Programación con OpenMP

Programación de Arquitecturas Multinúcleo
Grado en Ingeniería Informática

Curso 2023/24

Contenido

1. Introducción
2. Constructores
3. Cláusulas de alcance de datos
4. Funciones de librería
5. Variables de entorno
6. Tareas en OpenMP

Contenido

1. Introducción

- Modelo de programación `fork-join`, con generación de múltiples *threads*.
- Inicialmente se ejecuta un *thread* hasta que aparece el primer constructor paralelo, se crean *threads* esclavos y el que los pone en marcha es el maestro.
- Al final del constructor se sincronizan los *threads* y continúa la ejecución el maestro.

1. Introducción
1.1. Ejemplo inicial: ejemplo hello.c

```
#include <omp.h>
int main()
{
  int iam =0, np = 1;
   #pragma omp parallel private(iam, np)
    {
        #if defined (_OPENMP)
            np = omp_get_num_threads();
            iam = omp_get_thread_num();
        #endif
    printf("Hello from thread %d out of %d \n",iam,np);
    }
}
```

1. Introducción
1.2. Directivas

```
#pragma omp nombre-directiva [cláusulas]
```

Contenido

2. Constructores
2.1. Constructor `parallel`

```
#pragma omp parallel [cláusulas]
    bloque de instrucciones
```

- Se crea un grupo de *threads*. El que los pone en marcha actúa de maestro
- Con cláusula **`if`** se evalúa su expresión
  - si expresion<>0 se crean los *threads*
  - si expresion==0 no se crean más *threads*. Se ejecuta en secuencial
- Número de *threads* a crear se obtiene por variables de entorno o llamadas a librería
- Hay barrera implícita al final de la región
- Cuando dentro de una región hay otro constructor paralelo → anidamiento
  - cada esclavo crea otro grupo de *threads* esclavos de los que él sería el maestro
- Cláusulas (`private`, `firstprivate`, `default`, `shared`, `copyin` y `reduction`)
  → forma en que se accede a las variables

1. Introducción

1.1. Ejemplo inicial: ejemplo_hello_if.c

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

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

{
  int iam=0, np=1;
  int decision;
  decision=atoi(argv[1]);
  printf("decision=%d \n",decision);

  #pragma omp parallel private(iam, np) if(decision)
  {
    #if defined (_OPENMP)
     np = omp_get_num_threads();
     iam = omp_get_thread_num();
    #endif

    printf("Hello from thread %d out of %d \n",iam,np);

  }
}
```

2. Constructores
2.1. Constructor `parallel`

```
$ export OMP_NUM_THREADS=4

$ ./ejemplo_hello_if 1
decision=1
Hello from thread 1 out of 4
Hello from thread 2 out of 4
Hello from thread 0 out of 4
Hello from thread 3 out of 4

$ ./ejemplo_hello_if 0
decision=0
Hello from thread 0 out of 1
```

2. Constructores
2.1. Constructor `for`

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

- Las iteraciones del bucle `for` se ejecutan en paralelo por *threads* que ya existen
- Restricciones del `for`:
  - La parte de inicialización debe ser una asignación
  - La parte de incremento debe ser una suma o resta
  - La parte de evaluación es la comparación de una variable entera sin signo con un valor, utilizando un comparador mayor o menor (puede incluir igual)
  - Los valores que aparecen en las tres partes del `for` deben ser enteros
- Hay barrera implícita al final (a no ser que se utilice la cláusula `nowait`)
- Hay una serie de cláusulas (`private`, `firstprivate`, `lastprivate` y `reduction`) para indicar la forma en que se accede a las variables

2. Constructores
2.1. Constructor `for`

//ejemplo_for.c

```
#include <omp.h>
int main()
{
        int iam,np,i;
        #pragma omp parallel private(iam,np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif
                printf("\t Hello from thread %d out of %d\n",iam,np);


                #pragma omp for
                for(i=0;i<(np*2);i++)
                {
                  printf("\t\t Thread %d, contador %d \n",iam,i);
                }

        }
}
```

2. Constructores
2.1. Constructor `for`

```
$ ./ejemplo_for
    Hello from thread 0 out of 1
    Thread 0, contador 0
    Thread 0, contador 1

$ export OMP_NUM_THREADS=4

$ ./ejemplo_for
    Hello from thread 0 out of 4
          Thread 0, contador 0
          Thread 0, contador 1

    Hello from thread 3 out of 4
          Thread 3, contador 6
          Thread 3, contador 7

    Hello from thread 2 out of 4
          Thread 2, contador 4
          Thread 2, contador 5

    Hello from thread 1 out of 4
          Thread 1, contador 2
          Thread 1, contador 3
```

2. Constructores
2.1. Constructor `for`

```
//ejemplo_for_2.c
//fijando el numero de threads desde el código

#include <omp.h>
int main()
{
        int iam,np,i;

        omp_set_num_threads(3);

        #pragma omp parallel private(iam, np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif
                printf("Hello from thread %d out of %d\n",iam,np);


                #pragma omp for
                for(i=0;i<(np*2);i++)
                {
                  printf("Thread %d, contador %d \n",iam,i);
                }

}
```

2. Constructores
2.1. Constructor `for`

```
$ export OMP_NUM_THREADS=4

$ ./ejemplo_for_2

Hello from thread 1 out of 3
    Thread 1, contador 2
    Thread 1, contador 3

Hello from thread 0 out of 3
    Thread 0, contador 0
    Thread 0, contador 1

Hello from thread 2 out of 3
    Thread 2, contador 4
    Thread 2, contador 5
```

2. Constructores
2.1. Constructor `for`

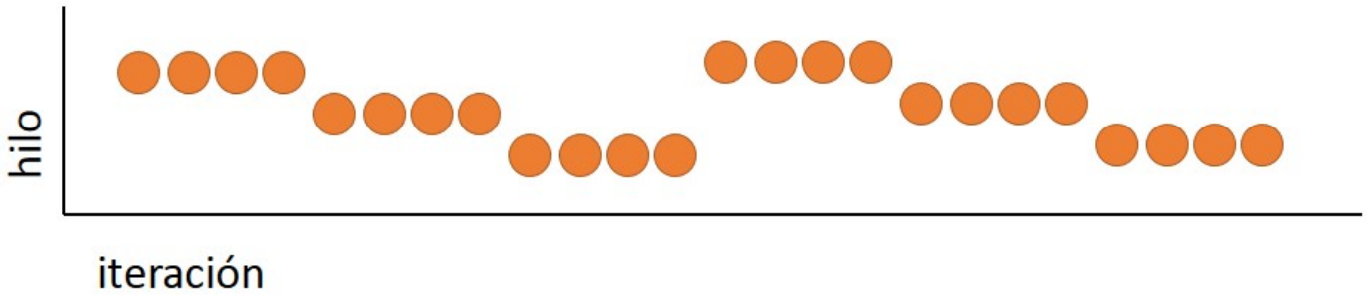
Cláusula `schedule:` cómo se reparten las iteraciones del `for` entre los threads:

- `schedule(static,tam)`
  - las iteraciones se dividen en grupos de tamaño `tam`, y la asignación de estos grupos se hace estáticamente a los threads. Si no se indica el tamaño se divide por igual entre los threads
- `schedule(dynamic,tam)`
  - las iteraciones se dividen en grupos de tamaño `tam`, y la asignación de estos grupos a los threads se hace dinámicamente cuando cada thread va acabando su trabajo
- `schedule(guided,tam)`
  - las iteraciones se asignan dinámicamente a los threads pero con tamaños decrecientes, empezando en tamaño `numiter/np` y acabando en `tam`
- `schedule(runtime)`
  - deja la decisión para el tiempo de ejecución
  - La decisión se obtienen de la variable de entorno `OMP_SCHEDULE`
    (esta variable se puede cambiar a mano o llamando a `omp_set_schedule()`)

2. Constructores
2.1. Constructor `for`

- Ejemplo:
  - for loop with 24 iterations and 3 OpenMP threads.

    schedule(static, 4):



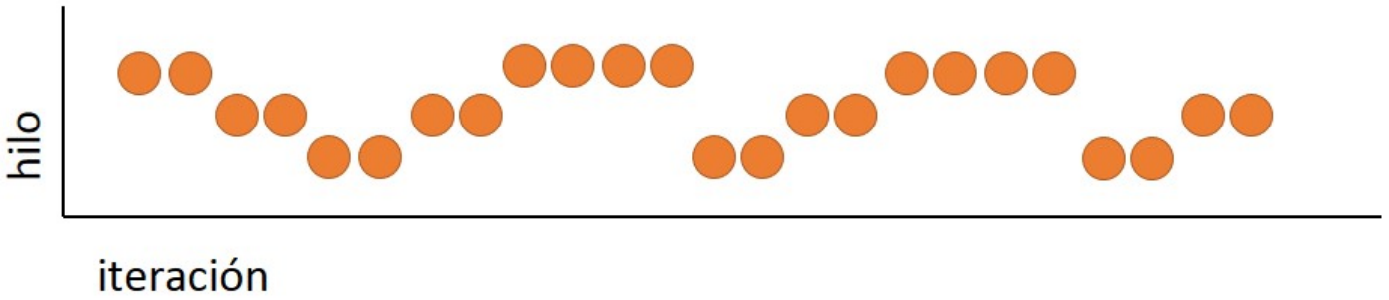
Las iteraciones se dividen entre los hilos antes de empezar la ejecución del bucle

El tamaño del bloque de reparto de iteraciones es 4

2. Constructores
2.1. Constructor `for`

- Ejemplo:
  - for loop with 24 iterations and 3 OpenMP threads.

    schedule(dynamic,2):



Cada hilo pide trabajo cuando se queda desocupado

El tamaño del bloque de reparto de iteraciones 2

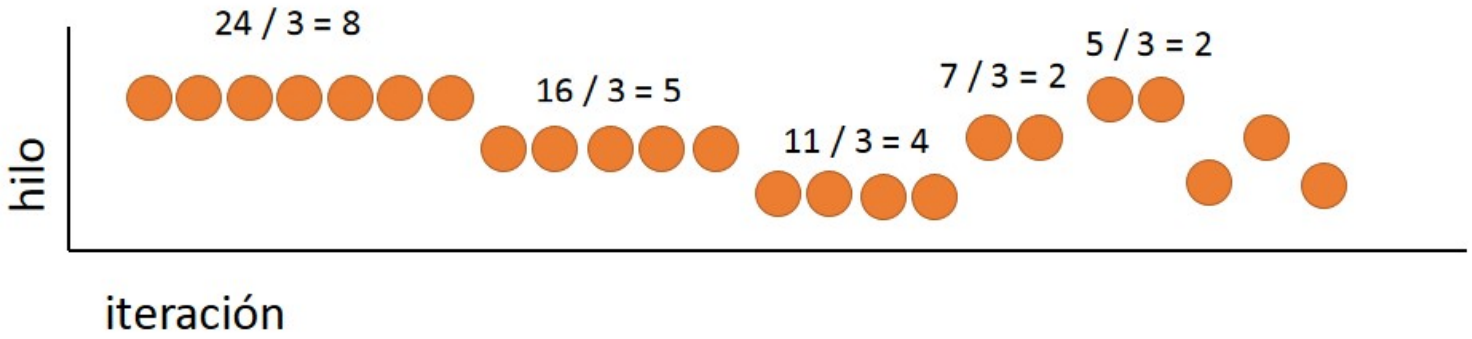
Asignación dinámica → Más sobrecarga de tiempo

Apropiado cuando cada iteración pueda tener diferente coste computacional

2. Constructores
2.1. Constructor `for`

- Ejemplo:
  - for loop with 24 iterations and 3 OpenMP threads.

    schedule(guided,1):



El bloque de reparto de iteraciones:

número de iteraciones aun no repartidas dividido por el número de hilos:

$$guided\_bloque = \lceil numiter_{paso\_i} \ / \ numhilos \rceil$$

2. Constructores
2.1. Constructor `for`

```
//ejemplo_for_static.c
#include <stdio.h>
#include <omp.h>

int main()
{
        int iam,np,i;
        #pragma omp parallel private(iam, np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif

                #pragma omp for schedule(static,2)
                for(i=0;i<(np*4);i++)
                {
                  printf("Thread %d, contador %d \n",iam,i);
                }
         }
}
```

2. Constructores
2.1. Constructor `for`

```
$ ./ejemplo_for_static
Thread 0, contador 0
Thread 0, contador 1

Thread 0, contador 8
Thread 0, contador 9

Thread 3, contador 6
Thread 3, contador 7

Thread 3, contador 14
Thread 3, contador 15

Thread 2, contador 4
Thread 2, contador 5

Thread 2, contador 12
Thread 2, contador 13

Thread 1, contador 2
Thread 1, contador 3

Thread 1, contador 10
Thread 1, contador 11
```

2. Constructores
2.1. Constructor `for`

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

int main()
{
        int iam,np,i=2;
        #pragma omp parallel private(iam, np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif

                #pragma omp for schedule(dynamic,1)
                for(i=0;i<(np*4);i++)
                {
                  printf("Thread %d, contador %d \n",iam,i);
                  sleep(iam);
                }
        }
}
```

2. Constructores
2.1. Constructor `for`

```
$ ./ejemplo_for_dynamic
Thread 0, contador 0
Thread 0, contador 1
Thread 0, contador 8
Thread 0, contador 9
Thread 3, contador 6
Thread 3, contador 7
Thread 3, contador 14
Thread 3, contador 15
Thread 2, contador 4
Thread 1, contador 2
Thread 1, contador 3
Thread 1, contador 10
Thread 1, contador 11
Thread 2, contador 5
Thread 2, contador 12
Thread 2, contador 13
```

2. Constructores
2.1. Constructor `for`

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

int main()
{
        int iam,np,i;
        #pragma omp parallel private(iam, np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif

                #pragma omp for schedule(guided,2)
                for(i=0;i<(np*4);i++)
                {
                  printf("Thread %d, contador %d \n",iam,i);
                  sleep(iam);
                }
        }
}
```

2. Constructores

2.1. Constructor `for`

```
$ ./ejemplo_for_guided
Thread 0, contador 0
Thread 0, contador 1
Thread 0, contador 2
Thread 0, contador 3

Thread 2, contador 4
Thread 2, contador 5
Thread 2, contador 6

Thread 1, contador 7
Thread 1, contador 8
Thread 1, contador 9

Thread 3, contador 10
Thread 3, contador 11

Thread 0, contador 12
Thread 0, contador 13

Thread 0, contador 14
Thread 0, contador 15
```

2. Constructores
2.1. Constructor `for`

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

int main()
{
        int iam,np,i;
        #pragma omp parallel private(iam, np,i)
        {
                #if defined (_OPENMP)
                  np = omp_get_num_threads();
                  iam = omp_get_thread_num();
                #endif

                #pragma omp for schedule(runtime)
                for(i=0;i<(np*4);i++)
                {
                  printf("Thread %d, contador %d \n",iam,i);
                  sleep(iam);
                }
        }
}
```

2. Constructores
2.1. Constructor `for`

```
$ export OMP_SCHEDULE="static,1“
$ ./ejemplo_for_runtime
Thread 2, contador 2
Thread 3, contador 3
Thread 1, contador 1
Thread 4, contador 4
Thread 5, contador 5
Thread 0, contador 0
Thread 0, contador 6
Thread 0, contador 12
Thread 0, contador 18
Thread 1, contador 7
Thread 2, contador 8
Thread 1, contador 13
Thread 3, contador 9
Thread 1, contador 19
Thread 2, contador 14
Thread 4, contador 10
Thread 5, contador 11
Thread 3, contador 15
Thread 2, contador 20
Thread 4, contador 16
Thread 3, contador 21
Thread 5, contador 17
Thread 4, contador 22
Thread 5, contador 23
```

2. Constructores
2.1. Constructor `for`

```
$ export OMP_SCHEDULE="static,4"
$ ./ejemplo_for_runtime
Thread 4, contador 16
Thread 0, contador 0
Thread 0, contador 1
Thread 0, contador 2
Thread 0, contador 3
Thread 5, contador 20
Thread 3, contador 12
Thread 2, contador 8
Thread 1, contador 4
Thread 1, contador 5
Thread 2, contador 9
Thread 1, contador 6
Thread 3, contador 13
Thread 1, contador 7
Thread 4, contador 17
Thread 2, contador 10
Thread 5, contador 21
Thread 2, contador 11
Thread 3, contador 14
Thread 4, contador 18
Thread 3, contador 15
Thread 5, contador 22
Thread 4, contador 19
Thread 5, contador 23
```

2. Constructores
2.2. Constructor sections

```
#pragma omp sections [cláusulas]
   {
    [#pragma omp section]
        bloque

    [#pragma omp section
        bloque
    ...
   }
```

- Cada sección se ejecuta por un único *thread*
- Hay barrera ímplicita al final de `sections` (a no ser que se utilice la cláusula `nowait`)
- Hay una serie de cláusulas (`private`, `firstprivate`, `lastprivate` y `reduction`) para indicar la forma en que se accede a las variables

2. Constructores
2.2. Constructor sections

```
// ejemplo_sections.c
#pragma omp parallel private(iam, np,i)
{
 #pragma omp sections
 {
   #pragma omp section
    printf("Soy el thread %d, en solitario en la seccion 1ª \n",iam);

   #pragma omp section
    printf("Soy el thread %d, en solitario en la sección 2ª \n",iam);

   #pragma omp section
    printf("Soy el thread %d, en solitario en la seccion 3ª \n",iam);

 }//sections
}//parallel
```

2. Constructores
2.2. Constructor sections

```
$ export OMP_NUM_THREADS=2

$ ./ejemplo_sections

Soy el thread 0, actuando en solitario dentro de la seccion 1ª

Soy el thread 1, actuando en solitario dentro de la seccion 2ª

Soy el thread 0, actuando en solitario dentro de la seccion 3ª
```

2. Constructores
2.3. Constructores combinados

```
#pragma omp parallel for [cláusulas]
   //bucle for
```

- Es forma abreviada de directiva `parallel` que tiene una única directiva `for`
- admite sus cláusulas (menos la `nowait`)

```
#pragma omp parallel sections [cláusulas]
   // conjunto de secciones
```

- Es forma abreviada de directiva `parallel` que tiene una única directiva `sections`
- admite sus cláusulas (menos la `nowait`)

2. Constructores
2.4. Constructores de ejecución secuencial

```
#pragma omp single [cláusulas]
// bloque
```

- El bloque se ejecuta por un único *thread*. No tiene por qué ser el maestro.
- Hay barrera implícita al final (a no ser que se utilice la cláusula `nowait`)

```
#pragma omp master
// bloque
```

- El bloque lo ejecuta el *thread* maestro.
- No hay sincronización ni al entrar ni salir.

```
#pragma omp ordered
// bloque
```

- Se usa dentro de un `for`
- El bloque se ejecuta en el orden en que se ejecutaría en secuencial

2. Constructores
2.4. Constructores de ejecución secuencial

- `ejemplo_single.c`
  - Barreras al final de cada single
- `ejemplo_master.c`
  - Ejecución solamente por thread maestro (el 0)
  - No hay barreras
- `ejemplo_ordered.c`
  - Se ordena la ejecución por iteraciones del bucle

2. Constructores
2.4. Constructores de ejecución secuencial

```
// ejemplo_single.c
#pragma omp parallel private(iam, np,i)
{
#pragma omp single
{
    printf("Soy el thread %d, actuando en solitario dentro del primer bloque\n",iam);
    sleep(1);
}
#pragma omp single
{
    printf("Soy el thread %d, actuando en solitario dentro del segundo bloque \n",iam);
    sleep(1);
}
#pragma omp single
{
    printf("Soy el thread %d, actuando en solitario dentro del tercer bloque \n",iam);
    sleep(1);
}

printf("Soy el thread %d, fuera de los singles\n",iam);
}//parallel
```

2. Constructores
2.4. Constructores de ejecución secuencial

```
$ export OMP_NUM_THREADS=4

$ ./ejemplo_single

Soy el thread 0, actuando en solitario dentro del primer bloque
Soy el thread 0, actuando en solitario dentro del segundo bloque
Soy el thread 0, actuando en solitario dentro del tercer bloque

Soy el thread 0, fuera de los singles
Soy el thread 3, fuera de los singles
Soy el thread 2, fuera de los singles
Soy el thread 1, fuera de los singles
```

2. Constructores
2.4. Constructores de ejecución secuencial

```
//ejemplo_master.c
#pragma omp parallel private(iam, np,i)
{

#pragma omp master                    {
   printf("Soy el thread %d, actuando en solitario dentro del primer
   bloque\n",iam);                      sleep(1);                    }

#pragma omp master                    {
   printf("Soy el thread %d, actuando en solitario dentro del segundo bloque
   \n",iam);                         sleep(1);                     }

#pragma omp master                    {
   printf("Soy el thread %d, actuando en solitario dentro del tercer bloque
   \n",iam);                         sleep(1);                     }


printf("Soy el thread %d, fuera de los bloques de master\n",iam);
}//parallel
```

2. Constructores
2.4. Constructores de ejecución secuencial

```
$ ./ejemplo_master

Soy el thread 0, actuando en solitario dentro del primer bloque

   Soy el thread 3, fuera de los bloques de master
   Soy el thread 2, fuera de los bloques de master
   Soy el thread 1, fuera de los bloques de master

Soy el thread 0, actuando en solitario dentro del segundo bloque
Soy el thread 0, actuando en solitario dentro del tercer bloque

   Soy el thread 0, fuera de los bloques de master
```

2. Constructores
2.4. Constructores de ejecución secuencial

```
//ejemplo_ordered.c

#pragma omp parallel private(iam, np,i)
{
#pragma omp for ordered
for(i=0;i<5;i++)
{
    printf("\t\tSoy el thread %d, antes del ordered. Iteracion %d\n",iam,i);

    #pragma omp ordered
    {
            printf("Soy el thread %d, actuando en la iteracion %d\n",iam,i);
            sleep(1);
    }//bloque ordered

}//for
}//parallel
```

2. Constructores
2.4. Constructores de ejecución secuencial

```
./ejemplo_ordered
    Soy el thread 0, antes del ordered en la iteracion 0
Soy el thread 0, actuando en la iteracion 0

    Soy el thread 3, antes del ordered en la iteracion 4
    Soy el thread 2, antes del ordered en la iteracion 3
    Soy el thread 1, antes del ordered en la iteracion 2
    Soy el thread 0, antes del ordered en la iteracion 1

Soy el thread 0, actuando en la iteracion 1
Soy el thread 1, actuando en la iteracion 2
Soy el thread 2, actuando en la iteracion 3
Soy el thread 3, actuando en la iteracion 4
```

2. Constructores
2.5. Constructores de sincronización

```
#pragma omp critical [nombre]
 //  bloque
```

- Asegura exclusión mutua en la ejecución del bloque.
- Todos los hilos, uno tras otro, irán ejecutando el bloque
- El nombre se puede usar para identificar secciones críticas distintas

```
#pragma omp barrier
```

- Sincroniza todos los threads en el equipo

```
#pragma omp atomic
 //Expresión
```

- Ejecuta operaciones atómicamente
  - La expresión debe ser: `x binop exp, x++, ++x, x--, --x`
    - `x` es una expresión con valor escalar
    - `binop` es un operador binario.
  - Asegura la carga y el almacenamiento de la variable `x` de forma atómica
    → la ubicación de memoria se protege contra más de una escritura

2. Constructores
2.5. Constructores de sincronización

```
//ejemplo_critical.c
#pragma omp parallel private(iam, np,i)
{

#pragma omp critical
{
     printf("Soy el thread %d, al inicio de la seccion critica
     \n",iam);

     sleep(1);

     printf("\t\tSoy el thread %d, al final de la seccion critica
     \n",iam);
}

}//parallel
```

2. Constructores
2.5. Constructores de sincronización

```
$ ./ejemplo_critical
Soy el thread 0, al inicio de la seccion critica
     Soy el thread 0, al final de la seccion critica

Soy el thread 3, al inicio de la seccion critica
     Soy el thread 3, al final de la seccion critica

Soy el thread 2, al inicio de la seccion critica
     Soy el thread 2, al final de la seccion critica

Soy el thread 1, al inicio de la seccion critica
     Soy el thread 1, al final de la seccion critica
```

2. Constructores
2.5. Constructores de sincronización

```
//ejemplo barrier.c
#pragma omp parallel private(iam, np,i)
{

     printf("Soy el thread %d, antes del barrier \n",iam);

     #pragma omp barrier

     printf("\t\tSoy el thread %d, despues del barrier \n",iam);

}//parallel
```

2. Constructores
2.5. Constructores de sincronización

```
$ export OMP_NUM_THREADS=6

$ ./ejemplo_barrier
Soy el thread 0, antes del barrier
Soy el thread 5, antes del barrier
Soy el thread 4, antes del barrier
Soy el thread 3, antes del barrier
Soy el thread 2, antes del barrier
Soy el thread 1, antes del barrier
      Soy el thread 0, despues del barrier
      Soy el thread 5, despues del barrier
      Soy el thread 4, despues del barrier
      Soy el thread 3, despues del barrier
      Soy el thread 2, despues del barrier
      Soy el thread 1, despues del barrier
```

2. Constructores
2.5. Constructores de sincronización

//ejemplo_atomic.c

```
int main()
{
        int iam,np,i,j;
        int count=0;

        #pragma omp parallel private(iam, np,i)
        {
                #pragma omp atomic
                    count++;
       }// parallel

        printf("Number of threads: %d\n", count);
}//main
```

2. Constructores
2.5. Constructores de sincronización

```
$ export OMP_NUM_THREADS=6

$ ./ejemplo_atomic
Number of threads: 6



$ export OMP_NUM_THREADS=4

$ ./ejemplo_atomic
Number of threads: 4
```

2. Constructores
2.5. Constructores de sincronización

Otro ejemplo: `critical_pi.c`

```
...
 #pragma omp parallel for shared(sum) private(i,x,acum) firstprivate(h) \\
    num_threads(nt)

        for(i=1; i <= n; i++)
        {
                x = h*((double) i - 0.5);
                acum = (4.0/(1.0 + x*x));

                #pragma omp critical
                {
                        sum += acum;
                }
        }
...
```

2. Constructores
2.5. Constructores de sincronización

Otro ejemplo: `atomic_pi.c`

```
...
#pragma omp parallel for shared(sum) private(i,x,acum) firstprivate(h) \\
    num_threads(nt)

        for(i=1; i <= n; i++)
         {
                x = h*((double) i - 0.5);
                acum = (4.0/(1.0 + x*x));

                #pragma omp atomic
                sum += acum;
        }
...
```

2. Constructores
2.5. Constructores de sincronización

Comparando prestaciones: `atomic_pi.c` versus `critical_pi.c`

```
$ critical_pi 4

Time = 23.6508 seconds
El valor aproximado de PI es: 3.1416046196492546, con un error de
    0.0000119660594615
```

```
$ atomic_pi 4

Time = 13.2825 seconds
El valor aproximado de PI es: 3.1415926535902865, con un error de
    0.0000000000004934
```

2. Constructores
2.6. Constructores de manejo de variables

```
#pragma omp flush [lista]
```

- Asegura que las variables que aparecen en la lista quedan actualizadas para todos los *threads*.
- Si no hay lista se actualizan todos los objetos compartidos.
- Se hace *flush* implícito:
  - al acabar `barrier`
  - al entrar o salir de `critical` u `ordered`
  - al salir de `parallel`, `for`, `sections` o `single`

- →`ejemplo_flush_2.c`

```
#pragma omp threadprivate (lista)
```

- Usado para declarar variables privadas a los threads.

2. Constructores

2.6. Constructores de manejo de variables

```
// ejemplo_flush_2.c
int main()
{
      int flag=0;
      #pragma omp parallel sections num_threads(2)  private(iam)
      {
           #pragma omp section
           {
            iam=omp_get_thread_num();
            printf("Thread %d, esperando dato: \n",iam);
            scanf("%d",&data);
            #pragma omp flush(data)
            flag = 1;
            #pragma omp flush(flag)
           }section

           #pragma omp section
           {
                iam=omp_get_thread_num();
                printf("ANTES: Thread %d, dato a procesar = %d \n",iam,data);
                while (!flag)
                     #pragma omp flush(flag)          // espera activa
                #pragma omp flush(data)               // obtención segura de datos
                printf("DESPUES: Thread %d: dato a procesar = %d \n", iam,data);
            }//section
       } //parallel sections
}
```

2. Constructores
2.6. Constructores de manejo de variables

```
$ ./ejemplo_flush_2

Thread 0, esperando dato:
12345    #tecleado en terminal por usuario


ANTES: Thread 1, dato a procesar = 0

DESPUES: Thread 1: dato a procesar = 12345
```

Contenido

3. Claúsulas de alcance de datos

- `private(variables)`:
  - Variables privadas a cada thread
  - No se copia valor del thread creador de la región paralela
  - No se guarda su valor al salir

- `firstprivate(variables)`
  - Variables privadas a cada thread
  - Sí se inicializan con el valor que tuviera en el thread creador de la región paralela
  - No se guarda su valor al salir

- `lastprivate(variables)`
  - Variables privadas a cada thread
  - No se copia valor del thread creador de la región paralela
  - Al salir, se quedan con el valor de la última iteración o sección

3. Claúsulas de alcance de datos

- `shared(variables)`
  - Variables compartidas por todos los threads.
- `default(shared|none)`
  - Indica cómo serán las variables por defecto.
- `reduction(operador:variables)`
  - Se obtienen valores de las variables aplicando el operador de reducción
- `copyin(variables)`
  - para asignar, al empezar la región paralela, el valor de las variables en el master a variables locales privadas a los threads

3. Claúsulas de alcance de datos

```
// ejemplo_private.c
int x=9999;

printf("\n Antes de pragma parallel x=%d \n\n",x);

#pragma omp parallel private(iam, np,i,x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d \n",iam,x);
    x=1000+iam;
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d \n",iam,x);
}

printf("\n Despues de pragma parallel x=%d \n\n",x);
```

3. Claúsulas de alcance de datos

```
$ ./ejemplo_private
Antes de pragma parallel x=9999

   Soy el thread 0, antes de actualizar, con x=0
        Soy el thread 0, despues de actualizar, con x=1000

   Soy el thread 3, antes de actualizar, con x=0
        Soy el thread 3, despues de actualizar, con x=1003

   Soy el thread 2, antes de actualizar, con x=0
        Soy el thread 2, despues de actualizar, con x=1002

   Soy el thread 1, antes de actualizar, con x=0
        Soy el thread 1, despues de actualizar, con x=1001


 Despues de pragma parallel x=9999
```

3. Claúsulas de alcance de datos

```
// ejemplo_firstprivate.c
int x=9999;

printf("\n Antes de pragma parallel x=%d \n\n",x);

#pragma omp parallel firstprivate(iam, np,i,x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d \n",iam,x);
    x=1000+iam;
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d \n",iam,x);
}

printf("\n Despues de pragma parallel x=%d \n\n",x);
```

3. Claúsulas de alcance de datos

```
$ ./ejemplo_firstprivate

Antes de pragma parallel x=9999

    Soy el thread 0, antes de actualizar, con x=9999
         Soy el thread 0, despues de actualizar, con x=1000

    Soy el thread 3, antes de actualizar, con x=9999
         Soy el thread 3, despues de actualizar, con x=1003

    Soy el thread 2, antes de actualizar, con x=9999
         Soy el thread 2, despues de actualizar, con x=1002

    Soy el thread 1, antes de actualizar, con x=9999
         Soy el thread 1, despues de actualizar, con x=1001


 Despues de pragma parallel x=9999
```

3. Claúsulas de alcance de datos

```
// ejemplo_lastprivate.c
 #pragma omp parallel private(iam, np,i)
 {
   printf("Soy el thread %d, antes de entrar en bucle, con x=%d \n",iam,x

   #pragma omp for lastprivate(x)
   for(i=0;i<11;i++)
   {
    printf("\t\t\t Soy el thread %d, antes de actualizar en for, i=%d
    x=iam*i;
    printf("\t\t\t\t Soy el thread %d, actualizando en for, i=%d x=%d\
   }
   printf("Soy el thread %d, despues de salir de bucle, con x=%d \n",iam,
 }//parallel

  printf("\n Despues de pragma parallel x=%d \n\n",x);
```

3. Claúsulas de alcance de datos

```
$ ./ejemplo_lastprivate
 Antes de pragma parallel x=9999

         Soy el thread 4, antes de entrar en bucle, con x=9999
            . . .
         Soy el thread 2, antes de entrar en bucle, con x=9999
                              Soy el thread 2, antes de actualizar en for, i=1 x=630901840
                                    Soy el thread 2, actualizando en for, i=1 x=2
                        . . .
                       Soy el thread 3, antes de actualizar en for, i=10 x=27
                                    Soy el thread 3, actualizando en for, i=10 x=30

         Soy el thread 0, despues de salir de bucle, con x=30
         Soy el thread 3, despues de salir de bucle, con x=30
            . . .
        Soy el thread 1, despues de salir de bucle, con x=30

 Despues de pragma parallel x=30
```

3. Claúsulas de alcance de datos

```
// ejemplo_reduction.c
int x=1000;  int iam =0, np = 1, i=0,j=0;
printf("\n Antes de pragma parallel x=%d \n\n",x);

#pragma omp parallel private(iam,np,i) reduction(+:x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d \n",iam,x);
    x=iam*10;
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d \n",iam,x);
} //parallel

printf("\n Despues de pragma parallel x=%d \n\n",x);
```

3. Claúsulas de alcance de datos

```
$ ./ejemplo_reduction

 Antes de pragma parallel x=1000

Soy el thread 0, antes de actualizar, con x=0
     Soy el thread 0, despues de actualizar, con x=0

Soy el thread 3, antes de actualizar, con x=0
     Soy el thread 3, despues de actualizar, con x=30

Soy el thread 2, antes de actualizar, con x=0
     Soy el thread 2, despues de actualizar, con x=20

Soy el thread 1, antes de actualizar, con x=0
     Soy el thread 1, despues de actualizar, con x=10


 Despues de pragma parallel x=1060
```

3. Claúsulas de alcance de datos

```
// ejemplo_copyin.c
int x;
#pragma omp threadprivate(x) //x privada a cada thread, no se copia valor del master

int main()
{
    x=9000; // lo ponemos en el master
    #pragma omp parallel private(iam,np,i) copyin(x)
          //sin copyin en cada tread tendriamos x=0, salvo el master
    {
          printf("Soy el thread %d, antes de actualizar, con x=%d \n",iam,x);
          x=x+iam;
          printf("\t\tSoy el thread %d, despues de actualizar, con x=%d \n",iam,x);
    }//parallel

    printf("\n Despues de pragma parallel x=%d \n\n",x); //x=1000
}//main
```

3. Claúsulas de alcance de datos

```
$ ./ejemplo_copyin

Soy el thread 2, antes de actualizar, con x=9000
     Soy el thread 2, despues de actualizar, con x=9002

Soy el thread 1, antes de actualizar, con x=9000
     Soy el thread 1, despues de actualizar, con x=9001

Soy el thread 0, antes de actualizar, con x=9000
     Soy el thread 0, despues de actualizar, con x=9000

Soy el thread 3, antes de actualizar, con x=9000
     Soy el thread 3, despues de actualizar, con x=9003


 Despues de pragma parallel x=9000
```

Contenido

4. Funciones de librería

Es necesario poner al principio: `#include <omp.h>`

- `void omp_set_num_threads(int num_threads);`
  - Para establecer el número de threads a usar en la siguiente región paralela
- `int omp_get_num_threads(void);`
  - Para consultar el número de threads que se están usando en una región paralela
- `int omp_get_thread_num(void);`
  - Para consultar el número del thread que soy
- `int omp_get_num_procs(void);`
  - Para consultar el número de procesadores (núcleos) del nodo

4. Funciones de librería

- `int omp_in_parallel(void);`
  - Devuelve valor distinto de cero si se ejecuta dentro de una región paralela

- `int omp_set_dynamic(int);`
  - Para permitir o desautorizar que el número de threads se pueda ajustar dinámicamente en las regiones paralelas

- `int omp_get_dynamic(void);`
  - Devuelve un valor distinto de cero si está permitido el ajuste dinámico del número de threads

- `int omp_set_nested(int);`
  - Para permitir o desautorizar el paralelismo anidado

→ **`ejemplo_nested.c`**

- `int omp_get_nested(void);`
  - Devuelve un valor distinto de cero si está permitido el paralelismo anidado

4. Funciones de librería

```
// ejemplo_nested.c
#pragma omp parallel private(iam, np,i)
{
     np = omp_get_num_threads();
     iam = omp_get_thread_num();
     printf("Hello from thread %d out of %d \n",iam,np);

     omp_set_dynamic(0);      // 0 --> habilita cambio dinamico de n°de hilos
     omp_set_nested(1);       // 1 --> habilita paralelismo anidado
     omp_set_num_threads(3);

     #pragma omp parallel private(iam, np,i)
     {
              np = omp_get_num_threads();
              iam = omp_get_thread_num();
              printf("\t\t Now: hello from thread %d out of %d \n",iam,np);
     }//parallel

}//parallel
```

4. Funciones de librería

```
$ ./ejemplo_nested

Hello from thread 0 out of 2

        Now: hello from thread 0 out of 3
        Now: hello from thread 2 out of 3
        Now: hello from thread 1 out of 3

Hello from thread 1 out of 2

        Now: hello from thread 2 out of 3
        Now: hello from thread 0 out of 3
        Now: hello from thread 1 out of 3
```

4. Funciones de librería

- `void omp_init_lock(omp_lock_t *lock);`
  - Para inicializar un candado. Un candado se inicializa como no bloqueado
- `void omp_destroy_lock(omp_lock_t *lock);`
  - Para destruir un candado
- `void omp_set_lock(omp_lock_t *lock);`
  - Para pedir un candado
- `void omp_unset_lock(omp_lock_t *lock);`
  - Para soltar un candado
- `int omp_test_lock(omp_lock_t *lock);`
  - Intenta pedir un candado pero no se queda bloqueado esperando

→ **`ejemplo_lock.c`**

4. Funciones de librería

```
// ejemplo_lock.c
omp_init_lock(&lck);

#pragma omp parallel shared(lck) private(id)
{
    id=omp_get_thread_num();
    omp_set_lock(&lck);
    printf("My thread id is %d.\n",id);
    omp_unset_lock(&lck);


    while (! omp_test_lock(&lck))     // espera activa
      skip(id);

    work(id);        //ahora tengo el candado abierto, entonces hago el trabajo
    omp_unset_lock(&lck);


}//parallel
omp_destroy_lock(&lck);
```

4. Funciones de librería

```
double omp_get_wtime(void);
```

Devuelve el tiempo de ejecución (en segundos)
El valor devuelto es privado del hilo que invoca la función

```
double omp_get_wtick(void);
```

Devuelve la precisión del reloj (segundos entre dos ticks consecutivos)

ejemplo_mm.c:

```
start=omp_get_wtime();
mm_i(a,b,c,t);
fin=omp_get_wtime();
tiempo=fin-start;
Mflops=((2.*t*t*t)/tiempo)/1000000.;

printf("segundos: %.6lf, Mflops: %.6lf, Mflops por thread: %.6lf\n",THREADS,t,tiempo,Mflops,Mflops/THREADS);

printf("Precision omp_get_wtick: numero de segundos entre sucesivos ticks de reloj usados por wtime=%lf \n",omp_get_wtick());
```

Contenido

5. Variables de entorno

`OMP_SCHEDULE`

- Establece el tipo de scheduling para `for` y `parallel for`

`OMP_NUM_THREADS`

- Establece el número de threads a usar

`OMP_DYNAMIC`

- Autoriza o desautoriza el ajuste dinámico del número de threads

`OMP_NESTED`

- Autoriza o desautoriza el anidamiento

Contenido

6. Tareas en OpenMP

- Cambiar OpenMP de "*thread-centric*" a "*task-centric*"
- Para expresar paralelismo irregular y no estructurado
- Para hacer paralelismo anidado de tareas
  - sin que conlleve un estricto paralelismo anidado de *threads* con sus consecuentes barriers implicitos que puedan ser innecesarios
- Para paralelizar bucles tipo `while`
  - Por ejemplo recorrido de una lista de punteros de longitud no conocida
- **Dos tipos de tareas:**
  - ***Tied task***: ligada a un *thread* fijo que la ejecutará.
    - Puede tener pausas o hacer otra cosa, pero ese *thread* acabará la tarea
  - ***Untied task***: a cualquier *thread* del *team* el *scheduler* le puede asignar una *untied task* que esté suspendida en ese momento

6. Tareas en OpenMP
6.1. Constructor task

```
#pragma omp task [clause [[,]clause] ...]
   structured-block
```

Clauses:

```
   if (scalar expression)
   untied
   default (shared | none)
   private (list)
   firstprivate (list)
   shared (list)
```

6. Tareas en OpenMP
6.2. Sincronización y Finalización de tareas

```
#pragma omp taskwait
```

- Tarea actual se suspende hasta finalización de sus tareas hijas
- Util para sincronización de tareas de grano fino

En general, un conjunto de tareas será forzada a completarse en alguno de estos casos:

- En una barrier implícita de *threads*
- En una barrier explícita de *threads* (`#pragma omp barrier`)
- En una barrier de tareas (`#pragma omp taskwait`)

6. Tareas en OpenMP
6.1. Ejemplos

```
//ejemplo de recorrido de una lista. Secuencial:

....

while(my_pointer)
{
    (void) do_independent_work (my_pointer);

    my_pointer = my_pointer->next ;

}
...
```

6. Tareas en OpenMP

6.1. Ejemplos

```
//ejemplo de recorrido de una lista. Paralelo:

my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer)
        {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }

    } // End of single - no implied barrier because nowait is used

    // no-single threads doing other work

} // End of parallel region (implied barrier) → threads waiting to pick up generated tasks
```

6. Tareas en OpenMP
6.1. Ejemplos

Ejemplo: Serie Fibonacci

F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2) (para n= 2, 3, …)

Secuencia valores resultado:

0,1,1, 2, 3, 5, 8, 13, 21, 34, 55,…

6. Tareas en OpenMP
6.1. Ejemplos

// Ejemplo Fibonacci. Secuencial recursivo:

```
long comp_fib_numbers(int n)
{
   // Basic algorithm: f(n) = f(n-1) + f(n-2)
   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 )
        return(n);

   fnm1 = comp_fib_numbers(n-1);
   fnm2 = comp_fib_numbers(n-2);
   fn = fnm1 + fnm2;

   return(fn);
}
```

6. Tareas en OpenMP

6.1. Ejemplos

```
// Ejemplo Fibonacci. Paralelo

long comp_fib_numbers(int n)
{
   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 )
        return(1);

   #pragma omp task shared(fnm1)
        {fnm1 = comp_fib_numbers(n-1);}

   #pragma omp task shared(fnm2)
        {fnm2 = comp_fib_numbers(n-2);}

   #pragma omp taskwait
   fn = fnm1 + fnm2;

   return(fn);
}
```

```
#pragma omp parallel shared(nthreads)
{
        #pragma omp single nowait
        {
                result = comp_fib_numbers(n);
        } // End of single
} // End of parallel region
```

6. Tareas en OpenMP

6.1. Ejemplos

```
$ ./ejemplo_fibo 4

Soy el thread 0

 Soy el thread 0, resolviendo fibo(4)

 Soy el thread 0, resolviendo fibo(2)

 Soy el thread 0, resolviendo fibo(0)

 Soy el thread 0, resolviendo fibo(1)

 Soy el thread 0, resolviendo fibo(3)

 Soy el thread 0, resolviendo fibo(1)

 Soy el thread 0, resolviendo fibo(2)

 Soy el thread 0, resolviendo fibo(0)

 Soy el thread 0, resolviendo fibo(1)

Soy el thread 1

Soy el thread 2

Soy el thread 3

Serie de Fibonacci para n=4. El resultado es 3
```

6. Tareas en OpenMP
6.1. Ejemplos

```
// ejemplo recorrido arbol en preorden
void traverse(bynarytree *p)
{
   process(p);

   if (p->left)
   {
        #pragma omp task
            traverse(p->left);
   }
   if (p->right)
   {
        #pragma omp task
            traverse(p->right);
   }
}
```

6. Tareas en OpenMP
6.1. Ejemplos

```
// ejemplo recorrido arbol en postorden
void traverse(bynarytree *p)
{
   if (p->left)
   {
        #pragma omp task
            traverse(p->left);
   }
   if (p->right)
   {
        #pragma omp task
            traverse(p->right);
   }

   #pragma omp taskwait

   process(p);

}
```

Referencias

The OpenMP API specification for parallel programming
https://www.openmp.org/specifications/

Contenido

Apéndice: Almacenamiento de matrices

En un array bidimensional: `double a[m][n]`

(fila i, columna j): `a[i][j]`

En un array unidimensional: `double a[m*n]`

(fila i, columna j): `a[i*n+j]`

Cuando la matriz es parte de otra:

***Leading dimension*:** posiciones de memoria entre 2 elementos consecutivos de la misma columna/fila

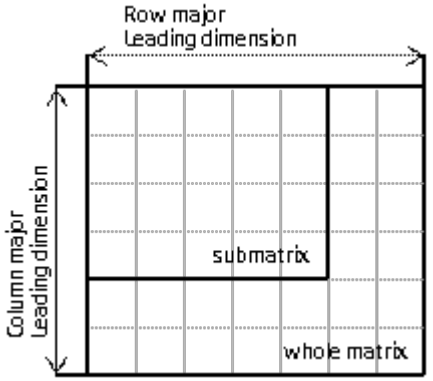
Dos opciones:

Si el almacenamiento es por filas:

(fila i, columna j): `a[i*ld+j]`

Si el almacenamiento es por columnas:

(fila i, columna j): `a[j*ld+i]`



Apéndice: Multiplicación matriz-vector

```
void matriz_vector(double *m,int fm,int cm,int ldm,double*v,double *w)
{
        int i,j,iam,nprocs;
        double s;

        for (i=0 ; i<fm ; i++)
        {
                s=0.;

                for(j = 0 ; j < cm ; j++)
                {
                        s += m[i*ldm+j] * v[j];
                }
                w[i]=s;
        }

}
```

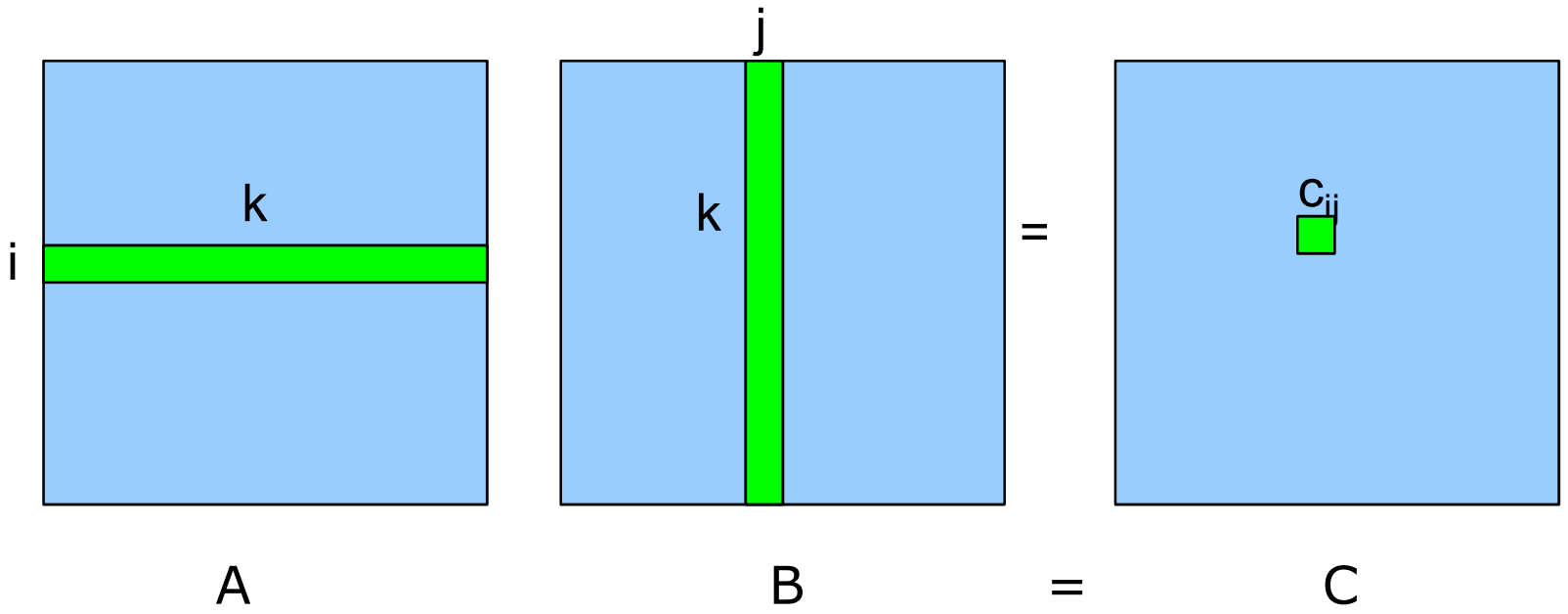
Apéndice: Suma de matrices

```
void suma_matrices (double *a,int fa,int ca,int lda, double *b,int
   fb,int cb,int ldb, double *c,int fc,int cc,int ldc)
{
   int i,j;

   for(i=0 ; i<fa ; i++)
   {
       for(j=0 ; j<ca ; j++)
       {
           c[i*ldc+j] = a[i*lda+j] + b[i*ldb+j];
       }
   }

}
```

Apéndice: Multiplicación de matrices

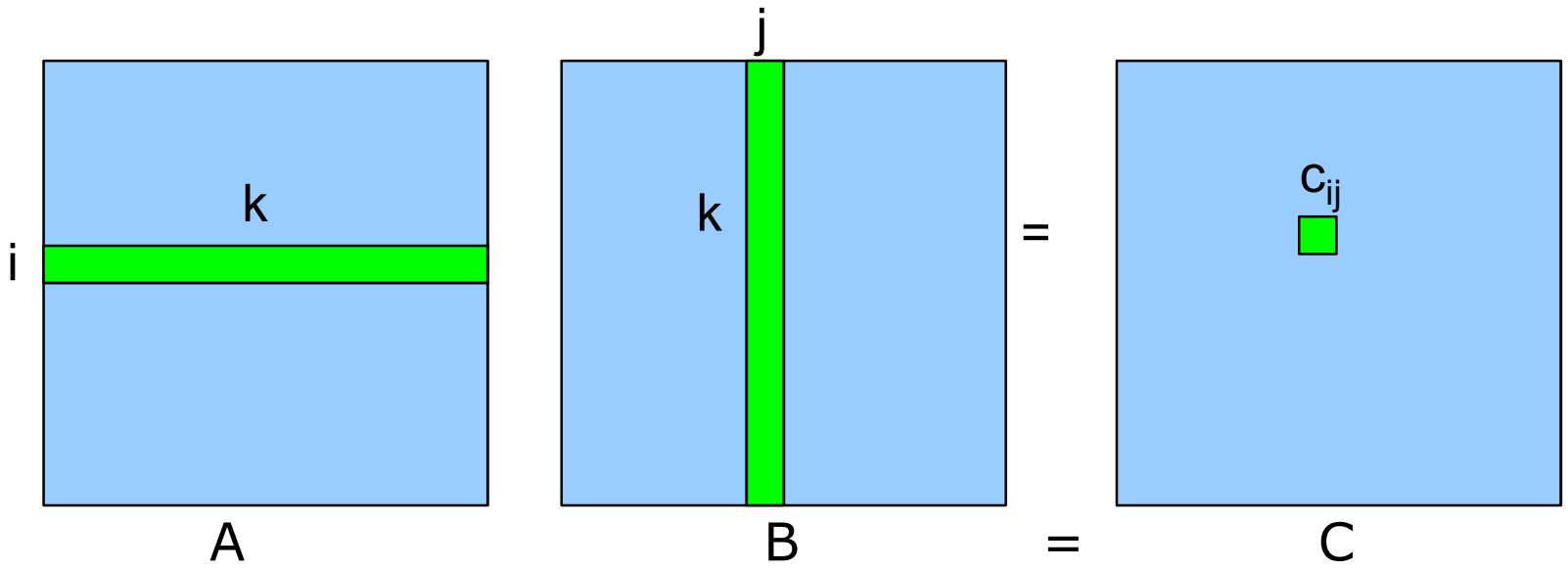


$$c_{ij} = a_{i0}\, b_{0j} + a_{i1}\, b_{1j} + \ldots + a_{i\,n-1}\, b_{n-1\,j}$$

Por cada elemento de C hay que realizar 2n operaciones

Total multiplicación matricial: $O(n)=2n^3$

Apéndice: Multiplicación de matrices

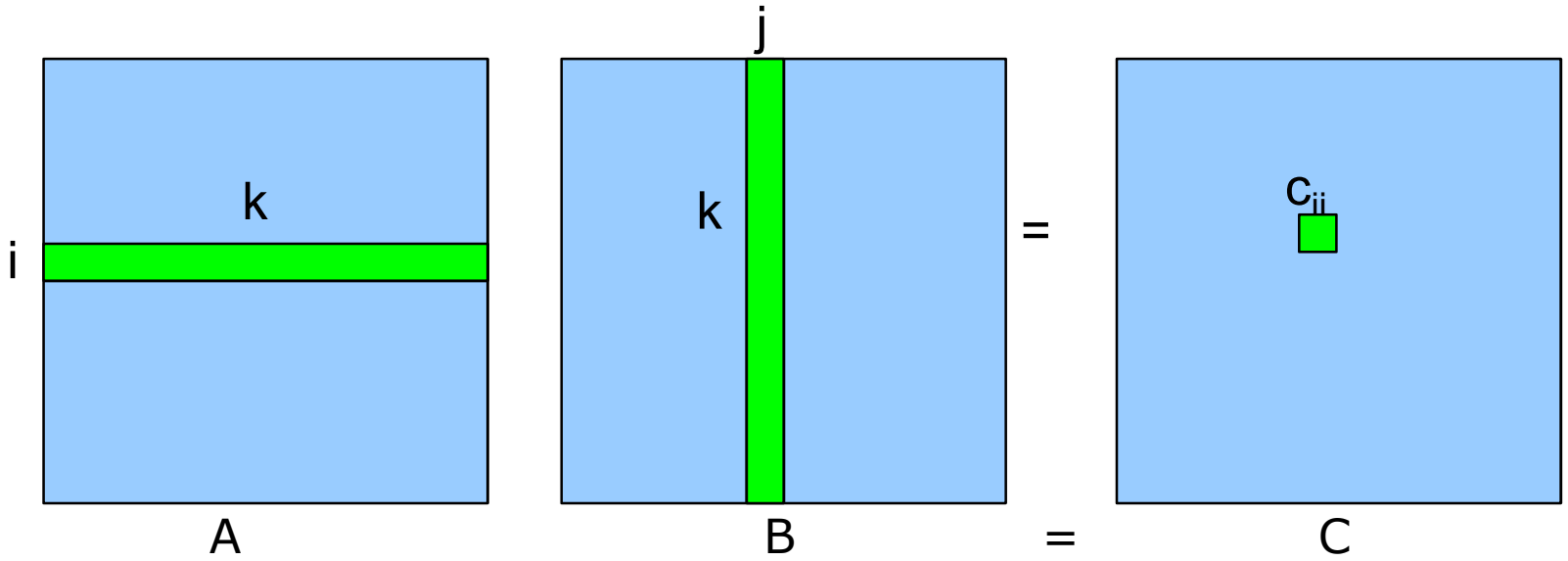


Matriz A: acceso por filas

- **Localidad espacial:**
  Se utilizan elementos contiguos de A cuando se calcula c_{ij}
  Espacio almacenamiento aprovechado: línea de cache

- **Localidad temporal:**
  Se reutiliza cada elemento de A cuando se va calculando una fila de C
  Espacio almacenamiento necesario: fila de A

Apéndice: Multiplicación de matrices



Matriz B: acceso por columnas

- **Localidad espacial:**

  Se utiliza elementos contiguos de B cuando se pasa de calcular c_{ij} a c_{ij+1}

  Espacio almacenamiento necesario: columna de B + líneas de caché

- **Localidad temporal:**

  Se reutiliza cada elemento de B cuando se avanza en una columna de C

  Espacio almacenamiento necesario: toda la matriz B

Apéndice: Operaciones por bloques

Objetivo: Mejorar la localidad de acceso a los datos

- **Aumentar Localidad Espacial**:
  - almacenar los datos acorde al esquema de acceso que se va a realizar
  - → Ideas ?

- **Aumentar Localidad Temporal**
  - reordenar operaciones para agrupar las que actúan con el mismo subconjunto de datos
  - → La que vamos a utilizar en las prácticas

Apéndice: Multiplicación de matrices por bloques:
mejorando localidad temporal

Hay un recorrido en matriz C, calculando de bloque en bloque

Por cada bloque de C:

- recorrer fila de bloques de A y columna de bloques de B
- multiplicando bloques y sumando bloques resultados

Estas operaciones con bloques:

- Son multiplicaciones y sumas matriciales
- Pero…con submatrices almacenadas dentro de matrices mayores
- Importante → *leading dimension*