UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computació Paral·lela (CPA)

Pràctica 1: Paral·lelització amb OpenMP

Curs 2017/18

Índex

1 Integració numèrica **2**
- 1.1 Paral·lelització de la primera variant . . . 3
- 1.2 Paral·lelització de la segona variant . . . 3
- 1.3 Execució en el cluster . . . 4

2 Processament d'imatges **4**
- 2.1 Descripció del problema . . . 4
- 2.2 Versió seqüencial . . . 5
- 2.3 Implementació paral·lela . . . 5

3 Nombres primers **7**
- 3.1 Algorisme seqüencial . . . 7
- 3.2 Algorisme paral·lel . . . 8
- 3.3 Contant primers . . . 9

Introducció

Aquesta pràctica consta de 3 sessions. 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`
Sessió 3	Nombres primers	`primo_grande.c`, `primo_numeros.c`

Per a compilar els codis, en la majoria de casos serà necessari enllaçar amb la biblioteca matemàtica (perquè s'usen funcions com `sqrt`), és a dir, afegir `-lm` en la línia de compilació. Recordar que la compilació de programes OpenMP requereix indicar l'opció de compilació corresponent, per exemple

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

Per a executar amb diversos fils, es pot fer per exemple:

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

Para a més detalls, consultar la documentació d'ús del clúster de pràctiques.

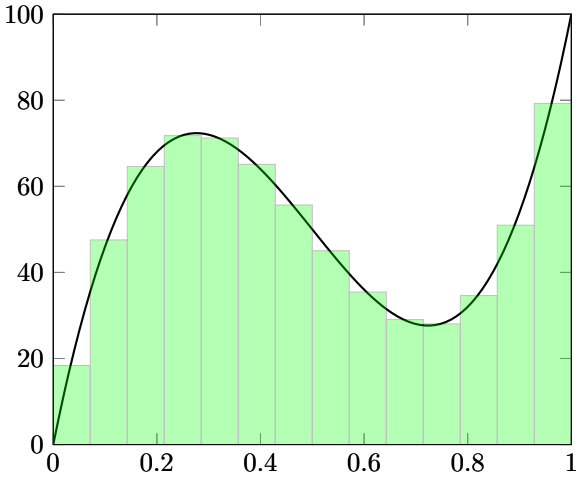

Figura 1: Interpretació geomètrica d'una integral.

1 Integració numèrica

Considerarem ací el càlcul numèric d'una integral de la forma

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

La tècnica que es va 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 pot veure's 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, compila el programa i executa'l. 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 com a segon argument del programa.

```
/* 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.

1.1 Paral·lelització de la primera variant

A continuació, modificarem el fitxer `integral.c` per a realitzar el càlcul de forma paral·lela utilitzant OpenMP. Començarem per la primera variant (`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.

Important: És convenient assegurar-se que el programa està efectivament usant el nombre de fils que hem indicat. Pera a açò, fes que el programa mostre, per mitjà d'un `printf`, el nombre de fils amb els que està treballant. El missatge amb el nombre de fils deuria mostrar-se només una vegada.

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.

```
#!/bin/sh
#PBS -l nodes=1,walltime=00:05:00
#PBS -q cpa
#PBS -d .

OMP_NUM_THREADS=3 ./pintegral 1
```

Figura 3: *Script* per a executar en el sistema de cues.

1.3 Execució en el cluster

Per a executar el programa en el clúster de pràctiques, hem de compilar en la màquina `kahan` i llançar un treball al sistema de cues, seguint les instruccions. Bàsicament, cal escriure un *script* amb les opcions del sistema de cues seguides pels comandos que es volen executar. En la Figura 3 hi ha un exemple en el qual es llança un programa OpenMP amb 3 fils d'execució amb un temps màxim de 5 minuts, en la cua `cpa` i amb l'opció d'executar en el directori actual (`.`). Per a llançar el treball al sistema de cues, suposant que el fitxer de l'*script* es diu per exemple `jobopenmp.sh`, caldria executar:

```
qsub jobopenmp.sh
```

Com a alternativa a indicar el nombre de fils en l'*script*, és possible assignar la variable `OMP_NUM_THREADS` en llançar el treball al sistema de cues:

```
qsub -v OMP_NUM_THREADS=3 jobopenmp.sh
```

Es recorda que es pot consultar l'estat de les cues amb l'ordre `qstat`, i cancel·lar treballs amb l'ordre `qdel`.

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.

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. Els alumnes hauran 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 un cert valor, anomenat 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 propers a la imatge, 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.

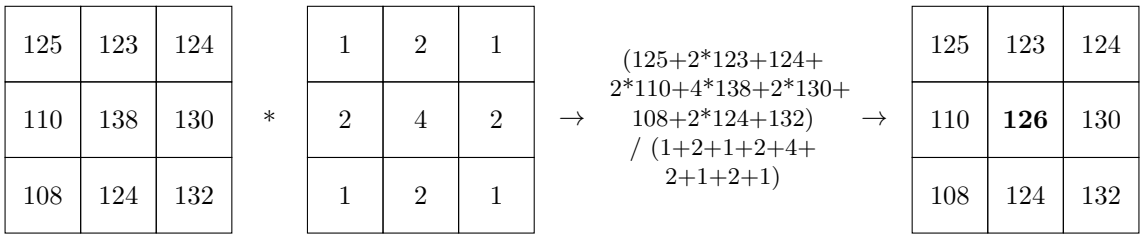

Figura 4: Model d'aplicació d'una mitjana ponderada en el filtrat d'imatges.

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 4 mostra un esquema. En aquesta figura, partim d'una imatge (esquerra) sobre la qual estem aplicant en el seu píxel central 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 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 5.

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 6 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 de la qual està especificat en `IMAGEN_ENTRADA`, aplicarà el filtrat tantes vegades com indica `NUM_PASOS` i amb 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.

L'alumne haurà de comprovar el correcte funcionament del programa i ajustar els valors del radi i el nombre de passos per a la imatge que vulga filtrar. S'adjunta una imatge de prova, procedent d'un famós benchmark[2], veure Figura 7. L'alumne podrà utilitzar les seues pròpies imatges, recomanant-se unes dimensions superiors als 4 Mpíxels.

2.3 Implementació paral·lela

Existeixen diferents aproximacions per a realitzar la implementació paral·lela mitjançant OpenMP. El treball se centrarà a analitzar els cinc bucles i decidir (i provar) quins bucles són susceptibles de ser paral·lelitzats i quines variables haurien de ser compartides o privades. Per a açò, l'alumne haurà de:

- Analitzar si les diferents iteracions tenen alguna dependència entre si (p.e. si la segona iteració utilitza dades generades per la primera) i si aquestes dependències poden resoldre's directament amb alguna clàusula d'OpenMP (p.e. en el cas de sumatoris).
- Una vegada analitzats els bucles susceptibles de ser paral·lelitzats, es procedirà a identificar les variables que hauran de ser privades a cada fil o compartides entre tots.
- Una vegada definida l'aproximació paral·lela, s'implementarà utilitzant les directives OpenMP corresponents.

[1] `http://www.irfanview.com`
[2] `http://en.wikipedia.org/wiki/Standard_test_image`

```
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 5: 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 6: Format d'un fitxer d'imatges PPM.

Figura 7: Imatge de referència (`Lenna.ppm`) abans (esquerra) i després (dreta) d'aplicar un filtrat d'un pas i radi 5.

- Els resultats variaran depenent dels bucles que es paral·lelitzen i per tant s'haurà de realitzar una experimentació que analitze la millor aproximació. Haurà d'instrumentar-se el codi perquè es puga mesurar el temps (`omp_get_wtime`). Per a açò, es mesurarà el temps necessari per a realitzar el procés amb les diferents combinacions. Es recomana utilitzar una taula en la qual s'arreplegue el temps de procés per a cadascun dels bucles per separat.

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 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 8. 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

```
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 8: Algorisme seqüencial per a determinar si n és primer.

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

Figura 9: Algorisme a paral·lelitzar: cerca el major primer que cap en un enter sense signe de 8 bytes.

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 9.

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

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?

Efectivament, OpenMP no permet la paral·lelització directa d'un bucle `for` llevat que estiga perfectament delimitat el seu inici, final i increment. Com en la funció `primo` el bucle pot acabar abans (quan ja s'ha adonat que el nombre no és primer), no és un bucle factible de paral·lelitzar amb `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,

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

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. Caldrà usar primitives i funcions d'OpenMP que permeten obtenir el nombre de fils i executar en paral·lel el bucle, però fent que cada fil realitze només una part d'ell. Desenvolupa aquesta nova versió paral·lela i mesura el temps d'execució necessari. Seria interessant provar tant repartint el bucle de forma cíclica entre els fils com repartint-ho per blocs consecutius.

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.

Observe's que el codi ha de realitzar-se de manera que:

1. El seu funcionament ha de ser correcte independentment del nombre de fils amb que s'execute. En particular, ha de funcionar bé amb un sol fil.

2. Si pot ser, ha de poder-se compilar i executar correctament encara que no s'utilitze OpenMP. Per a açò, cal recórrer a la compilació condicional (directiva `#ifdef`), de manera que es faça ús de funcions OpenMP solament si s'usa OpenMP en la compilació (o siga, si el símbol `_OPENMP` està definit).

Nota: Potser resulte 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 10. 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

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

veure si un nombre és primer. Desenvolupa aquest algorisme paral·lel per a resoldre el problema i mesura els temps d'execució.

No acaba de funcionar bé. Els nombres primers inicials són molt xicotets com per a suposar una càrrega de treball suficient que garantisca guany en repartir el treball entre múltiples fils. Açò es pot alleujar una mica afegint a la regió paral·lela una clàusula `if`, perquè s'execute en paral·lel només quan la grandària del problema siga major. Si bé en començar a treballar en paral·lel amb un problema major es va a repartir millor el treball, açò també suposa que tots els nombres anteriors a aqueix valor s'hauran calculat de forma seqüencial. Aquesta solució segueix sense comportar-se especialment bé.

Una estratègia alternativa seria paral·lelitzar el bucle del programa principal, que en aquest cas sí que sembla fàcilment paral·lelitzable mitjançant directives OpenMP. Desenvolupar una nova versió paral·lela per a aquest problema que faça aqueix bucle en paral·lel, usant per a la funció `primo` la funció seqüencial original. Mesurar temps d'execució per a aquest nou algorisme. Finalment, traure temps per a múltiples planificacions de repartiment del bucle, usant almenys les següents planificacions d'OpenMP:

- Estàtica sense *chunk*.
- Estàtica amb *chunk* 1.
- Dinàmica.

Recordeu que pot resultar còmode especificar la planificació amb la corresponent variable d'entorn, per a açò s'ha d'indicar en el codi `runtime` com a tipus de planificació.