Algoritmos Paralelos

Ejemplos prácticos de OPENMP (1)

(clase 10.11.15)

Prof. J.Fiestas

Ejemplos prácticos de OPENMP:

Uso de memoria en OPENMP:

Despues de Print 1, el valor de x puede ser 2 o 5, dependiendo del tiempo de ejecución de los hilos (threads) y de como se escribe en x.
x puede NO ser 5 en Print1 porque:

- Print1 puede ejecutarse antes de asignar el valor de x
- Si se ejecuta despues de asignar x, *thread 1* podria no ver el valor porque no se ha actualizado la memoria (flush) por *thread 0*

```
#include <stdio.h>
#include <omp.h>
int main(){
int x;
x = 2;
#pragma omp parallel num_threads(2) shared(x) {
if (omp_get_thread_num() == 0) {
x = 5;
} else {
/* Print 1: the following read of x has a race */
printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x ); }

if (omp_get_thread_num() == 0) {
/* Print 2 */
printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} else {
/* Print 3 */
printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} }
return 0; }
```

Ejemplos prácticos de OPENMP:

Uso de memoria en OPENMP:

Por ello, la barrera #pragma omp barrier luego de Print1 contiene flushes implícitos en todos los *threads*, para garantizar que el valor 5 sea imprimido por Print2 y Print3

```
#include <stdio.h>
#include <omp.h>
int main(){
int x;
x = 2;
#pragma omp parallel num_threads(2) shared(x) {
if (omp_get_thread_num() == 0) {
x = 5;
} else {
/* Print 1: the following read of x has a race */
printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x ); }
#pragma omp barrier
if (omp_get_thread_num() == 0) {
/* Print 2 */
printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} else {
/* Print 3 */
printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
} }
return 0; }
```

Variables de control interno (ICVs):

Cada tarea generada en una región en paralelo recibe una copia de ICVs, que controlan el comportamiento del programa.

OMP_SET_NUM_THREADS: define el número de threads que serán usados en la region que sigue

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

OMP_SET_MAX_ACTIVE_LEVELS: Limita el número de regiones en paralelo anidadas

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

OMP_SET_NESTED: Activa/desactiva paralelismo anidado

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

OMP_SET_DYNAMIC: Activa/desactiva ajuste dinámico del número de threads accesibles en la region en paralelo

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

Variables de control interno (ICVs):

La region en paralelo externa crea grupos de 2 *threads* que van a ejecutar la tarea
Cada tarea llama omp_set_num_threads(3) y crea un grupo de 3 *threads*, cada uno ejecutando una tarea dentro de la region paralela interna.
Cada tarea de la region en paralelo interna va a asignar 4 a *nthreadsvar*

```
int main (void) {
omp_set_nested(1);
omp_set_max_active_levels(8);
omp_set_dynamic(0);

omp_set_num_threads(2);
#pragma omp parallel {

      omp_set_num_threads(3);
      #pragma omp parallel {

      omp_set_num_threads(4);
      #pragma omp single {
      printf ("Inner: max_act_lev=%d, num_thds=%
max_thds=%d\n", omp_get_max_active_levels(),
omp_get_num_threads(), omp_get_max_threads(
}
}
```

El print de la region externa se ejecuta solo por uno de los *threads* del grupo, mientras que el print de la región interna hace lo mismo 2 veces, ya que existen dos *threads* en esta región en paralelo

```
#pragma omp barrier
#pragma omp single
{
printf ("Outer: max_act_lev=%d, num_thds=
%d,
max_thds=%d\n”,
omp_get_max_active_levels(),
omp_get_num_threads(),
omp_get_max_threads());
}
}
return 0;
}
```

```
Inner: max_act_lev=8, num_thds=3, max_thds=4
Inner: max_act_lev=8, num_thds=3, max_thds=4

Outer: max_act_lev=8, num_thds=2, max_thds=3
```

Constructor **parallel:**

Se puede usar en algoritmos paralelos de granularidad gruesa (*coarse grain parallel programms*)
En el ejemplo, cada *thread* en la región en paralelo, decide en que parte del array x va a trabajar, basada en el número total de *threads*

```
int main()
{
float array[10000];
sub(array, 10000);
return 0;
```

```
#include <omp.h>
void subdomain(float *x, int istart, int ipoints)
{
int i;
for (i = 0; i < ipoints; i++)
x[istart+i] = 123.456;
}
void sub(float *x, int npoints)
{
int iam, nt, ipoints, istart;
#pragma omp parallel default(shared)
private(iam,nt,ipoints,istart)
{
iam = omp_get_thread_num();
nt = omp_get_num_threads();
ipoints = npoints / nt; /* size of partition */
istart = iam * ipoints; /* starting array index */
if (iam == nt-1) /* last thread may do more */
ipoints = npoints - istart;
subdomain(x, istart, ipoints);  }
}
```

Constructor **sections:**

La cláusula firstprivate inicializa la copia privada de section_count en cada hilo.
Como las secciones modifican section_count, esto quiebra la independencia de las secciones.
Si las secciones son ejecutadas por hilos distintos, cada uno imprimira el valor 1.
Si el mismo hilo ejecuta ambas secciones, una imprimira 1 y la otra, 2.

```
int main( ) {
   int section_count = 0;
   omp_set_dynamic(0);
//    omp_set_num_threads(NT);
#pragma omp parallel
#pragma omp sections firstprivate( section_count )
{
#pragma omp section
   {
      section_count++;
      /* may print the number one or two */
      printf( "section_count %d\n", section_count );
   }
#pragma omp section
   {
      section_count++;
      /* may print the number one or two */
      printf( "section_count %d\n", section_count );
   }
}
   return 0;
}
```

Cláusula **nowait**:

En el ejempo, *static schedule* distribuye la misma secuencia numérica de iteración lógica entre los *threads* que ejecutan los 3 ciclos (*loops*) en paralelo. Esto permite el uso de nowait, incluso cuando hay una dependencia entre los ciclos. Siempre y cuando el mismo thread ejecute la misma secuencia numérica en la iteración, en cada ciclo. Note que el número de iteraciones es el mismo en cada ciclo.

```
#include <math.h>
void nowait_example2
(int n, float *a, float *b, float *c, float *y, float *z)
{
int i;
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)
c[i] = (a[i] + b[i]) / 2.0f;
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)
z[i] = sqrtf(c[i]);
#pragma omp for schedule(static) nowait
for (i=1; i<=n; i++)
y[i] = z[i-1] + a[i];
}
}
```

Cláusula **collapse**:

Los ciclos k,j estan asociados al constructor, es decir, las iteraciones k y j se colapsan en un ciclo con un espacio de iteración mas grande, y este último es dividido entre los *threads*

La secuencia de ejecución de las iteraciones k y j determinan el orden de iteración en el ciclo colapsado. Esto implica que el último valor de k será 2 y de j será 3.

Ya que klast y jlast son lastprivate, sus valores son asignados por la última iteración del ciclo k-j, colapsado.

Este ejemplo imprime: **2 3**

```
#include <stdio.h>
void test()
{
int j, k, jlast, klast;
#pragma omp parallel
{
#pragma omp for collapse(2) lastprivate(jlast, klast)
for (k=1; k<=2; k++)
for (j=1; j<=3; j++)
{
jlast=j;
klast=k;
}
#pragma omp single
printf("%d %d\n", klast, jlast);
}
}
```

Constructor **single:**

En el ejemplo, solo un *thread* imprime cada uno de los mensajes de avance del trabajo. El resto de *threads* evitará la region single y se detendra en la barrera implícita al final del constuctor single.

Si algun *thread* puede continuar sin necesidad de esperar, como en el caso del último trabajo, se puede incluir una cláusula nowait.
En este caso no es necesario saber cual de los *threads* ejecutará cual region single.

```
#include <stdio.h>
void work1() {}
void work2() {}
void single_example()
{
#pragma omp parallel
{
#pragma omp single
printf("Beginning work1.\n");
work1();
#pragma omp single
printf("Finishing work1.\n");
#pragma omp single nowait
printf("Finished work1 and beginning work2.\n");
work2();
}
}
```

Constructor **task**:

El ejemplo muestra como recorrer una estructura de árbol utilizando task
La función traverse debe llamarse desde una región en paralelo para que se ejecute de la misma forma.
Ya que no estan sincronizadas, las tareas no se harán en un orden determinado.

```
struct node {
struct node *left;
struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
if (p->left)
#pragma omp task // p is firstprivate by default
traverse(p->left);
if (p->right)
#pragma omp task // p is firstprivate by default
traverse(p->right);

process(p);
}
```

Constructor **task**:

Aqui se fuerza un recorrido post-orden del árbol, utilizando un taskwait, que garantiza que ambos nodos (izquierda y derecha) sean ejecutados antes que el nodo actual se procese (process(p))

```
struct node {
struct node *left;
struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
if (p->left)
#pragma omp task // p is firstprivate by default
traverse(p->left);
if (p->right)
#pragma omp task // p is firstprivate by default
traverse(p->right);
#pragma omp taskwait
process(p);
}
```

Ejercicio 1:

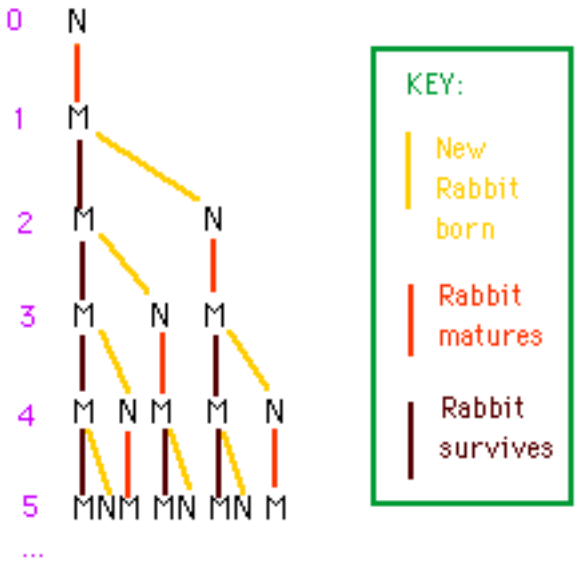
Programar la función Fibonacci en paralelo utilizando directivas OPENMP

La función debe ser llamada desde una región en paralelo, y utilizando el constructor **task** para calcular ambas componentes de la función recursiva.

Asegúrese que ambos términos sean calculados antes de ejecutar la suma y retornarla.

Desde la región paralela en el **main()** solo un thread debe llamar a la función para evitar competencia entre threads.

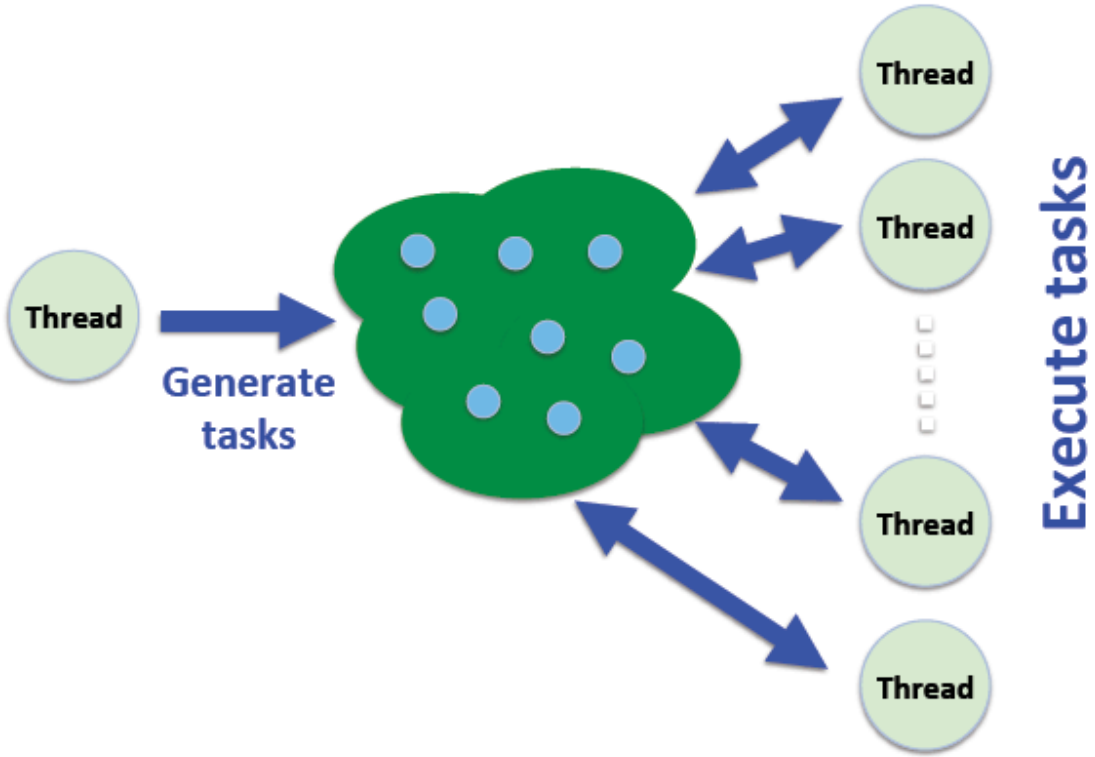
```
int fib(int n) {
int i, j;

if (n<2)
return n;

else {
i=fib(n-1);
j=fib(n-2);
return i+j;
}
}
```



Cláusula **depend** (dependencia)

```
#pragma omp task depend(dependency-type: list)
... structured block ...
```

*Dependencia tipo **in***: la tarea depende de todas las tareas antes generadas con al menos uno de sus elementos de lista en una cláusula in o inout
Dependencia tipo ***out*** y ***inout***: la tarea depende de todas las tareas antes generadas con al menos uno de sus elementos de lista en una cláusula in, out o inout



Constructor **task**:

depend(dependence-type : list)

La cláusula **depend** condiciona el orden de tareas (tasks)
El tipo puede **ser in, out, inout**
La dependencia se da por las referencias a variables dadas por tasks anteriores.
El código imprime "**x=2**", ya que la dependencia se da antes de la impresión

```
int main()
{
  int x = 1;
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task shared(x) depend(out: x)
      x = 2;
    #pragma omp task shared(x) depend(in: x)
      printf("x = %d\n", x);
  }
  return 0;
}
```

Constructor **task**:

depend(dependence-type : list)

El código imprimira
"**x=1**"
Ya que la dependencia con x=2 esta despues de la impresión

```
int main()
{
  int x = 1;
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task shared(x) depend(in: x)
      printf("x = %d\n", x);
    #pragma omp task shared(x) depend(out: x)
      x = 2;
  }
  return 0;
}
```

Constructor **task**:

El código imprimira "**x=2**" ya que la dependencia con x=2 se da al final de la secuencia de tasks, al tener que esperar que ambos se ejecuten (debido al **taskwait**)

```
int main()
{
  int x;
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task shared(x) depend(out: x)
      x = 1;
    #pragma omp task shared(x) depend(out: x)
      x = 2;
    #pragma omp taskwait
    printf("x = %d\n", x);
  }
  return 0;
}
```

Cláusula **depend** (competencia de dependencias)

El programa muestra una potencia concurrencia de ejecución de tareas usando dependencias múltiples con la cláusula depend en el constructor task.
Las dos últimas tareas son dependientes de la primera. Pero no hay dependencia entre las últimas dos tareas, que pueden ejecutarse en cualquier orden, o en competencia, si está disponible mas de un *thread*.
Por ello, posibles salidas son
"**x+1 = 3. x+2 = 4**" y
"**x+2 = 4. x+1 = 3**"

```
#include <stdio.h>
int main()
{
int x = 1;
#pragma omp parallel
#pragma omp single
{
#pragma omp task shared(x) depend(out: x)
x = 2;
#pragma omp task shared(x) depend(in: x)
printf("x + 1 = %d. ", x+1);
#pragma omp task shared(x) depend(in: x)
printf("x + 2 = %d\n", x+2);
}
return 0;
}
```

Constructor **taskgroup**

Permite agrupar y sincronizar tareas (*tasks*).
En el ejemplo, un *task* se crea en la región paralela, y luego un recorrido de árbol en paralelo es creado (**compute_tree()**). **Taskgroup** asegura que **start_background_work()** no participe de la sincronización y se pueda ejecutar en paralelo libremente. Lo contrario pasa con **taskwait** (donde todas las tareas participan de la sincronización)

#pragma omp taskgroup new-line

```
int main() {
int i;
tree_type tree;
init_tree(tree);
#pragma omp parallel
#pragma omp single {
#pragma omp task
start_background_work();
for (i = 0; i < max_steps; i++) {
#pragma omp taskgroup {
#pragma omp task
compute_tree(tree);
} // wait on tree traversal in this step
check_step();
}
} // only now is background work required to be complete
print_results();
return 0;
}
```

Ejercicio 2: **taskgroup**

Completar el código taskgroup.c , tal que:

- Se inicialice un árbol de 10 nodos con enteros aleatorios entre 1 y 50
- La función start_background_work() calcule la suma de los valores de los nodos del árbol
- La función compute_tree() recorra el árbol en postorden y eleve los valores al cuadrado (compute_something)
- Se impriman los valores de los nodos del árbol (puede ser en postorden)

Constructor **taskyield**

Los *tasks* ejecutan **something_useful()**, para luego hacer cálculos en una región crítica. **Taskyield** permite suspender el *task* actual si es que no puede acceder a la región crítica, y ejecutar otro *task* en vez de ese.

#pragma omp taskyield new-line

```
#include <omp.h>
void something_useful ( void );
void something_critical ( void );
void foo ( omp_lock_t * lock, int n )
{
int i;
for ( i = 0; i < n; i++ )
#pragma omp task
{
something_useful();
while ( !omp_test_lock(lock) ) {
#pragma omp taskyield
}
something_critical();
omp_unset_lock(lock);
}
}
```

Ejercicio 3: **multiplicación de matrices**

Programar la multiplicación de dos matrices en paralelo, utilizando OpenMP, de 1000 x 1000 elementos , que sean enteros entre 0 y 99
Calcular tiempos de ejecución para np=2,4,8,16 y representarlo en una gráfica
Calcule los FLOPs (Operaciones de coma flotante por segundo) y grafique FLOPs vs np