	→ Etiqueta del missatge.
MPI_Comm com,	→ És el comunicador de MPI que usarem en el pas de missatges.
MPI_Status* status);	→ Característiques que puguem necessitar del missatge rebut (es veurà més endavant).

- Enviar i rebre el missatge en un missatge bloquejant.

MPI_ISEND

int MPI_ISend (void* buf, → Adreça del buffer on estan les dades del missatge a enviar.

int count,	→ Número de dades del buffer.
MPI_Datatype datatype,	→ Tipus de dades que hi ha en el buffer.
int destination,	→Identificador del procés que rebrà el missatge.
int tag,	→ Etiqueta del missatge.
MPI_Comm com,	→ És el comunicador de MPI que usarem en el pas de missatges.
MPI_Request* request);	→ Serveix per saber si ja s'ha rebut el missatge.

- És igual que MPI_Send, la diferència és que no es bloquejant i, per tant, es pot continuar executant codi sense saber si han rebut el missatge enviat.
- Aquest missatge es rep per les comandes MPI_Recv o MPI_IRecv.

MPI_IRECV

int MPI_IRecv (void* buf, → Adreça del buffer on estan les dades del missatge que rep.

int count,	→ Número de dades del buffer.
MPI_Datatype datatype,	→ Tipus de dades que hi ha en el buffer.
int source,	→Identificador del procés que envia el missatge.
int tag,	→ Etiqueta del missatge.
MPI_Comm com,	→ És el comunicador de MPI que usarem en el pas de missatges.
MPI_Request* request);	→ Serveix per saber si ja s'ha rebut el missatge.

- És igual que MPI_Recv, la diferència és que no es bloquejant i, per tant, es pot continuar executant codi sense saber si s'ha rebut el missatge enviat.
- Aquest missatge es rep missatges de MPI_Send o MPI_ISend.

MPI_WAIT

```
int MPI_Wait ( MPI_Request* request,      → És el request d'una crida no
                                            bloquejant que s'ha fet.
               MPI_Status* status );      → Característiques que puguem
                                            necessitar de la crida feta.
```

- Es bloquejar esperant la finalització de una operació no-bloquejant que identifica request.
- A status tenim la informació de la operació completada.

MPI_TEST

```
int MPI_Test ( MPI_Request* request,      → És el request d'una crida no
                                            bloquejant que s'ha fet.
               int flag,                  → Indica si la rutina s'ha completat.
               MPI_Status* status );      → Característiques que puguem
                                            necessitar de la crida feta.
```

- És com MPI_WAIT però no es bloquejar. La variable flag ens infica si s'ha completat la crida no-bloquejant que identifica request.
- A status tenim la informació de la operació completada.

MPI_PROBE

```
int MPI_Probe ( int source,               → És el request d'una crida no
                                            bloquejant que s'ha fet.
                int tag,                  → Etiqueta del missatge.
                MPI_Comm com,             → És el comunicador de MPI que
                                            usarem en el pas de missatges.
                MPI_Status* status );     → Característiques que puguem
                                            necessitar del missatge rebut (es
                                            veurà més endavant).
```

- MPI_PROBE és una crida bloquejant que indica quan es rep un missatge del destinatari source amb l'etiqueta tag.

- Útil si no es sap quin proces t'enviarà un missatge o quina etiqueta. Source pot tenir valor MPI_ANY_SOURCE i tag pot tenir MPI_ANY_TAG.
- A status tenim la informació que ens pugui faltar sobre source o el tag.

MPI_BCAST

```
int MPI_Bcast (void* buf,            → Adreça del buffer on estan les
                                       dades del missatge a enviar.
               int count,            → Número de dades del buffer.
               MPI_Datatype datatype, → Tipus de dades que hi ha en el
                                       buffer.
               int root,             → És l'identificador de qui envia el
                                       missatge broadcast.
               MPI_Comm com);        → És el comunicador de MPI que
                                       usarem en el pas de missatges.
```

- Tots els processadors executaran aquesta crida, el procés que el seu identificador sigui el root enviarà el missatge i els altres ho rebran.

MPI_REDUCE

```
int MPI_Reduce (void* sendbuf,       → Adreça del buffer on estan les
                                       dades del missatge a enviar.
                void* recvbuf,       → Adreça on es reduiran tots els
                                       resultats
                int count,           → Número de dades del buffer.
                MPI_Datatype datatype, → Tipus de dades que hi ha en el
                                       buffer.
                MPI_Op op,           → Operació la qual es reduirà el
                                       resultat en la variable recvbuf.
                int root,            → És l'identificador de qui envia el
                                       missatge broadcast.
                MPI_Comm com);       → És el comunicador de MPI que
                                       usarem en el pas de missatges.
```

- Tots els processadors executaran aquesta crida, el procés que el seu identificador sigui el root farà l'acumulat a la variable recvbuf.
- Tipus de op: MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC, MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR.

Evitar deadlock en MPI:

Recorda: El deadlock es soluciona corregint l'ordre de les crides que es bloquegen!

```
void main ( int argc, char *argv[] ) {
        ...
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        ...
        //Imaginem que necessitem que el procés i enviï una dada al procés i+1
        //I que el procés i+1 necessiti enviar, també, una dada al procés i
        if (myrank == 0) {
                MPI_Send(&dada,1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD);
        }
        else {
                MPI_Recv(&dada,1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD,&status);
                if (myrank != nproc-1)
                        MPI_Send(&dada,1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD);
                //evitem deadlock!
        }
        //Ja hem fet la primera part
        if (myrank == nproc-1) {
                MPI_Send(&dada,1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD);
        }
        else {
                MPI_Recv(&dada,1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD,&status);
                if (myrank != 0)
                        MPI_Send(&dada,1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD);
                //evitem deadlock!
        }
        MPI_Finalize( );
}
```

Exemple MPI_Bcast

```
void main() {
        int clau;
        //inicialitza MPI
        ...
         MPI_Bcast(clau,1,MPI_INT,myrank,MPI_COMM_WORLD);
        if (myrank == 0) {
                clau = 10;
        }
        else {
                printf("Sóc el proces %d i he rebut la clau %d del proces 0",myrank,clau);
        }
}
```

Exemple MPI_Reduce

```
#define CLAU 5;

void main() {
        int result, global_result;
        int clau = CLAU;
        int * u;
        ...
        //inicialitza MPI
        ...
        u = malloc(N*N*sizeof(int));
        init_matrix(u);
         //COST MOLT ALT!!! Cada procés té parts de matriu que no calcularà
        //Suposem que contem el número de vegades que hi ha una clau en una matriu de
        // NxN. Cada procés farà N/nprocs files i suposem que aquesta divisió és entera
        result = 0;
        for (i = myrank*N; i < (myrank+1)*N; ++i) {
                for (j = 0; j < N;++j) {
                        if (u[i][j] == clau) ++result;
                }
        }
        MPI_Reduce(&result,&global_result,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
        if (myrank == 0) printf("El número de claus és: %d\n"global_result);
        free(u); //Alliberar memòria
        MPI_Finalize( );
        ...
}
```