Modelos programación HPC: OpenMP

Paralelismo y Sistemas Distribuidos

Grado en ciencia e ingeniería de datos

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

Licencia

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es

Índice

- Objetivos
- El modelo de tareas en OpenMP
- Visibilidad de variables
- Paralelización con tareas implícitas
  - Regiones paralelas
  - Bucles (loops)
    - Planificación
    - Fusión de bucles
  - Sincronizaciones
- Paralelización tareas explícitas
  - Control del número de tareas: cut-offs
  - Sincronizaciones y especificación de dependencias

Objetivos

- El objetivo al paralelizar un programa con OpenMP es:
  - Identificar las partes de código (tasks) que pueden ejecutarse de forma concurrente
  - Identificar las estructuras de datos que pueden accederse de forma concurrente
  - Asegurar que el resultado es correcto y eficiente
    - Identificar dependencias y variables compartidas que deben ser sincronizadas
- OpenMP es un modelo basado en directivas que se insertan en el código. El programador activa OpenMP mediante un flag de compilador.
  - En este caso, el compilador genera automáticamente el código para expresar el paralelismo indicado por las directivas
- Además de las directivas, se pueden incluir de forma manual funciones de la librería de OpenMP

Paralelismo de tareas vs datos

- Paralelizar un código siempre implica expresar las tareas que serán concurrentes, la *visibilidad* que queremos darle a nuestras variables, las dependencias entre tareas y las sincronizaciones
- En función del coste de acceso remoto a datos, expresaremos el paralelismo centrándonos en las tareas (más tareas y más dinámicas) o en los datos (tareas más estáticas para minimizar comunicaciones)

- https://www.openmp.org/
- https://www.openmp.org/resources/tutorials-articles/

Directivas OpenMP

- Las directivas (o constructs) OpenMP en general tienen la forma:
  - **#pragma omp contruct [clausula [clausula] …]**
- La “construct”= directiva indica que queremos hacer
  - Crear región paralela
  - Marcar una dependencia
  - Proteger una variable
  - Etc
- Las clausulas adaptan el comportamiento por defecto de esas directivas, y son opcionales
- Las directivas suelen afectar a un bloque estructurado (*structured block*)
  - Una o más líneas de código con un punto de entrada y uno de salida
- Las directivas tienen definida una visibilidad de las variables por defecto y una sincronización (o no) por defecto

Directivas OpenMP: ejemplo

```
void main(int argc,char *argv[])
{
#pragma omp parallel
printf("Hello 1\n");
printf("Hello 2\n");
}
```

```
void main(int argc,char *argv[])
{
#pragma omp parallel
{
printf("Hello 1\n");
printf("Hello 2\n");
}
}
```

- OpenMP tiene un número de threads por defecto (normalmente el número de threads de la máquina)
- Podemos modificarlo mediante variables de entorno
- export OMP_NUM_THREADS=2

Directivas OpenMP: ejemplo

- También Podemos definirlo explícitamente, pero esta opción es estática → para cambiarla hay que recompilar!

```
void main(int argc,char *argv[])
{
/* num_threads en una clausula de la directiva parallel */
#pragma omp parallel num_threads(4)
{
printf("Hello 1\n");
printf("Hello 2\n");
}
}
```

- Podemos usar funciones de librería para, por ejemplo, consultar cuantos threads hay o que thread somos del total

```
void main(int argc,char *argv[])
{
/* En este caso, el parallel afecta a a las 2 lineas */
#pragma omp parallel num_threads(4)
{
printf("Hello 1,, soy el thread %d de %d\n",omp_get_thread_num(),omp_get_num_threads());
printf("Hello 2\n");
}
}
```

Parallel

```
#pragma omp parallel [clausulas]
Structured_block
```

- Ámbito de las variables (data sharing)
  - Las variables declaradas antes del *parallel* son compartidas por defecto
  - Las variables declaradas dentro del *parallel* son privadas
- Sincronizaciones
  - La directiva parallel implica una sincronización global de todos los threads antes de continuar (barrier, implícito en este caso)

Visibilidad de variables

- SHARED = COMPARTIDAS (entre threads)
  - Las variables de tipo compartidas puede provocar condiciones de carrera (race conditions)
  - Si varios threads acceden a la misma posición de memoria para leer y escribir, debemos protegerlas.
- PRIVATE = PRIVADAS
  - Las variables que se privatizan no generan conflictos pero cada thread trabaja con una copia privada (sin valor inicial)
- FIRSTPRIVATE = COPIA el VALOR Y PRIVATIZA
  - Las variables configuradas como *firstprivate*, se privatizan y se hace una copia del valor anterior a la creación de la task
- Las variables configuradas como *private* o *firstprivate* **NO SE CONSERVA** su valor al salir de la task

Ejemplo

```
int var_shared=1,var_private=0,var_first_private=10;
/* En este caso, el parallel afecta a a las 2 lineas */
#pragma omp parallel num_threads(4) private(var_private) firstprivate(var_first_private)
{
int i;
int id=omp_get_thread_num();
for (i=0;i<10;i++){
        var_shared=id;
        var_first_private++;
        var_private++;
#pragma omp barrier
        printf("Soy %d : shared %d, var_private %d var_first_private %d\n",id,
       var_shared,var_private,var_first_private);
}
}
```

```
Soy 0 : shared 3, var_private 770765313 var_first_private 11
Soy 3 : shared 3, var_private 770734337 var_first_private 11
Soy 1 : shared 3, var_private 770761729 var_first_private 11
Soy 2 : shared 3, var_private 770756353 var_first_private 11
Soy 3 : shared 2, var_private 770734338 var_first_private 12
Soy 2 : shared 2, var_private 770756354 var_first_private 12
Soy 0 : shared 2, var_private 770765314 var_first_private 12
---
```

Ejemplo

```
#pragma omp parallel num_threads(4) private(var_private) firstprivate(var_first_private)
{
int i;
int id=omp_get_thread_num();
for (i=0;i<10;i++){
var_first_private++;
var_private=id;
#pragma omp barrier
printf("Soy %d : var_private %d var_first_private %d\n",id,var_private,var_first_private);
}
}
```

```
Soy 0 : var_private 0 var_first_private 11
Soy 3 : var_private 3 var_first_private 11
Soy 1 : var_private 1 var_first_private 11
Soy 2 : var_private 2 var_first_private 11
Soy 3 : var_private 3 var_first_private 12
Soy 2 : var_private 2 var_first_private 12
Soy 0 : var_private 0 var_first_private 12
Soy 1 : var_private 1 var_first_private 12
Soy 3 : var_private 3 var_first_private 13
...
```

PARALELISMO DE TAREAS

Tareas=taks

- En OpenMP el trabajo lo realizan los threads y el trabajo a realiza se especifica mediante las tareas, si no hay tareas no se hace nada (siempre hay una tarea “idle”)
- Si no hay threads no hay paralelismo
- Si no hay tareas los threads están “idle”
- Los threads se crean explícitamente
  - *parallel*
- Las tareas se pueden crear **implícitamente** o **explícitamente**
  - Implícita: *parallel* crea implícitamente 1 tarea por thread
  - Explícita: *task*
- Las tareas pueden ser ejecutadas de forma **inmediata** o **diferida,** depende de cómo las hayamos creado
  - *Parallel*: la tarea se asigna al thread de forma inmediata
  - *Task*: la tarea se pone una lista de tareas por hacer y el primer thread que esté “idle” la ejecutará

Creación de threads (y tareas)

- Creación de threads: únicamente con la directiva parallel
  - #pragma omp parallel: Crea N threads. N puede ser definido (menos a más prioridad)
    - Por defecto
    - Mediante variable de entorno OMP_SET_NUM_THREADS
    - Mediante clausula num_threads
  - En este caso, se crea 1 tarea por thread de forma implícita ya que:
    - Los threads son los componentes que ejecutan trabajo
    - El trabajo se especifica mediante las tareas, si no hay tareas no se hace nada

Worksharing: for (o parallel for)

- La directiva ”parallel” crea N threads que hacen, por defecto, lo mismo
- Un uso típico de OpenMP es la creación de threads para “cooperar” en solucionar un problema más rápido, por ejemplo, la ejecución de un bucle,

```
#pragma omp parallel for
For_loop
```

```
#define ITERS 12
void main(int argc,char *argv[])
{
int i;
#pragma omp parallel for num_threads(4)
for (i=0;i<ITERS;i++){
        printf("Soy %d de %d ejecutando iteracion %d \n",o
}
```

```
#pragma omp parallel num_threads(4)
#pragma omp for
for (i=0;i<ITERS;i++){
,,,
}
```

```
psd0@boada-1:~/ejemplos_openmp$ ./test_loop
Soy 0 de 4 ejecutando iteracion 0
Soy 0 de 4 ejecutando iteracion 1
Soy 0 de 4 ejecutando iteracion 2
Soy 1 de 4 ejecutando iteracion 3
Soy 1 de 4 ejecutando iteracion 4
Soy 1 de 4 ejecutando iteracion 5
Soy 2 de 4 ejecutando iteracion 6
Soy 2 de 4 ejecutando iteracion 7
Soy 2 de 4 ejecutando iteracion 8
Soy 3 de 4 ejecutando iteracion 9
Soy 3 de 4 ejecutando iteracion 10
Soy 3 de 4 ejecutando iteracion 11
```

Worksharing: for (o parallel for)

- Con ***for*** el trabajo no se replica, sinó que se reparte. Se puede aplicar a bucles con dimensiones conocidas.
- El compilador crear tareas que asigna a los diferentes threads de forma inmediata
- Distribuye las iteraciones entre threads dependiendo del scheduler que se aplica
  - Scheduler por defecto=STATIC
  - Hay una cláusula para cambiarlo: schedule(Schedule_name[,chuncl_size])
- Cada thread ejecuta de 0..N *chunks* de iteraciones
- 1 *chunck* es una secuencia de iteraciones consecutivas
- Cada scheduler se diferencia de otro en como se asignan las iteraciones a los threads (de forma estática o dinámica) y si el tamaño de los chuncks es fijo o variable

Posibles problemas

- Balanceo de carga
  - Para este caso tenemos la posibilidad de cambiar la distribución de iteraciones en threads → No todos los threads harán la mima cantidad de iteraciones
- Cantidad de trabajo
  - Si hay poco trabajo que hacer, hemos de limitar la cantidad de threads para evitar perder el tiempo
- Cantidad de iteraciones
  - Si hay pocas iteraciones, en algunos casos si se puede solucionar!!

Schedulings (algunos, hay más)

- *Static*:
  - las iteraciones se dividen en chuncks de chunck_size (o N/num_threads en su defecto)→ tamaño fijo
  - se asignan a los threads de forma estática y rotativa → asignación estática
- *Dynamic*:
  - las iteraciones se dividen en chuncks de chunck_size (1 en su defecto). Cada thread se asigna un chunck (por orden) → tamaño fijo.
  - A medida que terminan eligen uno nuevo de la lista de chucks → asignación dinámica
- Runtime → se selecciona el scheduling en el momento de la ejecución en función de la variable de entorno

Dynamic

- Este scheduling es útil cuando hay desbalanceo de carga (no todas las iteraciones tardan lo mismo)

```
#pragma omp parallel for num_threads(4) schedule(dynamic)
for (i=0;i<ITERS;i++){
printf("Soy %d de %d ejecutando iteracion %d \n",omp_get_thread_num(),omp_get_num_threads(),i);
}
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./loop_dynamic
Soy 3 de 4 ejecutando iteracion 0
Soy 3 de 4 ejecutando iteracion 4
Soy 3 de 4 ejecutando iteracion 5
Soy 3 de 4 ejecutando iteracion 6
Soy 3 de 4 ejecutando iteracion 7
Soy 3 de 4 ejecutando iteracion 8
Soy 2 de 4 ejecutando iteracion 2
Soy 2 de 4 ejecutando iteracion 10
Soy 2 de 4 ejecutando iteracion 11
Soy 3 de 4 ejecutando iteracion 9
Soy 1 de 4 ejecutando iteracion 3
Soy 0 de 4 ejecutando iteracion 1
```

Static vs dynamic

- Static es muy recomendable cuando el coste de las iteraciones es similar, ya que reduce al mínimo el tiempo de generar trabajo y maximiza la localidad de datos, especialmente cuando chunk_size=N/nthreads
- Dynamic tiene más overhead ya que los threads van a coger trabajo dinámicamente y eso puede generar conflictos. Sin embargo, ajusta mucho mejor la distribución de trabajo cuando hay desbalanceos.

FOR…algún detalle más

- Las variables declaradas antes de la directivas serán compartidas excepto el índice del bucle que se privatiza
- Sincronización
  - Se asume que todas las iteraciones son compartidas (si no es el caso, habrá que especificarlo)
  - Hay una sincronización implícita al final del for

Ejemplo

```
/* Los dos bucles se ejecutan con 4 threads. Solo se crean threads 1 vez (en el parallel). Bucle 2 no
   empieza hasta que termina bucle 1*/
#pragma omp parallel num_threads(4)
{
#pragma omp for  Schedule(static)
for (i=0;i<ITERS;i++){
printf("Bucle 1 Soy %d de %d ejecutando iteración %d \n",omp_
}
#pragma omp for  Schedule(static)
for (i=0;i<ITERS;i++){
printf("Bucle 2 Soy %d de %d ejecutando iteración %d \n",omp_
}
}
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./loop2
Bucle 1 Soy 1 de 4 ejecutando iteracion 3
Bucle 1 Soy 1 de 4 ejecutando iteracion 4
Bucle 1 Soy 1 de 4 ejecutando iteracion 5
Bucle 1 Soy 0 de 4 ejecutando iteracion 0
Bucle 1 Soy 0 de 4 ejecutando iteracion 1
Bucle 1 Soy 0 de 4 ejecutando iteracion 2
Bucle 1 Soy 3 de 4 ejecutando iteracion 9
Bucle 1 Soy 3 de 4 ejecutando iteracion 10
Bucle 1 Soy 3 de 4 ejecutando iteracion 11
Bucle 1 Soy 2 de 4 ejecutando iteracion 6
Bucle 1 Soy 2 de 4 ejecutando iteracion 7
Bucle 1 Soy 2 de 4 ejecutando iteracion 8
Bucle 2 Soy 1 de 4 ejecutando iteracion 3
Bucle 2 Soy 1 de 4 ejecutando iteracion 4
Bucle 2 Soy 1 de 4 ejecutando iteracion 5
Bucle 2 Soy 0 de 4 ejecutando iteracion 0
Bucle 2 Soy 0 de 4 ejecutando iteracion 1
Bucle 2 Soy 0 de 4 ejecutando iteracion 2
Bucle 2 Soy 2 de 4 ejecutando iteracion 6
Bucle 2 Soy 2 de 4 ejecutando iteracion 7
Bucle 2 Soy 2 de 4 ejecutando iteracion 8
Bucle 2 Soy 3 de 4 ejecutando iteracion 9
Bucle 2 Soy 3 de 4 ejecutando iteracion 10
Bucle 2 Soy 3 de 4 ejecutando iteracion 11
```

Modificamos la sincronización: nowait

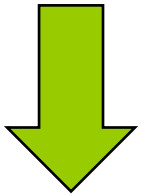
```
#pragma omp parallel num_threads(4)
{
#pragma omp for  schedule(static) nowait
for (i=0;i<ITERS;i++){
printf("Bucle 1 Soy %d de %d ejecutando iteracion %
}
#pragma omp for  schedule(static)
for (i=0;i<ITERS;i++){
printf("Bucle 2 Soy %d de %d ejecutando iteracion %
}
}
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./loop3
Bucle 1 Soy 1 de 4 ejecutando iteracion 3
Bucle 1 Soy 1 de 4 ejecutando iteracion 4
Bucle 1 Soy 1 de 4 ejecutando iteracion 5
Bucle 2 Soy 1 de 4 ejecutando iteracion 3
Bucle 1 Soy 3 de 4 ejecutando iteracion 9
Bucle 1 Soy 3 de 4 ejecutando iteracion 10
Bucle 1 Soy 3 de 4 ejecutando iteracion 11
Bucle 2 Soy 3 de 4 ejecutando iteracion 9
Bucle 2 Soy 3 de 4 ejecutando iteracion 10
Bucle 2 Soy 3 de 4 ejecutando iteracion 11
Bucle 1 Soy 0 de 4 ejecutando iteracion 0
Bucle 1 Soy 0 de 4 ejecutando iteracion 1
Bucle 1 Soy 0 de 4 ejecutando iteracion 2
Bucle 2 Soy 0 de 4 ejecutando iteracion 0
Bucle 2 Soy 0 de 4 ejecutando iteracion 1
Bucle 2 Soy 0 de 4 ejecutando iteracion 2
Bucle 1 Soy 2 de 4 ejecutando iteracion 6
Bucle 1 Soy 2 de 4 ejecutando iteracion 7
Bucle 1 Soy 2 de 4 ejecutando iteracion 8
Bucle 2 Soy 2 de 4 ejecutando iteracion 6
Bucle 2 Soy 2 de 4 ejecutando iteracion 7
Bucle 2 Soy 2 de 4 ejecutando iteracion 8
Bucle 2 Soy 1 de 4 ejecutando iteracion 4
Bucle 2 Soy 1 de 4 ejecutando iteracion 5
```

- Entre bucles salen meclados
- Entre threads ordenados

Cantidad de trabajo

```
/* ITERS=12 */
#pragma omp parallel for num_threads(48)
for (i=0;i<ITERS;i++){
        for (j=0;j<10000;j++){
        do_work(i,j);
        }
}
```



```
#pragma omp parallel for num_threads(48) collapse(2)
for (i=0;i<ITERS;i++){
        for (j=0;j<10000;j++){
        do_work(i);
        }
}
```

Con 12 iteraciones, no podemos ocupar 48 threads, pero en realidad SI hay iteraciones, solo que en bucle interno

Le decimos al compilar que considere 2 bucles, no 1. Han de estar perfectamente anidados!!

TAREAS EXPLÍCITAS

Tareas

- Las directivas de *worksharing* (for) nos permiten ejecutar fácilmente el paralelismo de bucles, pero muchas aplicaciones no siguen ese patrón, sino que tienen un paralelismo a nivel de función

```
#pragma omp task [clausulas]
Structured_block
```

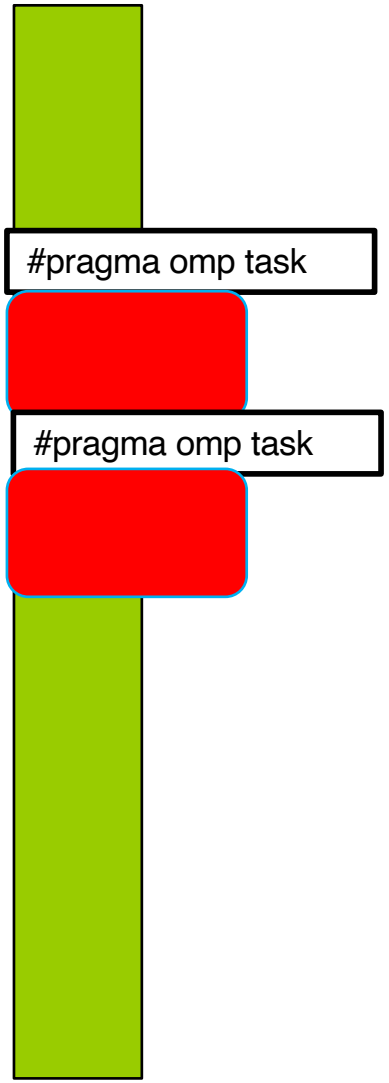
- La directiva, solo crea la tarea. El thread que crea la tarea no la ejecuta de forma inmediata, que lo pone en la lista y continua con la tarea que el estaba ejecutando
- Por defecto, las tareas nuevas se añaden a una lista de tareas y se ejecutan en *modo diferido*, es decir, a medida que los threads van consumiendo tareas van ejecutando las que están en la lista de pendientes
- No existe una sincronización explícita
- Para que haya threads, se tienen que haber creado antes con la directiva parallel

Ejemplo tasks

Queremos crear 2 tasks para dividir el vector en 2 partes

```
#define N 40960
int a[N],b[N],c[N];
void vector_add(int *A, int *B, int *C, int n)
{
int i;
#pragma omp task
for (i=0; i< n/2; i++)
C[i] = A[i] + B[i];
#pragma omp task
for (i=n/2; i< n; i++)
C[i] = A[i] + B[i];
}
void main()
{
init_vector(a,N,1);
init_vector(b,N,2);
init_vector(c,N,0);


vector_add(a, b, c, N);
}
```



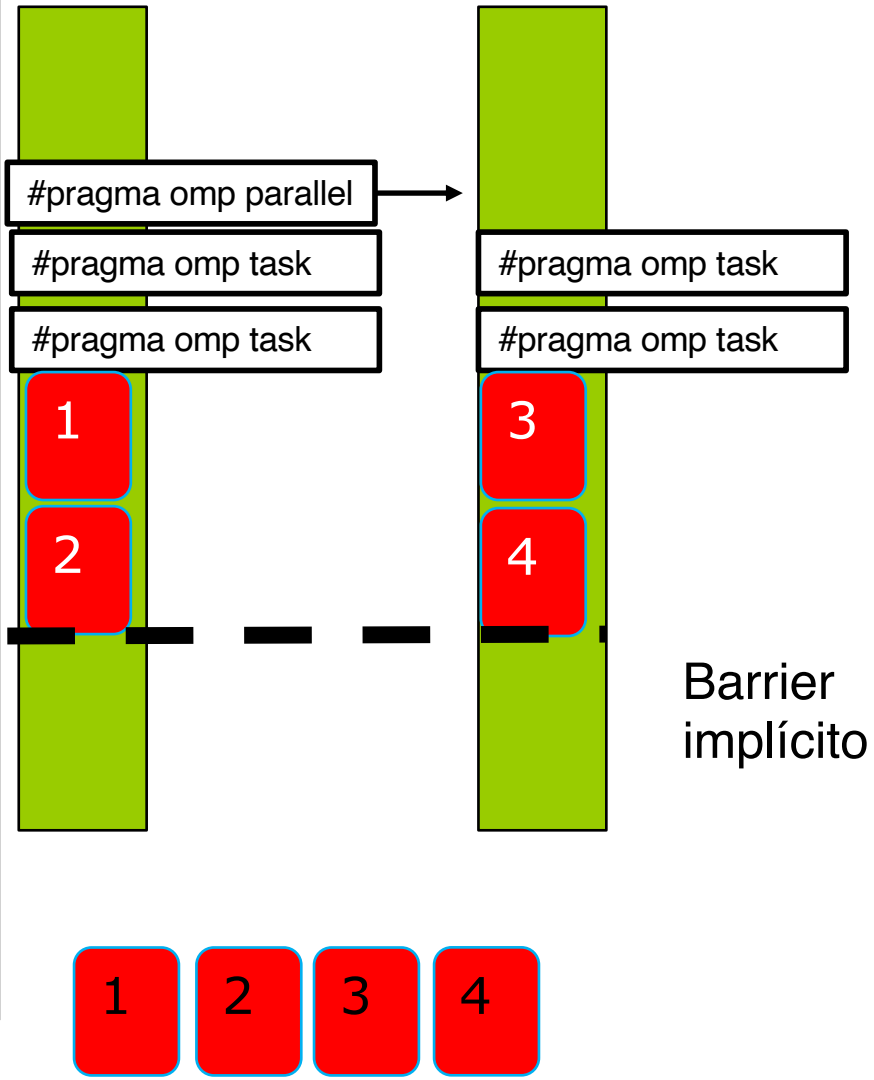
Si no hay threads, no hay paralelismo!!!

Ejemplo tasks

Queremos crear 2 tasks para dividir el vector en 2 partes

```
#define N 40960
int a[N],b[N],c[N];
void vector_add(int *A, int *B, int *C, int n)
{
int i;
#pragma omp task
for (i=0; i< n/2; i++)
C[i] = A[i] + B[i];
#pragma omp task
for (i=n/2; i< n; i++)
C[i] = A[i] + B[i];
}
void main()
{
init_vector(a,N,1);
init_vector(b,N,2);
init_vector(c,N,0);
#pragma omp parallel num_threads(2)

vector_add(a, b, c, N);
}
```



Esta opción crearía
2xnum_threads tareas

Directiva: Single

```
#pragma omp single [clausulas]
Structured_block
```

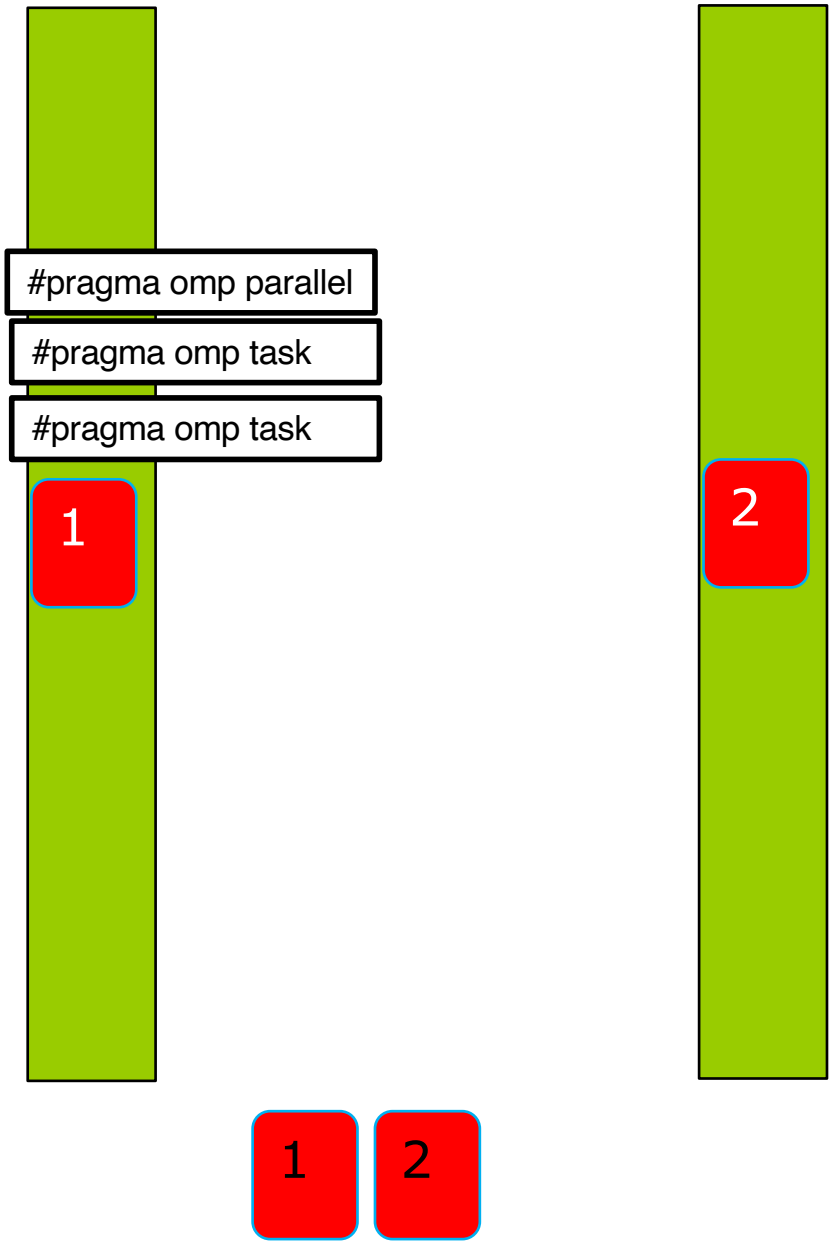
- Cuando queremos que un trozo de código SOLO lo ejecute un thread, podemos utilizar la directiva single
- Tienen un barrier implícito, todos los threads esperarán antes de continuar

```
#pragma omp parallel num_threads(4)
{
id=omp_get_thread_num();
printf("Hello 1 soy el thread %d de %d th=%d\n",id,omp_get_num_threads(),omp_get_thread_num());
#pragma omp single
{
printf("Este mensaje sale solo 1 vez\n");
} /* BARRIER */
printf("Y este ya lo hacen todos\n");
}
printf("Despues del parallel\n");
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./single
Hello 1 soy el thread 2 de 4 th=2
Hello 1 soy el thread 0 de 4 th=0
Este mensaje sale solo 1 vez
Hello 1 soy el thread 1 de 4 th=1
Hello 1 soy el thread 3 de 4 th=3
Y este ya lo hacen todos
Y este ya lo hacen todos
Y este ya lo hacen todos
Y este ya lo hacen todos
Despues del parallel
```

Ejemplo tasks

```
#define N 40960
int a[N],b[N],c[N];
void vector_add(int *A, int *B, int *C, int n)
{
int i;
#pragma omp task
for (i=0; i< n/2; i++)
C[i] = A[i] + B[i];
#pragma omp task
for (i=n/2; i< n; i++)
C[i] = A[i] + B[i];
}
void main()
{
init_vector(a,N,1);
init_vector(b,N,2);
init_vector(c,N,0);
#pragma omp parallel num_threads(2)
#pragma omp single
vector_add(a, b, c, N);
}
```



Tasks

- Visibilidad de variables
  - Las variables DECLARADAS ANTES del parallel se consideran compartidas –> Hay que pensar si hay algún conflicto
  - Las variables DECLARAS entre el parallel y la creación de la task se consideran FIRSTPRIVADAS
  - Las variables DECLARADAS dentro de la task son PRIVADAS
- Sincronización
  - No hay una sincronización explícita que espere a la finalización de las tareas

Sincronizaciones: ejemplo

```
int acum_vector(int *A, int n)
{
int acum_1=0,acum_2=0;
#pragma omp task
for (i=0; i< n/2; i++) acum_1=acum_1+A[i];
#pragma omp task
for (i=n/2; i< n; i++) acum_2=acum_2+A[i];
return acum_1+acum_2;
}
void main()
{
int total;
init_vector(a,N,1);
#pragma omp parallel
#pragma omp single
total=acum_vector(a, N);
/* if (total!=N) ERROR!!!! */
}
```

Error1: acum_1 y acum_2 no estarán calculados cuando se haga el return

Error2: acum_1 y acum_2 son firstprivate por defecto, el valor no se conserva fuera de la task!!

Sincronizaciones: ejemplo

```
int acum_vector(int *A, int n)
{
int i;
/* acum_1 y acum_2 son firstprivate por defecto */
int acum_1=0,acum_2=0;

#pragma omp task shared(acum_1)
for (i=0; i< n/2; i++) acum_1=acum_1+A[i];
#pragma omp task shared(acum_2)
for (i=n/2; i< n; i++) acum_2=acum_2+A[i];
#pragma omp taskwait
return acum_1+acum_2;
}
void main()
{
int total;
init_vector(a,N,1);
#pragma omp parallel
#pragma omp single
total=acum_vector(a, N);
/* if (total!=N) ERROR!!!! */
}
```

acum_1 y acum_2 serán shared

Esperamos a que esten calculados

SINCRONIZACIONES

Sincronizaciones

- Cuando una variable es accedida por múltiples threads a la vez, tanto en lectura como escritura, aparece el problema de “race condition” o condición de carrera
- El problema es que el resultado es incoherente ya es producto de la mezcla de instrucciones de lenguaje máquina que el programador asume que se van a hacer de forma consecutiva y que no es así
- Las sincronizaciones se usan para imponer un determinado orden y proteger el acceso a datos compartidos
- OpenMP ofrece diferentes mecanismos mediante directivas y funciones de la librería
- Estas opciones van de más restrictivas pero muy rápidas a más flexibles pero más lentas

Sincronizaciones:opciones

- Directivas
  - Critical
  - Atomic
  - Barrier
  - Ordered
  - Flush
- Cláusulas que Podemos usar en algunas directivas
  - **Reduction**
  - Depend
- Locks (funciones)

Rance condition

- Race condition example

```
void main(int argc,char *argv[])
{
        int acum=0,i;
        #pragma omp parallel for
        for (i=0;i<1000000;i++){
                acum=acum+1;
        }
        printf("Acum %d\n",acum);
}
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./sincro1
Acum 937499
```

La variable acum es shared, todos los threads acceden a la vez

Reduction

- Es la más eficiente. Crea una variable local y luego acumula los resultados parciales
- Sólo se puede aplicar a algunas operaciones: +, -, *, &, |, ^, && y ||

```
#pragma omp parallel for reduction(+:acum)
for (i=0;i<1000000;i++){
acum=acum+1;
}
printf("Acum %d\n",acum);
```

```
nct00018@login1:~/psd/ejemplos_openmp> ./sincro2
Acum 1000000
```

Critical, Atomic

- Critical se puede aplicar a un “stuctured block”, atomic se aplica a una única línea.
  - Critical acepta de forma condicional una etiqueta que permite identificar race conditions diferentes entre las cuales no hay conflictos
- En este caso no hay diferencia

```
#pragma omp parallel for
for (i=0;i<n;i++){
#pragma omp critical
total=total+A[i];
}
```

```
#pragma omp parallel for
for (i=0;i<n;i++){
#pragma omp atomic
total=total+A[i];
}
return total;
```

Locks

- Locks no tienen restricciones, se pueden usar en cualquier contexto
- Hay que declarar tantos “locks” como zonas haya que proteger

```
omp_lock_t mi_lock;
int a[N];
int vector_acum(int *A, int n)
{
int i,total=0;
#pragma omp parallel for
for (i=0;i<n;i++){
omp_set_lock(&mi_lock);
total=total+A[i];
omp_unset_lock(&mi_lock);
}
return total;
}
void main()
{
int acum;
init_vector(a,N,1);
omp_init_lock(&mi_lock);
acum=vector_acum(a, N);
omp_destroy_lock(&mi_lock);
printf("La suma de los elementos de a es
%d\n",acum);
}
```

Barrier

- Una directiva barrier marca un punto en el cual, dado un conjunto de threads, ninguno de ellos puede continuar la ejecución hasta que todos hayan alanzado el barrier y todas las tasks explícitas generadas hayan acabado

```
#pragma omp barrier
```

- Aplica a barriers explícitos y por defecto (como en el parallel)

Ordered

- La clausula/directiva *ordered* nos permite paralelizar bucles que no son 100% paralelos
- La cláusula aplicada a un bucle índice cuantos niveles de anidamiento hemos de considerar (en el ejemplo 2: i , j)
- Dentro del bucle hemos de indicar de qué iteraciones dependemos (**sink**) y cuando la parte de la iteración que genera la dependencia está terminada (**source**)

```
#pragma omp for schedule(static,1) ordered(2)
for ( i = 1; i < N; i++ )
for ( j = 1; j < N; j++ ) {
#pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
foo (i, j);
#pragma omp ordered depend(source)
}
}
```

Consideramos 2 niveles (i,j)

Dependemos de la i-1,j y i,j-1

La dependencia de la iteración i,j está lista

Flush

- La cláusula flush permite indicar que hay que sincronizar en memoria todos los accesos a memoria pendientes

Dependencias

- Mediante la cláusula depend podemos indicar las dependencias que existen entre tasks. En tiempo de ejecución la librería se encarga de identificar las dependencias y saber que tasks están libres (o no) de dependencias.

```
#pragma omp task [depend (in : var_list)] [depend (out : var_list)
[depend (inout : var_list)]
```

- Para calcular las dependencias solo se tiene en cuenta las tareas creadas anteriormente en el tiempo en el mismo nivel (sibling tasks).
- Cuando una tarea (T1) se especifica, por ejemplo, que tiene una dependencia IN, se mira si existe una task hermana ANTERIOR con la misma variable marcada con OUT (T2). Si existe, esas dos tareas tienen una dependencia y T2 no empezará hasta que T1 haya terminado

TASKS CUT-OFFS

Límites en la generación de tasks

- Cuando la generación de tareas se hace de forma recursiva o iterativa, se pueden llegar a generar muchas tareas o tareas con muy poca carga de trabajo
  - En ambos casos se incrementa el overhead
- Existen diferentes alternativas para limitar la cantidad de tasks o la cantidad de trabajo que ejecutan
  - Modificación del código (añadir algún condicional que impida la creación de tasks)
  - Mediante cláusulas de la directiva tasks que generan un código condicional automáticamente (lo genera el compilador)
- Estas estrategias se conocen como mecanismos de “cut-off”

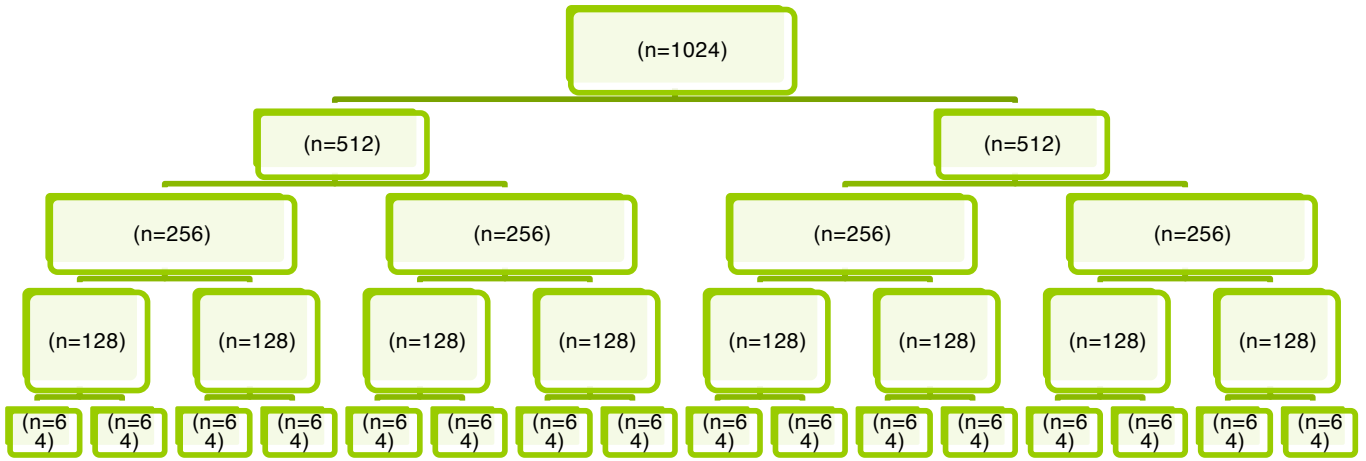
Ejemplo

```
void vector_add(int *A, int *B, int *C, int n)
{
        int i;
        for (i=0;i<n;i++) C[i] = A[i] + B[i];
}
void vector_add_rec(int *A, int *B, int *C, int n)
{
int i,size;
/* Caso no recursivo */
if (n<=MIN_N){  vector_add(A,B,C,n);return;}
/* Caso recursivo */
#pragma omp task
vector_add_rec(A,B,C,n/2);
#pragma omp task
vector_add_rec(&A[n/2],&B[n/2],&C[n/2],n-(n/2));
}
void main()
{
…
#pragma omp parallel
#pragma omp single
vector_add_rec(a, b, c, N);
…
}
```

Con esta condición controlamos el tamaño mínimo

Ejemplo

N=1024
MIN_N=64



Tipos de límites

- Dependiendo del problema, nos puede interesar un cut-off estático o dinámico
- Estático: cuando se cumple una determinada condición ya no generamos más tasks
  - A partir de un tamaño de problema → más sencillo con el código ya que hemos de incluir el caso NO recursivo
  - A partir de un número de niveles en la recursividad
- Dinámico: La condición que determina si hemos de generar más tasks varía en el tiempo
  - Ejemplo: queremos tener un máximo de N tasks pendientes. Las tasks se van ejecutando, por lo tanto este valor va variando
  - Este caso es más complejo!!

Cláusula final [+ mergeable]

- La cláusula "*final(condición)*" (aplicada a la directiva task) indica al compilador que:
  - Si la condición es cierta, las task que se genera y todas las que ella genere, se ejecutarán de forma secuencial por la task que ha encontrado la directiva. En este caso, la task se crea
- Si añadimos la cláusula *mergeable*, el compilador puede decidir no crear la task (es lo más lógico)

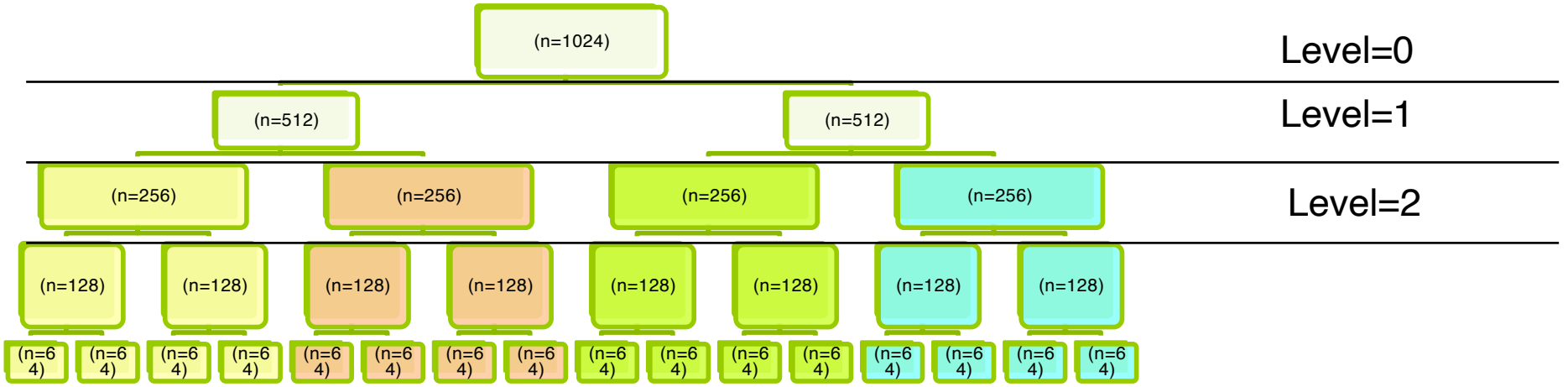
Ejemplo

```
void vector_add(int *A, int *B, int *C, int n)
{
        int i;
        for (i=0;i<n;i++) C[i] = A[i] + B[i];
}
void vector_add_rec(int *A, int *B, int *C, int n,int level)
{
int i,size;
/* Caso no recursivo */
if (n<=MIN_N){  vector_add(A,B,C,n);return;}
/* Caso recursivo */
#pragma omp task  final(level>=MAX_LEVELS) mergeable
vector_add_rec(A,B,C,n/2,level+1);
#pragma omp task final(level>=MAX_LEVELS) mergeable
vector_add_rec(&A[n/2],&B[n/2],&C[n/2],n-(n/2),level+1);
}
void main()
{
…
#pragma omp parallel
#pragma omp single
vector_add_rec(a, b, c, N,0);
…
}
```

Hemos de modificar el código para saber cuantos niveles tenemos.

Ejemplo

MAX_LEVELS=2
N=1024
MIN_N=64



PARALELIZACIÓN BASADA EN DATOS

Paralelización considerando datos como primer criterio

- Normalmente, ,estos casos aplican a entornos de memoria distribuida, en los que utilizaremos MPI como modelo de programación
- En cualquier caso, tambien podemos usar OpenMP siempre que sea dentro de un nodo
- Se calcula lo que se conoce como las ”*owner compute rules*” , quien soy y que tengo que hacer

Ejemplo

```
void main(int argc,char *argv[])
{
/* En este caso, el parallel afecta a a las 2 lineas */
#pragma omp parallel num_threads(4)
{
int id,num_th,i;
int first_iter,last_iter,num_iters;

id=omp_get_thread_num();
num_th=omp_get_num_threads();
num_iters=ITERS/num_th;
first_iter=(id*num_iters);
last_iter=first_iter+num_iters;
if (id==(num_th-1)) last_iter=ITERS;

for (i=first_iter;i<last_iter;i++){
                do_work(i);
}
}
}
```

Este código hace una distribución similar a la que hace un ”parallel for schedule(static)”

¿Quien soy dentro del grupo?

¿Qué tengo que hacer?

RESUMEN

OpenMP

- Parallel → Única directive que crea threads
  - Control del número de threads (num_threads(x))
- Visibilidad variables: aplica a todas las directivas
  - Shared, private, firstprivate
  - Hay que saber como aplica por defecto a cada directiva
- Paralelismo de bucles
  - For
  - **Desbalanceo de carga: cláusula schedule para modificar la asignación de trabajo a threads**
- Tasks: paralelismo más functional pero se puede usar para todo

OpenMP

- Sincronizaciones: aplica a todas las directivas.
  - Entre tasks
    - Barrier/nowait
    - Depend
    - taskwait
    - Hay que saber como aplica por defecto a cada directive
  - Protección de acceso a variables compartidas
    - Reduction
    - Critical
    - Atomic
    - Locks
- Control del **overhead en la creación de tasks**
  - Final(condición)