Programació i computació paral·lela

PID_00250605

Ivan Rodero Castro
Francesc Guim Bernat

Temps mínim de dedicació recomanat: 6 hores

Universitat Oberta de Catalunya

Ivan Rodero Castro

Enginyer d'informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) d'assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau com a màster i doctorat. Des del 2009 fa docència i recerca a Rutgers University (Nova Jersey), on és també el director associat del Rutgers Discovery Informatics Institute. Consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya des del 2010. Centra la seva recerca en l'àrea dels sistemes paral·lels i distribuïts, incloses la computació d'altes prestacions, la *green computing*, la informàtica en núvol i les dades massives.

Francesc Guim Bernat

Enginyer d'Informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) en assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau, com a màster i doctorat. Des de 2008 fa docència com a consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya. Des de l'any 2008 és arquitecte de processadors a la companyia Intel Corporation.

Els textos i imatges publicats en aquesta obra estan subjectes –llevat que s'indiqui el contrari– a una llicència de Reconeixement-NoComercial-SenseObraDerivada (BY-NC-ND) v.3.0 Espanya de Creative Commons. Podeu copiar-los, distribuir-los i transmetre'ls públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), no en feu un ús comercial i no en feu obra derivada. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.ca

Índex

Introducció

En aquest mòdul didàctic estudiarem els models principals de programació d'aplicacions paral·leles utilitzats en la computació d'altes prestacions, els esquemes bàsics de programació d'aplicacions paral·leles i els entorns utilitzats per a l'execució d'aquest tipus d'aplicacions en el context de la computació d'altes prestacions.

En els primers apartats ens centrarem en els models de programació tant per a memòria compartida com distribuïda. En relació amb els models de memòria compartida estudiarem principalment OpenMP i també farem èmfasi en la programació gràfica, la qual té cada cop més presència en la computació d'altes prestacions. A continuació estudiarem els models principals per a memòria distribuïda, tant de pas de missatges (Message Passing Interface - MPI) com els que proporcionen abstraccions d'espais d'adreces compartits utilitzant memòria distribuïda.

Un cop presentats els models de programació més rellevants per a la computació d'altes prestacions estudiarem algunes de les tècniques més populars que permeten el desenvolupament d'aplicacions paral·leles, tant per al seu disseny inicial com per a convertir aplicacions seqüencials en paral·leles. En concret, estudiarem tant tècniques per a l'obtenció de paral·lelisme (per exemple, a partir del paral·lelisme de dades) com diversos patrons d'algoritmes paral·lels.

Objectius

Els materials didàctics d'aquest mòdul contenen les eines necessàries per a assolir els objectius següents:

1. Conèixer els models de programació per a memòria compartida i saber programar aplicacions paral·leles amb OpenMP.

2. Aprendre els conceptes fonamentals per a programar dispositius GPU amb els models de programació per a computació gràfica CUDA i OpenCL.

3. Aprendre els conceptes fonamentals per a programar aplicacions paral·leles amb MPI i conèixer models de programació per a memòria distribuïda basats en espais d'adreces compartits com ara PGAS.

4. Aprendre les tècniques i patrons bàsics per al disseny d'algoritmes paral·lels i ser capaç de desenvolupar aplicacions basades amb aquestes tècniques i els models de programació estudiats en aquest mòdul didàctic.

5. Conèixer el funcionament i els components que formen els sistemes de gestió d'aplicacions en sistemes paral·lels per a la computació d'altes prestacions.

6. Aprendre els conceptes fonamentals de les polítiques de planificació de treballs en entorns d'altes prestacions.

1. Models de programació per a memòria compartida

Tal com vam veure en el mòdul "Introducció a la computació d'altes prestacions" d'aquesta l'assignatura, els sistemes multiprocessador o multinucli de memòria compartida són aquells en els quals qualsevol dels processadors o nuclis poden accedir a qualsevol espai de la memòria. Per tant, hi ha un únic espai d'adreces i, per tant, a cada ubicació de memòria té una única adreça dins d'un rang d'adreces. Les alternatives principals per a la programació de sistemes de memòria compartida són:

- Utilitzar processos convencionals oferts pel sistema operatiu.
- Utilitzar fluxos, com per exemple *Pthreads*.
- Utilitzar un nou llenguatge de programació. Un exemple és el llenguatge Ada, tot i que no és una solució gaire estesa.
- Utilitzar una biblioteca amb un llenguatge de programació existent.
- Modificar la sintaxi d'un llenguatge de programació seqüencial existent per tal de crear un llenguatge de programació paral·lela. Un exemple és UPC[1].
- Utilitzar un llenguatge de programació seqüencial existent i complementar-lo amb directives de compilació per tal d'especificar paral·lelisme. Un exemple és OpenMP.

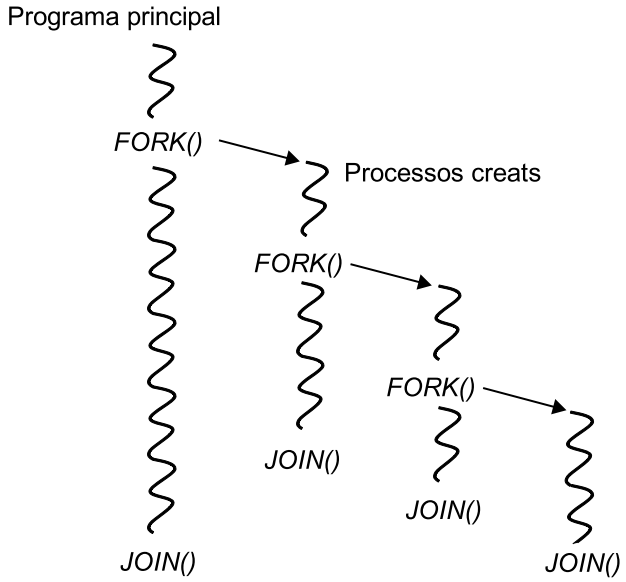
[(1)] De l'anglès *unified parallel C*.

En les seccions següents estudiarem les característiques bàsiques de la programació basada en processadors/fluxos i els possibles problemes associats. A continuació estudiarem OpenMP com a model de programació per a memòria compartida adequat per a la computació d'altes prestacions.

1.1. Programació amb processos i fluxos

Un dels possibles mecanismes per a implementar paral·lelisme amb memòria compartida és l'ús de processos convencionals utilitzant el model *fork-join* tal com mostra la figura 1.

Figura 1. Model *fork-join* utilitzat per a implementar paral·lelisme amb processos



Es tracta d'un model SPMD[2] en què, generalment, el codi corresponent al procés pare és diferent del que executen els processos creats que s'acostumen a anomenar *esclaus*. A més, el procés pare és el que s'encarrega d'esperar la finalització dels processos esclaus (*JOIN* en la figura 1). L'estructura del codi que implementa el model *fork-join* es mostra a continuació.

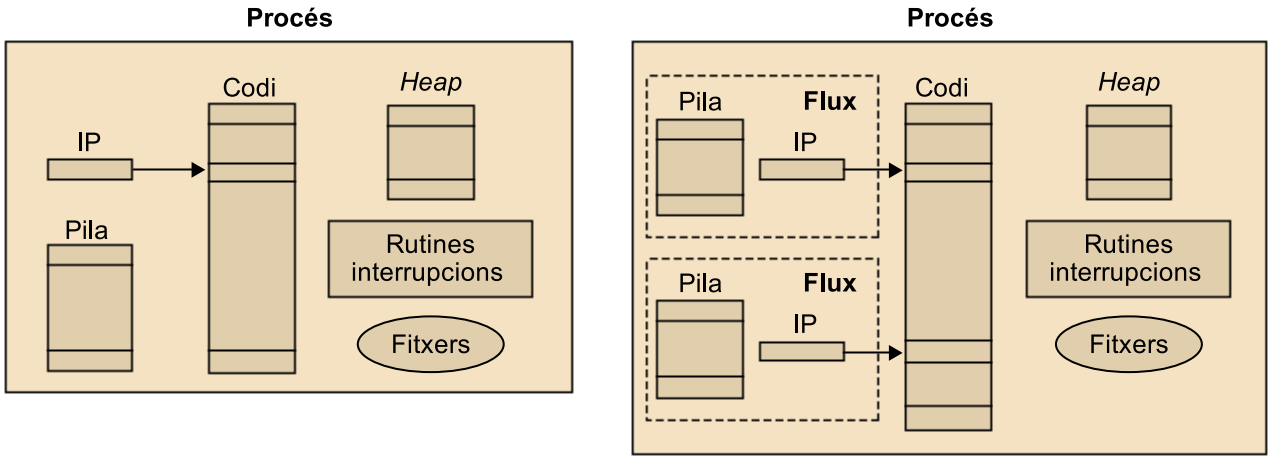
(2)De l'anglès *single-program multiple-data*

```
pid = fork();
if (pid==0) {
// codi a executar per un procés esclau
}
else{
   // codi a executar pel procés pare
}
if (pid==0) exit;
else wait(0);
...
```

El problema principal que fa que aquesta solució no sigui efectiva és la gran sobrecàrrega associada a la creació i gestió dels processos. En canvi, el model introduït anteriorment s'aplica majoritàriament en la programació per a memòria compartida.

L'alternativa més evident a utilitzar processos tradicionals és utilitzar fluxos. Mentre que els processos són molt pesats, ja que són codis completament independents amb les seves pròpies variables i estructures de memòria, els fluxos comparteixen un espai global de memòria entre les diferents rutines tal com mostra la figura 2.

Figura 2. Diferències entre processos i fluxos



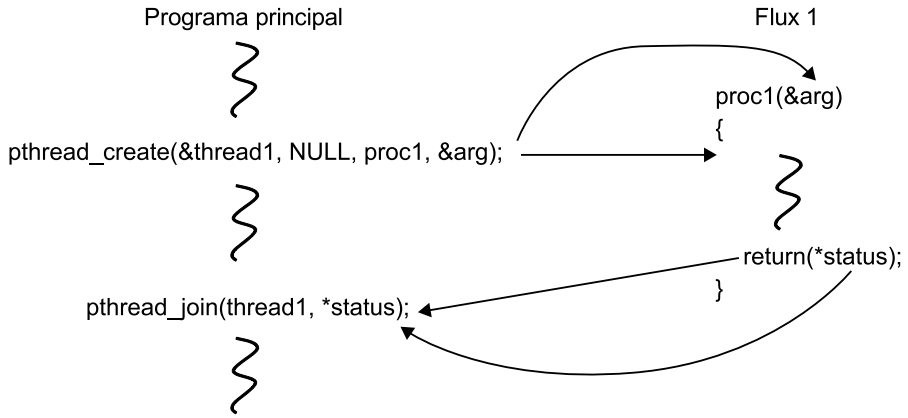
Esquerra: procés convencional. Dreta: procés amb dos fluxos

En un altre mòdul de l'assignatura vam presentar la biblioteca de fluxos *Pthreads*, que compleix els estàndards POSIX. També vam veure un codi que exemplificava el model *fork-join*, que es pot representar tal com mostra la figura 3.

Vegeu també

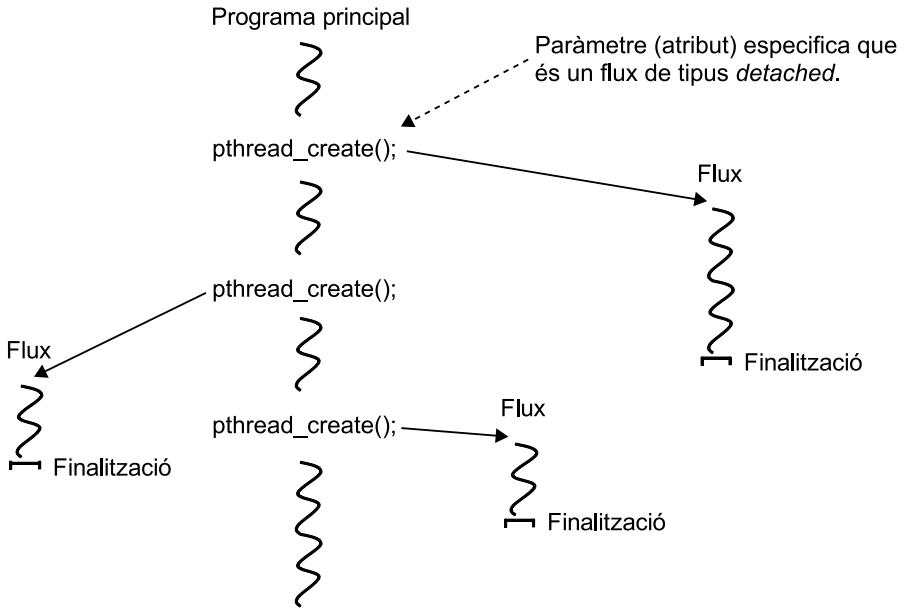
La biblioteca de fluxos *Pthreads* s'ha vist en el mòdul "Introducció a la computació d'altes prestacions" d'aquesta assignatura.

Figura 3. Execució d'un flux mitjançant *Pthreads*



La biblioteca de *Pthreads* també permet utilitzar fluxos que no necessiten que el programa principal hagi d'esperar la seva finalització (mitjançant *join*). Aquests tipus de fluxos s'anomenen *detached* i quan finalitzen són destruïts i els seus recursos alliberats. La figura 4 mostra un esquema de funcionament de fluxos de tipus *detached*.

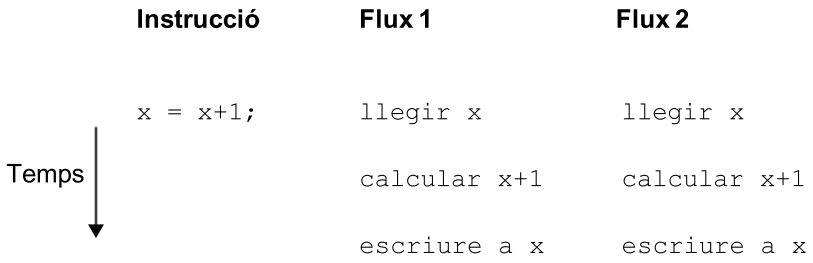
Figura 4. Execució de fluxos de tipus *detached*



Quan s'utilitzen fluxos cal tenir en compte que l'espai de memòria és compartit i poden haver-hi problemes tant de consistència com de rendiment.

Podem dir que una rutina és segura si l'execució de múltiples instàncies de la rutina en diferents fluxos produeixen resultats correctes. Així doncs, caldrà assegurar-se que les rutines són segures quan accedeixen a dades compartides. En l'exemple de la figura 5 es poden veure dos fluxos que incrementen el valor de la variable compartida `x`. Per a dur a terme la instrucció cal llegir primer la variable `x`, després calcular `x+1` i finalment escriure el resultat a `x`. Com que tots dos fluxos llegeixen el valor original de `x` a la vegada, el valor final de `x` només reflectirà un únic increment.

Figura 5. Exemple d'accés a dades compartides mitjançant fluxos



Un mecanisme per a assegurar que només un flux accedeix a un recurs específic al mateix temps és definir seccions del codi que només es puguin executar un sol cop a la vegada. Aquest tipus de seccions de codi s'anomenen *seccions crítiques* i el mecanisme associat a la utilització d'aquestes es coneix com a *exclusió mútua*.

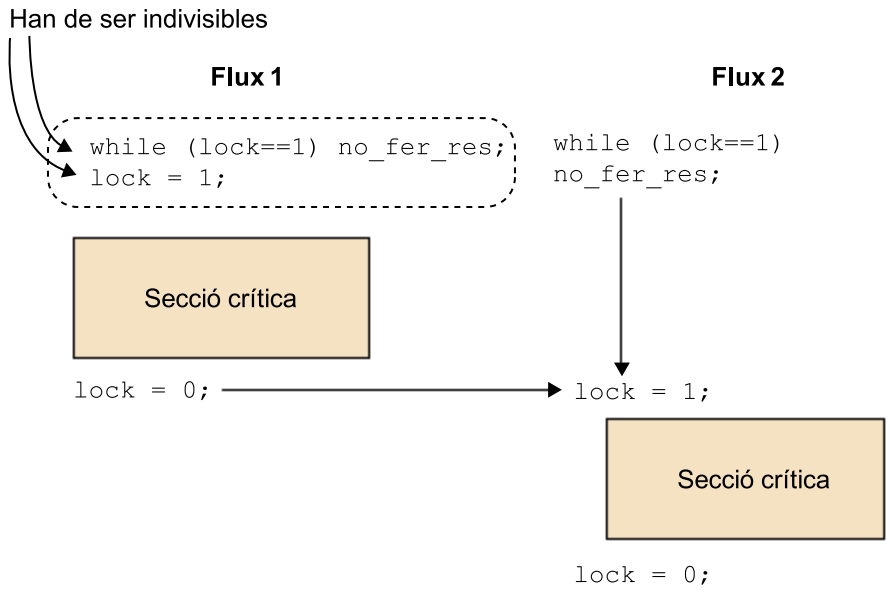
El mecanisme més senzill per a assegurar exclusió mútua en seccions crítiques és la utilització de *locks*. Un *lock* és una variable d'un bit que quan és 1 indica que un flux ha entrat en la secció crítica i quan és 0 indica que cap flux no

hi ha entrat. Una possible implementació d'exclusió mútua és mitjançant una espera activa (*busy waiting* o *spin loops*) que consisteix en els passos següents, que també il·lustra la figura 6:

- Esperar fins que el *lock* indiqui que no hi ha cap flux en la secció crítica.
- Bloquejar la secció crítica canviant el valor del *lock* a 1.
- Executar el codi de la secció crítica.
- Desbloquejar la secció crítica i canviar el valor del *lock* un altre cop a 0.

L'inconvenient principal de l'espera activa és que consumeix molts cicles de CPU que no són realment útils.

Figura 6. Esquema d'espera activa per a accedir a una secció crítica des de dos fluxos



La biblioteca de *Pthreads* implementa els *locks* directament amb variables *lock* d'exclusió mútua que s'anomenen variables de tipus *mutex*. Les seccions crítiques es defineixen com es mostra a continuació:

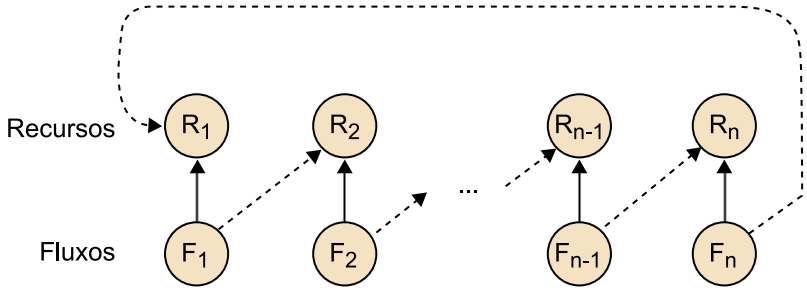
```
pthread_mutex_lock(&mutex1);
   Secció crítica
pthread_mutex_unlock(&mutex1);
```

Si un flux arriba a la crida `pthread_mutex_lock` quedarà bloquejat fins que el *lock* indicat estigui obert. Si hi ha diversos fluxos bloquejats en aquest lloc, quan el *lock* s'obri el sistema determinarà el flux que pot accedir a la secció crítica. Cal tenir en compte que només el flux que tanca un *lock* el pot tornar a obrir.

Un dels problemes principals de l'exclusió mútua és el conegut com a *interbloqueig*[3]. L'interbloqueig de dos fluxos es produeix quan cadascun dels dos necessita el recurs que bloqueja l'altre flux. Aquest fenomen també es pot produir de manera circular involucrant múltiples fluxos, com il·lustra la figura 7. Aquest fenomen es coneix com a *abraçada mortal*.

(3)En anglès, *deadlock*.

Figura 7. Abraçada mortal involucrant *n* fluxos



Per tal de prevenir els interbloqueigs, la biblioteca de *Pthreads* ofereix l'operació `pthread_mutex_trylock()`, que permet consultar si un *lock* està obert o tancat abans de bloquejar el flux. Aquesta operació tanca el *lock* i retorna 0 si prèviament estava obert, o bé retorna `EBUSY` si està tancat.

Activitat

Busqueu per la vostra banda altres mecanismes per a prevenir interbloqueigs, com ara semàfors o monitors.

1.2. OpenMP

És possible desenvolupar programes paral·lels per a memòria compartida amb *Pthreads*, però caldrà gestionar els fluxos i tenir en compte problemes de sincronització com els que hem vist anteriorment. Òbviament, això no afavoreix la productivitat del programador i fa que es necessitin abstraccions de més alt nivell que facilitin la programació i l'execució eficient d'aplicacions paral·leles com és la que proporciona OpenMP.

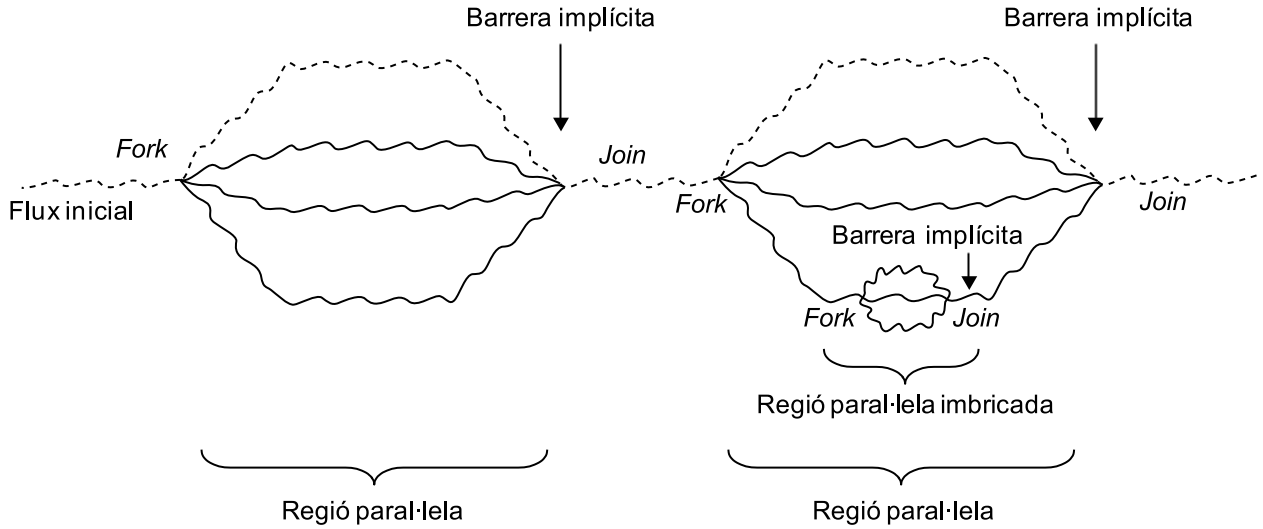
OpenMP consisteix en una interfície de programació que estén llenguatges de programació, com ara C/C++ i Fortran, mitjançant l'ús de directives o "pragmes". OpenMP és l'estàndard actual per a la programació de sistemes de memòria compartida, com ara sistemes multinucli i computadors d'altres prestacions basats en multiprocessadors amb memòria virtual compartida. Típicament OpenMP s'utilitza en sistemes amb un nombre no massa gran de processadors, tot i que hi ha implementacions de sistemes de memòria compartida amb centenars de nuclis. També hi ha versions d'OpenMP per a d'altres tipus de sistemes, com ara clústers i acceleradors, però en aquests casos el rendiment dels programes OpenMP pot ser inferior al dels que utilitzen entorns de programació específics per a aquests sistemes.

OpenMP per a clústers

El sistema *blacklight* del Pittsburgh Supercomputing Center exposa 4.096 nuclis amb un model de memòria compartida.

El model d'execució d'OpenMP és *fork-join*, com il·lustra la figura 8. Això vol dir que inicialment el programa treballa amb un únic flux fins que arriba a una regió paral·lela, és a dir, quan arriba al constructor `#pragma omp parallel` que posa en marxa un nombre de fluxos esclaus (part *fork* del model). El flux que treballa inicialment és el flux mestre d'aquest conjunt d'esclaus. Els fluxos estan enumerats des de 0 fins al nombre de fluxos –1. El mestre i els esclaus treballen en paral·lel en el bloc que apareix a continuació del constructor. En acabar la regió paral·lela hi ha una sincronització de tots els fluxos, els esclaus moren i únicament queda el flux mestre (part *join* del model). A partir d'aquest moment el mestre continua treballant seqüencialment tret que comenci una altra regió paral·lela. Tal com il·lustra la figura, també cal tenir en compte que hi pot haver més d'un nivell de regions paral·leles (regió paral·lela imbricada).

Figura 8. Model *fork-join* d'OpenMP



El codi següent mostra un exemple de programa en OpenMP en què podem observar alguns elements necessaris en programes OpenMP, com ara la inclusió del fitxer de capçaleres `omp.h`, en què es defineixen les funcions d'OpenMP, la utilització de directives (comencen per `#pragma omp`) per a indicar al compilador la manera en què s'ha de distribuir el treball o la utilització de clàusules per a indicar operacions concretes com per exemple la reducció. En concret, el codi retorna la suma de la multiplicació dels elements de dos vectors.

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

int main (int argc, char *argv[])
{
   int   i, n;
   float a[100], b[100], sum;

   /* Algunes inicialitzacions - regió seqüencial */
   n = 100;
```

```
    for (i=0; i < n; i++)
       a[i] = b[i] = i * 1.0;
    sum = 0.0;

    #pragma omp parallel for reduction(+:sum)   /* Regió paral·lela */
    for (i=0; i < n; i++)
       sum = sum + (a[i] * b[i]);

    printf("Suma = %f\n",sum);   /* Regió seqüencial */
}
```

1.2.1. Directives OpenMP

Les directives OpenMP segueixen les convencions dels estàndards per a directives de compilació en C/C++, diferencien entre majúscules i minúscules, només es pot especificar un nom de directiva per directiva, i cada directiva s'aplica almenys a la sentència que segueix, que pot ser un bloc estructurat. En directives llargues es pot utilitzar el caràcter `\` al final de la línia.

El format general és el següent:

```
#pragma omp nom_directiva [clàusules, ...]
```

en què

- `#pragma omp` es demana a totes les directives OpenMP per a C/C++.
- `nom_directiva` és un nom vàlid de directiva, i ha d'aparèixer després del `pragma` i abans de qualsevol clàusula. En el nostre exemple és `parallel for`.
- `[clàusules, ...]` són opcionals. Les clàusules poden anar en qualsevol ordre i repetir-se quan sigui necessari, a menys que hi hagi alguna restricció. En el nostre cas són `reduction` i `private`.

1.2.2. Creació de fluxos

Tal com s'ha indicat anteriorment, la directiva amb què es creen els fluxos esclaus és `parallel` i s'utilitza de la manera següent:

```
#pragma omp parallel [clàusules]
   bloc
```

Amb aquest `pragma`:

- Es crea un grup de fluxos i el flux que els posa en marxa pren el rol de flux mestre.

- El nombre de fluxos a crear s'obté per la variable d'entorn `OMP_NUM_THHREADS` o explícitament amb una crida a la biblioteca tal com veurem més endavant. Si el valor és igual a 0 s'executa de manera seqüencial.

- Hi ha una barrera implícita al final de la regió, de manera que el flux mestre espera que acabin tots els esclaus per a continuar amb l'execució seqüencial.

- Quan dins d'una regió hi ha un altre constructor `parallel`, cada esclau crearia un altre grup de fluxos esclaus dels quals seria el mestre. Això s'anomena *paral·lelisme imbricat* i, tot i que es poden programar d'aquesta manera, en algunes implementacions d'OpenMP la creació de grups d'esclaus dins d'una regió paral·lela no està suportada.

- Les clàusules de compartició de variables que suporta la directiva `parallel` són principalment: `private`, `firstprivate`, `lastprivate`, `default`, `shared`, `copyin` i `reduction`. El significat d'aquestes clàusules i d'altres de compartició es veurà més endavant.

1.2.3. Clàusules de compartició de variables

En els casos anteriors hem vist que les directives OpenMP tenen clàusules amb què poden indicar com es du a terme la compartició de variables, les quals estan totes en memòria compartida. Cada directiva té una sèrie de clàusules que s'hi poden aplicar. Aquestes són:

- `private(llista)`. Les variables de la llista són privades dels fluxos, que vol dir que cada flux té una variable privada amb aquest nom (les quals poden tenir valors diferents en diferents fluxos). Les variables no s'inicialitzen abans d'entrar en la regió paral·lela (per tant, no podem esperar un valor inicial concret en entrar en la regió paral·lela) i no es guarda el seu valor en sortir de la regió paral·lela (per tant, no podem esperar que l'últim valor d'una variable privada es conservi en finalitzar la regió paral·lela).

- `firstprivate(llista)`. Les variables són privades dels fluxos i s'inicialitzen en entrar a la regió paral·lela amb el valor que tingués la variable corresponent del flux mestre.

- `lastprivate(llista)`. Són privades dels fluxos i en sortir de la regió paral·lela queden amb el valor de l'última iteració (si estem en un bucle

`for` paral·lel) o secció (veurem més endavant el funcionament de les seccions).

- `shared(llista)`. Indica les variables compartides per tots els fluxos. Per defecte totes són compartides, per la qual cosa en la majoria dels casos no fa realment falta utilitzar aquesta clàusula.

- `default(shared|none)`. Indica com seran les variables per defecte. Si s'especifica `none` caldrà indicar explícitament amb la clàusula `shared` les que es vulgui que siguin compartides.

- `reduction(operador:llista)`. Les variables de la llista s'obtenen per l'aplicació de l'operador, que ha de ser associatiu.

- `copyin(llista)`. S'utilitza per a assignar el valor de la variable en el mestre a variables del tipus `threadprivate`.

1.2.4. Clàusules de divisió del treball

Una vegada s'han generat amb `parallel` un conjunt de fluxos esclaus, aquests i el mestre poden treballar en la resolució d'un problema en paral·lel. Hi ha dues maneres de dividir el treball entre el fluxos: mitjançant les directives `for` i `sections`. Mentre que la primera directiva fa referència al paral·lelisme de dades, la segona fa referència al paral·lelisme funcional.

La directiva for té la forma següent:

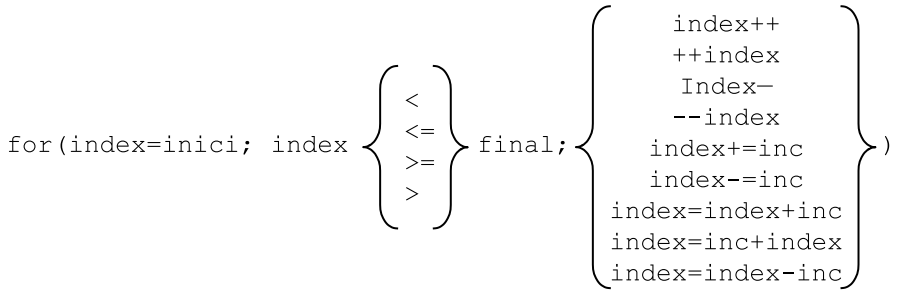
```
#pragma omp for [clàusules]
   bucle for
```

en què:

- Les iteracions les executen en paral·lel els fluxos que ja hi ha, creats prèviament amb `parallel`.

- El bucle `for` ha de tenir una forma especial (forma canònica), tal com mostra la figura 9. La part d'inicialització ha de tenir una assignació; la part de l'increment una suma o resta; la de l'avaluació és la comparació d'una variable entera amb signe amb un valor, utilitzant un comparador major o menor (pot incloure igual); i els valors que apareixen en les tres parts del `for` han de ser enters.

Figura 9. Forma canònica dels bucles en OpenMP



Activitat

Busqueu quina diferencia hi ha entre fer un `index++` o bé un `++index`, i quina de les dues opcions és més eficient.

- Hi ha una barrera implícita al final del bucle, tret que s'utilitzi la clàusula `nowait`.

- Les clàusules de compartició de variables que admet són `private`, `first-private`, `lastprivate` i `reduction`.

- Pot aparèixer una clàusula `schedule` per a indicar de quina manera es divideixen les iteracions del for entre els fluxos, és a dir la política de planificació.

Les polítiques de planificació disponibles a OpenMP són:

- `schedule(static, mida)`. Les iteracions es divideixen segons la mida de bloc que s'indica. Per exemple, si el nombre d'iteracions del bucle és de 100 i aquestes estan numerades de 0 a 99, i la mida de bloc és 2, es consideren 50 blocs cadascun dels quals amb dues iteracions (blocs amb iteracions 0-1, 2-3, etc.), i si disposem de 4 fluxos, els blocs d'iteracions s'assignen als fluxos cíclicament, amb la qual cosa al flux 0 corresponen les iteracions 0-1, 8-9, etc., al flux 1 les iteracions 2-3, 10-11, etc., al 2 les 4-5, 12-13, etc., i al 3 les 6-7, 14-15, etc. Si no s'indica cap mida de bloc, les iteracions es divideixen per igual entre els fluxos amb blocs de mida màxima: si tenim 12 iteracions i 4 fluxos, s'assignen al flux 0 les iteracions 0-2, a l'1 les 3-5, al 2 les 6-8 i al 4 les 9-11.

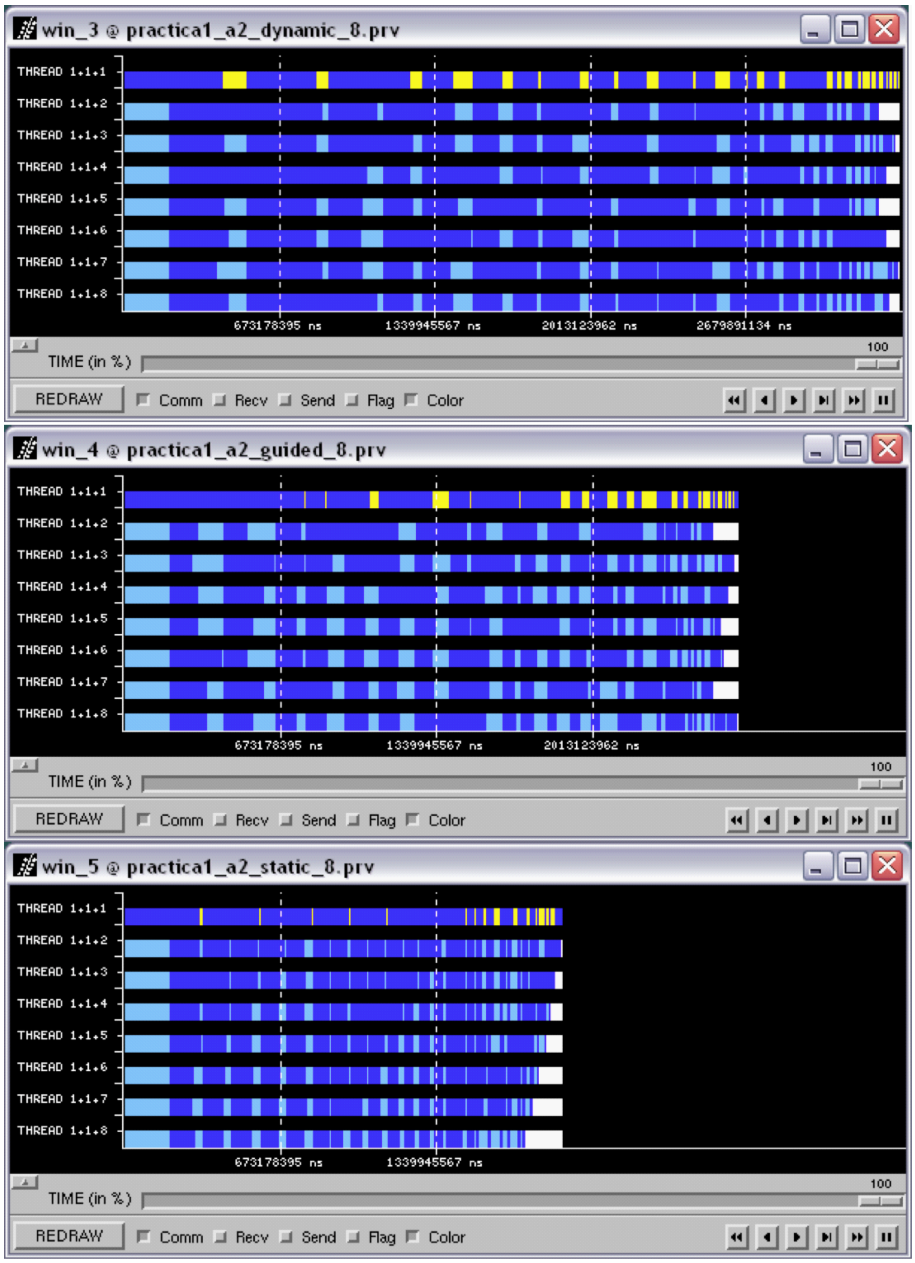
- `schedule(dynamic, mida)`. Les iteracions s'agrupen segons la mida de bloc especificada i s'assignen als fluxos dinàmicament quan van acabant el seu treball. En el cas anterior de 100 iteracions, mida de bloc 2 i 4 fluxos, s'assignen inicialment al flux i les iteracions `2i` i `2i+1`, i a partir d'aquí la resta de blocs de dues iteracions s'assignen als fluxos segons es vagin quedant sense feina. Quan el volum de computació de cada iteració no es coneix a priori, pot ser preferible utilitzar una assignació dinàmica, ja que permet reduir el desbalanceig.

- `schedule(guided, mida)`. Les iteracions s'assignen dinàmicament als fluxos però amb mida de bloc decreixent fins a arribar a la mida de bloc que s'indica (si no s'indica cap valor per defecte és 1). La mida inicial de bloc i la manera en què decreix depèn de la seva implementació.

- `schedule(runtime)`. Decideix la política de planificació en temps d'execució mitjançant la variable d'entorn `OMP_SCHEDULE`.

La figura 10 mostra les traces obtingudes de l'execució d'una aplicació OpenMP utilitzant les polítiques de planificació `static`, `dynamic` i `guided`. Es pot apreciar que el temps d'execució (eix de les *x*) pot variar molt significativament en funció de la política de planificació.

Figura 10. Traces d'execució extretes amb l'aplicació Paraver del Barcelona Supercomputing Center



Activitat

Amb vista a practicar amb l'OpenMP i entendre'n el funcionament, us recomanem descarregar Paraver en un entorn Linux (pot ser una màquina virtual) i fer algunes proves senzilles d'OpenMP amb Paraver per a entendre'n el comportament.

La directiva sections té la forma següent:

```
#pragma omp sections [clàusules]
   {
      [#pragma omp section]
         bloc
      [#pragma omp section
         bloc
      ...
      ]
   }
```

en què:

- Cada secció s'executa mitjançant un flux. La manera en què es distribueixen les seccions entre els fluxos depèn de la implementació concreta d'OpenMP.
- Hi ha una barrera final tret que s'utilitzi la clàusula `nowait`.
- Les clàusules de compartició de variables que admet són `private`, `firstprivate`, `lastprivate` i `reduction`.

Atès que `for` i `sections` s'utilitzen amb molta freqüència immediatament després d'haver creat els fluxos amb `parallel`, hi ha dues directives combinades que s'utilitzen molt i que s'especifiquen a continuació:

```
#pragma omp parallel for [clàusules]
   bucle for
```

i

```
#pragma omp parallel sections [clàusules]
```

Aquestes són una manera abreviada de posar la directiva `parallel` que conté una única directiva `for` o `sections`. Les dues tenen les mateixes clàusules que `for` i `sections`, excepte `nowait`, que en aquest cas no està permesa ja que acaba el `parallel`, amb el qual el mestre ha d'esperar que acabin tots els esclaus per a seguir l'execució seqüencial.

1.2.5. Directives de sincronització

Dins d'una regió paral·lela pot ser necessari sincronitzar els fluxos en l'accés a algunes variables o zones de codi per a evitar situacions d'interbloqueig. OpenMP ofereix diverses primitives amb aquesta finalitat:

- `single`. El codi afectat per la directiva l'executarà un únic flux. Els fluxos que no estan treballant durant l'execució de la directiva esperen al final. Admet les clàusules `private`, `firstprivate` i `nowait`. No està permès fer bifurcacions cap a un bloc o des d'un bloc `single`. És útil per a seccions de codi que poden ser no segures per a la seva execució paral·lela (per exemple, per a entrada/sortida).

- `master`. El codi només l'executa el flux mestre. La resta de fluxos no executen aquesta secció de codi.

- `critical`. Protegeix una secció de codi perquè hi pugui accedir un únic flux a la vegada. S'hi pot associar un nom de la forma:

```
#pragma omp critical [nom]
```

  de manera que hi pot haver seccions crítiques protegint zones diferents del programa, de manera que a seccions amb noms diferents hi poden accedir diversos fluxos a la vegada. Les regions que tinguin el mateix nom es tracten com la mateixa regió. Totes les que no tenen nom es consideren la mateixa. No està permès fer bifurcacions cap a un bloc o des d'un bloc critical.

- `atomic`. Assegura que una posició de memòria es modifiqui sense que múltiples fluxos intentin escriure-la de manera simultània. S'aplica a la sentència que segueix a la directiva. Només s'assegura en mode exclusiu l'actualització de la variable, però no l'avaluació de l'expressió.

- `barrier`. Sincronitza tots els fluxos. Quan un flux arriba a la barrera, espera que arribin els altres i, quan han arribat tots, aquestes segueixen la seva execució.

- `threadprivate`. S'utilitza perquè variables globals es converteixin en locals i persistents en un flux per mitjà de múltiples regions paral·leles.

- `ordered`. Assegura que el codi s'executi en l'ordre en què les iteracions s'executen en la seva forma seqüencial. Pot aparèixer només una vegada en el context d'una directiva `for` o `parallel for`. No està permès fer bifurcacions cap a un bloc o des d'un bloc `ordered`. Només hi pot haver un únic flux executant-se simultàniament en una secció `ordered`. Una iteració d'un bucle no pot executar la mateixa directiva `ordered` més d'una

vegada, i no ha d'executar més d'una directiva `ordered`. Un bucle amb una directiva `ordered` ha de contenir una clàusula `ordered`.

- `flush`. Té la forma:

```
#pragma omp flush [llista-de-variables]
```

i assegura que el valor de les variables s'actualitza en tots els fluxos en què són visibles. Si no hi ha una llista de variables s'actualitzaran totes.

1.2.6. Funcions i variables

Les funcions que típicament s'utilitzen més en OpenMP són les relacionades amb el nombre de fluxos, és a dir, `omp_set_num_threads`, `omp_get_num_threads` i `omp_get_thread_num`. Altres funcions importants són:

- `omp_get_max_threads`. Obté la màxima quantitat possible de fluxos.
- `omp_get_num_procs`. Retorna el nombre màxim de processadors que es poden assignar al programa.
- `omp_in_parallel`. Retorna un valor diferent de 0 si s'executa dins d'una regió paral·lela.

Es pot posar o treure el que el nombre de fluxos es pugui assignar dinàmicament en les regions paral·leles (`omp_set_dynamic`), o es pot comprovar si està permès l'ajust dinàmic, amb la funció `omp_get_dynamic`, que retorna un valor diferent de 0 si està permès l'ajust dinàmic del nombre de fluxos.

Es pot permetre desautoritzar el paral·lelisme imbricat amb `omp_set_nested`, o comprovar si està permès, amb `omp_get_nested`, que retornen un valor diferent de 0 si n'està.

També hi ha funcions per a la gestió de *locks* com les que es mostren a continuació:

- `void omp_init_lock(omp_lock_t *lock)` per a inicialitzar un *lock*, que s'inicialitza com a no bloquejat.
- `void omp_init_destroy(omp_lock_t *lock)` per a destruir un *lock*.
- `void omp_set_lock(omp_lock_t *lock)` per a bloquejar un *lock*.
- `void omp_unset_lock(omp_lock_t *lock)` per a desbloquejar un *lock*.
- `void omp_test_lock(omp_lock_t *lock)` per a comprovar si un *lock* està bloquejat o no i així poder evitar bloquejos indesitjats.

1.2.7. Entorn de compilació i execució

Per a compilar un programa OpenMP cal disposar d'un compilador que pugui interpretar les directives que apareixen en el codi. OpenMP està suportat pel compilador GCC des de la versió 4.1 i també per d'altres de propietaris, com ara l'ICC d'Intel. En aquest subapartat utilitzarem el llenguatge C per a exemplificar l'ús d'OpenMP. L'opció de compilació per a GCC és `-fopenmp` o `-openmp` i per a ICC és `-openmp`. Així doncs, la línia següent serviria per a compilar un programa OpenMP amb GCC:

```
gcc -fopenmp -o programa programa.c
```

L'execució del fitxer binari (`programa` en el nostre exemple) té lloc com qualsevol altre programa, però cal determinar quants fluxos intervindran en l'execució paral·lela. Podem utilitzar una variable d'entorn (`OMP_NUM_THREADS`) que ens indica el nombre de fluxos a utilitzar. Si no s'inicialitza aquesta variable, quan executem el nostre programa OpenMP tindrà un valor per defecte, que acostuma a coincidir amb el nombre de nuclis del node en què estem treballant.

Abans de l'execució es poden establir els valor de la variable, per exemple des de l'intèrpret d'ordres. Per exemple, si establim el valor de la variable com a continuació:

Nota

Depenent de l'intèrpret de comandes que s'estigui fent servir, caldrà emprar el comandament 'set' en comptes d''export'.

```
export OMP_NUM_THREADS=6
```

Establirem el nombre de fluxos a utilitzar en la regió paral·lela a sis, independentment del nombre de nuclis de què disposem. Això vol dir que és poden utilitzar més fluxos que processadors hi ha al nostre sistema. En termes generals, el rendiment de l'execució d'un programa OpenMP, utilitzant més fluxos que processadors disponibles en el sistema, és inferior al que podem aconseguir utilitzant un flux per processador per la sobrecàrrega que genera la creació i gestió dels fluxos. De fet, podem arribar a observar que el temps d'execució del programa paral·lel sigui més elevat que el del programa seqüencial.

És molt important recordar que la mida del problema és vital per a poder explotar el paral·lelisme. Així doncs, encara que disposem de diversos nuclis la mida del problema (el nombre d'intervals) pot no ser suficientment gran perquè es noti l'efecte de la paral·lelització, especialment si tenim en compte que hi ha zones que s'executen seqüencialment.

Tot i això, encara que no disposem de suficients processadors per a executar tots els fluxos, podria ser que s'obtingués un rendiment millor gràcies a unes millors particions i localitat de les dades. Per exemple, pot succeir que, en

dividir el problema en diversos fluxos, les dades a tractar a cada iteració d'un bucle hi càpiguen en una pàgina de memòria cau i, en conseqüència, es pugui accelerar l'execució de manera molt significativa.

Activitat

Com que actualment la majoria de computadors personals disposen de múltiples nuclis, podeu experimentar i buscar explicació als temps que s'obtenen de l'execució d'algun programa OpenMP variant el nombre de fluxos, essent aquest nombre tant més petit com més gran que el nombre de nuclis disponibles.

Hi ha quatre variables d'entorn. A més d'OMP_NUM_THREADS, també hi ha disponible:

- `OMP_SCHEDULE` indica el tipus de planificació per a `for` i `parallel for`.
- `OMP_DYNAMIC` autoritza o desautoritza l'ajust dinàmic del nombre de fluxos.
- `OMP_NESTED` autoritza o desautoritza el paral·lelisme imbricat. Per defecte no està autoritzat.

2. Models de programació gràfica

En els darrers anys, la capacitat de càlcul dels processadors gràfics[4] ha esdevingut més elevada que la dels processadors multinucli més avançats per a dur a terme certes tasques. Això ha fet que aquest tipus de dispositius estiguin esdevenint molt populars per al còmput d'algoritmes de propòsit general i no sols per a la generació de gràfics. La computació de propòsit general sobre GPU es coneix popularment com a GPGPU[5].

(4)GPU, de l'anglès *graphics processing unit*.

(5)De l'anglès *general-purpose computing on graphics processing unit*.

Les GPU proporcionen rendiments molt elevats només per a certes aplicacions per les seves característiques arquitecturals i de funcionament. De manera general, podem dir que les aplicacions que poden aprofitar millor les capacitats de les GPU són les que compleixen les dues condicions següents:

- Treballen sobre vectors de dades grans.
- Tenen un paral·lelisme de gra fi tipus SIMD.

Un dels inconvenients principals a l'hora de treballar amb GPU és la dificultat per al programador per a transformar programes dissenyats per a CPU tradicionals en programes que es puguin executar de manera eficient en una GPU. Per aquest motiu s'han desenvolupat models de programació, tant propietaris (CUDA) com oberts (OpenCL), que proporcionen al programador un nivell d'abstracció, més proper a la programació per a CPU, que li simplifiquen considerablement la tasca.

En aquest apartat estudiarem els models principals de programació per a GPU orientats a aplicacions de propòsit general, específicament CUDA i OpenCL, que són els estàndards actuals per a la programació de processadors gràfics.

2.1. CUDA

CUDA[6] és una especificació inicialment propietària desenvolupada per Nvidia com a plataforma per als seus productes GPU. CUDA inclou les especificacions de l'arquitectura i un model de programació associat. En aquest subapartat estudiarem el model de programació CUDA; no obstant això, ens referirem a dispositius compatibles amb CUDA en les GPU que implementen l'arquitectura i especificacions definides a CUDA.

(6)De l'anglès *compute unified device architecture*.

2.1.1. Arquitectura compatible amb CUDA

La figura 11 mostra l'arquitectura d'una GPU genèrica compatible amb CUDA. Aquesta arquitectura està organitzada en una sèrie de multiprocessadors[7] els quals tenen una quantitat elevada de fluxos d'execució. En la figura, dos SM

(7)SM, de l'anglès *streaming multiprocessors*.

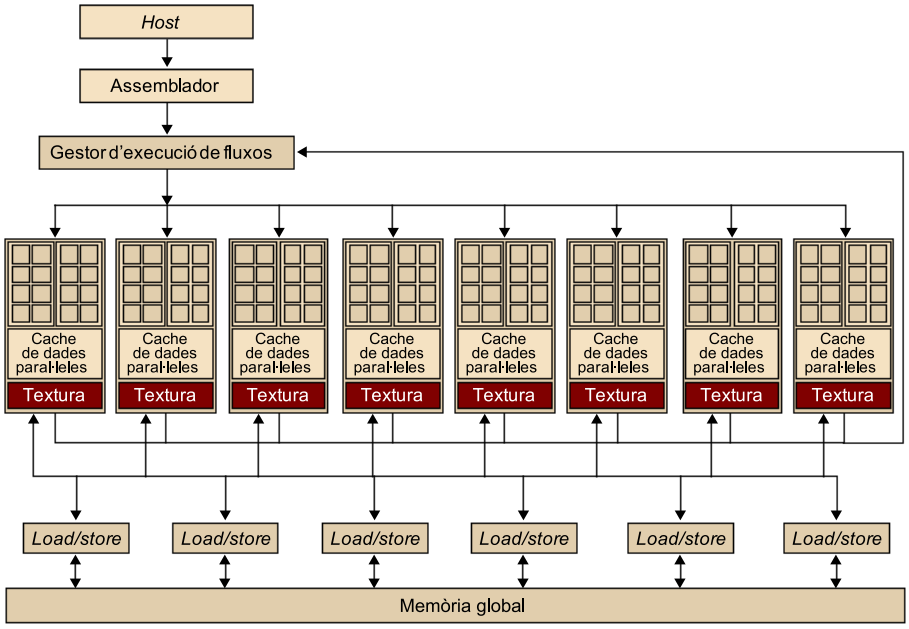
formen un bloc, tot i que el nombre d'SM per bloc depèn de la implementació concreta del dispositiu. A més, cada SM de la figura té un nombre de *streaming processors* (SP) que comparteixen la lògica de control i memòria cache d'instruccions.

La GPU té una memòria DRAM (que pot ser de tipus GDDR[8]), la qual està indicada en la figura 11 com a memòria global[9]. Aquesta memòria es diferencia de la memòria DRAM de la placa base del computador en què essencialment s'utilitza per a gràfics (*framebuffer*). Per a aplicacions gràfiques aquesta manté les imatges de vídeo i la informació de les textures. En canvi, per a càlculs de propòsit general aquesta funciona com a memòria externa amb molta amplada de banda però amb una latència una mica més elevada que la memòria típica del sistema. Tot i així, per a aplicacions massivament paral·leles, l'amplada de banda més gran compensa la latència més elevada.

(8)De l'anglès *graphics double data rate*.

(9)Depenent del tipus d'arquitectura de la GPU que s'estigui fent servir, pot ser que el tipus de memòria sigui diferent o fins amb un tipus d'arquitectura global diferent (per exemple, memòria física compartida entre la GPU i la CPU).

Figura 11. Esquema de l'arquitectura d'una GPU genèrica compatible amb CUDA



Noteu que aquesta arquitectura genèrica és molt similar a la G80, la qual suporta fins a 768 fluxos per SM, el qual suma un total de 12.000 fluxos en un únic xip.

L'arquitectura GT200, posterior a la G80, té 240 SP i supera el TFlop de pic teòric de rendiment. Com que els SP són massivament paral·lels, es poden arribar a utilitzar encara més fluxos per aplicació que la G80. La GT200 suporta 1.024 fluxos per SM i en total suma prop de 30.000 fluxos per xip. Per tant, la tendència mostra clarament que el nivell de paral·lelisme que suporten les GPU està augmentant ràpidament. Serà molt important, doncs, intentar explotar aquest nivell tan elevat de paral·lelisme quan es desenvolupin aplicacions de propòsit general per a GPU.

2.1.2. Entorn de programació

La CUDA es va desenvolupar per tal d'augmentar la productivitat en el desenvolupament d'aplicacions de propòsit general per a GPU. Des del punt de vista del programador, el sistema està compost per un *host*, que és una CPU tradicional, com per exemple un processador d'arquitectura Intel, i d'un o més dispositius[10] que són GPU.

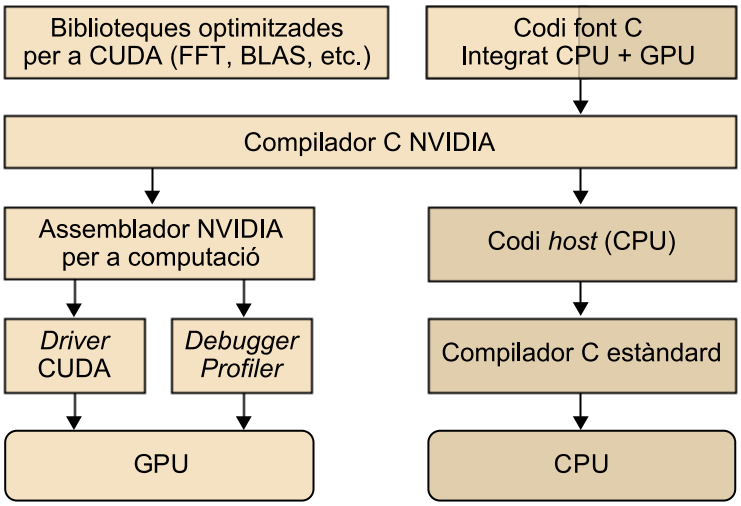
(10)En anglès, *devices*.

La CUDA pertany al model SIMD, per tant, està pensada per a explotar el paral·lelisme de dades. Això vol dir que un conjunt d'operacions aritmètiques es poden executar sobre un conjunt de dades de manera simultània. Afortunadament, moltes aplicacions tenen parts amb un nivell molt elevat de paral·lelisme de dades.

Un programa en CUDA consisteix en una o més fases que es poden executar o bé en el *host* (CPU) o bé en el dispositiu GPU. Les fases en què hi ha molt poc o gens de paral·lelisme entre dades s'implementen en el codi que s'executarà en el *host*, i les fases amb un nivell de paral·lelisme entre dades elevat s'implementen en el codi que s'executarà en el dispositiu.

Tal com mostra la figura 12, el compilador de NVIDIA (del qual `nvcc` és el binari principal) s'encarrega de proporcionar la part del codi corresponent al *host* i al dispositiu durant el procés de compilació. El codi corresponent al *host* és simplement codi ANSI C, que es compila mitjançant el compilador de C estàndard del *host*, com si fos un programa per a CPU convencional. El codi corresponent al dispositiu també és ANSI C però amb extensions que inclouen paraules clau per a poder definir funcions que tracten les dades en paral·lel. Aquestes funcions s'anomenen *nuclis* o *kernels*.

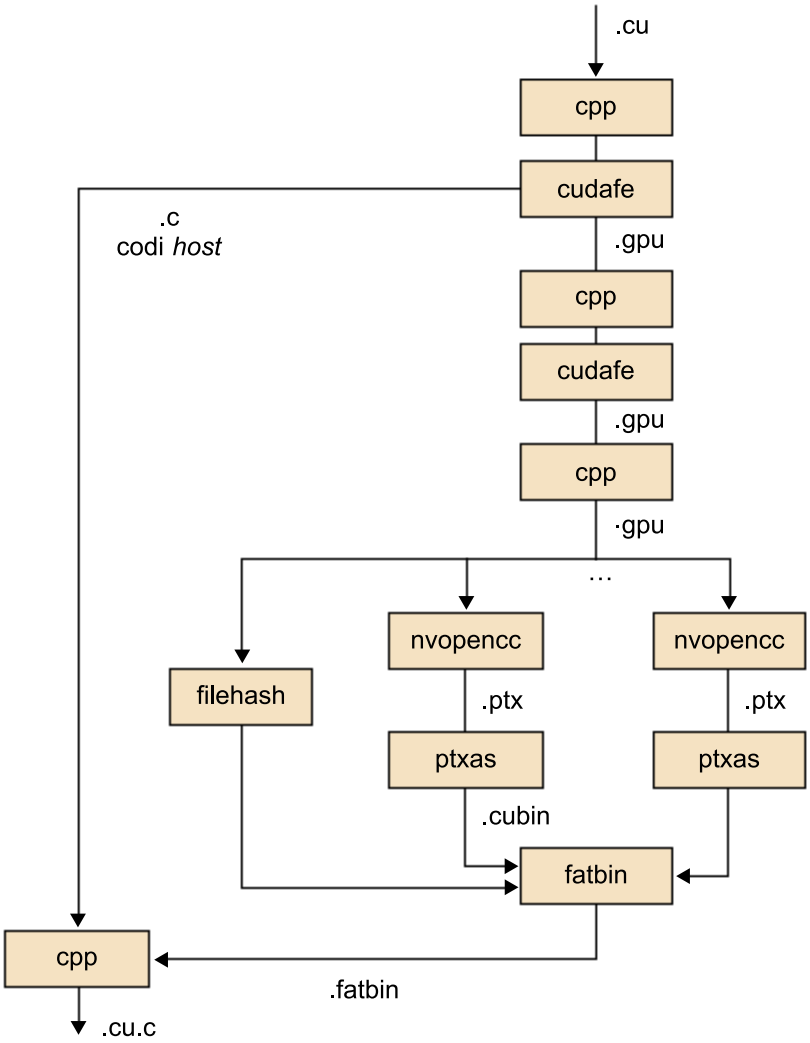
Figura 12. Esquema de blocs de l'entorn de compilació de la CUDA



El codi del dispositiu es torna a compilar amb `nvcc` i llavors ja es pot executar en el dispositiu GPU. En situacions en què no hi ha cap dispositiu disponible o bé el nucli és més apropiat per a una CPU, també es poden executar els nuclis

en una CPU convencional mitjançant eines d'emulació que proporciona la plataforma CUDA. La figura 13 mostra totes les etapes del procés de compilació i la taula 1 descriu com el compilador `nvcc` interpreta els diferents tipus de fitxers d'entrada.

Figura 13. Passos en la compilació de codi CUDA, des de .cu fins a .cu.c



Taula 1. Interpretació de nvcc dels fitxers d'entrada

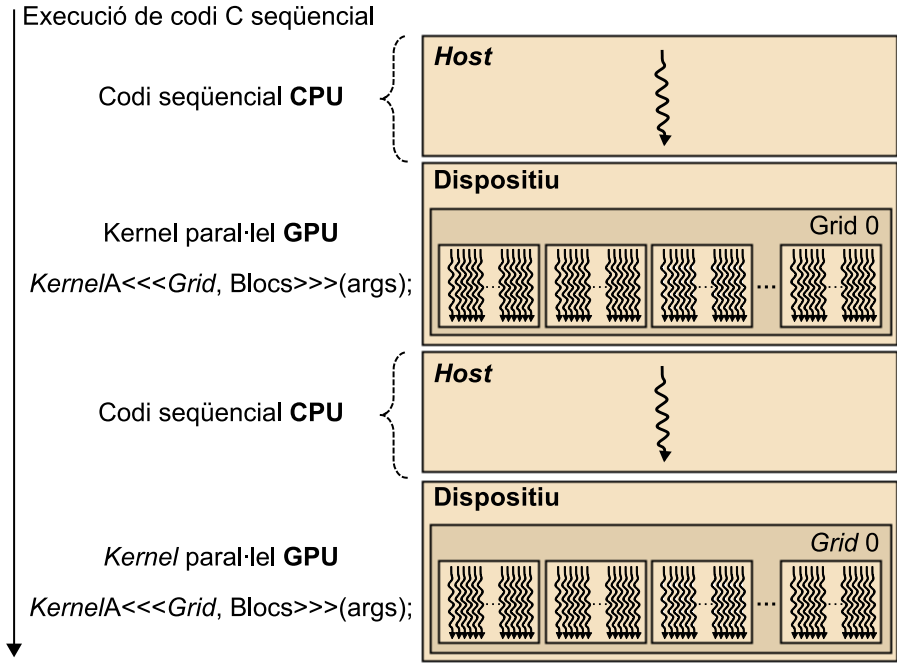
.cu	Codi font CUDA que conté tant el codi del *host* com les funcions del dispositiu.
.cup	Codi font CUDA preprocessat que conté tant el codi del *host* com les funcions del dispositiu.
.c	Fitxer de codi font C.
.cc, .cxx, .cpp	Fitxer de codi font C++.
.gpu	Fitxer intermedi gpu.
.ptx	Fitxer assemblador intermedi ptx.
.o, .obj	Fitxer d'objecte.
.a, .lib	Fitxer de biblioteca.

.res	Fitxer de recurs.
.so	Fitxer d'objecte compartit.

Per a explotar el paral·lelisme entre dades, els nuclis han de generar una quantitat de fluxos d'execució força elevada (per exemple, unes desenes o centenars de milers de fluxos). Hem de tenir en compte que els fluxos de CUDA són molt més lleugers que els fluxos de CPU. De fet podrem assumir que per a generar i planificar aquests fluxos només necessitarem uns pocs cicles de rellotge a causa del suport de maquinari, en contrast amb els fluxos convencionals per a CPU que normalment requereixen milers de cicles.

L'execució d'un programa típic CUDA es mostra en la figura 14. L'execució comença amb execució en el *host* (CPU). Quan s'invoca un nucli, l'execució es mou cap al dispositiu (GPU) on es genera un nombre molt elevat de fluxos. El conjunt de tots aquest fluxos que es generen quan s'invoca un nucli s'anomena *grid*. En la figura 14 es mostren dos *grids* de fluxos. Un nucli finalitza quan tots els seus fluxos finalitzen la seva execució en el *grid* corresponent. Un cop finalitzat el nucli, l'execució del programa continua en el *host* fins que s'invoca un altre nucli.

Figura 14. Etapes en l'execució d'un programa típic CUDA



2.1.3. Model de memòria

És important remarcar que la memòria del *host* i del dispositiu són espais de memòria completament separats. Això reflecteix la realitat que els dispositius són típicament targetes que tenen la seva pròpia memòria DRAM. Per a executar un nucli en el dispositiu GPU normalment cal seguir aquests passos:

- Reservar memòria en el dispositiu (pas 1 de la figura 15).

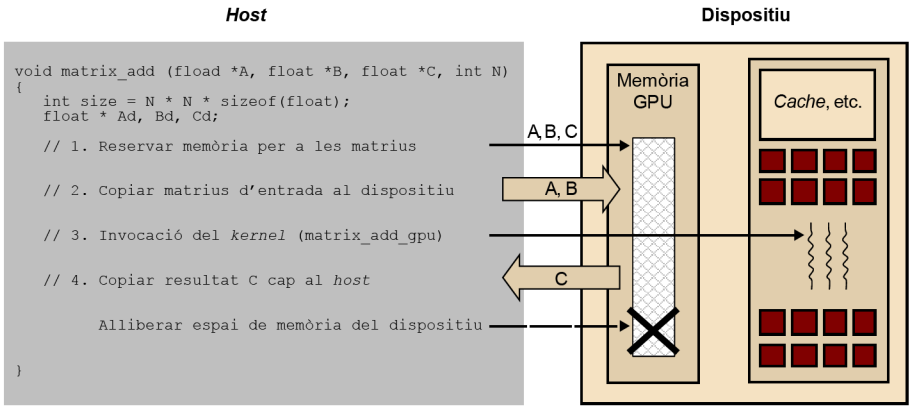
- Transferir les dades necessàries des del *host* a l'espai de memòria assignat al dispositiu (pas 2 de la figura 15).

- Invocar l'execució del nucli en qüestió (pas 3 de la figura 15).

- Transferir les dades amb els resultat des del dispositiu cap al *host* i alliberar la memòria del dispositiu (si ja no és necessària) un cop finalitzada l'execució del nucli (pas 4 de la figura 15).

L'entorn CUDA proporciona una interfície de programació que simplifiquen aquestes tasques al programador. Per exemple, només cal especificar quines dades cal transferir des del *host* al dispositiu i viceversa.

Figura 15. Esquema dels passos per a l'execució d'un nucli en una GPU



Durant tot aquest apartat utilitzarem el mateix programa d'exemple que fa la suma de dues matrius. L'exemple següent mostra el codi corresponent a la implementació seqüencial en C estàndard de la suma de matrius per a arquitectura de tipus CPU.

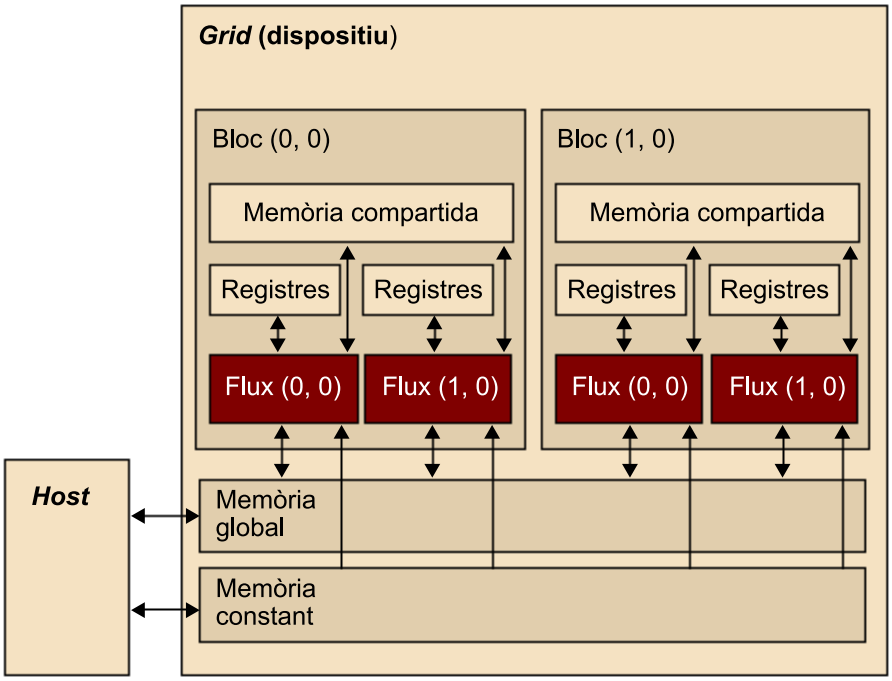
```
void matrix_add_cpu (float *A, float *B, float *C, int N)
{
   int i, j, index;
   for (i=0; i<N; i++){
      for (j=0; j<N; j++){
         index = i+j*N;
         C[index] = A[index] + B[index];
      }
   }
}

int main(){
   matrix_add_cpu(a, b, c, N);
}
```

La figura 16 mostra el model de memòria de CUDA exposat al programador en termes d'assignació, transferència i utilització dels diferents tipus de memòria del dispositiu. En la part inferior de la figura es troben les memòries de tipus global i constant. Aquestes memòries són aquelles a les quals el *host* pot transferir dades de manera bidireccional, per *grid*. Des del punt de vista del dispositiu, es pot accedir a diferents tipus de memòria amb els modes següents:

- Accés de lectura/escriptura a la memòria global, per *grid*.
- Accés només de lectura a la memòria constant, per *grid*.
- Accés de lectura/escriptura als registres, per flux.
- Accés de lectura/escriptura a la memòria local, per flux.
- Accés de lectura/escriptura a la memòria compartida, per bloc.
- Accés de lectura/escriptura.

Figura 16. Model de memòria de CUDA



La interfície de programació per a assignar i alliberar memòria global del dispositiu consisteix en dues funcions bàsiques: `cudaMalloc()` i `cudaFree()`. La funció `cudaMalloc()` es pot cridar des del codi del *host* per a assignar un espai de memòria global per a un objecte. Com haureu observat, aquesta funció és molt similar a la funció `malloc()` de la biblioteca de C estàndard, ja que CUDA és una extensió del llenguatge C i cerca mantenir les interfícies tan similars a les originals com sigui possible.

La funció `cudaMalloc()` té dos paràmetres. El primer paràmetre és l'adreça del punter cap a l'objecte un cop s'hagi assignat l'espai de memòria. Aquest punter és genèric i no depèn de cap tipus d'objecte, per tant s'haurà de fer *cast* a tipus (`void **`). El segon paràmetre és la mida de l'objecte que es vol assignar en bytes. La funció `cudaFree()` allibera l'espai de memòria de l'objecte indicat com a paràmetre de la memòria global del dispositiu. El codi següent

mostra un exemple de com s'utilitzen aquestes dues funcions. Després de fer el `cudaMalloc(), Matriu` apunta a una regió de la memòria global del dispositiu que s'hi ha assignat.

```
float *Matriu;
int mida = AMPLADA * LLARGÀRIA * sizeof(float);
cudaMalloc((void **) &Matriu, mida);
...
cudaFree(Matriu);
```

Un cop que un programa ha assignat memòria global del dispositiu per als objectes o estructures de dades del programa, es poden transferir les dades que faran falta per a la computació des del *host* cap al dispositiu. Això es fa mitjançant la funció `cudaMemcpy()`, que permet transferir dades entre memòries. Cal tenir en compte que la transferència és asíncrona.

La funció `cudaMemcpy()` té quatre paràmetres. El primer és un punter en l'adreça de destinació en què s'han de copiar les dades. El segon paràmetre apunta a les dades que s'han de copiar. El tercer paràmetre especifica el nombre de bytes que s'han de copiar. Finalment, el quart paràmetre indica el tipus de memòria involucrat en la còpia, que pot ser un dels següents:

- `cudaMemcpyHostToHost`: de la memòria del *host* cap a la memòria del mateix *host*.
- `cudaMemcpyHostToDevice`: de la memòria del *host* cap a la memòria del dispositiu.
- `cudaMemcpyDeviceToHost`: de la memòria del dispositiu cap a la memòria del *host*.
- `cudaMemcpyDeviceToDevice`: de la memòria del dispositiu cap a la memòria del dispositiu.

Cal tenir en compte que aquesta funció es pot utilitzar per a copiar dades de la memòria d'un mateix dispositiu però no entre diferents dispositius. A continuació es mostra un exemple de transferència de dades entre *host* i dispositiu basat en l'exemple de la figura 15:

```
void matrix_add (fload *A, float *B, float *C, int N)
{
   int size = N * N * sizeof(float);
   float * Ad, Bd, Cd;

   // 1. Reservar memòria per a les matrius
   cudaMalloc((void **) &Ad, size);
   cudaMalloc((void **) &Bd, size);
   cudaMalloc((void **) &Cd, size);
   // 2. Copiar matrius d'entrada al dispositiu
   cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    ...
    // 4. Copiar resultat C cap al host
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    ...
```

A més dels mecanismes d'assignació de memòria i transferència de dades, CUDA també suporta diferents tipus de variables. Els diferents tipus de variables que utilitzen els diversos tipus de memòria són utilitzades en diversos àmbits i tindran cicles de vida diferents, com resumeix la taula 2.

Taula 2. Diferents tipus de variables de CUDA

Declaració variables	Tipus de memòria	Àmbit	Cicle de vida
Per defecte (diferents de vectors)	Registre	Flux	Nucli
Vectors per defecte	Local	Flux	Nucli
__device__, __shared__, int SharedVar;	Compartida	Bloc	Nucli
__device__, int GlobalVar;	Global	*Grid*	Aplicació
__device__, __constant__, int ConstVar;	Constant	*Grid*	Aplicació

2.1.4. Definició de nuclis

El codi que s'executa en el dispositiu (nucli) és la funció que executen els diferents fluxos durant la fase paral·lela, cadascun en el rang de dades que li correspon. Cal recordar que la CUDA segueix el model SPMD i, per tant, tots els fluxos executen el mateix codi. A continuació es mostra la funció o nucli de la suma de matrius i la seva crida.

```
__global__ matrix_add_gpu (fload *A, float *B, float *C, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j*N;
   if (i<N && j<N){
         C[index] = A[index] + B[index];
   }
}

int main(){
   dim3 dimBlock(blocksize, blocksize);
   dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
   matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

```
}
```

Podem observar la utilització de la paraula clau específica de la CUDA `__global__` davant de la declaració de `matrix_add_gpu()`. Aquesta paraula clau indica que aquesta funció és un nucli i que cal cridar-lo des del *host* per a generar el *grid* de fluxos que executarà el nucli en el dispositiu. A més de `__global__` hi ha dues altres paraules clau que es poden utilitzar davant de la declaració d'una funció:

- La paraula clau `__device__` indica que la funció declarada és una funció CUDA de dispositiu. Una funció de dispositiu s'executa únicament en un dispositiu CUDA i només es pot cridar des d'un nucli o des d'una altra funció de dispositiu. Aquestes funcions no poden tenir ni crides recursives ni crides indirectes a funcions mitjançant punters.

- La paraula clau `__host__` indica que la funció és una funció de *host*, és a dir, una funció simple de C que s'executa en el *host* i, per tant, que pot ser cridada des de qualsevol funció de *host*. Per defecte totes les funcions en un programa CUDA són funcions de *host* si és que no s'especifica cap paraula clau en la definició de la funció.

Les paraules clau `__host__` i `__device__` es poden utilitzar simultàniament en la declaració d'una funció. Aquesta combinació fa que el compilador generi dues versions de la mateixa funció: una que s'executa en el *host* i que només es pot cridar des d'una funció de *host*, i una altra que s'executa en el dispositiu i que només es pot cridar des del dispositiu o funció de nucli.

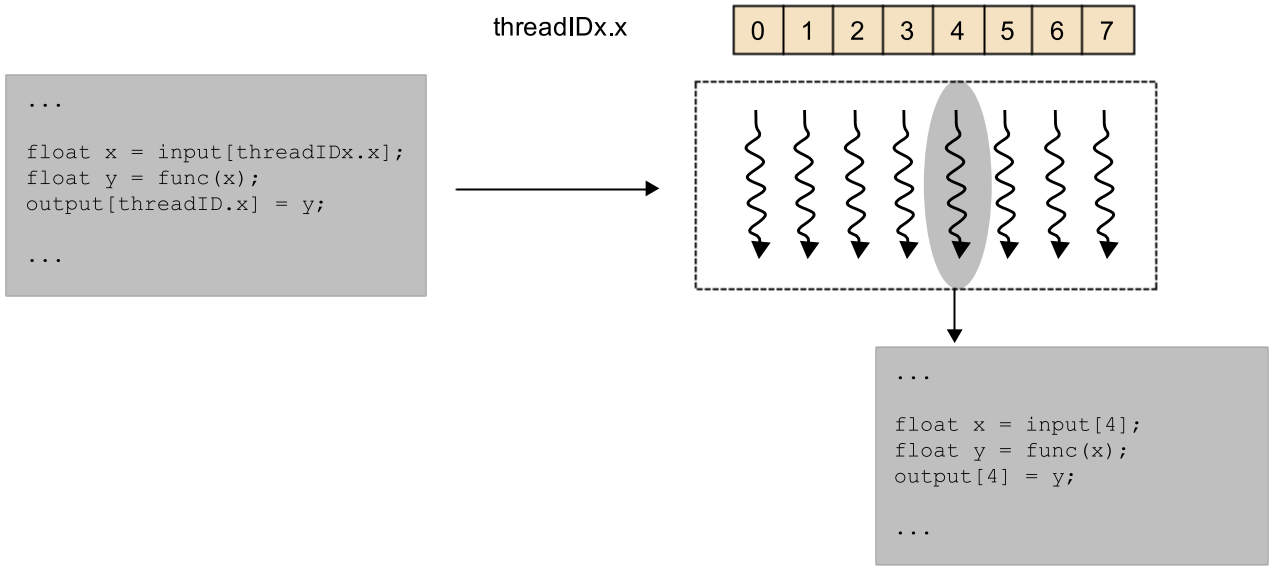
2.1.5. Organització de fluxos

En la CUDA, un nucli s'executa mitjançant un conjunt de fluxos (per exemple, un vector o una matriu de fluxos). Com que tots els fluxos executen el mateix nucli (model SIMT[11]) es necessita un mecanisme que permeti diferenciar-los, i així poder assignar la part corresponent de les dades a cada flux d'execució. La CUDA incorpora paraules clau per a fer referència a l'índex d'un flux (per exemple `threadIdx.x` i `threadIDx.y` si tenim en compte dues dimensions).

[11] De l'anglès *single instruction multiple threads*.

La figura 17 mostra com el nucli fa referència a l'identificador de flux i que durant la seva execució, a cadascun del fluxos l'identificador es substitueix pel valor que li correspon. Per tant, les variables `threadIdx.x` i `threadIdx.y` tindran diferents valors per a cadascun dels fluxos d'execució. Noteu que les coordenades reflecteixen l'organització multidimensional dels fluxos d'execució tot i que l'exemple de la figura 17 només fa referència a una dimensió (`threadIdx.x`).

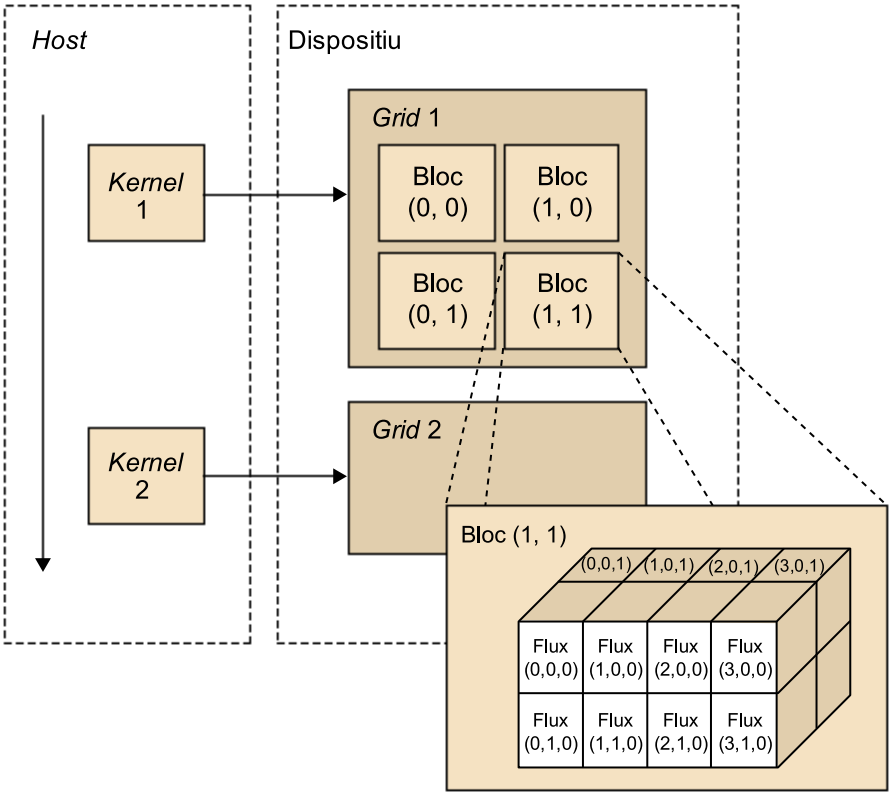
Figura 17. Execució d'un nucli CUDA en un vector de fluxos



Normalment, els *grid* que s'utilitzen en CUDA estan formats per molts fluxos (de prop de milers o fins i tot de milions de fluxos). Els fluxos d'un *grid* estan organitzats en una jerarquia de dos nivells, com es pot veure en la figura 18. També es pot observar com el `kernel 1` crea el `Grid 1` per a la seva execució. El nivell superior d'un *grid* consisteix en un o més blocs de fluxos. Tots els blocs d'un *grid* tenen el mateix nombre de fluxos i han d'estar organitzats de la mateixa manera. En la figura 18, el `Grid 1` consta de quatre blocs i està organitzat com una matriu de 2x2 blocs.

Cada bloc d'un *grid* té una coordenada única en un espai de dues dimensions mitjançant les paraules clau `blockIdx.x` i `blockIdx.y`. Els blocs s'organitzen en un espai de tres dimensions amb un màxim de 512 fluxos d'execució. Les coordenades dels fluxos d'execució en un bloc s'identifiquen amb tres índexs (`threadIdx.x`, `threadIdx.y` i `threadIdx.z`) tot i que no totes les aplicacions necessiten utilitzar les tres dimensions de cada bloc de fluxos. En la figura 18, cada bloc està organitzat en un espai de 4x2x2 fluxos d'execució.

Figura 18. Exemple d'organització dels fluxos d'un grid en CUDA



Quan el *host* fa una crida a un nucli s'han d'especificar les dimensions del *grid* i dels blocs de fluxos mitjançant paràmetres de configuració. El primer paràmetre especifica les dimensions del *grid* en termes de nombre de blocs, i el segon especifica les dimensions de cada bloc en termes de nombre de fluxos. Tots dos paràmetres són de tipus dim3, que essencialment és una estructura de C amb tres camps (x, y, z) de tipus enter sense signe. Com que els *grids* són grups de blocs en dues dimensions, el tercer camp de paràmetres de configuració del *grid* s'ignora (qualsevol valor serà vàlid). A continuació teniu un exemple en què dues variables d'estructures de tipus dim3 defineixen el *grid* i els blocs de l'exemple de la figura 18:

```
// Configuració de les dimensions de grid i blocs
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

// Invocació del kernel (suma de matrius)
matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

La CUDA també ofereix un mecanisme per a sincronitzar els fluxos d'un mateix bloc mitjançant la funció de tipus barrera `__syncthreads()`. Quan es crida la funció `__syncthreads()`, el flux que l'executa quedarà bloquejat fins que tots els fluxos del seu bloc arribin a aquest mateix punt. Això serveix per a assegurar que tots els fluxos d'un bloc han completat una fase abans de passar a la següent.

2.2. OpenCL

L'OpenCL és una interfície estàndard, oberta, lliure i multiplataforma per a la programació paral·lela. La motivació principal per al desenvolupament de l'OpenCL va ser la necessitat de simplificar la tasca de programació portable i eficient de la creixent quantitat de plataformes heterogènies, com ara la CPU multinucli, la GPU o fins i tot sistemes encastats. L'OpenCL va ser concebuda per Apple tot i que la va acabar desenvolupant el grup Khronos, que és el mateix que va impulsar l'OpenGL i n'és responsable.

L'OpenCL consisteix en tres parts: l'especificació d'un llenguatge multiplataforma, una interfície per a l'entorn de computació i una interfície per a coordinar la computació paral·lela entre processadors heterogenis. L'OpenCL utilitza un subconjunt de C99 amb extensions per al paral·lelisme i utilitza l'estàndard de representació numèrica IEEE 754 per a garantir la interoperabilitat entre plataformes.

Hi ha moltes similituds entre l'OpenCL i la CUDA tot i que l'OpenCL té un model de gestió de recursos més complex, ja que suporta múltiples plataformes i portabilitat entre diferents fabricants. L'OpenCL suporta models de paral·lelisme tant per a dades com per a tasques. En aquest subapartat ens centrarem en el model de paral·lelisme per a dades, el qual és equivalent al de la CUDA.

2.2.1. Model de paral·lelisme per a dades

De la mateixa manera que en la CUDA, un programa en l'OpenCL està format per dues parts: els nuclis que s'executen en un o més dispositius, i un programa *host* que invoca i controla l'execució dels nuclis. Quan s'invoca un nucli, el codi s'executa en feines elementals[12] que corresponen als fluxos de la CUDA. Les feines elementals i les dades associades a cada feina elemental es defineixen a partir del rang d'un espai d'índexs N-dimensional[13]. Les feines elementals formen grups de feines[14] que corresponen al blocs de la CUDA. Les feines elementals tenen un identificador global que és únic. A més, els grups de feines elementals s'identifiquen dins del rang *N*-dimensional i, per a cada grup, cadascuna de les feines elementals té un identificador local que anirà des de 0 fins a la mida del grup-1. Per tant, la combinació de l'identificador del grup i de l'identificador local dintre del grup també identifica de manera única una feina elemental. La taula 3 resumeix algunes de les equivalències entre l'OpenCL i la CUDA.

(12) En anglès, *work items*.

(13) En anglès, *NDRanges*.

(14) En anglès, *work groups*.

Taula 3. Algunes correspondències entre OpenCL i CUDA

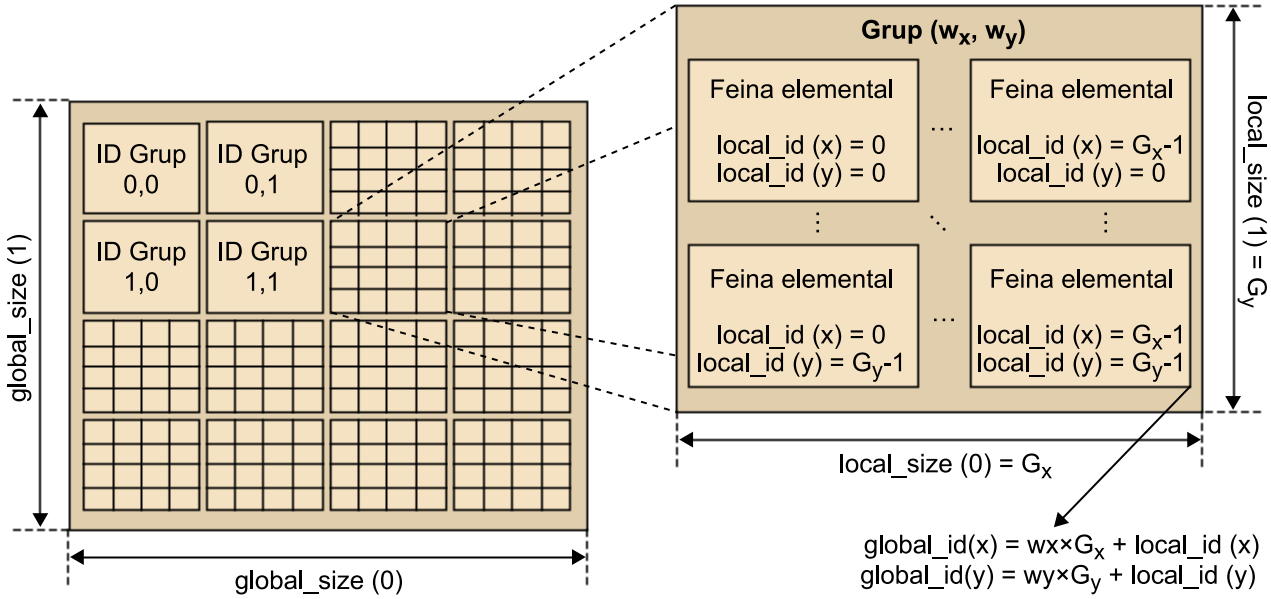
OpenCL	CUDA
Nucli	Nucli
Programa *host*	Programa *host*

OpenCL	CUDA
NDRange (rang N-dimensional)	*Grid*
Feina elemental (*work item*)	Flux
Grup de feines (*work group*)	Bloc
`get_global_id(0);`	`blockIdx.x * blockDim.x + threadIdx.x`
`get_local_id(0);`	`threadIdx.x`
`get_global_size(0);`	`gridDim.x*blockDim.x`
`get_local_size(0);`	`blockDim.x`

La figura 19 mostra el model de paral·lelisme per a dades de l'OpenCL. El rang N-dimensional (equivalent al *grid* en la CUDA) conté les feines elementals. En l'exemple de la figura, el nucli utilitza un rang de dues dimensions. Mentre que en la CUDA cada flux té uns valors de `blockIdx` i `threadIdx` que es combinen per a identificar el flux, en l'OpenCL disposem d'interfícies per a identificar les feines elementals de les dues maneres que hem vist. Per una banda, la funció `get_global_id()`, donada una dimensió, retorna l'identificador únic de feina elemental en la dimensió especificada. En l'exemple de la figura, les crides `get_global_id(0)` i `get_global_id(1)` retornen l'índex de les feines elementals en les dimensions *X* i *Y*, respectivament. Per l'altra banda, la funció `get_local_id()`, donada una dimensió, retorna l'identificador de la feina elemental dins del seu grup en la dimensió especificada. Per exemple, `get_local_id(0)` és equivalent a `threadIdx.x` en la CUDA.

L'OpenCL també disposa de les funcions `get_global_size()` i `get_local_size()` que, donada una dimensió, retornen la quantitat total de feines elementals i la quantitat de feines elementals dintre d'un grup en la dimensió especificada, respectivament. Per exemple, `get_global_size(0)` retorna la quantitat de feines elementals que és equivalent a `gridDim.x*blockDim.x` en la CUDA.

Figura 19. Exemple de rang N-dimensional



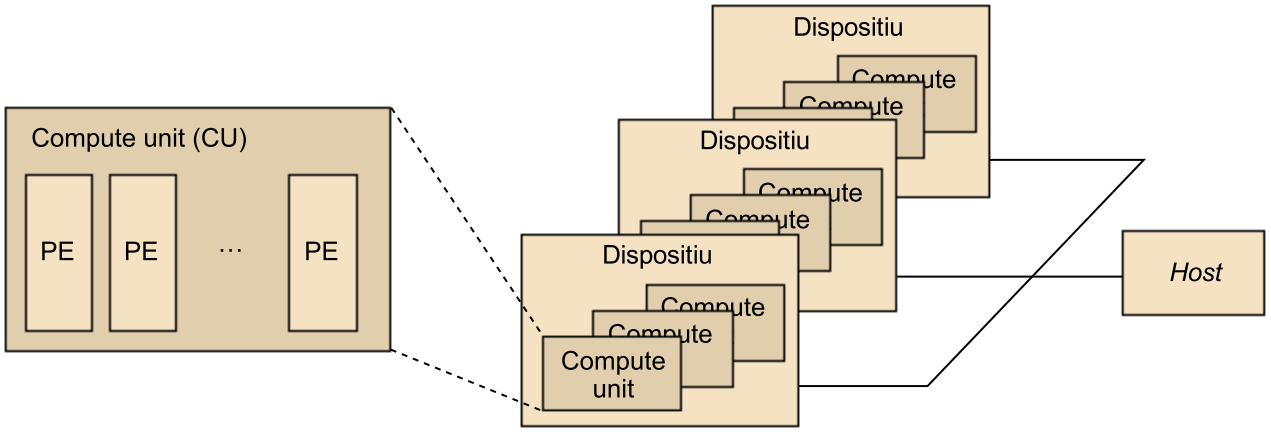
Es poden observar les feines elementals, els grups que formen i identificadors associats.

Les feines elementals dintre del mateix grup es poden sincronitzar utilitzant barreres que són equivalents a `__syncthreads()` de la CUDA. En canvi, les feines elementals de diferents grups no es poden sincronitzar, excepte si és per la terminació del nucli o la invocació d'un de nou.

2.2.2. Arquitectura conceptual

La figura 20 mostra l'arquitectura conceptual de l'OpenCL, la qual està formada per un *host* (típicament una CPU que executa el programa *host*) connectat a un o més dispositius OpenCL. Un dispositiu OpenCL està compost per una o més *compute units* (CU) que corresponen als SM de la CUDA. Finalment, un CU està format per un o més *processing elements* (PE) que corresponen als SP de la CUDA. El programa s'acabarà executant en els PE.

Figura 20. Arquitectura conceptual de l'OpenCL



2.2.3. Model de memòria

En un dispositiu OpenCL hi ha disponible una jerarquia de memòria que inclou tipus de memòria diferents, tal com mostra la figura 21: global, constant, local i privada.

- La memòria global és la que es pot utilitzar per a totes les unitats de càlcul d'un dispositiu.

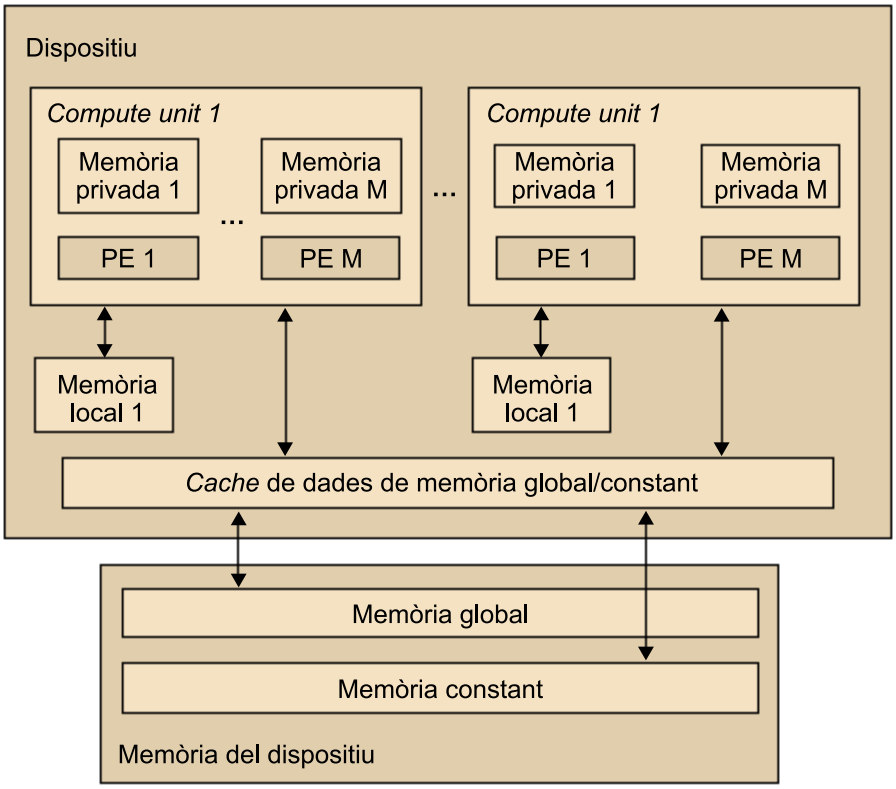
- La memòria constant és la memòria que es pot utilitzar per a emmagatzemar dades constants per a accés només de lectura per a totes les unitats de càlcul d'un dispositiu durant l'execució d'un nucli. El processador *host* és responsable d'assignar i iniciar els objectes de memòria que resideixen en l'espai de memòria.

- La memòria local és la memòria que es pot utilitzar per a les feines elementals d'un grup.

- La memòria privada és la que es pot utilitzar únicament per a una unitat de càlcul única. Això és similar als registres en una única unitat de càlcul o un únic nucli d'una CPU.

Així doncs, tant la memòria global com la constant corresponen als tipus de memòria amb el mateix nom de la CUDA, la memòria local correspon a la memòria compartida de la CUDA i la memòria privada correspon a la memòria local de la CUDA.

Figura 21. Arquitectura i jerarquia de memòria d'un dispositiu OpenCL



Per a crear i manejar objectes en la memòria d'un dispositiu, l'aplicació que s'executa en el *host* utilitza la interfície d'OpenCL, ja que els espais de memòria del *host* i dels dispositius són principalment independents l'un de l'altre. La interacció entre l'espai de memòria del *host* i dels dispositius pot ser de dos tipus: copiant dades explícitament o bé mapant/desmapant regions d'un objecte OpenCL en la memòria. Per copiar dades explícitament, el *host* posa ordres a la cua per a transferir dades entre la memòria de l'objecte i la memòria del *host*. Aquestes ordres de transferència poden ser o bé bloquejants o bé no bloquejants. Quan s'utilitza una crida bloquejant es pot accedir a les dades des del *host* amb seguretat, un cop la crida ha finalitzat. En canvi, per a fer una transferència no bloquejant, la crida a la funció d'OpenCL finalitza immediatament i es desenvia tot i que la memòria des del *host* no és segura per a utilitzar. La interacció mapant/desmapant objectes en la memòria permet que el *host* pugui tenir en el seu espai de memòria una regió corresponent a objectes OpenCL. Les ordres de mapeig/desmapeig també poden ser bloquejants o no bloquejants.

2.2.4. Gestió de nuclis i de dispositius

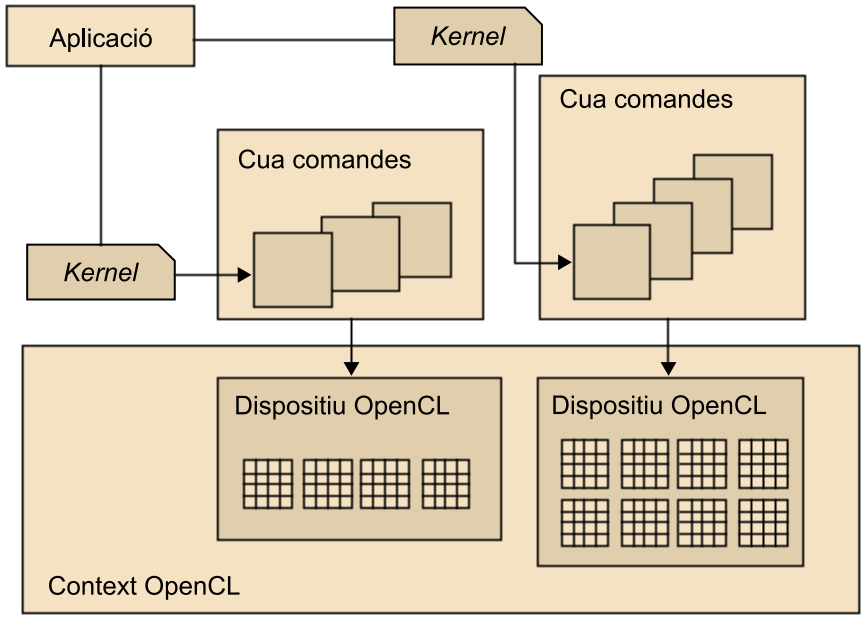
Els nuclis de l'OpenCL tenen la mateixa estructura que els de la CUDA. Per tant, són funcions que es declaren començant amb la paraula clau `__kernel`, que és equivalent a `__global` de la CUDA. A continuació es mostra la implementació del nostre exemple de suma de matrius en OpenCL. Fixeu-vos que els arguments del nucli corresponents a les matrius estan declarats com a `__global`, ja que es troben en la memòria global, i les dues matrius d'entrada també

estan declarades com a const, ja que només caldrà fer accessos en la modalitat de lectura. En la implementació del cos del nucli s'utilitza `get_global_id()` en les dues dimensions de la matriu per tal de definir l'índex associat. Aquest índex s'utilitza per a seleccionar les dades que corresponen a cadascuna de les feines elementals que instanciï el nucli.

```
__kernel void matrix_add_opencl (   __global const float *A,
              __global const float *B,
              __global float *C,
              int N) {
  int i = get_global_id(0);
  int j = get_global_id(1);
  int index = i + j*N;
  if (i<N && j<N){
        C[index] = A[index] + B[index];
  }
}
```

El model de gestió de dispositius de l'OpenCL és molt més sofisticat que el de la CUDA, ja que l'OpenCL permet abstraure diferents plataformes maquinari. Els dispositius es gestionen mitjançant contextos. Com mostra la figura 22, per a gestionar un o més dispositius (dos en l'exemple de la figura), primer cal crear un context que contingui els dispositius mitjançant o bé la funció `clCreateContext()` o bé la funció `clCreateContextFromType()`. Normalment cal indicar a les funcions `CreateContext` la quantitat i el tipus de dispositius del sistema mitjançant la funció `clGetDeviceIDs()`. Per a executar qualsevol tasca en un dispositiu, primer cal crear una cua d'ordres per al dispositiu mitjançant la funció `clCreateCommandQueue()`. Un cop s'ha creat una cua per al dispositiu, el codi que s'executa en el *host* pot inserir-hi un nucli i els paràmetres de configuració associats. Un cop el dispositiu estigui disponible per a executar el nucli següent de la cua, aquest nucli s'eliminarà de la cua i passarà a ser executat.

Figura 22. Gestió de dispositius en OpenCL mitjançant contextos



A continuació es mostra el codi corresponent al *host* per a executar la suma de matrius mitjançant el nucli de l'exemple de codi anterior. Suposem que la definició i inicialització de variables s'ha dut a terme correctament i que la funció del nucli `matrix_add_opencl()` està disponible. En el primer pas es crea un context i, un cop s'han obtingut els dispositius disponibles, es crea una cua d'ordres que utilitzarà el primer dispositiu disponible dels que hi hagi en el sistema. En el segon pas es defineixen els objectes que ens caldran en la memòria (les tres matrius A, B i C). Les matrius A i B es defineixen de lectura, la matriu C es defineix d'escriptura. Noteu que en la definició de les matrius A i B s'utilitza l'opció `CL_MEM_COPY_HOST_PTR`, que indica que les dades de les matrius A i B es copiaran dels punters especificats (`srcA` i `srcB`). En el tercer pas es defineix el nucli que s'executarà posteriorment mitjançant `clCreateKernel` i també s'especifiquen els arguments de la funció `matrix_add_opencl`. A continuació s'envia el nucli a la cua d'ordres definida prèviament i, finalment, es llegeixen els resultats de l'espai de memòria corresponent a la matriu C per mitjà del punter `dstC`. En aquest exemple no hem entrat en detall en molts dels paràmetres de les diferents funcions de la interfície de l'OpenCL, per tant es recomana repassar l'especificació d'aquests que està disponible en la pàgina web http://www.khronos.org/opencl/.

```
main(){
  // Inicialització de variables, etc.
  (...)

  // 1. Creació del context i cua al dispositiu
  cl_context context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
  // Per obtenir la llista de dispositius GPU associats al context
  size_t cb;
  clGetContextInfo( context,   CL_CONTEXT_DEVICES,   0,   NULL, &cb);
  cl_device_id *devices = malloc(cb);
```

```
clGetContextInfo( context,   CL_CONTEXT_DEVICES,   cb,   devices,   NULL);
cl_cmd_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0 , NULL);

// 2. Definició dels objectes a memòria (matrius A, B i C)
cl_mem memobjs[3];
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
               sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
              sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);

// 3. Definició del kernel i arguments
cl_program program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "matrix_add_opencl", NULL);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memobjs[2]);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&N);

// 4. Invocació del kernel
size_t global_work_size[1] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
      NULL, 0, NULL, NULL);

// 5. Lectura dels resultats (matriu C)
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
      dstC, 0, NULL, NULL);
(...)
```

En aquesta secció s'ha parlat d'OpenCL per tal d'accelerar aplicacions emprant les GPU. Ara bé, durant els darrers anys Intel ha proposat també emprar OpenCL per tal de programar acceleradors de tipus FPGA.

Activitat

Us proposem identificar similituds i diferències entre l'ús d'OpenCL per a GPU i FPGA.

3. Models de programació per memòria distribuïda

En aquest apartat ens centrarem en els models principals de programació per a memòria distribuïda, MPI, que és l'estàndard *de facto* des de fa anys, i els models PGAS, que intenten donar solució a reptes importants que presenten les característiques dels futurs sistemes d'altres prestacions.

3.1. MPI

L'MPI[15] és una biblioteca de pas de missatges que es pot utilitzar tant en programes C com Fortran, des dels quals es fan crides a funcions de l'MPI per a la gestió i comunicació de processos.

(15)De l'anglès *message passing interface*.

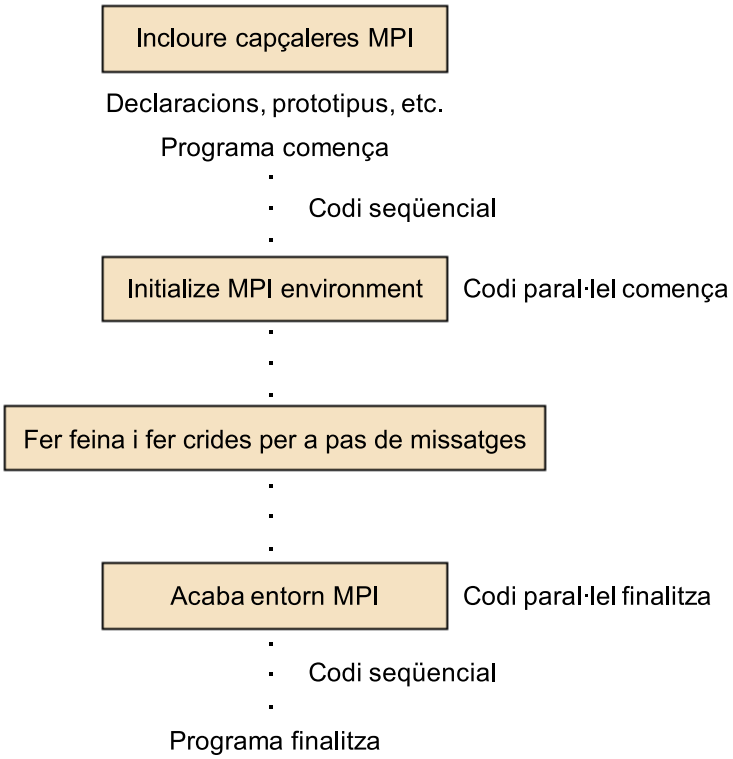
Vegeu també

La biblioteca MPI s'ha vist en el mòdul "Introducció a la computació d'altes prestacions" d'aquesta assignatura.

La figura 23 mostra l'estructura general d'un programa MPI en què es pot apreciar com inicialment hi ha un únic procés que executa el codi de manera seqüencial fins que s'inicialitza l'entorn MPI, que és quan el codi paral·lel comença. En la secció de codi paral·lel es farà la tasca concreta i s'utilitzarà pas de missatges per a comunicar els diferents processos MPI. Finalment, un cop acabat el treball que es vol executar en paral·lel, s'indica la finalització del codi paral·lel i es torna a executar la resta del codi de manera seqüencial fins que acaba l'execució del programa.

Cal tenir present que, a diferència de l'OpenMP, el nombre de processos no es pot establir en temps d'execució, sinó que s'especifica quan es vol executar el programa tal com veurem més endavant. També cal recordar que tots els processos MPI executaran exactament el mateix codi.

Figura 23. Estructura general d'un programa MPI



A continuació es mostra l'exemple bàsic de programa MPI (variant del típic "hello world") en què es pot observar l'estructura general d'un programa MPI. Aquesta és una versió similar a la que s'ha vist en el primer mòdul de l'assignatura però exemplificant-ne la utilització en Fortran.

```
    program hello

    use fmpi
!   include "mpif.h"
    call MPI_INIT( ierr )
    call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
    call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

    write (*,*) "Hello from ",myid
    write (*,*) "Numprocs is ",numprocs

    call MPI_FINALIZE(ierr)
    stop
    end
```

Una de les maneres més habituals d'utilitzar l'MPI és mitjançant el model SPMD. En aquest cas caldrà diferenciar el codi que executa el procés mestre (típicament el procés amb rang igual a zero) del que executen els processos esclaus, tal com es mostra a continuació.

```
main (int argc, char *argv[])
```

```
{
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // Consulta el rang del procés

   if (myrank == 0)
      master(); // codi que executa el procés mestre
   else
      slave();  // codi que executen els processos esclaus

   MPI_Finalize();
}
```

3.1.1. Comunicadors

Un comunicador permet definir una sèrie de processos entre els quals es poden realitzar comunicacions. Cada procés pot estar en diferents comunicadors i tindrà un identificador en cadascun (també anomenat *rang*), estant els identificadors entre 0 i el nombre de processos en el comunicador menys 1. En posar-se en marxa l'MPI, tots els processos són a `MPI_COMM_WORLD`, la qual cosa és una constant que identifica un comunicador que inclou a tots els processos.

A part de `MPI_COMM_WORLD`, també es poden definir comunicadors amb un nombre inferior de processos, per exemple per a comunicar les dades en cadascuna de les files de processos en una graella, amb la qual cosa es pot fer *broadcast* en les diferents files de processos sense que les comunicacions dels uns afectin els altres. Tot i això, també es poden fer servir en les comunicacions punt a punt.

Hi ha dos tipus de comunicadors: els intracomunicadors i els intercomunicadors. Els intracomunicadors s'utilitzen per a enviar missatges entre els processos en aquest comunicador, i els intercomunicadors s'utilitzen per a enviar missatges entre processos en diferents comunicadors. Nosaltres ens centrem en comunicacions entre processos d'un mateix comunicador. Les comunicacions entre processos en comunicadors diferents poden tenir sentit si en dissenyar biblioteques es creen comunicadors i un usuari de la biblioteca vol comunicar un procés del seu programa amb un altre de la biblioteca MPI.

Un comunicador consta d'un grup, que és una col·lecció ordenada de processos als quals s'associen identificadors, i un context, que és un identificador que associa el sistema al grup. Addicionalment, a un comunicador s'hi pot associar una topologia virtual.

Activitat

Exploreu com es poden definir topologies virtuals de comunicadors i quina en pot ser la utilitat.

En l'MPI hi ha dos tipus bàsics de comunicacions: les comunicacions punt a punt, que involucren dos processos que intercanvien missatges només entre ells, i les comunicacions col·lectives, que involucren un conjunt de processos que realitzen tots ells una tasca concreta col·lectivament.

3.1.2. Comunicacions punt a punt

En l'exemple anterior els processos no interaccionen. El normal és que els processos no treballin de manera independent, sinó que intercanviïn informació per mitjà de pas de missatges. Per a enviar missatges entre dos processos (un d'origen i un de destinació) s'utilitzen les comunicacions punt a punt. Més endavant es mostra un fragment de codi MPI en què el procés amb rang 0 li envia l'enter *x* al procés amb rang 1. El programa mostra la manera normal de treballar amb pas de missatges. S'executa el mateix programa en tots els processos, però processos diferents executen parts diferents del codi. En l'exemple, s'utilitzen funcions `MPI_Send` i `MPI_Recv` per a enviar i rebre missatges respectivament. Aquestes dues funcions són bloquejants i les seves definicions es mostren a continuació.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
                  int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
                  int source, int tag, MPI_Comm comm,
                  MPI_Status *status)
```

Els paràmetres es descriuen a continuació:

- `buf` conté l'inici de la zona de memòria de la qual s'agafaran les dades a enviar o bé on s'emmagatzemaran les dades que es reben.

- `count` indica el nombre de dades a enviar o l'espai disponible per a rebre'n (no el nombre de dades del missatge que es rep, ja que la mida la determina qui envia).

- `datatype` és el tipus de dades a transferir, i ha de ser un tipus `MPI (MPI_Datatype)`, per exemple, `MPI_CHAR` o `MPI_INT`.

- `dest` i `source` són els identificadors dels processos als quals s'envia i dels quals es rep el missatge respectivament. Es pot utilitzar la constant `MPI_ANY_SOURCE` per a indicar que es pot rebre des de qualsevol procés.

- `tag` s'utilitza per a diferenciar entre missatges, i ha de coincidir en el procés que envia i el que rep. Es pot utilitzar `MPI_ANY_TAG` per a indicar que el missatge és compatible amb missatges amb qualsevol identificador.

- `comm` és el comunicador dins del qual es fa la comunicació. És del tipus MPI `MPI_Comm`, i en l'exemple s'utilitza l'identificador de comunicador format per a tots els processos (`MPI_COMM_WORLD`).

- `status` referencia una variable de tipus `MPI_Status`. En el programa no s'utilitza, però conté informació del missatge que s'ha rebut, i es pot consultar per a identificar alguna característica del missatge, per exemple, la seva longitud, el procés origen, etc.

A continuació es mostra un fragment de codi MPI en què el procés amb rang 0 li envia l'enter *x* al procés amb rang 1:

```
 MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
   if (myrank == 0) {
         int x;
         MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
   }
   else if (myrank == 1) {
         int x;
         MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD,status);
}
```

En els missatges s'han d'utilitzar tipus de dades MPI. Per raons de portabilitat, l'MPI redefineix els seus tipus de dades elementals. La taula següent mostra els tipus estàndard per a C i Fortan:

Taula 4. Tipus de dades MPI per a C i Fortran

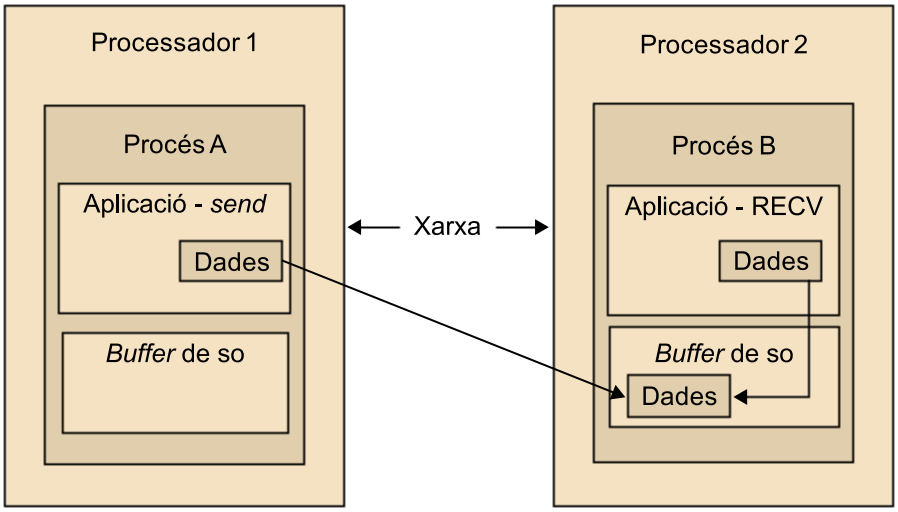
Tipus de dades C		**Tipus de dades Fortran**	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_WCHAR	wchar_t wide character		
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER MPI_INTEGER1 MPI_INTEGER2 MPI_INTERGER4	integer integer*1 integer*2 integer*4
MPI_LONG	signed		
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int		
MPI_SIGNED_CHAR	Signed char		
MPI_UNSIGNED_CHAR	Unsigned char		
MPI_UNSIGNED_SHORT	Unsigned short int		
MPI_UNISGNED	Unsigned int		
MPI_SIGNED_CHAR	Signed char		

Tipus de dades C		Tipus de dades Fortran	
MPI_UNSIGNED_CHAR	Unsigned char		
MPI_FLOAT	Float	MPI_REAL MPI_REAL2 MPI_REAL4 MPI_REAL8	Real Real*2 Real*4 Real*8
MPI_DOUBLE	Double	MPI_DOUBLE_PRECISION	Double precision
MPI_LONG_DOUBLE	Long double		
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	Float_Complex	MPI_COMPLEX	Complex
MPI_C_DOUBLE COMPLEX	Double_Complex	MPI_DOUBLE_COMPLEX	Double complex
MPI_C_LONG_DOUBLE_COMPLEX	Long double_Complex		
MPI_C_BOOL	_Bool	MPI_LOGICAL	Logical
MPI_C_LONG_DOUBLE_COMPLEX	Long double_Complex		
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	Int_8_t Int_16_t Int_32_t Int_64_t		
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	Uint_8_t uint_16_t uint_32_t uint_64_t		
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	Data packed or un-packed with MPI_PACK()/ MPI_UNPACK	MPI_PACKED	Data packed or un-packed with MPI_PACK()/ MPI_UNPACK

La comunicació que hem vist anteriorment mitjançant `MPI_Send` i `MPI_Recv` és de tipus bloquejant, ja que es fa una sincronització entre el procés que envia i el que rep el missatge. Amb `MPI_Send` i `MPI_Recv` el procés que envia el missatge es continua executant un cop acabat l'enviament, i el que rep, si quan sol·licita la recepció del missatge, aquest encara no ha arribat, es queda bloquejat fins que el missatge arribi.

L'MPI proporciona altres possibilitats per a enviaments bloquejants, per exemple `MPI_Ssend`, `MPI_Bsend` i `MPI_Rsend`. Aquestes funcions es diferencien en la manera de gestionar l'enviament, podent gestionar el *buffer* de comunicació o determinar que el procés prengui les dades directament de la memòria. En la figura 24 s'il·lustra el funcionament de pas de missatges mitjançant *buffer*.

Figura 24. Funcionament de pas de missatge a MPI mitjançant *buffer*.



La funció `MPI_Sendrecv` combina en una crida l'enviament i la recepció entre dos processos però continua essent una funció bloquejant.

L'MPI també proporciona comunicació no bloquejant, en què el procés receptor sol·licita el missatge i si aquest no ha arribat continua la seva execució. Les funcions són `MPI_Isend` i `MPI_Irecv`. Si el procés receptor no ha rebut el missatge pot arribar un moment en què per força hagi d'esperar-lo per a continuar l'execució. Les definicions d'aquestes dues funcions es mostren a continuació:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm,
       MPI_Status *status)
```

També es disposa de la funció `MPI_Wait()` per a esperar l'arribada d'un missatge, i de `MPI_Test()` per a comprovar si l'operació s'ha completat.

A continuació es mostra el mateix exemple del desenvolupament del fragment de codi MPI en què el procés amb rang 0 li envia l'enter *x* al procés amb rang 1, però ara utilitzant crides asíncrones en canvi de crides síncrones o bloquejants.

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
    if (myrank == 0) {
       int x;
       MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
       compute(); // Realitza algun càlcul mentre es realitza l'enviament
       MPI_Wait(req1, status);
    }
    else if (myrank == 1) {
```

```
        int x;
        MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

3.1.3. Comunicacions col·lectives

A més de les comunicacions punt a punt, l'MPI ofereix una sèrie de funcions per a dur a terme comunicacions en què intervenen tots els processos d'un comunicador. Sempre que sigui possible fer les comunicacions per mitjà d'aquestes comunicacions col·lectives, és convenient fer-ho així, ja que facilita la programació evitant possibles errors, i si les funcions estan optimitzades per al sistema on estem treballant és possible que es desenvolupin programes més eficients amb comunicacions col·lectives i no pas amb comunicacions punt a punt.

Algunes de les comunicacions col·lectives més útils es presenten a continuació:

a) Barrera:

```
int MPI_Barrier(MPI_Comm comm)
```

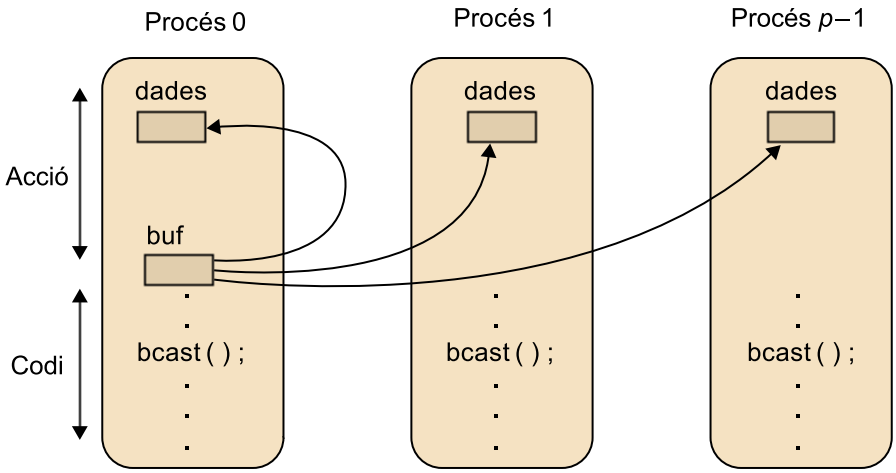
Estableix una barrera. Tots els processos esperen que tots arribin a la barrera, per a continuar l'execució un cop hi hagin arribat. S'utilitza un comunicador que estableix el grup de processos que s'estan sincronitzant. Totes les funcions de comunicacions col·lectives retornen un codi d'error.

b) *Broadcast*:

```
int MPI_Bcast(void *buffer, int count,
        MPI_Datatype datatype, int root, MPI_comm comm)
```

Du a terme una operació de *broadcast* (comunicació un a tots), en què s'envien `count` dades del tipus `datatype` des del procés arrel (`root`) a la resta de processos en el comunicador. Tots els processos que intervenen han de cridar la funció indicant el procés que actua com a arrel. En l'arrel, les dades que s'envien es prenen de la zona apuntada pel *buffer*, i els que reben emmagatzemen en la memòria reservada en el *buffer*. La figura 25 il·lustra el funcionament d'aquesta operació.

Figura 25. Funcionament de l'operació *broadcast* de l'MPI

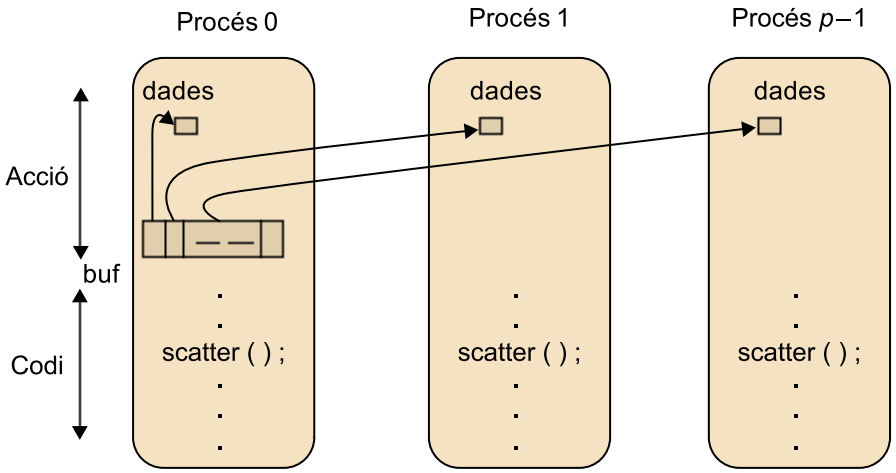


c) *Scatter*:

```
int MPI_Scatter (void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm)
```

Es pot utilitzar `MPI_Scatter` per a enviar des d'un procés missatges diferents a la resta de processos. En el procés arrel el missatge es divideix en segments de mida `sendcount`, i el segment i-èsim s'envia al procés `i`. Si es volen enviar blocs de mides diferents als diferents processos o si els blocs a enviar no estan seguits en la memòria, es pot utilitzar la funció `MPI_Scatterv`. La figura 26 il·lustra el funcionament del `MPI_Scatter`.

Figura 26. Funcionament de l'operació *scatter* de l'MPI



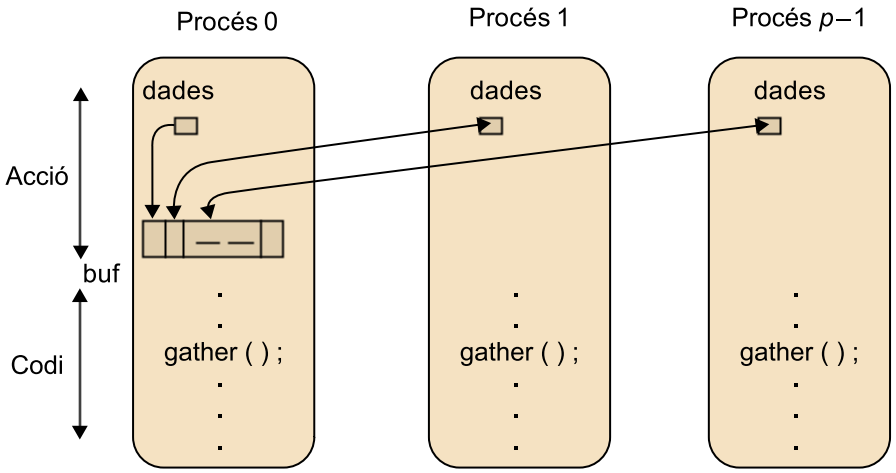
d) *Gather*:

```
int MPI_Gather(void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
```

```
            int root, MPI_Comm comm)
```

Aquesta és la funció inversa del `MPI_Scatter`. Tots els processos (inclosa l'arrel) envien al procés arrel `sendcount` dades des de `sendbuf`, i l'arrel les emmagatzema a `recvbuf` per l'ordre dels processos. Si es vol que cada procés enviï blocs de mides diferents s'ha d'utilitzar `MPI_Gatherv`. La figura 27 il·lustra el funcionament de `MPI_Gather`.

Figura 27. Funcionament de l'operació *gather* de l'MPI



Per a enviar blocs de dades de tots a tots els processos s'utilitza:

```
int MPI_Allgather (void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            MPI_Comm comm)
```

en què el bloc enviat per l'i-èsim procés s'emmagatzema com a bloc i-èsim en `recvbuf` en tots els processos. Per a enviar blocs de mides diferents s'utilitza `MPI_Allgatherv`.

Per a enviar blocs de dades diferents als diferents processos s'utilitza:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
            MPI_Datatype sendtype, void *recvbuf,
            int recvcount, MPI_Datatype recvtype,
            MPI_Comm comm)
```

en què, de cada procés i, el bloc j s'envia al procés j, que l'emmagatzema com a bloc i en `recvbuf`. Hi ha el corresponent `MPI_Alltoallv`.

e) Reducció:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op,
```

```
              int root, MPI_Comm comm)
```

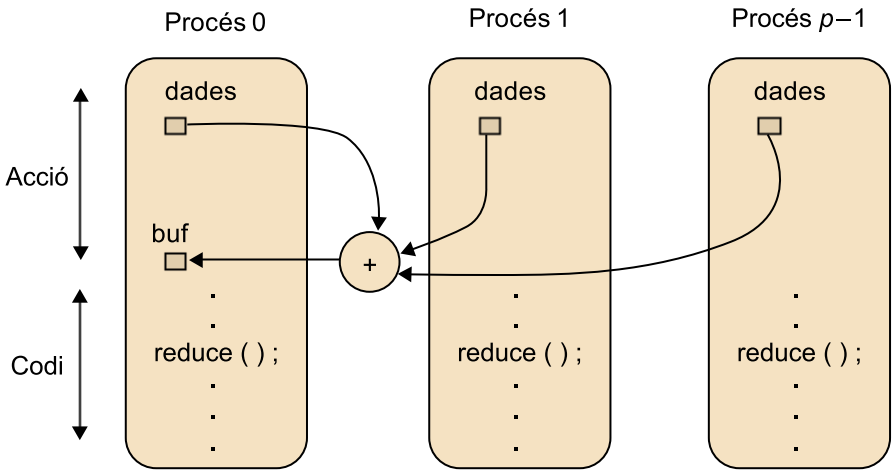
Du a terme una reducció de tots a un, és a dir, es recullen els valors distribuïts en els diferents processos i s'aplica una operació a tots ells. Això vol dir que les dades sobre les quals es farà la reducció es prenen de la zona apuntada per `sendbuf` i el resultat es deixa en l'apuntada per `recvbuf`. Si el nombre de dades (`count`) és superior a 1, l'operació es fa element a element en els del *buffer* per separat. El resultat es deixa en el procés `root`. L'operació que s'aplica a les dades és determinada per `op`. Les operacions que s'admeten per a fer reduccions es mostren en la taula següent:

Taula 5. Tipus de dades MPI per a C i Fortran

Operació	Significat	Tipus C permesos
MPI_MAX MPI_MIN MPI_SUM MPI_PROD	màxim mínim suma producte	Enters i punt flotant Enters i punt flotant Enters i punt flotant Enters i punt flotant
MPI_LAND MPI_LORD MPI_LXORD	AND lògic OR lògic XORD lògic	Enters Enters Enters
MPI_BAND MPI_BOR MPI_BXOR	AND bit a bit OR bit a bit XOR bit a bit	Enters i bytes Enters i bytes Enters i bytes
MPI_MAXLOC MPI_MINLOC	Màxim i localització Mínim i localització	Parelles de tipus Parelles de tipus

La figura 28 il·lustra el funcionament de `MPI_Reduce` i a continuació es mostra un exemple d'utilització d'aquesta operació en el qual s'utilitza per a sumar els valors d'un interval a partir de les sumes parcials de cadascun dels processos MPI i reducció.

Figura 28. Funcionament de l'operació de reducció de l'MPI




```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
```

```
{
   int MyProc, tag=1, size;
   char msg='A', msg_recpt ;
   MPI_Status *status ;
   int root ;
   int left, right, interval ;
   int number, start, end, sum, GrandTotal;
   int mystart, myend;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   root = 0;
   if (MyProc == root)      /* El procés arrel llegeix els límits de l'interval */
   {
      printf("Give the left and right limits of the interval\n");
      scanf("%d %d", &left, &right);
      printf("Proc root reporting : the limits are : %d %d\n", left, right);
   }
   MPI_Bcast(&left, 1, MPI_INT, root, MPI_COMM_WORLD); /*Bcast limits a tots*/
   MPI_Bcast(&right, 1, MPI_INT, root, MPI_COMM_WORLD);
   if (((right - left + 1) % size) != 0)
      interval = (right - left + 1) / size + 1 ;  /*Fixa límits locals de suma*/
   else
      interval = (right - left + 1) / size;
   mystart = left + MyProc*interval ;
   myend = mystart + interval ;
   /* estableix els límits dels intervals correctament */
   if (myend > right) myend = right + 1 ;
   sum = root;         /* Suma localment a cada procés MPI */
   if (mystart <= right)
      for (number = mystart; number < myend; number++) sum = sum + number ;
   /* Fa la reducció al procés arrel */
   MPI_Reduce(&sum, &GrandTotal, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD) ;
   MPI_Barrier(MPI_COMM_WORLD);
   /* El procés arrel retorna els resultat */
   if(MyProc == root)
      printf("Proc root reporting : Grand total = %d \n", GrandTotal);
   MPI_Finalize();
}
```

Quan tots els processos han de rebre el resultat de l'operació s'utilitza la funció:

```
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

3.1.4. Compilació i execució

En aquest primer exemple veurem com es compila i s'executa un codi MPI. La manera de fer-ho pot variar d'una implementació a una altra. Una manera normal de compilar és la següent (per MPICH, per exemple).

```
mpicc programa.c -o programa
```

en què `mpicc` crida el compilador de C que estigui establert i el munta amb la biblioteca de l'MPI. S'indica el codi a compilar i les opcions que tradicionalment es passen al compilador C.

Un cop generat el codi, la manera d'executar-lo també depèn de la compilació. El més habitual és utilitzar `mpirun` passant-hi el codi a executar i una sèrie d'arguments que indiquen els processos a posar en marxa i la distribució dels processos en els processadors. Per exemple, si es crida de la manera següent:

```
mpirun -np 4 programa
```

es posen en marxa quatre processos (`-np 4`) en què tots executen el mateix codi. No es diu a quins processadors s'assignen els processos, per la qual cosa s'utilitzarà l'assignació per defecte que estigui establerta en el sistema: pot ser que tots els processos s'assignin al mateix node (que pot ser monoprocessador o tenir varis nuclis) o que, si hi ha diferents nodes en el sistema, s'assigni un procés a cada node.

Hi ha diferents maneres d'indicar com s'assignen els processos en executar el programa. Per exemple, mitjançant:

```
mpirun -np 4 -machinefile nodes.txt programa
```

s'indica que els processos s'assignin segons s'indica en el fitxer `nodes.txt`, el qual assenyala els servidors que es poden fer servir per a executar els processos MPI. Es tracta d'una llista amb els noms dels servidors, com per exemple:

```
node1
node2
```

En aquest exemple, i si els estem executant en el `node0`, els quatre processos s'assignaran en l'ordre següent: el procés 0 al node en què estem executant el programa (`node0`), el procés 1 al `node1`, el 2 al `node2`, i una vegada s'hagi acabat la llista de màquines del fitxer es comença una altra vegada pel principi, de manera que el procés 3 s'assigna al `node1`. La forma d'assignació depèn de la implementació, i pot ser que no s'assigni cap procés al node on som o que aquest entri en l'assignació cíclica un cop acabada la llista de màquines del fitxer. Per exemple, amb l'opció `-nolocal` podem evitar que s'utilitzi el

servidor des del qual estem executant el programa (cosa que pot ser útil, per exemple, en el cas d'estar utilitzant un node *login* que no s'hauria de fer servir per a la computació).

3.2. Llenguatges PGAS

La programació de sistemes distribuïts moltes vegades s'ha equiparat amb el paradigma de pas de missatges. Tot i així, és possible implementar abstraccions de nivell més alt per sobre del sistema de memòria distribuïda. Durant l'última dècada, diferents grups de recerca han desenvolupat aquestes abstraccions, de la qual cosa ha resultat una família de llenguatges coneguts com a *llenguatges PGAS*[16].

(16)De l'anglès *partitioned global address space*.

L'espai d'adreces global fa referència al fet que el llenguatge proporciona un únic espai de memòria per sobre de les memòries virtuals de les màquines distribuïdes. Aquests llenguatges òbviament no proporcionen memòria compartida, ja que no es pot esperar que el maquinari pugui encarregar-se de la seva coherència. Tot i així, l'espai d'adreces global permet definir estructures de dades globals, cosa que millora el model de memòria distribuïda en el qual el programador ha de recordar un conjunt d'estructures de dades independents que actuen com una estructura de dades distribuïda difícil de mantenir. Amb els llenguatges PGAS, els programadors es poden centrar en el comportament d'un únic procés i poden distingir les dades locals de les dades no locals (remotes), però aquests llenguatges permeten simplificar la programació, ja que eliminen els detalls del pas de missatges, i millora així la productivitat del programador. Són els compiladors els que generen totes les crides de comunicació quan hi ha referències a adreces de memòria que no són locals.

Les implementacions de llenguatges PGAS més importants i que a continuació estudiarem són UPC, Co-Array Fortran i Titanium, les quals estenen C, Fortran i Java, respectivament.

3.2.1. UPC

L'UPC[17] va ser desenvolupat prop de l'any 2000. L'UPC proporciona al programador una visió global de l'espai de memòria en què quan els elements (per exemple matrius) es defineixen com a compartits (*shared*) es distribueixen en les memòries dels diferents processos, de tal manera que els elements compartits es distribueixen cíclicament o amb blocs cíclics. Aquest tipus de distribució fa que es pugui balancejar millor la càrrega, però en general també fa que es redueixi la localitat de les dades. Tot i així, podem millorar la localitat de les dades assignant els fragments de les dades compartides directament al processadors de tal manera que moltes dades contigües quedin en el mateix procés.

(17)De l'anglès *unified parallel C*.

Com que l'UPC estén el llenguatge de programació C es poden utilitzar apuntadors. Els apuntadors d'UPC poden ser o bé privats (locals respecte al procés) o bé compartits entre tots els processos. Com que els apuntadors poden ser privats o compartits i, a més, poden apuntar a dades privades o compartides, hi ha quatre tipus d'apuntadors tal com mostra la figura 29.

Figura 29. Tipus d'apuntador en l'UPC

		Propietat de l'apuntador	
		Private	***Shared***
Propietat de la referència	***Private***	*Private-private,* `p1`	*Private-shared,* `p2`
	Shared	*Shared-private,* `p3`	*Shared-shared,* `p4`

Aquestes propietats estan associades als tipus del llenguatges, com es mostra a continuació, en què es poden veure diversos exemples de declaració d'apuntadors en UPC:

```
int *p1;   /* apuntador privat apuntant a dades locals */
shared int *p2; /* apuntador privat apuntant a l'espai compartit */
int *shared p3; /* apuntador compartit apuntant a dades locals *;
shared int *shared p4; /* apuntador compartit apuntant a l'espai compartit */
```

El codi de la suma de vectors que es mostra a continuació il·lustra l'ús del segon cas (apuntador privat apuntant a l'espai compartit).

```
shared int v1[N], v2[N], v1v2sum[N];


void main()
{
   int i;
   shared int *p1, *p2;
   p1=v1;
   p2=v2;
   upc_forall(i=0; i<N; i++; p1++; p2++; i)
   {
      v1v2sum[i]=*p1+*p2;
   }
}
```

El codi anterior també il·lustra l'ús d'un altre element bàsic de l'UPC, el `upc_forall`. Aquesta abstracció distribueix les iteracions de bucles de C normal (`for`) en els processos utilitzant una clàusula d'afinitat (que és la quarta clàusula del `upc_forall`). La clàusula d'afinitat del bucle indica on s'han executat les iteracions del bucle. En l'exemple anterior, el procés que executa

la iteració i-èsima és la que està associada a `i` (el propietari). El `upc_forall` és una operació global en què la major part del codi s'executa de manera local en un procés.

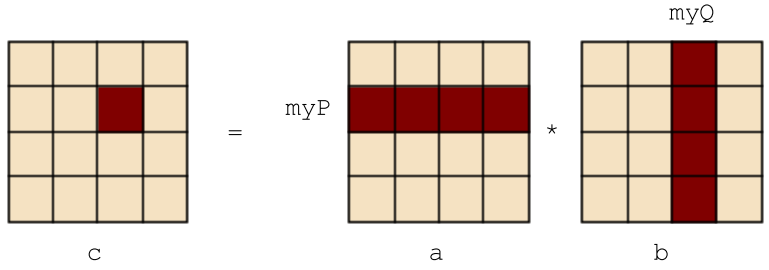
3.2.2. Co-Array Fortran

Co-Array Fortran (CAF) és una extensió de Fortran desenvolupada al final dels anys noranta. El CAF es caracteritza per la seva elegància i simplicitat. La idea principal és afegir una extensió al llenguatge anomenada `co-array`. El `co-array` és el mecanisme per a la comunicació entre processadors. Aprofitant que el Fortran utilitza parèntesis per a referir-se a les matrius, el CAF afegeix claudàtors als noms de variables per a referir-se a co-arrays. Per exemple, el codi que es mostra a continuació defineix tres matrius amb co-arrays.

```
real, dimension(n,n)[p,*]::a,b,c
...
do k=1,n
   do q=1,p
           c(i,j)[myP,myQ]=c(i,j)[myP,myQ]+a(i,k)[myP,q]*b(k,j)[q,myQ]
   enddo
enddo
```

L'especificació dels co-arrays indica que la memòria d'una variable que es distribueix per tots els processos s'anomena *imatge* en CAF. Per tant, en l'exemple anterior, a cada procés hi ha assignats fragments de les matrius `a`, `b` i `c`. A més, el fet que el co-array es defineixi per tenir dues dimensions fa que els processos es puguin organitzar en l'estructura lògica de dues dimensions de `p` files i `q` columnes. Cal tenir en compte que el símbol ***** indica que cada fila de processos s'ha d'omplir al màxim, per tant, si `num_images()=pq`, llavors hi ha `p` files i `q` columnes de processos. Cada procés trobarà la seva ubicació en l'organització de dues dimensions cridant la rutina `this_image()` i definint els dos paràmetres, `myP` i `myQ`, per a indicar un fragment concret de la matriu. La gestió de la partició de dades, inclosa la inicialització dels seus valors, és responsabilitat del programador. En la figura 30 es mostra un exemple en què `myP=2` i `myQ=3`.

Figura 30. Exemple d'organització de co-array amb CAF



El codi anterior mostra que cada procés, definit per `[myP,myQ]`, actualitza el seu fragment de la matriu quan calcula el producte escalar. Per a indicar les files s'accedeix a les dades en els processos `[myP, q]`, i per a indicar les columnes s'accedeix a les dades en els processos `[q, myQ]`. Les crides de comunicació per a aquests accessos remots les genera el compilador, cosa que fa que la programació se simplifiqui substancialment.

3.2.3. Titanium

El llenguatge Titanium (Ti) és una extensió del Java que es pot executar en computadors de memòria distribuïda. Té un model de memòria similar a l'UPC i, de la mateixa manera que altres llenguatges PGAS, genera codi de comunicació entre nodes basant-se en una biblioteca de comunicacions unidireccional. Una de les qualitats que el diferencia dels altres llenguatges PGAS que hem vist és el fet que és orientat a objectes. També es diferencia del Java pel fet d'afegir les anomenades *regions*, les quals suporten una gestió de memòria d'altes prestacions com a alternativa al mecanisme de *garbage collection*. Hi ha altres funcionalitats del Java que s'han restringit o limitat, però les funcionalitats més típiques són suportades. Per exemple, s'han afegit matrius de dues dimensions per tal de fer el Ti més apropiat per a la computació científica.

Una prestació d'eficiència important del Ti és la iteració desordenada, `foreach`. Aquesta simplifica la tasca tant del programador com del compilador i funciona permetent iterar sobre el domini d'una variable. A continuació es mostra com s'utilitza per a multiplicar matrius.

```
public static void matMul(   double [2d] a,
                  double [2d] b,
                  double [2d] c)
{
   foreach (ij in c.domain())
   {
      double [1d] aRowi=a.slice(1, ij[1]);
      couble [1d] bColj=b.slice(2, ij[2]);
      foreach (k in aRowi.domain())
      {
         c[ij]+=aRowi[k]*bColj[k];
      }
   }
}
```

D'aquesta manera `foreach`, al contrari de `forall`, permet la concurrència sobre múltiples índexs dins d'un mateix bloc.

El Titanium força la sincronització global mitjançant barreres i el concepte de variable compartida entre tots els processos que s'anomenen `single`. Per exemple, en una simulació és força habitual que cada procés faci els seus càl-

culs sobre dades locals i periòdicament coordini les seves accions. La barrera assegura que tots els processos paren els seus càlculs al mateix temps per a actualitzar o llegir de la memòria. La utilització de variables tipus `single` garanteix que tots els processos estiguin en la mateixa fase del còmput. Per exemple, una simulació de partícules podria utilitzar aquest concepte tal com es mostra a continuació:

```
int single stepCount=0;
int single endCount=100;
for (; stepCount<endCount; stepCount++)
{
   Llegeix partícules remotes
   Calcula les forces de la meva
   Ti.barrier();
   Escriu les meves partícules utilitzant noves forces
   Ti.barrier();
}
```

4. Esquemes algorítmics paral·lels

En aquest apartat analitzarem esquemes algorítmics dels que més s'utilitzen en la resolució de problemes en entorns paral·lels. Estudiarem un conjunt d'esquemes de referència que constitueix la base per a la resolució d'una gamma amplia de problemes reals.

4.1. Paral·lelisme de dades

Amb aquesta tècnica es treballa amb moltes dades que es tractaran de manera igual o similar. Típicament es tracta d'algoritmes numèrics en què les dades es troben en vectors o matrius i es treballa amb aquests fent un mateix processament. La tècnica és apropiada per a memòria compartida i normalment s'obté el paral·lelisme dividint el treball dels bucles entre els diferents fluxos (per exemple, en OpenMP `#pragma omp for`). Els algoritmes en què apareixen bucles on no hi ha dependències de dades, o aquestes dependències es poden evitar, són adequats per a paral·lelismes de dades. Si es distribueixen les dades entre els processos treballant en memòria distribuïda parlem de partició de dades.

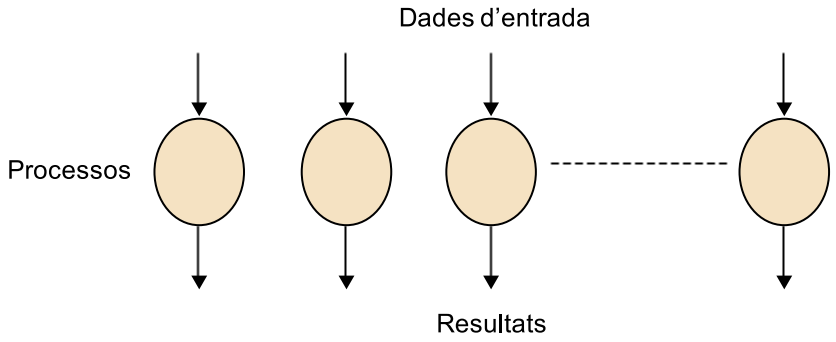
Aquest tipus de paral·lelisme és el que hi ha en la multiplicació de matrius en l'OpenMP, ja que en aquest cas es paral·lelitza la multiplicació simplement indicant amb `#pragma omp parallel for` en el bucle més extern que cada flux calculi una sèrie de files de la matriu resultat. L'aproximació d'una integral també segueix aquest paradigma.

La suma seqüencial de n nombres es fa amb un únic bucle `for`, el qual es pot paral·lelitzar amb `#pragma omp parallel for`, amb la clàusula `reduction` per a indicar la manera en què els fluxos accedeixen a la variable en què s'emmagatzema la suma. La paral·lelització es fa d'una manera molt simple, i es parla de paral·lelisme implícit, ja que el programador no és el responsable de la distribució del treball ni de l'accés a les variables compartides.

En alguns casos ens pot interessar (per exemple, per a disminuir els costos de gestió dels fluxos) posar en marxa un treball per a cadascun dels fluxos, i que el programador s'encarregui de la distribució i accés a les dades, amb la qual cosa es complica la programació. En aquest cas parlem de paral·lelisme explícit.

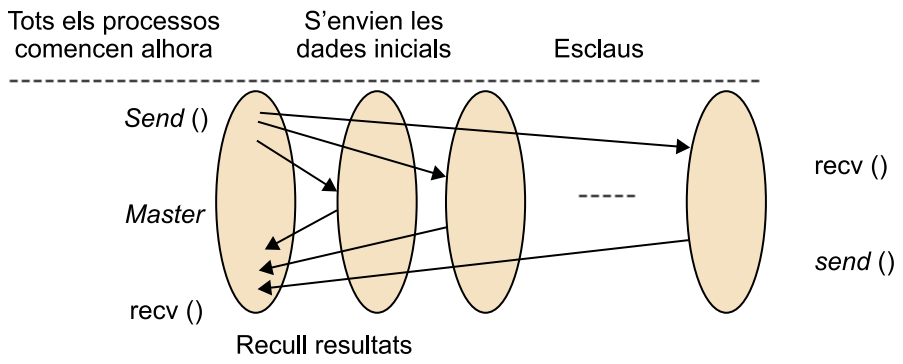
Aquest tipus de problemes que estan compost d'un conjunt de tasques independents, com mostra la figura 20, també es coneix com a *embarrassingly parallel*. En aquest tipus d'aplicacions no hi ha o hi ha molt poca comunicació entre processos i cada procés pot fer la seva tasca sense necessitar cap interacció amb els altres processos.

Figura 20. Model d'execució de múltiples processos amb tasques independents



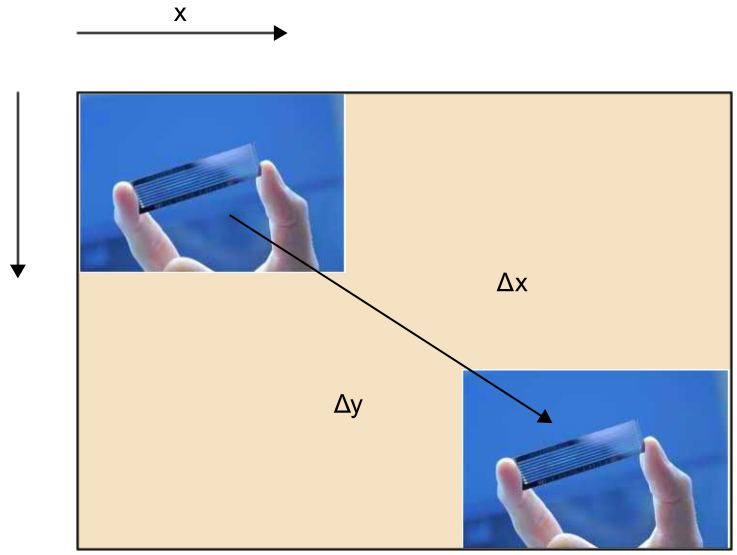
Si pensem en una implementació estàtica amb l'MPI d'aquest model basat en el patró mestre-esclau, veurem que el procés mestre crea un conjunt de processos esclaus que se sincronitzen mitjançant el pas de missatges al principi, fan la seva tasca independent i finalment es tornen a sincronitzar al final quan el procés mestre recull els resultats dels altres processos, com il·lustra la figura 32.

Figura 21. Implementació estàtica de tasques independents amb l'MPI basada en mestre-esclau



Tal com hem comentat anteriorment, aplicacions com ara el processament d'imatges són típiques d'aquest model. Un exemple clar és el desplaçament d'una imatge en el qual cada píxel s'ha de desplaçat un increment concret, com mostra la figura 33.

Figura 22. Exemple d'aplicació de desplaçament d'una imatge



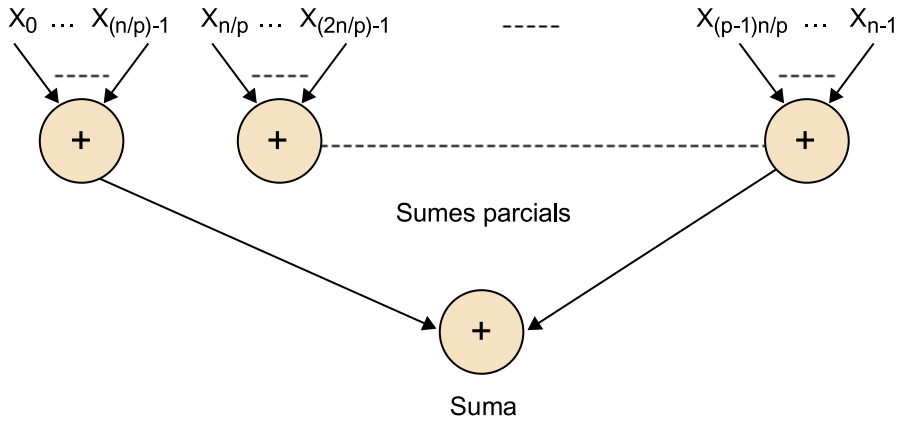
4.2. Partició de dades

En aquest tipus d'aplicacions es treballa amb volums de dades grans que normalment estan en vectors o matrius i s'obté paral·lelisme dividint el treball en zones diferents de les dades entre diferents processos. A diferència del paral·lelisme de dades, en aquest cas s'han de distribuir les dades entre els processos, i aquesta distribució determina el repartiment del treball. El més normal és dividir l'espai de dades en regions i que en l'algoritme hi hagi un intercanvi de dades entre regions adjacents, per la qual cosa s'han d'assignar regions adjacents als processadors intentant que les comunicacions no siguin gaire costoses. Hi ha comunicacions que comparteixen dades i que estan distribuïdes en la memòria del sistema, per la qual cosa hi ha un cost de comunicacions que sol ser força elevat, de manera que per a obtenir bones prestacions és necessari que el volum de computació sigui molt més gran que el de comunicació.

Una manera concreta de partir dades és la de dividir i conquerir. Aquest tipus de problemes es caracteritzen pel fet de dividir el problema en subproblemes com el problema principal. Normalment es divideix en encara més subproblemes de manera recursiva.

La figura 34 mostra un exemple de suma d'una seqüència de nombres mitjançant un patró model de dividir i conquerir en el qual cada procés fa una suma parcial que finalment és agregada per tal d'obtenir el resultat final.

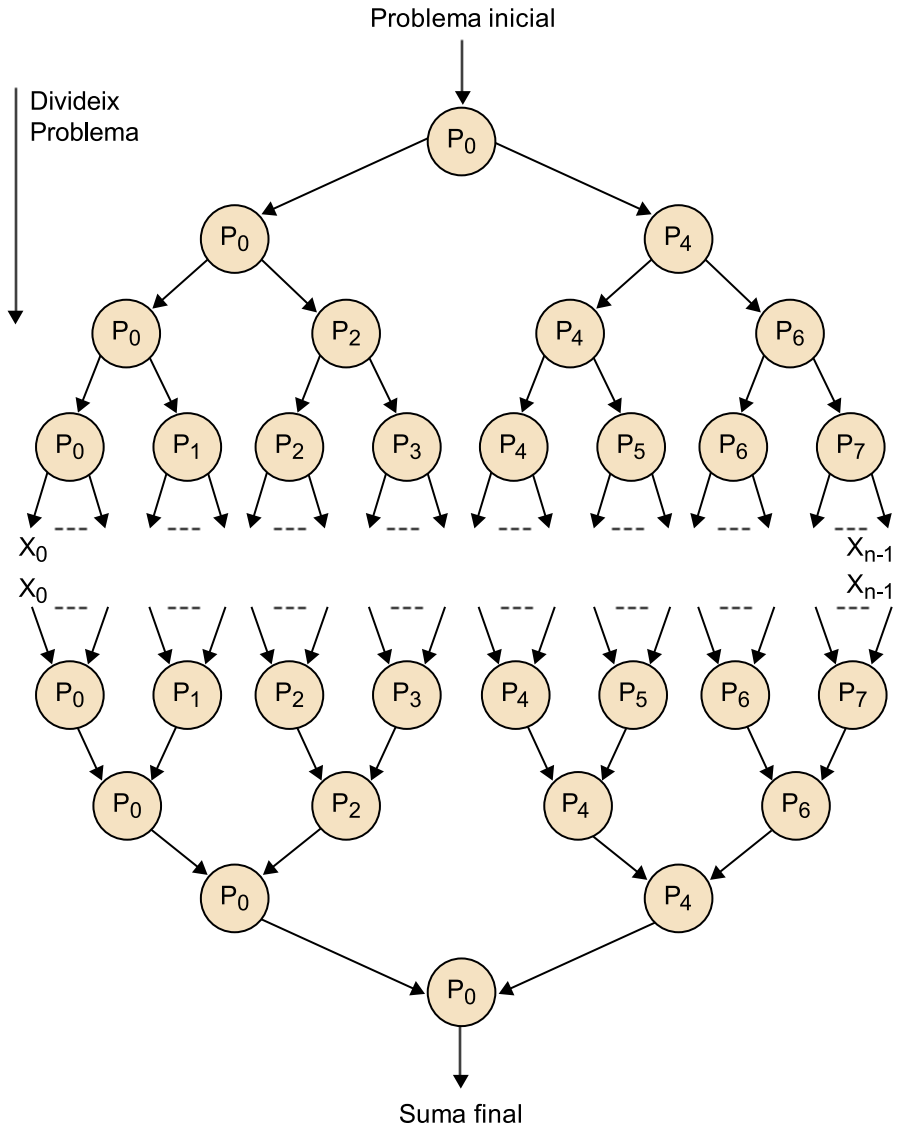
Figura 34. Exemple de suma d'una seqüència de nombres mitjançant un patró model de dividir i conquerir



4.3. Esquemes paral·lels en arbre

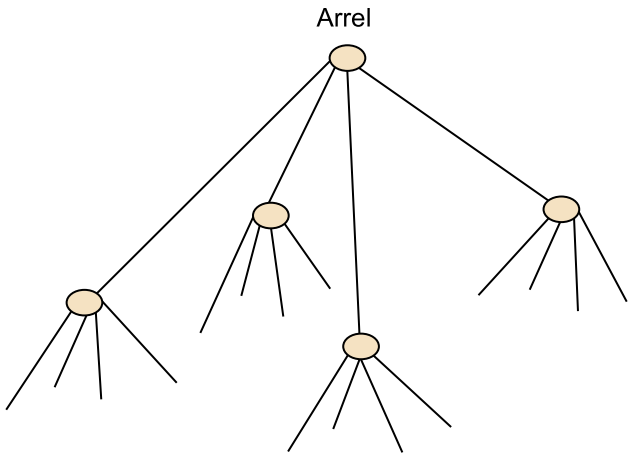
Molts problemes tenen una representació en forma d'arbre o graf i la seva solució s'obté recorrent l'arbre o graf fent computacions. Una tècnica per a desenvolupar programes paral·lels és, donat un problema, obtenir el graf de dependències que representa les computacions a fer (nodes del graf) i les dependències de dades (arestes), i a partir del graf decidir l'assignació de tasques als processos i les comunicacions entre aquests, que estaran determinades per les arestes que comuniquen nodes assignats a processos diferents. Un exemple és la suma prefixa, com mostra la figura 35. Veiem com en la primera fase es divideix el problema a mesura que es generen fills i, en la segona, es recullen els resultats a mesura que es recorre l'arbre.

Figura 35. Exemple de suma prefixa mitjançant l'esquema paral·lel en arbre



Els arbres també poden tenir més de dos fills, com il·lustra la figura 36, en què els nodes tenen quatre fills.

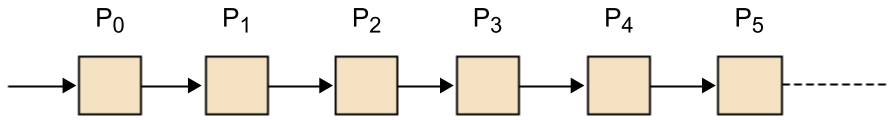
Figura 36. Esquema paral·lel en arbre amb quatre fills



4.4. Computació en *pipeline*

Amb el patró *pipeline* es resol un problema descomponent-lo en una sèrie de tasques successives, de manera que les dades flueixen en una direcció per l'estructura de processos. De fet, es pot veure com una forma de descomposició funcional en què el problema es divideix en diferents funcions que s'han de dur a terme successivament, com mostra la figura 37. Té una estructura lògica de pas de missatges, ja que entre les tasques consecutives s'han de comunicar dades. Pot tenir interès quan no hi ha un únic conjunt de dades a tractar, sinó una sèrie de conjunts de dades, que passen a ser computades una darrera l'altra, o bé problemes en què hi ha un paral·lelisme funcional, amb operacions de diferent tipus sobre el conjunt de dades i que s'han d'executar una darrera l'altra.

Figura 37. Esquema de *pipeline* senzill



Per exemple, suposem que hem de sumar els elements d'un vector mitjançant un algoritme com el següent:

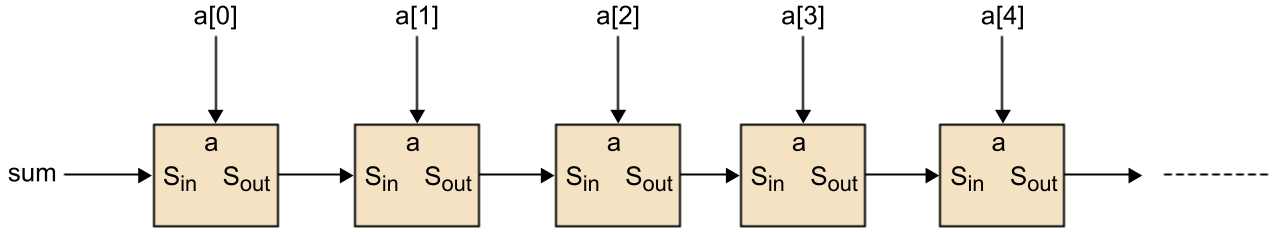
```
for (i = 0; i < n; i++)
   sum = sum + a[i];
```

i podem desplegar el bucle de la manera següent:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
...
```

D'aquesta manera ens trobem amb el *pipeline* que mostra la figura 38:

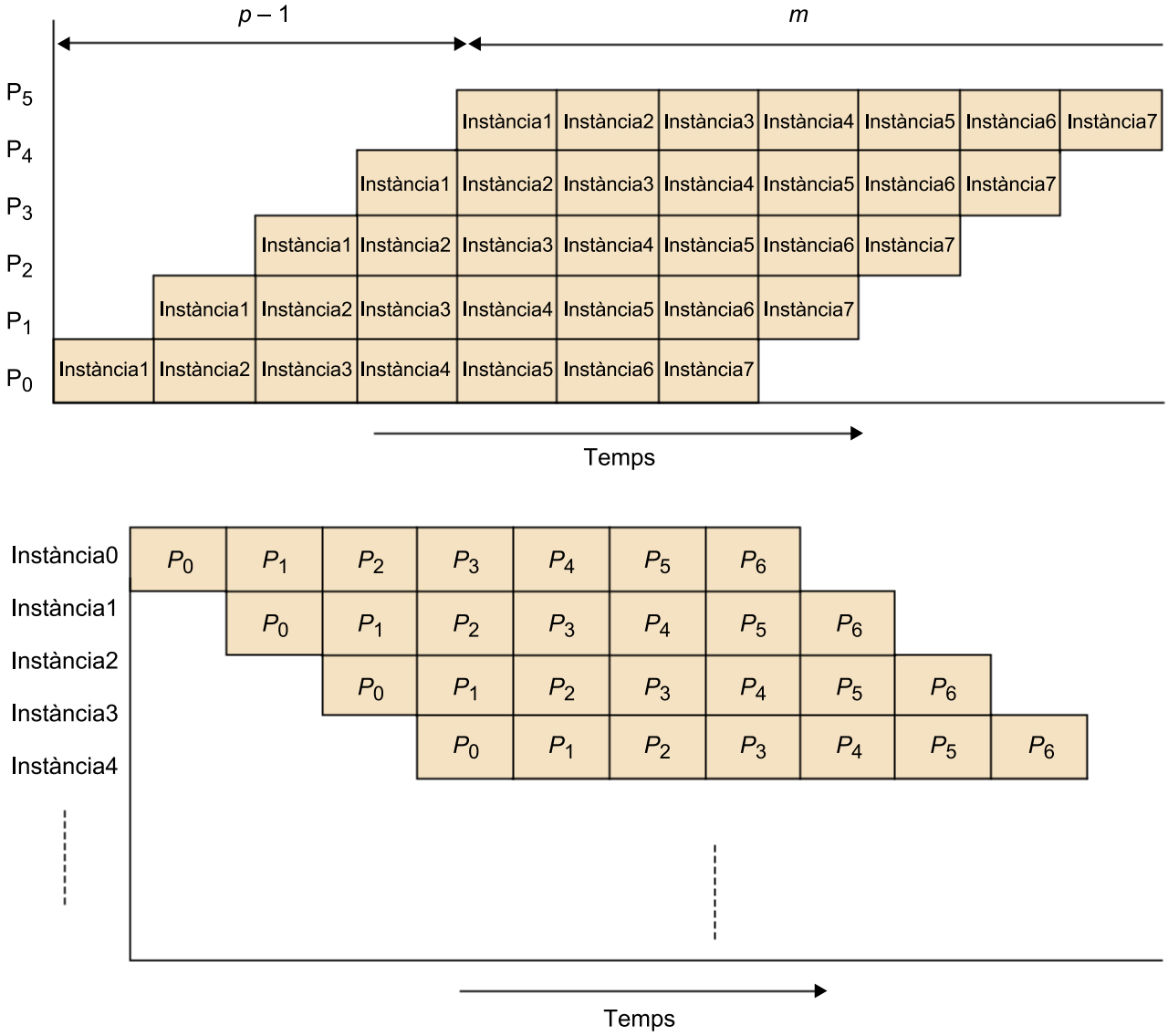
Figura 38. *Pipeline* de l'exemple de la suma dels elements d'un vector



Si suposem que, en general, el problema es pot dividir en una sèrie de tasques seqüencials, l'esquema de *pipeline* pot proporcionar un temps d'execució més curt en els tipus de computacions següents:

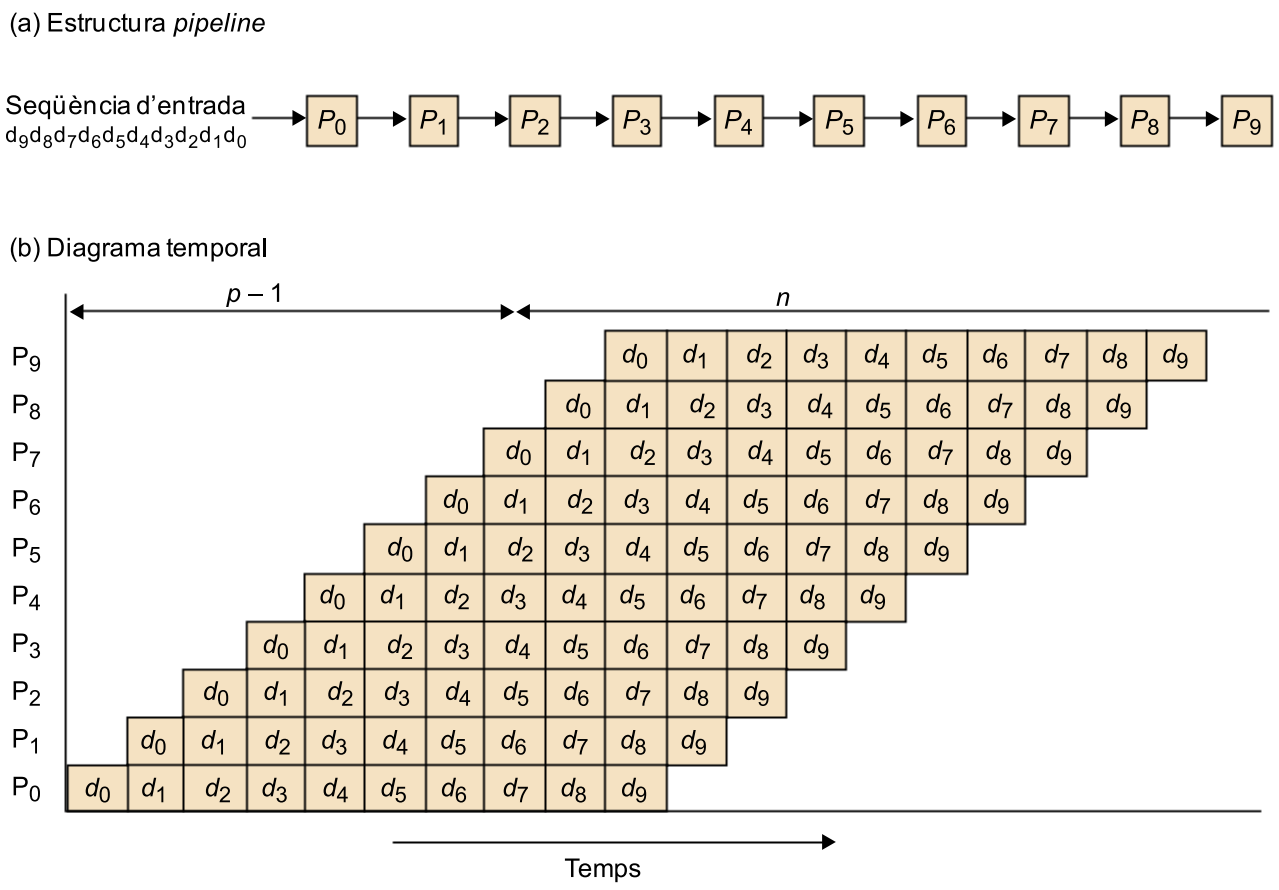
1) Si més d'una instància del problema complet es pot executar alhora, tal com mostra el diagrama de la figura 39.

Figura 39. Diagrama espaciotemporal del primer tipus de *pipeline*



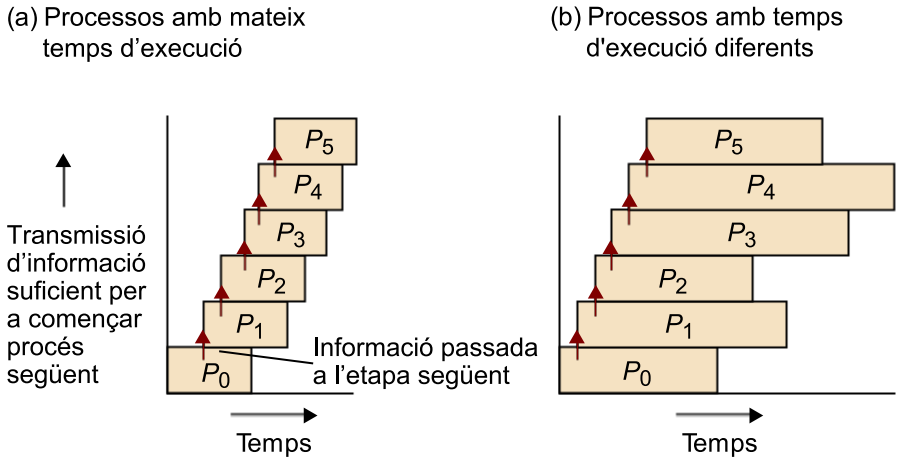
2) Si una sèrie d'elements de dades s'ha de processar i cada una requereix múltiples operacions, com mostra la figura 40.

Figura 40. Diagrama espaciotemporal del segon tipus de *pipeline*



3) Si la informació necessària per a començar l'execució del procés següent es pot passar al procés següent abans que el procés actual hagi acabat les seves operacions tal com il·lustra la figura 41. Un exemple d'aquest tipus és la cerca d'una certa informació en un conjunt de dades.

Figura 41. Diagrama espaciotemporal del tercer tipus de *pipeline*



4.5. Esquema mestre-esclau

En el paradigma mestre-esclau tenim un flux/procés diferenciat, anomenat *mestre*, que posa en marxa els altres processos, anomenats *esclaus*, els assigna treball i recull les solucions parcials que generen.

En alguns casos es parla de *granja de processos* quan un conjunt de processos treballa de manera conjunta però independentment en la resolució d'un problema. Aquest paradigma pot tenir similituds amb el mestre-esclau si considerem que els processos que constitueixen la granja són els esclaus (o els esclaus i el mestre si és que aquest també intervé en la computació).

Un altre paradigma relacionat amb aquests dos és el dels treballadors replicats. En aquest cas, els treballadors (fluxos o processos) actuen de manera autònoma, executen tasques que possiblement donen lloc a noves tasques que executaria el mateix treballador o un altre. Es gestiona una bossa de tasques, cada treballador pren una tasca, que s'inclou en la bossa, i una vegada acabada la tasca que té assignada en pren una altra de la bossa per a treballar-hi, i així mentre queden tasques per a executar. Es planteja el problema de la terminació, ja que és possible que un treballador sol·liciti una tasca i a la bossa no n'hi hagi més, però això no vol dir que no quedin tasques per executar, ja que és possible que un altre procés estigui computant i generi noves tasques. La condició per a finalitzar serà que no quedin tasques en la bossa i que no hi hagi treballadors treballant. Hi ha diferents maneres d'abordar el problema de la terminació.

Aquesta és la forma típica de treball en OpenMP. Inicialment treballa un únic flux, i al arribar a un constructor paral·lel posa en marxa una sèrie de fluxos esclaus dels que es converteix en el mestre.

En programació per pas de missatges, també s'acostuma a utilitzar aquest patró. En MPI, en posar-se en marxa els processos amb una ordre com `mpirun -np 8 programa`, s'inicialitzen vuit processos tots iguals, però l'entrada i sortida se sol fer mitjançant un procés, que actua com a mestre, que genera el problema, distribueix les dades, darrera del qual comença la computació, en la qual el mestre pot intervenir o no, i al final el mestre rep els resultats dels esclaus.

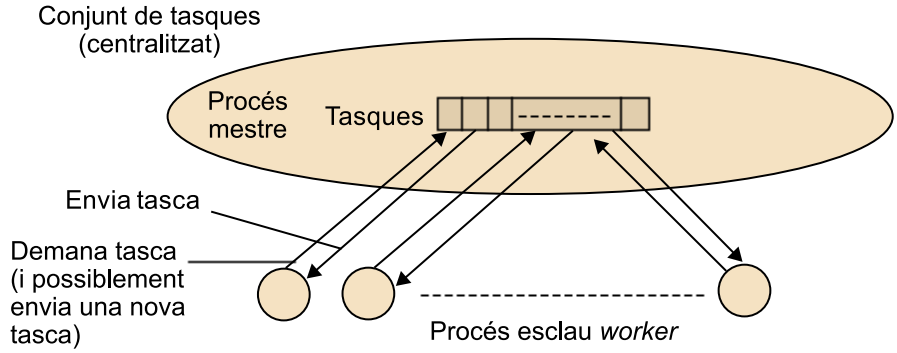
L'assignació del treball als esclaus es pot fer de manera estàtica o dinàmica:

- En l'assignació estàtica el mestre decideix els treballs que assigna als esclaus i fa l'enviament.

- En l'assignació dinàmica el mestre genera els treballs i els emmagatzema en una bossa de tasques que s'encarrega de gestionar. Els esclaus demanen treball de la bossa de tasques a mesura que queden lliures per a executar noves tasques. D'aquesta manera s'equilibra la càrrega dinàmicament, però

hi pot haver una sobrecàrrega de la gestió de la bossa i aquesta pot convertir-se en un coll d'ampolla. A més, en alguns problemes, els esclaus en executar una tasca generen noves tasques que s'han incloure en la bossa, de manera que es generen més comunicacions amb el mestre. Un exemple d'això es mostra en la figura 42.

Figura 42. Exemple d'assignació dinàmica mitjançant un conjunt de tasques centralitzat



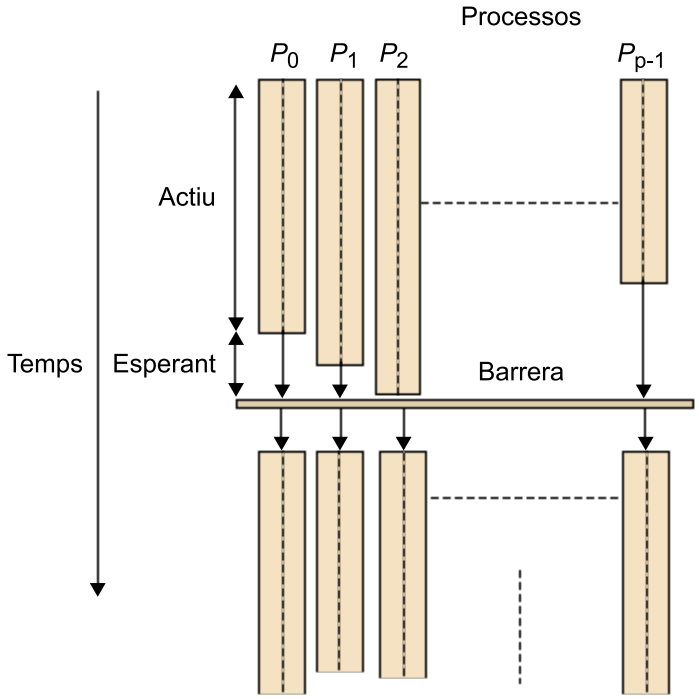
4.6. Computació síncrona

Parlem de *paral·lelisme síncron* quan es resol un problema per iteracions successives en què tots els processos se sincronitzen. Les característiques generals són:

- Cada procés du a terme el mateix treball sobre una porció diferent de les dades.
- Part de les dades d'una iteració s'utilitza en la següent, de manera que al final de cada iteració hi ha una sincronització, que pot ser local o global.
- El treball acaba quan es compleix algun criteri de convergència, per al qual normalment es necessita sincronització global.

Hi ha molts problemes que es resolen amb aquest tipus de patró. El mecanisme principal és la utilització de barreres allà on els processos s'han de sincronitzar. Els processos poden continuar l'execució quan tots arriben a aquest punt, com mostra la figura 43.

Figura 43. Processos arribant a una barrera en diferents moments



Activitat

Busqueu possibles implementacions de barreres tant per a memòria compartida com distribuïda. Alguns exemples típics de possibles implementacions són:

a) centralitzada (lineal)

b) en forma d'arbre

c) *butterfly*

4.7. Programació amb Python per a sistemes d'altes prestacions

Python és un llenguatge de programació d'alt nivell i propòsit general molt utilitzat i amb adopció creixent. La seva filosofia de disseny busca llegibilitat en el codi i la seva sintaxi permet als programadors expressar conceptes en menys línies de codi del que seria possible en llenguatges com C. També proveeix estructures per a permetre programes més entenedors tant a petita com a gran escala.

Python suporta diversos paradigmes de programació, inclosa la programació orientada a objectes, la imperativa i també la funcional o procedimental. Presenta un sistema dinàmic i una gestió de la memòria automàtica i té una biblioteca estàndard gran i exhaustiva. Hi ha intèrprets de Python per a molts sistemes operatius diferents.

Python està esdevenint força popular en entorns d'altes prestacions, ja que permet treballar més ràpidament i integrar sistemes amb més eficàcia. Python és fàcil d'aprendre i amb ell proporciona guanys de productivitat pràcticament immediats. En canvi, les solucions basades en els models de programació per a computació d'altes prestacions com ara MPI proporcionen rendiments superiors.

CPython, la implementació de referència de Python, és un programari lliure i de codi obert que té un model de desenvolupament basat en la comunitat, de la mateixa manera que la major part de les altres implementacions. Python també disposa d'extensions com Cython, que és un llenguatge basat en Python que suporta crides a funcions en llenguatge C i la declaració de tipus de dades i classes del llenguatge C. Cython és un compilador estàtic i fa que escriure extensions C de Python sigui tan fàcil com amb Python. Això permet al compilador generar codi C molt eficient a partir del codi Cython. El codi C es genera un sol cop i a continuació es compila amb els principals compiladors de C/C++.

El llenguatge Python estàndard no és especialment adequat per a càlculs numèrics. Per exemple, la utilització de llistes/matrius pot ser molt flexible, però també són lentes per a processar-se en càlculs numèrics. NumPy és un paquet fonamental per a la computació numèrica. Defineix, per exemple, les matrius numèrics i els tipus de matriu i les operacions bàsiques sobre aquestes. Numpy també disposa de rutines per a àlgebra lineal i es pot enllaçar amb llibreries optimitzades com BLAS/LAPACK. El rendiment amb NumPy és molt més proper al que es pot aconseguir amb el llenguatge C amb Python per defecte.

Hi ha llibreries d'interès per a la computació d'altes prestacions, com ara SciPy, la qual és una col·lecció d'algoritmes numèrics i eines específiques per a dominis científics concrets com ara processament de senyal, optimització, estadística, etc. SymPy és una biblioteca Python per a matemàtiques simbòliques. El seu objectiu és convertir-se en un sistema d'àlgebra computacional tot mantenint el codi tan simple com sigui possible per a ser comprensible i fàcilment extensible. SymPy està escrit completament en Python.

També es disposa d'eines per a la utilització del model de programació MPI per Python. Mpi4py proporciona una interfície per a MPI que és orientada a objectes i és molt similar a l'estàndard per C++. Aquesta permet la comunicació d'objectes Python com ara matrius de NumPy. A continuació es mostren dos exemples senzills de la utilització de Mpi4py.

Exemple sense comunicació:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

```
print "I am rank", rank
```

Exemple amb comunicació d'objecte Python:

```
from mpi4py import MPI


comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

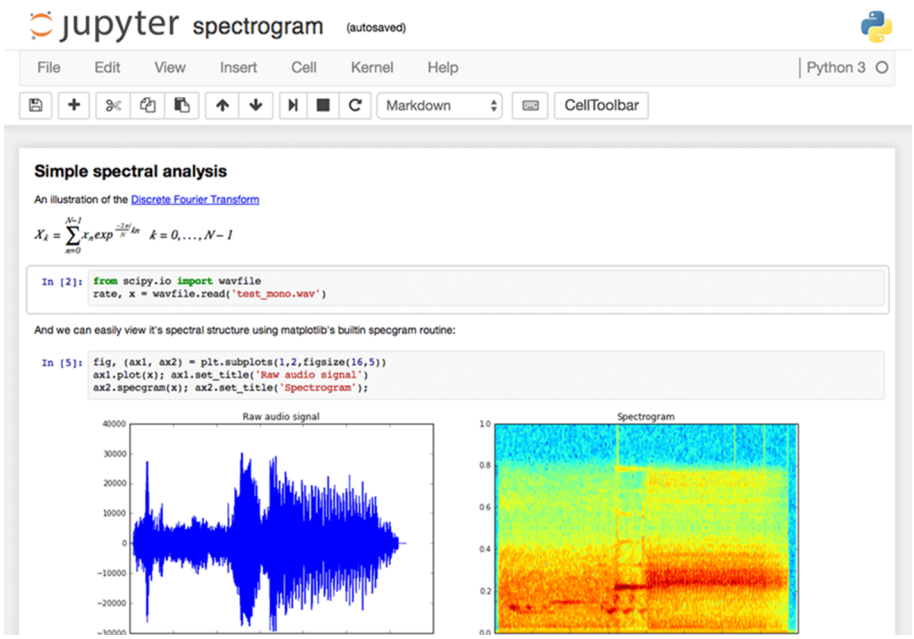
Altres projectes relacionats que faciliten l'ús d'MPI per Python són OOMPI, Pypar, pyMPI o Scientific Python.

Eines de més alt nivell també han guanyat molta popularitat, com per exemple iPython i Jupyter Notebook.

iPython és un intèrpret de comandes per a la computació interactiva en diversos llenguatges de programació, originalment desenvolupat per al llenguatge de programació Python, que ofereix les principals funcions següents:

- Contenidors interactius.
- Una interfície amb suport per codi, text, expressions matemàtiques, gràfics en línia i altres recursos.
- Suport per a visualitzar i usar dades interactives.
- Intèrprets flexibles que permeten carregar-se en projectes propis.
- Eines per a la computació paral·lela.

El Jupyter Notebook és una aplicació web de codi obert que us permet crear i compartir documents que continguin codis en viu, equacions, visualitzacions i text. Els usos inclouen: neteja i transformació de dades, simulació numèrica, modelització estadística, visualització de dades i molt més. Un exemple d'ús de Jupyter Notebook es mostra en la figura següent.


jupyter spectrogram (autosaved)
File Edit View Insert Cell Kernel Help
Python 3
Markdown
CellToolbar
Simple spectral analysis
An illustration of the Discrete Fourier Transform
$X_k = \sum_{n=0}^{N-1} x_n exp^{\frac{-2\pi i}{N} kn} \quad k = 0, \ldots, N-1$
In [2]:
from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
And we can easily view it's spectral structure using matplotlib's builtin specgram routine:
In [5]:
fig, (ax1, ax2) = plt.subplots(1,2,figsize(16,5))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
Raw audio signal
Spectrogram

Bibliografia

Gaster, B.; Howes, L.; Kaeli, D. R.; Mistry, P.; Schaa, D. (2011). *Heterogeneous computing with OpenCL*. Burlington, Massachusetts: Morgan Kaufmann.

Grama, A.; Karypis, G.; Kumar, V.; Gupta, A. (2003). *Introduction to parallel computing*. Reading, Massachusetts: Addison-Wesley.

Hwang, K.; Fox, G. C.; Dongarra, J. J. (2012). *Distributed and cloud computing: from parallel processing to the Internet of things*. Burlington, Massachusetts: Morgan Kaufmann.

Jain, R. (1991). *The Art of computer system performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley-Interscience.

Kirk, D. B.; Hwu, W. W. (2010). *programming massively parallel processors: a hands-on approach*. Burlington, Massachusetts: Morgan Kaufmann.

Lin, C.; Snyder, L. (2008). *Principles of parallel programming*. Reading, Massachusetts: Addison Wesley.

Munshi, A. (2009). *The OpenCL specification*. Khronos OpenCL Working Group.

Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. (2011). *OpenCL programming guide*. Reading, Massachusetts: Addison-Wesley Professional.

Nvidia (2007). *Nvidia CUDA compute unified device architecture*. Technical Report.

Nvidia (2007). *The CUDA compiler driver NVCC*.

Pacheco, P. (2011). *An introduction to parallel programming*. Burlington, Massachusetts: Morgan Kaufmann.

Sanders, J.; Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Reading, Massachusetts: Addison-Wesley Professional.