UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computació Paral·lela (CPA)

Pràctica 1: Paral·lelització amb OpenMP

Curs 2021/22

Índex

Introducció

Aquesta pràctica consta de 3 sessions, corresponents a cada un dels 3 apartats d'este butlletí. La següent taula mostra el material de partida per a realitzar cadascun dels apartats:

Sessió 1	Integració numèrica	`integral.c`
Sessió 2	Processament d'imatges	`imagenes.c`, `Lenna.ppm`, `Lenna1k.ppm`
Sessió 3	Nombres primers	`primo_grande.c`, `primo_numeros.c`

Les pràctiques d'esta assignatura estan preparades per a ser realitzades sobre els ordinadors del laboratori, amb el sistema operatiu Linux, o bé accedint al mateix entorn a través de l'escritori remot DSIC-LINUX des de `https://polilabs.upv.es/`. Serà necessari també connectar-se des d'aquest entorn al clúster de càlcul `kahan` mitjançant `ssh`, como es comenta en l'apartat 1.3.

Començarem per crear una carpeta en la unitat `W` per al codi font de la pràctica. S'aconsella que la ruta d'eixa carpeta siga curta i sense espais, perquè més endavant necessitarem teclejar-la sovint. Per exemple, podria ser `W/cpa/prac1` (d'ara en avant suposarem que eixa és la ruta de la carpeta). Guardarem en eixa carpeta els fitxers `.c` i `.ppm` de la pràctica (directament, sense subdirectoris).

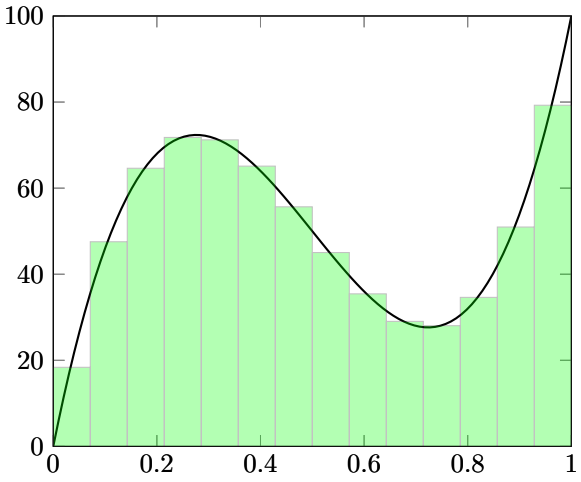

Figura 1: Interpretació geomètrica d'una integral.

1 Integració numèrica

En este primer apartat aprendreu a compilar programes OpenMP i a realitzar la paral·lelització de bucles senzills. Veureu com executar progrmes, tant sobre la màquina local com sobre el clúster de càlcul `kahan`.

Per a fer-ho, considerarem el problema del càlcul numèric d'una integral de la forma

$$\int_a^b f(x)dx.$$

La tècnica que anem a utilitzar per a calcular numèricament la integral és senzilla, i consisteix a aproximar mitjançant rectangles l'àrea corresponent a la integral, tal com pot veure's en la Figura 1. Es pot expressar l'aproximació realitzada de la següent forma

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i) \cdot h = h \cdot \sum_{i=0}^{n-1} f(x_i), \tag{1}$$

on n és el nombre de rectangles considerat, $h = (b-a)/n$ és l'amplària dels rectangles, i $x_i = a + h \cdot (i + 0.5)$ és el punt mitjà de la base de cada rectangle. Com més gran siga el nombre de rectangles, millor serà l'aproximació obtinguda.

El codi del programa seqüencial que realitza el càlcul es troba en el fitxer `integral.c`, del que es pot veure un extracte en la Figura 2. En particular, podem veure que hi ha dues funcions diferents per al càlcul de la integral, que corresponen a dues variants amb petites diferències entre si. En ambdues hi ha un bucle que realitza el càlcul de la integral, i que correspon al sumatori que apareix en l'equació (1).

El que es pretén fer en aquesta pràctica és paral·lelitzar mitjançant OpenMP les dues variants del càlcul de la integral.

En primer lloc, compileu el programa. Per a fer-ho, obriu una terminal sobre la carpeta on es troba el fitxer `integral.c` y executeu l'ordre:

```
$ gcc -Wall -o integral integral.c -lm
```

En fer-ho, s'haurà creat en la mateixa carpeta un programa executable anomenat `integral`. El significat de les opcions de compilació és el següent:

- `-o fitxer_executable`: especifica el nom del fitxer executable o d'eixida (*output*).

```
/* Calcul de la integral d'una funcio f. Variant 1 */
double calcula_integral1(double a, double b, int n)
{
   double h, s=0, result;
   int i;
   h=(b-a)/n;
   for (i=0; i<n; i++) {
      s+=f(a+h*(i+0.5));
   }
   result = h*s;
   return result;
}
/* Calcul de la integral d'una funcio f. Variant 2 */
double calcula_integral2(double a, double b, int n)
{
   double x, h, s=0, result;
   int i;
   h=(b-a)/n;
   for (i=0; i<n; i++) {
      x=a;
      x+=h*(i+0.5);
      s+=f(x);
   }
   result = h*s;
   return result;
}
```

Figura 2: Codi seqüencial per a calcular una integral.

- `-lm`: compilar amb la llibreria matemàtica. Esta opció és necessària si el programa utilitza funcions matemàtiques com `sin`, `cos`, `pow`, `exp`...
- `-Wall` (opcional): mostrar totes les advertències (*warnings*).

A continuació, executeu el programa. En executar-lo li indicarem mitjançant un argument (1 o 2) quina de les dues variants del càlcul de la integral volem utilitzar. Per exemple, per a utilitzar la primera variant:

```
$ ./integral 1
```

El resultat de la integral eixirà per pantalla. El resultat serà el mateix independentment de la variant escollida per al seu càlcul.

Opcionalment, es pot indicar el valor de n (nombre de rectangles) com a segon argument del programa (per defecte són 100000 rectangles). Per exemple:

```
$ ./integral 1 500000
```

1.1 Paral·lelització de la primera variant

En esta secció haureu de modificar el codi d'`integral.c` per tal de realitzar el càlcul de forma paral·lela utilitzant OpenMP. En lloc de modificar el fitxer original, feu una còpia amb un altre nom (per exemple, `pintegral.c`).

Comenceu per fer que el programa simplement mostre el nombre de fils amb els què s'executa. Per a fer-ho, modifiqueu el programa com es mostra de forma esquemàtica en la Figura 3. És a dir, afegiu una

```
...
#include <omp.h>
...
int main(int argc, char *argv[]) {
   ...
   printf("Nombre de fils: %d\n", funcio_openmp_adequada());
   ...
}
```

Figura 3: Modificació (incompleta) per a mostrar el nombre de fils.

sentència `printf` en la funció `main` per a mostrar el nombre de fils, el qual s'obté mitjançant la funció adequada d'OpenMP. S'ha d'incloure a més el fitxer de capçalera `omp.h`.

Per a compilar el programa, heu d'afegir l'opció `-fopenmp`, per exemple:

```
$ gcc -fopenmp -Wall -o pintegral pintegral.c -lm
```

I per a executar el programa amb múltiples fils, per exemple 4:

```
$ OMP_NUM_THREADS=4 ./pintegral 1
```

Tingueu en compte que **no ha d'haver cap espai** en "`OMP_NUM_THREADS=4`".

En executar el programa, es mostrarà el nombre de fils utilitzats. Es mostra només 1 fil en lloc de 4? Si és així, tingueu en compte que la funció d'OpenMP per a obtindre el nombre de fils torna el nombre de fils actius, i fora d'una regió paral·lela només hi ha 1 fil actiu. Heu de solucionar este problema per a que es mostre el nombre de fils amb què realment s'ha executat el programa. A més, heu de fer que el missatge amb el nombre de fils es mostre només una vegada.

Una vegada solucionat això, ens ocuparem de la paral·lelització de la primera variant de càlcul de la integral (`calcula_integral1`). Podem començar col·locant una directiva `parallel for` sense preocupar-nos de si les variables són compartides, privades o d'un altre tipus, i seguidament compilem i executem el programa modificat, per a veure què és el que ocorre. Ens adonarem que el resultat és incorrecte. Per a solucionar-ho pot ser necessari indicar el tipus d'algunes de les variables que intervenen en el bucle, mitjançant la utilització de clàusules com `private` o `reduction`.

Una vegada s'haja modificat el fitxer i s'haja compilat per a produir l'executable, es comprovarà que el resultat de la integral és el mateix que en el cas seqüencial, i que no varia en tornar a executar el programa, ni en canviar el nombre de fils.

1.2 Paral·lelització de la segona variant

Considerarem ara la paral·lelització de la segona variant (`calcula_integral2`). Com veiem en la Figura 2, el codi d'aquesta segona variant és pràcticament idèntic al de la primera. L'única diferència és que s'utilitza una variable auxiliar `x`, la qual cosa òbviament no afecta en res al resultat del càlcul.

Paral·lelitza ara aquesta segona versió. De nou, comprova que el resultat és el mateix que en el cas seqüencial, i que no varia en tornar a executar ni en canviar el nombre de fils.

1.3 Execució en el cluster

En este apartat utilitzareu el clúster de càlcul `kahan` per a llançar execucions, el què vos permetrà fer ús d'un major nombre de *cores*.

`kahan` es un clúster compost per 4 nodes de càlcul, cadascun d'ells amb 64 *cores*, i un node *front-end* al qual es connecten els usuaris per a compilar i llançar execucions. Tots els *cores* d'un mateix node comparteixen la

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=5:00
#SBATCH --partition=cpa

OMP_NUM_THREADS=3 ./pintegral 1
```

Figura 4: Fitxer de treball per a executar en el sistema de cues.

memòria d'eixe node, però no poden accedir a la memòria dels altres nodes. En el material de l'assignatura podeu trobar més detalls sobre el clúster `kahan`.

Per a treballar amb `kahan`, heu de connectar-vos al node *front-end* per mig de `ssh`:

```
$ ssh -l login@alumno.upv.es kahan.dsic.upv.es
```

on `login` és el teu nom d'usuari UPV. Després d'executar l'ordre, estaràs al teu directori `home` de `kahan`.

Si executeu el comandament `ls` comprovareu que hi ha un directori `W`, que correspon a la teua unitat `W` de la UPV. Recorda que els fitxers de la pràctica deurien estar en una carpeta de `W`, per exemple `W/cpa/prac1`, como es va indicar en l'apartat d'introducció. Comprova que la següent ordre mostra un llistat dels fitxers de codi font de la pràctica, entre els quals estarà el fitxer `pintegral.c` creat anteriorment:

```
$ ls W/cpa/prac1
```

A continuació tenim que compilar el programa, igual que hem fet anteriorment, però aquesta vegada en `kahan`. És important tindre en compte que l'executable generat no deu estar en la carpeta `W`, perquè eixa carpeta no és accessible des dels nodes de càlcul de `kahan`. Por a açò, crearem una carpeta (amb `mkdir`) en la que deixar els executables, ens situarem en ella (`cd`) i compilarem:

```
$ mkdir prac1
$ cd prac1
$ gcc -Wall -fopenmp -o pintegral ~/W/cpa/prac1/pintegral.c -lm
```

on al compilar indiquem la ruta del fitxer `pintegral.c` (el caràcter `~` indica el directori `home` i habitualment es pot teclejar amb `AltGr+4`).

Ara pots executar el programa, per exemple:

```
$ OMP_NUM_THREADS=4 ./pintegral 1
```

Acabes d'executar el programa en el *front-end*, **no en els nodes de càlcul** de `kahan`. Encara que és possible executar un programa en el *front-end*, només s'hauria de fer per a execucions molt curtes. El *front-end* només deu usar-se per a connectar-se per `ssh`, compilar i llançar treballs sobre el clúster de la manera que s'explica a continuació.

L'execució de treballs en el clúster s'ha de fer mitjançant el **sistema de cues SLURM**. Per a fer-ho cal crear un **fitxer de treball**, el qual és un *script* amb diferents opcions del sistema de cues seguides pels comandaments que es vol executar. En la Figura 4 hi ha un exemple d'un fitxer de treball, en el qual s'executa un programa OpenMP amb 3 fils d'execució (última línia del fitxer). Les línies que comencen per `#SBATCH` especifiquen diferents opcions del sistema de cues. En aquest cas el treball utilitzarà la cua (partició) denominada *cpa*, usant un node del clúster (amb els seus 64 cores) i amb un temps màxim de 5 minuts per a la seua execució.

Copieu el text de la Figura 4 en un fitxer (per exemple `jobopenmp.sh`) i guardeu-ho en la carpeta `W/cpa/prac1`. Canvieu el nombre de fils amb què s'executarà el programa, posant el valor que vulgueu. Tindria sentit canviar el nombre de nodes per un valor distint de 1?

A continuació s'ha de llançar el treball al sistema de cues, per a la qual cosa s'utilitza l'ordre `sbatch`. Suposant que el fitxer de treball s'anomena per exemple `jobopenmp.sh`, només caldria executar en el *front-end* (des del directori on està l'executable, en aquest cas `~/prac1`):

```
$ sbatch ~/W/cpa/prac1/jobopenmp.sh
```

En la terminal es mostrarà l'identificador del treball.

Una vegada llançat, el sistema de cues s'encarrega d'assignar al nostre treball els recursos que necessite (en este cas un node del clúster) quan aquests estiguen disponibles, mantenint al nostre treball en espera fins aleshores. D'esta manera, s'assegura que els nodes assignats a un treball no són utilitzats per cap altre treball.

Què ocorre amb els missatges que hauria de mostrar el programa? Eixos missatges no es mostren en la terminal, sinò que es deixen en un fitxer. Per exemple, si l'identificador del treball és 620, després de la seua execució es crearà el fitxer `slurm-620.out`. Podem mostrar el seu contingut amb l'ordre `cat`, per exemple:

```
$ cat slurm-620.out
Nombre de fils: 16
Valor de la integral = 1.000000000041
```

També podem copiar el fitxer generat a la carpeta `W/cpa/prac1`, i una vegada allí visualitzar-lo amb qualsevol editor de text:

```
$ cp slurm-620.out ~/W/cpa/prac1
```

Es pot consultar l'estat de les cues amb l'ordre `squeue`, per exemple:

```
$ squeue
      JOBID PARTITION       NAME        USER ST       TIME  NODES NODELIST(REASON)
       620     cpa       jobopenmp      john  R       0:01    1     kahan01
```

Per a cada treball es mostra el seu estat (ST), entre els que destaquem: en cua o pendent (PD), en execució (R) o finalitzat amb èxit (CD).

També es pot cancel·lar un treball amb l'ordre `scancel`:

```
$ scancel 620
```

1.4 Mesura de temps

En molts casos interessa mesurar el temps d'execució del programa, o d'una part d'ell, per a poder comparar-lo amb el temps del programa seqüencial i determinar la millora obtinguda.

Per a mesurar el temps d'un fragment d'un programa utilitzarem la funció `omp_get_wtime()` d'OpenMP, que torna el temps transcorregut (en segons) des d'un instant inicial fixe. La forma d'utilitzar esta funció s'il·lustra en la Figura 5.

Mesureu el temps d'execució del programa paral·lel en el clúster utilitzant un nombre de rectangles de 500000000 (500 milions), amb 1 fil i amb 16 fils, comprovant la reducció de temps obtinguda.

Podríem augmentar el nombre de fils indefinidament, millorant sempre el temps d'execució? Si no és així, fins a quin nombre de fils penses que es produiria millora?

2 Processament d'imatges

Aquest apartat de la pràctica se centra en la implementació en paral·lel del filtrat d'una imatge utilitzant OpenMP. L'objectiu és aprofundir en el coneixement d'OpenMP i de la resolució de dependències de dades entre fils.

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

int main(int argc, char *argv[]) {
   double t1, t2;
   ...
   t1 = omp_get_wtime();
   ...  /* Fragment de codi a cronometrar */
   t2 = omp_get_wtime();
   printf("Tiempo: %f\n", t2-t1);
}
```

Figura 5: Mesura de temps d'un fragment de codi.

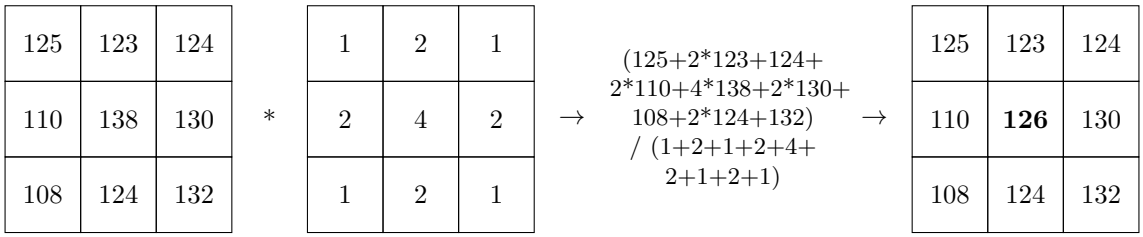

Figura 6: Model d'aplicació d'una mitjana ponderada en el filtrat d'imatges.

Els exercicis d'aquest apartat es realitzaran partint de la versió seqüencial d'un programa que permet llegir una imatge (en format PPM), aplicar diverses etapes de filtrat basat en la mitjana ponderada amb ràdio variable i escriure la imatge resultant en un arxiu amb el mateix format. Haureu de realitzar una versió paral·lela mitjançant OpenMP d'aquest programa, aprofitant els diferents bucles en els quals s'estructura el mètode.

2.1 Descripció del problema

El filtrat d'imatges consisteix en substituir els valors dels píxels d'una imatge per valors que depenen dels seus veïns. El filtrat es pot utilitzar per a reduir soroll, reforçar contorn, enfocar o desenfocar una imatge, etc.

El filtrat de mitjana consisteix a substituir el valor de cada píxel per la mitjana dels valors dels píxels veïns. Per veïns podem entendre als píxels que disten com a molt una certa distància, anomenada radi, en ambdues coordenades cartesianes. El filtrat de mitjana redueix notablement el soroll aleatori, però produeix un important efecte de desenfocament. Generalment, s'utilitza una màscara que pondera el valor dels punts veïns, seguint un ajust parabòlic o lineal. Aquest filtrat produeix millors efectes, encara que té un cost computacional més elevat. Més encara, el filtrat és un procés iteratiu que pot requerir diverses etapes.

2.2 Versió seqüencial

El material per a la pràctica inclou el fitxer `imagenes.c` amb la implementació seqüencial del filtrat. El filtrat utilitzat aplica una màscara ponderada que proporciona més pes als punts més pròxims al punt que s'està processant. La Figura 6 mostra un esquema. En aquesta figura, partim d'una imatge (esquerra) sobre el píxel central de la qual (amb el valor de color 138) s'aplica el filtrat, utilitzant la màscara quadràtica a la seua dreta. Aquesta aplicació implica realitzar el càlcul que es mostra, el resultat del qual apareix en negreta en la imatge resultant a la dreta. Aquest filtrat es realitzarà per a cadascun dels punts de la imatge.

L'algorisme per tant recorre les dues dimensions de la imatge, i per a cada píxel, utilitza dos bucles interns

```
for (p=0;p<pasos;p++) {
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++) {
      resultado.r = 0;
      resultado.g = 0;
      resultado.b = 0;
      tot=0;
      for (k=max(0,i-radio);k<=min(n-1,i+radio);k++) {
         for (l=max(0,j-radio);l<=min(m-1,j+radio);l++) {
           v = ppdBloque[k-i+radio][l-j+radio];
           resultado.r += ppsImagenOrg[k][l].r*v;
           resultado.g += ppsImagenOrg[k][l].g*v;
           resultado.b += ppsImagenOrg[k][l].b*v;
           tot+=v;
         }
      }
      resultado.r /= tot;
      resultado.g /= tot;
      resultado.b /= tot;
      ppsImagenDst[i][j].r = resultado.r;
      ppsImagenDst[i][j].g = resultado.g;
      ppsImagenDst[i][j].b = resultado.b;
    }
  }
  memcpy(ppsImagenOrg[0],ppsImagenDst[0],n*m*sizeof(struct pixel));
}
```

Figura 7: Bucles principals per al processament d'imatges.

```
P3       <- Cadena constant que indica el format (ppm, color RGB)
512 512  <- Dimensions de la imatge (nombre de columnes i nombre de files)
255      <- Major nivell d'intensitat
224 137 125 225 135 ... <- 512x512x3 valors. Cada punt són tres valors consecutius (R,G,B)
```

Figura 8: Format d'un fitxer d'imatges PPM.

que recorren els píxels que disten com a molt el valor del radi en cadascuna de les dimensions, evitant eixir dels límits de les coordenades. El procés de filtrat es repeteix diverses vegades per a tota la imatge, i per tant el procés implica cinc bucles niats: passos, files, columnes, radi per files, radi per columnes, tal com mostra la Figura 7.

Per a la lectura i escriptura de la imatge utilitzem el format PPM, un format simple basat en text, que pot ser visualitzat per diferents programes, com `irfanview`[1] o `display` (disponible en el clúster de pràctiques). El format de la imatge es mostra en la Figura 8 i pot comprovar-se veient el contingut del fitxer amb `head`, `more` o `less`.

Per tant, el programa llegirà el contingut d'un fitxer d'imatge el nom del qual està especificat en `IMAGEN_ENTRADA`, aplicarà el filtrat tantes vegades com indica `NUM_PASOS` utilitzant el radi `VAL_RADIO` i ho escriurà en el fitxer `IMAGEN_SALIDA`. La reserva de memòria la realitza la funció de lectura de la imatge, garantint que tots els píxels de la imatge es troben consecutius.

Comproveu el correcte funcionament del programa. Per a fer-ho, compileu-lo i executeu-lo en la màquina local. Observareu que el programa ha creat el fitxer `lenna-fil.ppm` (el nom especificat en `IMAGEN_SALIDA`) amb el resultat d'aplicar el filtre sobre la imatge del fitxer `Lenna.ppm`, obtinguda d'un conegut benchmark[2]

[1] `http://www.irfanview.com`

[2] `http://en.wikipedia.org/wiki/Standard_test_image`

Figura 9: Imatge de referència (`Lenna.ppm`) abans (esquerra) i després (dreta) d'aplicar un filtrat d'un pas i radi 5.

(veure Figura 9).

Feu una còpia del fitxer `lenna-fil.ppm` que heu obtingut, anomenant-la `ref.ppm`. Eixe fitxer conté la eixida del programa original, que utilitzareu com a referència per a comparar-la més avant amb l'exida dels programes modificats que haureu de desenvolupar, per tal de comprovar si funcionen correctament.

A continuació, modifiqueu el programa per a què mostre el nombre de fils amb el qual s'ha executat, tal com vau fer en la sessió anterior, i també el temps d'execució emprat per la funció que realitza el filtrat de la imatge.

Compileu-lo i executeu-lo en el clúster. Per a açò, com es va explicar a l'apartat 1.3, deveu connectar-se mitjançant `ssh` al node *front-end* de `kahan`, canviar al directori `~/prac1` i compilar. En aquest cas també deuràs copiar els fitxers d'imatges des de la carpeta del codi font (`~/W/cpa/prac1`) a la carpeta d'executables (`~/prac1`), per exemple amb l'ordre:

```
$ cp ~/W/cpa/prac1/*.pmm ~/prac1/
```

A continuació, crea un fitxer de treball i llança el treball mitjançant `sbatch` (de nou, vore apartat 1.3).

2.3 Implementació paral·lela

Existeixen diferents aproximacions per a realitzar la implementació paral·lela mitjançant OpenMP, depenent del bucle que s'elegisca per a la seua paral·lelització. El treball se centrarà a analitzar els cinc bucles de la Figura 7 i decidir (i provar) quins bucles són susceptibles de ser paral·lelitzats i quines variables haurien de ser compartides o privades. Per a açò, per a cada un dels 5 bucles haureu de seguir els següent passos:

1. Decidir si el bucle es pot paral·lelitzar. Per a fer-ho, cal analitzar si les diferents iteracions del bucle en qüestió tenen alguna dependència entre elles (per exemple, si la segona iteració utilitza dades generades per la primera) i si estes dependències es poden resoldre directament amb alguna clàusula d'OpenMP (com en el cas de sumatoris).

   Si el bucle no es pot paral·lelitzar, els següents passos no es realitzarien.

```
Tiempo secuencial: _____________

Tiempo de versiones paralelas:
            Version 1  Version 2  ...
  2 hilos   _________  _________  ...
  8 hilos   _________  _________  ...
 32 hilos   _________  _________  ...
```

Figura 10: Plantilla per a mesura de temps.

2. Escriure le(s) directive(s) per a paral·lelitzar el bucle, prestant atenció a quines variables hauran de ser privades per a cada fil, i quines compartides per tots.

3. Executar el programa modificat (preferiblement en el clúster).

4. Comprovar que el fitxer generat pel programa modificat (`lenna-fil.ppm`) és exactament igual que el produït pel programa original (`ref.ppm`). Per a fer-ho, executeu l'ordre:

   ```
   $ cmp lenna-fil.ppm ref.ppm
   ```

   Si els fitxers són iguals, l'ordre anterior no mostrarà cap missatge.

A continuació es proporcionen algunes orientacions que poden ser d'utilitat:

- És més senzill començar considerant el bucle més intern, donat que hi han menys variables implicades, passant després a considerar el bucle immediatament per damunt, i així successivament fins a arribar al bucle més extern.
- No es pot utilitzar la clàusula `reduction` sobre un membre d'una variable de tipus `struct`. Si ho necessiteu, substituïu l'`struct` per una variable per cada membre del mateix.
- Algunes versions paral·leles, encara que siguen correctes, poden ser extremadament lentes, i inclús empitjorar al augmentar el nombre de fils. Per estar raó es recomana utilitzar inicialment només 2 fils en executar les versions paral·leles.
- Tingueu en compte que el clúster s'utilitza de forma compartida per tots el alumnes, per la qual cosa heu d'evitar omplir la cua amb els teus treballs. Podeu executar múltiples ordres dins d'un mateix fitxer de treball.

Convé que vos feu la següent pregunta: té sentit paral·lelitzar dos o més bucles a la vegada?

Tracteu d'obtindre una cosa pareguda al que es mostra en la Figura 10, en la qual es registra el temps d'execució del programa seqüencial, junt amb el temps corresponent a cadascuna de les versions paral·leles desenvolupades, per a distint nombre de fils. Si una versió paral·lela és més lenta amb 2 fils que el programa seqüencial, no és recomana que l'executeu amb més fils, perquè només empitjorarà.

Es fàcil veure que hi ha algunes versions paral·leles molt més eficients que altres. Quines són les versions més eficients i per què?

3 Nombres primers

En aquesta sessió es va a treballar amb el conegut problema de veure si un nombre és primer o no. Encara que per a aquest problema hi ha diversos algorismes (alguns d'ells més eficients però quelcom complicats), es va a utilitzar l'algorisme seqüencial típic.

En aquest cas, la paral·lelització no és "trivial". No sempre es pot obtenir un bon algorisme paral·lel amb OpenMP afegint tan sols un parell de directives. Quan es puga, així ha de fer-se tant per comoditat com per

```
Funció primer(n)
  Si  n és parell  i  no és el número 2  llavors
    p <- fals
  si no
    p <- vertader
  Fi si
  Si    p llavors
    s <- arrel quadrada de n
    i <- 3
    Mentre  p  i  i <= s
      Si  n és divisible de forma exacta per  i llavors
        p <- fals
      Fi si
      i <- i + 2
    Fi mentre
  Fi si
  Retorna p
Fi funció
```

Figura 11: Algorisme seqüencial per a determinar si n és primer.

claredat i independència de la plataforma. Però en ocasions (i aquesta és una d'elles), cal parar-se a pensar i indicar explícitament un repartiment de la càrrega entre fils. Açò és el que va a haver-hi que realitzar aquesta pràctica.

3.1 Algorisme seqüencial

L'algorisme seqüencial clàssic per a veure si un nombre és primer es mostra en la Figura 11. Consisteix a mirar si és divisible per algun nombre inferior a ell (diferent de l'1). Si és divisible de forma exacta per algun nombre inferior a ell i diferent de la unitat, el nombre no és primer.

Comprovar si un nombre és primer o no seguint aquest algorisme té un cost xicotet, sempre que el nombre no siga molt gran. Noteu que el bucle deixa d'executar-se si es descobreix que el nombre és compost (en aqueix cas no fa falta seguir mirant). Per açò, és obvi que el pitjor cas (el major cost) succeirà quan el nombre a comprovar siga primer o siga no primer però compost per factors grans.

Amb la intenció d'obtenir un codi que tinga un cost una mica més elevat, es va a procedir a cercar un nombre primer gran. No té sentit paral·lelitzar una tasca el cost de la qual siga molt xicotet (excepte per a il·lustrar o ensenyar coses bàsiques).

El problema inicial a resoldre en aquest apartat de la pràctica va a ser, per tant, no el veure si un nombre és primer (açò serà un component fonamental) sinó cercar el nombre primer més gran que càpia en un enter sense signe de 8 bytes. El procés per a obtenir aquest nombre suposa partir del major nombre que càpia en aqueix tipus de dades i anar cap avall fins a trobar un nombre que siga primer, utilitzant l'algorisme anterior per a veure si cada nombre és primer o no. Habitualment el nombre més gran serà imparell (tot a uns en binari) i per a cercar primers es pot anar decrementant de dos en dos per a no mirar els parells, que no fa falta. L'algorisme es mostra en la Figura 12.

Revisa el programa proporcionat, `primo_grande.c`. Aquest programa utilitza els algorismes abans proposats per a cercar i mostrar per pantalla el nombre primer més gran que cap en una variable sencera sense signe de 8 bytes.

3.2 Algorisme paral·lel

En este apartat haureu d'implementar una versió paral·lela amb OpenMP de la funció que esbrina si un nombre és primer o no. Com la part costosa d'aqueixa funció és un bucle `for`, sembla immediata la paral·lelització mitjançant l'ús de `parallel for`. Prova a fer-ho. Què succeeix?

```
Funció primer_gran
  n <- sencer més gran possible
  Mentre n no siga primer
    n <- n - 2
  Fi mentre
  Retorna n
Fi funció
```

Figura 12: Algorisme a paral·lelitzar: cerca el major primer que cap en un enter sense signe de 8 bytes.

```
#pragma omp parallel ...
{
   for (i = ...; p && i <= ...; i += ...)
      if (n % i == 0) p = 0;
}
```

Figura 13: Esquema de paral·lelització del bucle de la funció `primer`.

Efectivament, OpenMP no permet utilitzar la directiva `parallel for` en este cas. Recorda que la directiva `parallel for` equival a una directiva `parallel` seguida d'una directiva `for`. Esta segona directiva s'encarrega de repartir automàticament les iteraciones del bucle entre els fils, però per a poder fer-ho és necessari que l'inici, final i increment del bucle estiguen perfectament delimitats. En la función `primer`, l'inici i l'increment de la variable del bucle (`i`) estan clars, però el valor final és desconegut en un principi: és possible que arribe a `s`, o bé que acabe abans perquè `p` passe a ser fals. Per tant, OpenMP no permet la directiva `for` (ni tampoc `parallel for`).

En realitat, el que no permet que OpenMP puga paral·lelitzar el bucle és incloure la comprovació de primer dins de la condició del bucle. Què passaria si es llevara eixa part de la condició? La funció seguiria sent perfectament correcta, però què inconvenient té? [Nota: Si no s'és capaç de veure l'inconvenient, es pot provar el programa eliminant aqueixa part de la condició, fins i tot en la seua versió paral·lela, ja que en aqueix cas sí es pot paral·lelitzar de forma molt fàcil. Però cal tenir en compte que el temps d'execució del programa s'incrementarà moltíssim.]

Una vegada que s'haja comprès la inconveniència de realitzar així la versió paral·lela, caldrà cercar una altra forma de fer-ho. En aquesta ocasió no va a ser tan trivial com la paral·lelització d'altres programes. Una forma de fer-ho és realitzar un repartiment explícit del bucle entre els fils de l'equip. És a dir, fer de forma explícita el que OpenMP fa de forma automàtica mitjançant la directiva `for`.

La paral·lelització del bucle tindrà la forma que es pot veure en la Figura 13, on cada fil de la regió paral·lela ha de realitzar només un subconjunt de les iteracions del bucle original. Les iteracions que realitza cada fil estan determinades pel valor inicial de `i`, el seu increment i el seu valor final. Per tant, haureu de modificar alguns d'estos valors per a aconseguir que cada fil processe un subconjunt d'iteracions diferent.

Per exemple, suposant `s`=19, els valors de la variable `i` de les diferents iteracions del bucle complet són: 3, 5, 7, 9, 11, 13, 15, 17, 19. Una manera de repartir les iteracions seria assignar a cada fil un bloc d'iteracions consecutives. Per exemple, per a 3 fils:

Fil 0	3	5	7
Fil 1	9	11	13
Fil 2	15	17	19

Una altra manera de repartir les iteracions és fer-ho de forma cíclica:

Fil 0	3	9	15
Fil 1	5	11	17
Fil 2	7	13	19

Este repartiment cíclic és més fàcil d'implementar, per la qual cosa és el que es recomana que utilitzeu en este cas. Lògicament, el repartiment haurà de funcionar correctament per a qualsevol valor de `s` i del nombre

```
Funció conta_primers(últim)
  n <- 2 (per l'1 i el 2)
  i <- 3
  Mentre i <= últim
    Si  i és primer  llavors
      n <- n + 1
    Fi si
    i <- i + 2
  Fi mentre
  Retorna n
Fi funció
```

Figura 14: Algorisme que conta la quantitat de nombres primers que hi ha entre 1 i un valor donat.

de fils. Necessitareu utilitzar funcions d'OpenMP per a obtindre el nombre de fil i l'índex de cada fil. Eviteu cridar a estes funcions en cada iteració del bucle, per qüestions d'eficiència. Desenvolupeu esta nova versió paral·lela i mesura el temps d'execució necessari.

És important conservar la condició d'eixida del bucle quan es veu que el nombre no és primer. D'aquesta manera, (si es fa correctament) quan qualsevol fil descobrisca que el nombre no és primer, tots (tard o d'hora) deixaran de processar el bucle.

Nota: És convenient afegir el modificador de C `volatile` a la variable que s'utilitza com a control del bucle: `volatile int p;` El modificador `volatile` del llenguatge C li indica al compilador que no optimitze l'accés a eixa variable (que no la carregue en registres i que qualsevol accés a ella es faça efectivament sobre memoria), de forma que la seua modificació per part d'un fil serà visible abans en la resta de fils[3].

3.3 Contant primers

Ja que s'està treballant amb nombres primers, plantegem aquest nou problema relacionat: contar la quantitat de nombres primers que hi ha entre l'1 i un nombre gran, per exemple 100000000. [Nota: Si tarda molt, preneu un nombre final menor. Idealment hauria de tardar 1 o 2 minuts l'algorisme seqüencial.]

L'algorisme per a realitzar aquest procés seria el mostrat en la Figura 14. Proveu la versió seqüencial d'aquest algorisme, realitzant mesura del temps d'execució.

Atès que es disposa d'una versió paral·lela amb OpenMP de la funció per a veure si un nombre és primer, és trivial realitzar una versió paral·lela del nou problema sense més que utilitzar la versió paral·lela per a veure si un nombre és primer.

Tanmateix, eixa paral·lelització no condueix a bones prestacions. La raó és que els nombre primers inicials son molt menuts i cada un d'ells suposa molt poc treball, de forma que no hi ha guany al repartir el treball entre múltiples fils.

Una alternativa millor seria paral·lelitzar el bucle del programa principal (que en aquest cas sí que sembla fàcilment paral·lelitzable mitjançant directives OpenMP) i utilitzar la funció `primo` seqüencial original. Desenvolupeu una nova versió paral·lela basada en esta aproximació i mesureu el temps d'execució. Finalment, tragueu temps per a múltiples planificacions de repartiment del bucle, usant almenys les següents planificacions d'OpenMP:

- Estàtica sense especificar grandària de *chunk*.
- Estàtica amb *chunk* 1.
- Dinàmica.

Recordeu que en la directiva `for` d'OpenMP, la planificació es pot indicar de dues maneres alternatives:

[3] Aquest tipus de comportament pot aconseguir-se també usant la sentència `flush` d'OpenMP.

1. Especificant directament la planificació amb una clàusula `schedule` en la directiva `for` d'OpenMP (per exemple, `schedule(static,6)`).

2. Utilitzant la clàusula `schedule(runtime)` en la directiva `for`, y donant-li valor a la variable d'entorn `OMP_SCHEDULE` en executar el programa (per exemple, `OMP_SCHEDULE="static,6"`). Esta opció té l'avantatge de que es pot canviar la planificació sense tindre que recompilar el programa.