Open MP

Computació Paral·lela I Massiva

Grau en Informàtica

Carles Aliagas 2013
carles.aliagas@urv.net

Model de programació memòria compartida: OpenMP

1. Model d'execució
2. Directives i compilació
3. Constructor *parallel*
4. Particionat: *for, sections, single*
5. Sincronització:
   master,critical,barrier,atomic,flush,ordered
6. Entorn de les dades:
   private,firstprivate,shard,default,reduction,copyin
7. Exercicis
8. Funcions de llibreria
9. Funcions de bloqueig

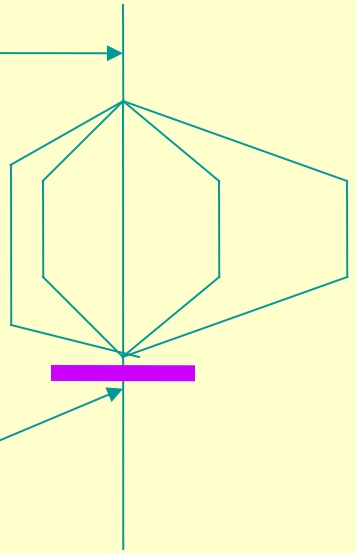
1.- Model Execució

- MIMD: Memòria Compartida
- Model fork & join amb threads

Comença el thread màster

es creen threads secundaris per a executar les porcions

tots els threads s'esperen a que hagin acabat els altres per a que continuï el thread màster



2.- Directives i compilació

- Es parteix d'un codi escrit en alt nivell
- S'afegeixen directives de la forma:

```
#pragma omp ..........
```

- Es compila usant el compilador de OpenMP
- Genera codi en llenguatge màquina i crides a sistema per a manipular els threads

3.- Contructor *Parallel*

```
#pragma omp parallel [clàusules]
   {
   }
```

- El codi que hi ha dins de les claus s'executarà de manera paral·lela per un conjunt de threads
- Les clàusules determinen alguna característica sobre l'execució d'aquest codi:

  `if (expressió)`

  es crearan el conjunt de threads només si l'expressió és certa

  Altres clàusules determinen l'àmbit i comportament de variables:

  private,....,reduction (mes endavant ja es veurà)

Contructor *Parallel*

- Al final de l'execució es produeix una sincronització per barrera.
- Tots els threads esperen i finalitzen al final de la zona paral·lela.
- Només el thread màster continua la seva execució de manera seqüencial.

4.- Particionat

- Decidir com es divideix la feina de la zona paral·lela per a que sigui la unitat d'assignació als threads paral·lels.
- `for, sections, single`.

Constructor *for*

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

- Cada iteració del bucle és una unitat d'assignació de feina per a un thread.
- Els bucles for tenen certes restriccions:
  - el format és:

```
for(assignació; var cmp valor;
 increment)
```

Constructor *for*

- Les clàusules són:
  - àmbit de les variables (*private,..., reduction*)
  - `schedule`: Determina el particionat i assignació d'iteracions als threads.
  - `ordered`:permet la inclusió de directives *ordered* dins el bucle *for*
  - `nowait`: elimina la barrera per defecte que hi ha al final d'un bucle *for*

Constructor *for*

- clàusula `schedule`:
  - static: `schedule(static,niter)`
    - Les iteracions es divideixen en trossos de *niter* iteracions. S'assignen en ordre del número de thread al principi del bucle de manera estàtica.
    - Si no s'especifica *niter* els trossos seran de mida N/nombre de threads.
    - Es coneix quin thread farà cada iteració !!

Constructor *for*

- clàusula `schedule`:
  - dynamic: `schedule(dynamic,niter)`
    - Les iteracions es divideixen en trossos de *niter* iteracions. S'assigna inicialment un tros a cada thread i es van assignant més trossos a mesura que acaben el que tenien assignats.
    - Si no s'especifica *niter* els trossos seran de 1 iteració.
    - Es balancejen les iteracions entre els threads.

Constructor *for*

- clàusula `schedule`:
  - guided: `schedule(guided,mida)`
    - Les iteracions es divideixen en trossos de mida dinàmica començant per N/num_threads i acabant per *mida*. L'assignació és dinàmica i es fa a mesura que acaben els trossos ja assignats.
    - Si no s'especifica *mida* es considera el valor de 1.
    - Són dinàmics l'assignació d'iteracions i la mida del bloc a assignar.

Constructor *for*

- clàusula `schedule`:
  - runtime: `schedule(runtime)`
    - La decisió del tipus d'scheduling es deixa fins la moment de l'execució. Es consulta la variable d'entorn `OMP_SCHEDULE`

exemple:a1.c

```
#define N 10000000

int a[N],b[N];
long long s=0;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
}
```

exemple:a1.c

```
#pragma omp parallel for
for (i=0;i<N;i++)
 b[i] += a[i];

 printf("Valor i %d, de b[i] %d \n",i-1,b[i-1]);

for (i=0;i<N;i++)
 s+=b[i];

 printf("Valor %d, de b %d suma total: %ld\n",
   i-1,b[i-1],s);

}
```

Constructor *section*

```
#pragma omp sections [clausules]
  {
   [#pragma omp section]
     {}
   [#pragma omp section
     {} ]
  }
```

- Cada secció és una unitat d'assignació de feina per a un thread.
- La primera definició de *section* és opcional.

Constructor `section`

- Les clàusules són:
  - àmbit de les variables (*private,..., reduction*)
  - `nowait`: elimina la barrera per defecte que hi ha al final d'un bucle *for*

Constructor *single*

```
#pragma omp single [clausules]
   {
   }
```

- El bloc s'ha d'executar per només un únic thread. Hi ha una sincronització per barrera.
- les clàusules són:
  - private,firstprivate
  - nowait

5.- Sincronització

- master
- critical
- barrier
- atomic
- flush
- ordered

Constructor *master*

```
#pragma omp master
  {
  }
```

- El bloc s'ha d'executar pel thread master. thread.
- No hi ha una sincronització per barrera ni a l'entrada ni a la sortida.

Constructor *critical*

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

- Crea una secció crítica d'accés al bloc. Només és permet que un únic thread estigui executant dins de la secció critica.
- Poden haver varis trossos de codi separats que pertanyin a la mateixa secció crítica. Només cal que tinguin el mateix *nom*

Constructor *barrier*

```
#pragma omp barrier
```

- Força una sincronització per barrera. On tots els threads a mesura que arriben s'esperen a que arribin tots els altres.
- Quan tots han arribat, tots continuen la seva execució.

Constructor *atomic*

```
#pragma omp atomic
  expressió;
```

- *Expressió* modifica el valor d'una posició de memòria.
- Aquesta modificació es fa de manera atòmica.
- Un cop un thread està fent la modificació els altres hauran d'esperar-se si volen fer també aquesta modificació.
- És més eficient que critical, ja que el bloqueig va per zones de memòria i no per codi

Constructor *flush*

```
#pragma omp flush [(list)]
```

- Sincronitza els objectes compartits que hi ha a *list*.
  - Una dada que està en un registre es copia a memòria
  - Una dada de memòria s'escriu i es torna llegir.
  - ....
- si no hi ha list es sincronitzen tots els objectes compartits.

Constructor *flush*

- Es fa de manera implicita, sempre que no hi hagi un *nowait* amb els següents constructors:
  - *barrier*
  - al entrar i sortir de *critical* i de *ordered*
  - al sortir de *parallel, for, sections* i *single*

Constructor *ordered*

```
#pragma omp ordered
   {
   }
```

- provoca que el codi del bloc s'executi com si s'estés executant de manera seqüencial
- S'usa dins d'un constructor *for*.

6.- Entorn de les dades

- Aquí es veu els diferents àmbits d'accés i comportament de les dades compartides i locals dels threads quan s'executen.
- Per defecte tota variable declarada és ***compartida*** per tots els threads
- Es poden definir privades amb una directiva:
  - threadprivate
- o amb clàusules dins dels constructors estudiats fins al moment.
  - private, firstprivate, lastprivate, shared, default, reduction i copyin

Directiva *threadprivate*

```
#pragma omp threadprivate (list)
```

- Indica una llista de variable que seran privades als threads que la manipulen.
- La mateixa variable estarà creada tantes vegades com threads hi hagin i cadascuna estarà manipulada només pel seu thread

Clàusula *private*

```
... private (list)
```

- Indica una llista de variable que seran privades als threads que la manipulen.
- El valor inicial es indeterminat.

Clàusula *firstprivate*

```
... firstprivate (list)
```

- El valor inicial és el que tenia la variable quan es va inicialitzar per primera vegada en el programa.
- Es fa una còpia del valors a totes les zones privades de cada thread

Clàusula *lastprivate*

```
... lastprivate (list)
```

- Al sortir de la zona compartida el valor copiat a la variable original és el que ha calculat l'última iteració (en ordre seqüencial) del bucle *for* o l'ultima secció d'un contructor *sections*

Clàusula *shared*

```
... shared (list)
```

- Les variables de la llista seran compartides per tots els threads de la zona compartida.

Clàusula *default*

```
... default (shared|none)
```

- Les variables de la zona compartida seran per defecte compartides (shared) o s'haurà d'especificar de manera explícita amb alguna clàusula (none).

Clàusula *reduction*

```
... reduction (op:list)
```

- Les variables de la llista es crearan de manera privada a tots els threads de la zona compartida i a mesura que vagin acabant s'anirà copiant el seu valor privat a la dada original fent una operació *op.*
- Al final la dada tindrà els acumulats de la operació *op* de totes les copies privades

Clàusula *copyin*

```
... copyin (list)
```

- Les variables de la llista es crearan de manera privada a tots els threads de la zona compartida i seran inicialitzats amb el valor que en aquell moment tingui la copia del *master thread*.

Exemples

exemple:a4.c

```
#define N 10000

int a[N],b[N],c[N];
long long s=0;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
 c[i]=0;
}
```

exemple:a4.c

```
#pragma omp parallel
{
  #pragma omp for nowait
    for (i=0;i<N;i++)
     b[i] += a[i];

  #pragma omp for nowait
    for (i=0;i<N;i++)
     c[i] += a[i];
}
```

exemple:a4.c

```
/* checksum */
s=0;
for (i=0;i<N;i++)
 s+=b[i];

 printf("Valor %d, de b %d suma total: %ld\n",
   i-1,b[i-1],s);

/* checksum */
s=0;
for (i=0;i<N;i++)
 s+=c[i];

 printf("Valor %d, de c %d suma total: %ld\n",
   i-1,c[i-1],s);
}
```

exemple:a5.c

```
#define N 1000000
#define MAX 4

int a[N],b[N],ind[N];
long long s=0;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
 ind[i]=i%MAX;
}
```

exemple:a5.c

```
#pragma omp parallel for
for (i=0;i<N;i++)
 #pragma omp critical (sc_b)
 b[ind[i]] += a[i];

for (i=0;i<MAX;i++)
{
 printf("Valor %d, de b %d \n",i,b[i]);
 s+=b[i];
}

 printf("Suma total de b: %ld\n",s);

}
```

exemple:a7.c

```
#define N 10000000

int a[N],b[N];
long long s=0;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
}
```

exemple:a7.c

```
#pragma omp parallel for
for (i=0;i<N;i++)
 b[i] += a[i];

 printf("Valor i %d, de b[i] %d \n",i-1,
   b[i-1]);

#pragma omp parallel for reduction(+:s)
for (i=0;i<N;i++)
 s+=b[i];

 printf("Valor %d, de b %d suma total:
   %ld\n",i-1,b[i-1],s);
}
```

exemple:a12.c

```
#define N 1000000
#define MAX 4

int a[N],b[N],ind[N];
long long s=0;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
 ind[i]=i%MAX;
}
```

exemple:a12. BAD

```
#pragma omp parallel for
for (i=0;i<N;i++)
 b[ind[i]] += a[i];

for (i=0;i<MAX;i++)
{
 printf("Valor %d, de b %d \n",i,b[i]);
 s+=b[i];
}
 printf("Suma total de b: %ld\n",s);
}
```

exemple:a12. OK

```
#pragma omp parallel for
for (i=0;i<N;i++)
 #pragma omp atomic
 b[ind[i]] += a[i];

for (i=0;i<MAX;i++)
{
 printf("Valor %d, de b %d \n",i,b[i]);
 s+=b[i];
}
 printf("Suma total de b: %ld\n",s);
}
```

7.- exercicis

exercici:8 reines

```
#define R 8

int r[R+2];

main()
{
int i,a,b;

 r[1]=1;
 /*printf ("Reina 1, pos 1 \n"); */
 a=2; r[2]=1;
 /*printf ("Reina 2, pos 1 \n"); */
```

exercici:8 reines

```
while (a > 0)
  {
   if (r[a] > R) /* s'han explorat totes les
   alternatives d'una reina */
    {
     a--;
     r[a]++;
    }
   else
    {
     b=1;
     for (i=1;i<a;i++)
       b = b && ((r[a] != r[i]) &&
        (r[a] != r[i]+(a-i)) &&
        (r[a] != r[i]+(i-a)));
```

exercici:8 reines

```
        if (b)
         {   /* s'han explorat totes les reines */
          if (a == R)          {
              psol(r,R);
              r[a]++;
           }
           else {  /* de moment OK, pasem a la
                      seguent reina*/
             a++;
             r[a]=1;
           }
          }
         else /* alternativa incorrecta, mirem la
        seguent */
           r[a]++;
          }
        }
}
```

exercici:8 reines

```
psol(int* r,int nr)
{
int i;

 printf("Sol: ");
 for(i=1;i<=nr;i++) printf("%d ",r[i]);
 printf("\n");
}
```

8.- Funcions de llibreria

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num
- omp_get_num_procs
- omp_in_parallel

omp_set_num_threads

```
#include <omp.h>
void omp_set_num_threads( int num_threads)
```

- Funció cridada des de la zona sequèncial del codi
- Estableix el nombre de threads que s'han de crear dins d'una zona paral·lela
- Té preferència sobre la variable d'entorn `OMP_NUM_THREADS`

omp_get_num_threads

```
#include <omp.h>
int omp_get_num_threads( void )
```

- Funció cridada des de la regió paral·lela. Si es crida des d'una zona seqüencial retorna 1.
- Torna el nombre de threads que s'han creat per a executar una zona paral·lela.

omp_get_max_threads

```
#include <omp.h>
int omp_get_max_threads( void )
```

- Torna el nombre de threads que es poden crear com a molt per a executar una zona paral·lela.
- S'utilitza per a definir una taula de threads i no tenir problemes de fora de rang.

omp_get_thread_num

```
#include <omp.h>
int omp_get_thread_num( void )
```

- Torna el número del thread que crida a la funció
- Els threads s'enumeren de 0,... (omp_get_num_threads) -1
- El thread master té número 0. Si es crida des d'una zona sequèncial torna 0.

omp_get_num_procs

```
#include <omp.h>
int omp_get_num_procs( void )
```

- Torna el nombre màxim de processadors que es poden assignar a l'execució de l'aplicació.

omp_in _parallel

```
#include <omp.h>
int omp_in_parallel( void )
```

- Torna un booleà indicant si ens trobem en una zona paral·lela que s'està executant de manera paral·lela.

Exemple:a3

```
#define N 10000000

int a[N],b[N];
long long s=0;

int jo,nthr,porcio;

main()
{
int i;

/* inicialitzacio, no en paral.lel */

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
}
```

Exemple:a3

```
#pragma omp parallel shared(a,b) private(i,jo,nthr,porcio)
{
 jo = omp_get_thread_num();
 nthr = omp_get_num_threads();
 porcio= N/nthr;

 /* falla l'arrodoniment */
 printf("jo: %d, porcio: %d de %d a %d\n", jo, porcio,
jo*porcio, (jo+1)*porcio);

 for (i=jo*porcio;i<(jo+1)*porcio;i++)
 b[i] += a[i];
}

 printf("Valor i %d, de b[i] %d \n",i-1,b[i-1]);

for (i=0;i<N;i++)
 s+=b[i];

 printf("Valor %d, de b %d suma total: %ld\n",i-1,b[i-1],s);

}
```

9.- Funcions de bloqueig

Funcions que implementen un test&set mitjançant software.

- omp_init_lock
- omp_destroy_lock
- omp_set_lock
- omp_unset_lock
- omp_test_lock

L'estat d'un lock pot ser:
ocupat/no_ocupat

omp_init _lock

```
#include <omp.h>
void omp_init_lock( omp_lock_t * lock )
```

- Inicialitza l'estructura d'un lock per a ser usat a posteriori
- L'estat inicial d'un Lock inicialitzat és *no_ocupat*.

omp_destroy _lock

```
#include <omp.h>
void omp_destroy_lock( omp_lock_t * lock )
```

- Elimina la informació d'un lock. A partir d'aquest moment no es podrà utilitzar
- L'estat del Lock ha de ser *no_ocupat*.

omp_set _lock

```
#include <omp.h>
void omp_set_lock( omp_lock_t * lock )
```

- Bloqueja un thread mentre el lock estigui *ocupat.* Un cop ha estat desocupat l'ocupa ell i aquest thread pot continuar la seva exucució.

omp_unset _lock

```
#include <omp.h>
void omp_unset_lock( omp_lock_t * lock )
```

- Allibera un lock.
- L'estat del lock passa a *no_ocupat.*
- L'únic que pot alliberar un lock és el thread que l'havia ocupat.

omp_test _lock

```
#include <omp.h>
int omp_test_lock( omp_lock_t * lock )
```

- Funció equivalent a fer un `omp_set_lock`. La diferencia és que el thread que intenta ocupar el lock no queda bloquejat .
- La funció retorna 0 (FALSE) si el lock ja està ocupat i diferent de 0 (TRUE) si ha pogut ocupar el lock.

Exemple: a16

```
#include <omp.h>

#define N 1000000
#define TRUE 1
#define FALSE 0

int a[N],b[N];
long long s=0;

main()
{
omp_lock_t bloq;
int i,jo, primers =0;

/* inicialitzacio, no en paral.lel */

omp_init_lock(& bloq);

for(i=0;i<N;i++)
{
 a[i]=1;
 b[i]=2;
}
```

Exemple: a16

```
#pragma omp parallel for private(jo)
for (i=0;i<N;i++)
{
 omp_set_lock(&bloq);
  primers ++;
  if ( primers < 15 )
   {
    jo = omp_get_thread_num();
    printf(" El thread %d executa la iteracio
        %d\n",jo,i);
   }
 omp_unset_lock(&bloq);
 b[i] += a[i];
}

 printf("Valor i %d, de b[i] %d \n",i-1,b[i-1]);
```

Exemple: a16

```
primers = 0;

#pragma omp parallel for reduction(+:s)
for (i=0;i<N;i++)
{
 if (omp_test_lock(&bloq))
  {
   primers ++;
   if ( primers < 15 )
    {
     jo = omp_get_thread_num();
     printf(" El thread %d executa la iteracio
        %d\n",jo,i);
    }
   omp_unset_lock(&bloq);
  }

 s+=b[i];
}
 printf("Valor %d, de b %d suma total: %ld\n",
        i-1,b[i-1],s);
}
```