COMPUTACIÓN DE ALTA PERFORMANCE

Curso 2025

Sergio Nesmachnow (sergion@fing.edu.uy)

Centro de Cálculo

TEMA 7: OpenMP

Contenido

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Desarrollo de aplicaciones paralelas con OpenMP
- Conceptos y sintaxis
- Modelo de ejecución y memoria
- La API de OpenMP
  - Crear y distribuir trabajos
  - Paralelización de ciclos
  - Master, secciones y variables
  - Asignación de trabajo
  - Tareas

El paradigma de memoria compartida

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Modelo de programación para aplicaciones en sistemas de memoria compartida (multitarea o multithreading)
- Se dispone de un recurso compartido común (la memoria del sistema)
- Los procesos concurrentes se comunican y sincronizan mediante el uso del recurso compartido
- Las tareas o threads comparten un espacio de direccionamiento global, donde leen y escriben (de manera asincrónica)
- Son necesarios mecanismos de protección para el acceso a recursos (variables) compartidos
- Implementable sobre sistemas UMA y sistemas NUMA
- Permite el desarrollo de aplicaciones híbridas (memoria compartida y distribuida)

OpenMP

- Interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas
- Modelo de programación portable y escalable que proporciona una interfaz simple y flexible para el desarrollo de aplicaciones paralelas.
- Definido conjuntamente por proveedores de hardware y de software.
- Modelo de ejecución fork-join
  - Para threads: paralelismo sincrónico, barreras implícitas de sincronización
- Modelo de ejecución con grafo de dependencias
  - Para tareas: paralelismo asincrónico, el uso de barreras no es estrictamente necesario
- Permite el desarrollo de aplicaciones híbridas (memoria compartida y distribuida)

OpenMP

- Objetivo: desarrollar un estándar portable y eficiente, para programación paralela
- Plataforma objetivo: memoria compartida
- Motivación: evitar el manejo explícito de threads
  - Ejemplo de multiplicación de matrices: en el algoritmo secuencial se conoce el código (ciclo de iteraciones) a ejecutar en paralelo, la conversión del programa es casi mecánica
  - OpenMP incluye directivas para implementar de modo simple este tipo de paralelismo
- Permite implementar paralelismo implícito y explícito.
- El número de threads/tareas no es fijado en tiempo pre-ejecución.
- Sintaxis basada en el uso de directivas (indicadas al compilador)

OpenMP: sintaxis

- Sintaxis basada en el uso de directivas (indicadas al compilador)
- En lenguaje C/C++
  - Directivas de compilación (pragmas)

```
#pragma omp nombre_de_directiva [cláusula [[,] clausula …]
```

  - La API provee servicios en tiempo de ejecución
  - Los prototipos de las funciones y las definiciones de los tipos de datos están disponibles en omp.h
  - Las directivas son ignoradas por el compilador en caso de no reconocer OpenMP.
- En lenguaje FORTRAN
  - Comentarios específicamente formateados

```
sentinela nombre_de_directiva [cláusula [[,] clausula …]
```

  - Sentinela es !OMP, COMP o *$OMP en formato fijo, o !$OMP en formato libre
  - La API provee servicios en tiempo de ejecución, las definiciones de las funciones están disponibles en el módulo omp_lib

OpenMP: compilación

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- gcc 4.2 y posteriores incluyen soporte para OpenMP 3.0
  - Modelos de paralelismo basado en threads y tareas

  En C: debe utilizarse la opción `-fopenmp`

  ```
  $ gcc -o omp_helloc -fopenmp omp_hello.c
  ```

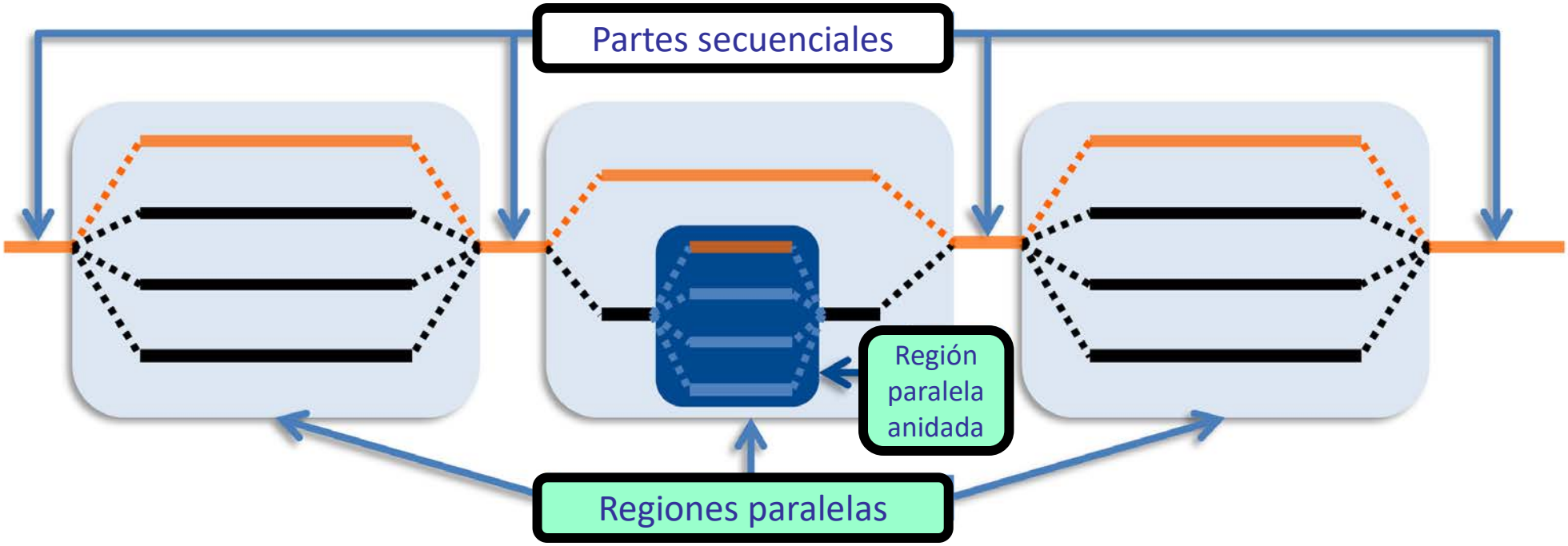
  En FORTRAN: debe utilizarse la opción `-fopenmp`

  ```
  $ gfortran -o omp_hellof -fopenmp omp_hello.f
  ```

- gcc 4.9 y superiores proven soporte para OpenMP 4.0
  - Incluye soporte para aceleradores y GPU

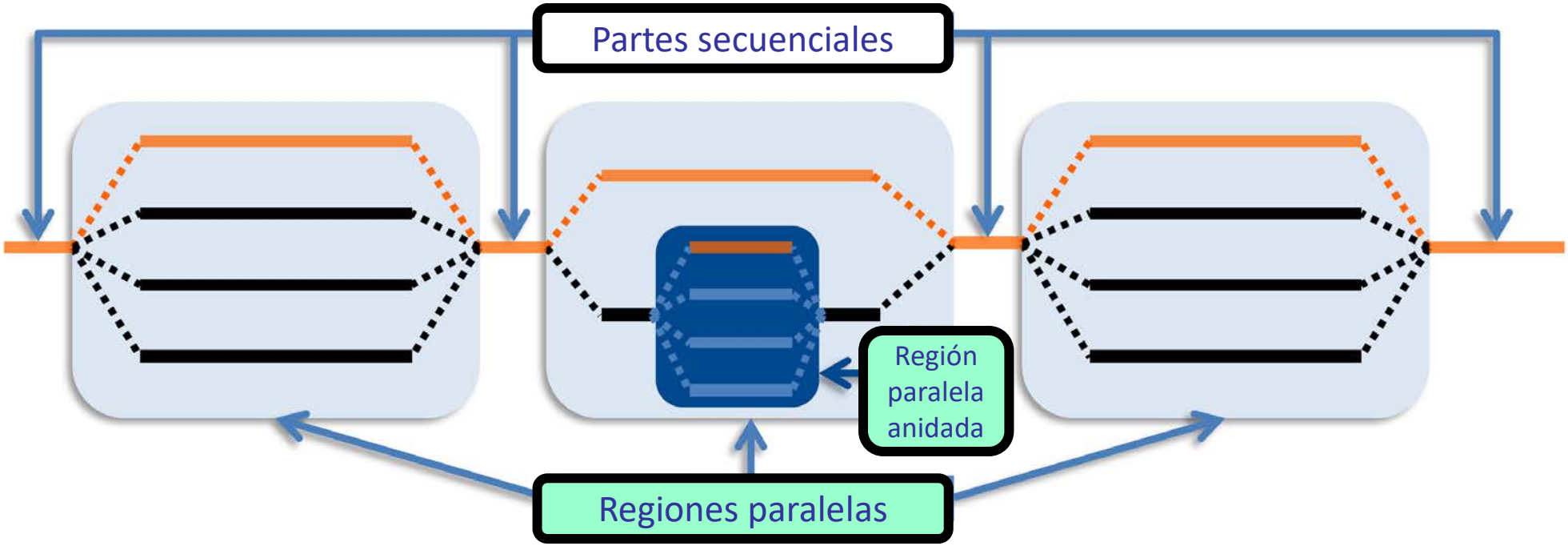
Modelo de ejecución

- Basado en el modelo fork join
  - Un pool de threads que cooperan para resolver un problema
  - Un thread distinguido (master) se encarga de la coordinación del trabajo.
  - Usualmente se ejecuta un thread por recurso de cómputo (pero pueden ejecutarse más o menos)
  - Diferentes threads pueden seguir diferentes flujos de control



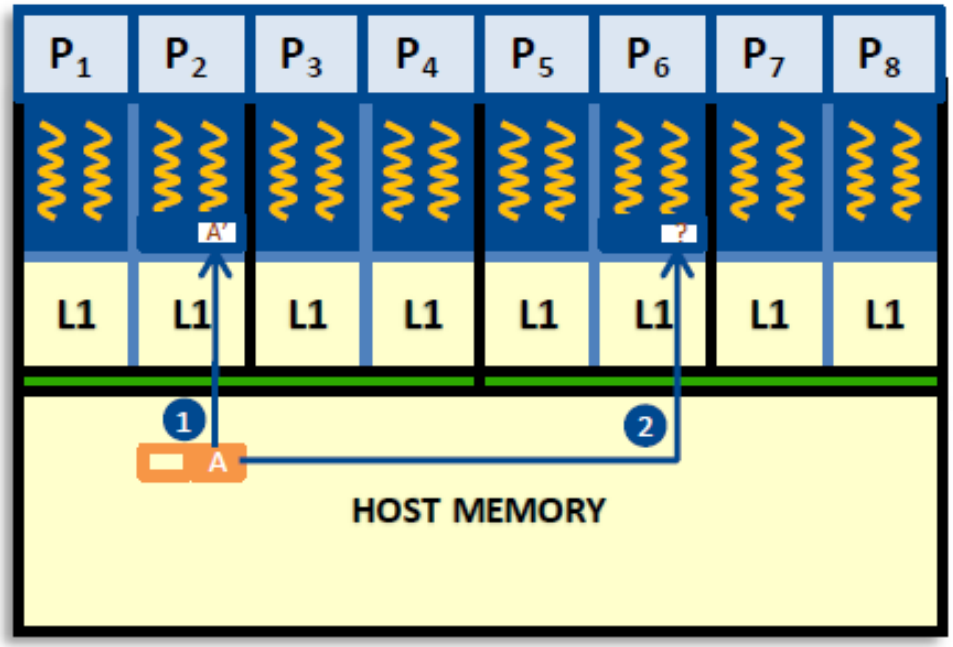
Modelo de ejecución

- Modelo fork join
  - Todos los programas OpenMP comienzan con un único master thread.
  - El master thread ejecuta secuencialmente hasta encontrar una region paralela, donde crea un equipo/pool de threads (fork).
  - Los threads completan la region paralela, se sincronizan y terminan, y solo permanece el master thread que ejecuta secuencialmente (join).



Modelo de memoria

- Modelo de memoria de consistencia laxa
  - Diferentes threads pueden ver diferentes valores para la misma variable compartida (los threads manejan información no consistente)
  - La consistencia solamente se garantiza en ciertos puntos especiales
  - Puntos explícitos (indicados mediante la directiva flush)
  - Puntos implícitos (indicados mediante otras directivas)



OpenMP: bloques estructurados

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- La mayoría de las directivas se aplican a bloques estructurados

```
#pragma omp directiva [clause[[,] clause]...]
{bloque estructurado}
```

- Bloque de una o más sentencias, con un único punto de entrada, un único punto de salida y una barrera implícita al final del bloque
  - Saltar fuera del bloque no está permitido
  - Finalizar el programa (abort/exit) si está permitido

```
#pragma omp directive-name clause1(...) clause2(...)
{
  conjunto de instrucciones sin saltos externos
}
```

```
#pragma omp directive-name clause1(...) clause2(...)
for (int i = 0; i < SIZE; i++) {
   A [ i ] = 0;
}
```

```
#pragma omp directive-name clause1(...) clause2(...)
{
  conjunto de instrucciones
  if ( expr ) exit(0);
}
```

```
#pragma omp directive-name clause1(...) clause2(...)
for (int i = 0; i < SIZE; i++) {
   A [ i ] = 0;
   if ( i == INDEX ) break;
}
```

OpenMP: número de threads

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- El número máximo de threads está controlado por tres vías:

1. Una variable interna de nombre nthreads-var, sobre la que se definen las funciones de set/get:

```
void omp_set_num_threads(int value); // seguido de una región paralela
int omp_get_num_threads(void);       // retorna el número de threads actual
int omp_get_max_threads(void);       // retorna el máximo número de threads
```

2. La variable de entorno OMP_NUM_THREADS

```
$ export OMP_NUM_THREADS=<valor>
$ ./myProgram
```

3. La cláusula num_threads (expresión), que define el número máximo de threads y sobreescribe el valor de nthreads-var

OpenMP: número de threads

- Ejemplo: crear una region paralela con tres threads

1. Con la cláusula num_threads

```
#include <stdio.h>
void main (void) {
    #pragma omp parallel num_threads(3){
            printf("Hello world\n”);
    }
}
```

2. Con la variable interna nthreads-var

```
#include <stdio.h>
#include <omp.h>
void main (void) {
    omp_set_num_threads(3);
    #pragma omp parallel {
            printf("Hello world\n”);
    }
}
```

```
$ gcc -fopenmp myHello.c -o myHello
$ ./myHello
Hello world!
Hello world!
Hello world!
```

OpenMP: número de threads

- Ejemplo: crear una región paralela con tres threads

3. Con la variable de entorno (la solución más flexible)

```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel{
            printf ("Hello world … \n”);
    }
    #pragma omp parallel
            printf ("...and godbye \n”);
    }
}
```

OpenMP: número de threads

```
$ gcc -fopenmp myHello.c -o myHello
$ OMP_NUM_THREADS=2 ./myHello
Hello world...
Hello world...
...and goodbye!
...and goodbye!
```

← Con dos threads

Con tres threads →

```
$ OMP_NUM_THREADS=3 ./myHello
Hello world...
Hello world...
Hello world...
...and goodbye!
...and goodbye!
...and goodbye!
```

OpenMP: replicar trabajo

- Ejecución en paralelo de dos bloques de código

```
#include stdio.h
void main (void){
    do_work_1();
    do_work_2();
}
```

- Simplemente se pueden incluir en una region paralela

```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel num_threads (2){
            do_work_1();
            do_work_2();
    }
}
```

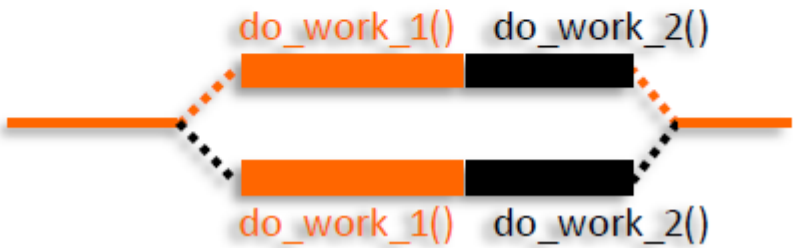
OpenMP: replicar trabajo

```
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```

← Secuencial

Paralelo con dos threads →

```
$ time ./myProgram
real    0m4.104s
user    0m0.000s
sys     0m0.000s
```



OpenMP: identificar threads

- Dentro de una region paralela, cada thread tiene su propio identificador
  - En el rango [0, …, N-1], siendo N el número de threads en ejecución
  - La función para obtener el identificador es int omp_get_thread_num(void);
  - La función retorna 0 si se invoca fuera de una región paralela.
- Ejemplo

```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel num_threads (4){
            int id = omp_get_thread_num();
            printf("Hello world! I am the thread %d\n", id);
    }
}
```

OpenMP: identificar threads

```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel num_threads (4){
            int id = omp_get_thread_num();
            printf("Hello world! I am the thread %d\n", id);
    }
}
```

```
$ ./myThreadId
Hello world! I am the thread 2.
Hello world! I am the thread 1.
Hello world! I am the thread 0.
Hello world! I am the thread 3.
```

OpenMP: distribuir trabajo

- Cuando dos bloques de código pueden ejecutar en paralelo, se puede utilizar el identificador de thread para distribuir el trabajo

```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel num_threads (2){
            int id = omp_get_thread_num();
            if (id == 0){
                    do_work_1();
            }
            if (id == 1){
                    do_work_2();
            }
    }
}
```

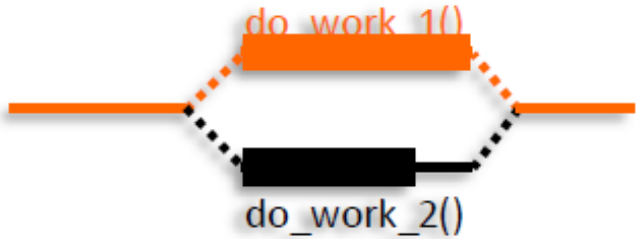
OpenMP: distribuir trabajo

```
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```

← Secuencial

Paralelo con dos threads →

```
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



OpenMP: distribuir trabajo

- El identificador de thread debe ser utilizado con cuidado
  - No es una buena idea confiar en el número de threads
  - Existen mecanismos más seguros para compartir trabajo

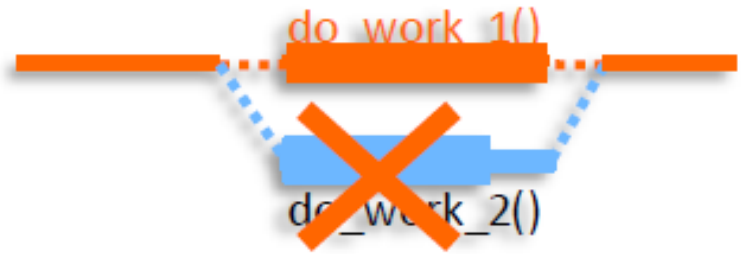
```
#include stdio.h
#include omp.h
void main (void){
    #pragma omp parallel {
            int id = omp_get_thread_num();
            if (id == 0){
                    do_work_1();
            }
            if (id == 1){
                    do_work_2();
            }
    }
}
```

Es correcto el código ?

OpenMP: distribuir trabajo

- El código funciona correctamente solo cuando se utilizan dos threads

```
$ export OMP_NUM_THREADS=1
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



Es correcto el código ?

OpenMP: distribuir trabajo

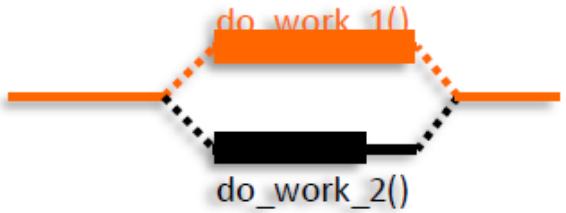
- Se podría corregir la sección paralela del código?

```
#pragma omp parallel {
    int id = omp_get_thread_num();
    if (id == 0){
        do_work_1();
    }
    if (id == 1 || omp_get_num_threads() < 2){
        do_work_2();
    }
}
```

No es eficiente para más de dos threads

```
$ export OMP_NUM_THREADS=1
$ time ./myProgram
real    0m4.003s
user    0m0.000s
sys     0m0.000s
```


```
$ export OMP_NUM_THREADS=2
$ time ./myProgram
real    0m2.604s
user    0m0.000s
sys     0m0.000s
```



Distribuir iteraciones en un loop

- Objetivo: loop independiente

```
double A[SIZE];
void main(void){
    for(int i = 0; i < SIZE; i++){
            A[i] = 0;
    }
}
```

- Se debe garantizar la independencia entre las iteraciones
- Una solución: calcular una cota inferior y una cota superior de procesamiento para cada thread
  - Usando la dimension del problema, el número de threads y el identificador de threads
  - Permiten aplicar una descomposición de dominio sobre el espacio de iteraciones.

Distribuir iteraciones en un loop

- Considerando una cota inferior y una cota superior de procesamiento para cada thread

```
double A[SIZE];
void main(void){
    #pragma omp parallel {
        int id = omp_get_thread_num();
        int nt = omp_get_num_threads();
        int lb = id*(SIZE/nt);
        int ub = (id+1)*(SIZE/nt);
        if (id == nt-1) {
            ub += (SIZE % nt);
        }
        for(int i = lb; i < ub; i++){
            A[i] = 0;
        }
    }
}
```

El último thread realiza más trabajo !

Distribuir iteraciones en un loop

- Considerando una cota inferior y una cota superior de procesamiento para cada thread

```
double A[SIZE];
void main(void){
    #pragma omp parallel {
        int id = omp_get_thread_num();
        int nt = omp_get_num_threads();
        int lb = id*(SIZE/nt);
        int ub = (id+1)*(SIZE/nt)+((id == nt-1) ? (SIZE % nt): 0);
        for(int i = lb; i < ub; i++){
            A[i] = 0;
        }
    }
}
```

OpenMP: constructor if

- Evita la creación de regiones paralelas si no corresponde (o no es conveniente)

```
#pragma omp parallel if (expresión)
    {bloque estructurado}
```

- Ejecuta en paralelo si se cumple la condición (sinó, solo se usa un thread).
- Siempre se crea un pool de threads y un entorno de datos

```
double A[SIZE];
void main(void){
    #pragma omp parallel if (SIZE>256){
        int id = omp_get_thread_num();
        int nt = omp_get_num_threads();
        int lb = id*(SIZE/nt);
        int ub = (id+1)*(SIZE/nt)+((id == nt-1) ? (SIZE % nt): 0);
        for(int i = lb; i < ub; i++){
            A[i] = 0;
        }
    }
}
```

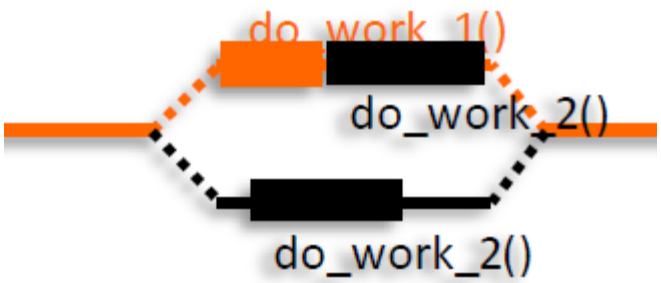
OpenMP: constructor master

- Solo el thread master ejecuta un bloque estructurado

```
#pragma omp master
    {bloque estructurado}
```

- Los threads que no son master no ejecutan el bloque
- No hay barrera implícita al inicio
- No hay barrera implícita al final

```
#pragma omp parallel num_threads(2){
int id = omp_get_thread_num();
#pragma omp master
do_work_1();    // ejecuta solo en un thread (el master)
do_work_2(id); // ejecuta en los N threads
```



OpenMP: constructor for

- Divide las iteraciones de un ciclo (for) entre los threads del pool

```
#pragma omp for [lista de cláusulas]
    {bloque estructurado (loop)}
```

- Las cláusulas pueden ser private(lista), firstprivate(lista), reduction(operador:lista) y otros (que veremos más adelante)

```
#pragma omp for [lista de cláusulas]
    for(expresión inicial; condición; expresión de incremento)
```

- Las iteraciones del ciclo deben ser independientes (la responsabilidad se delega al programador).
- Las variables de la iteración pueden ser enteros, punteros e iteradores (C++).

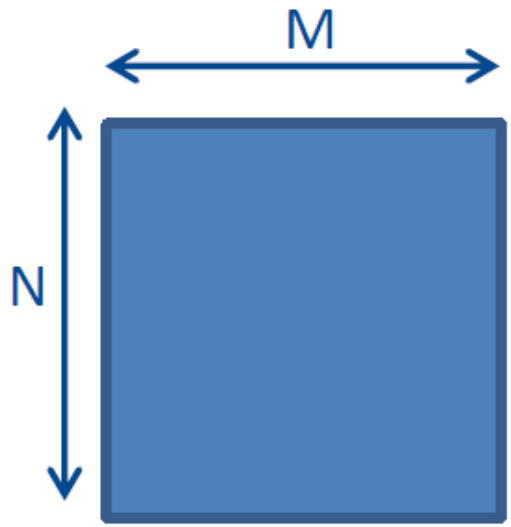
OpenMP: constructor parallel for

- Combinación del constructor parallel con el constructor for

```
#pragma omp parallel for [lista de cláusulas]
    {bloque estructrurado (loop)}
```

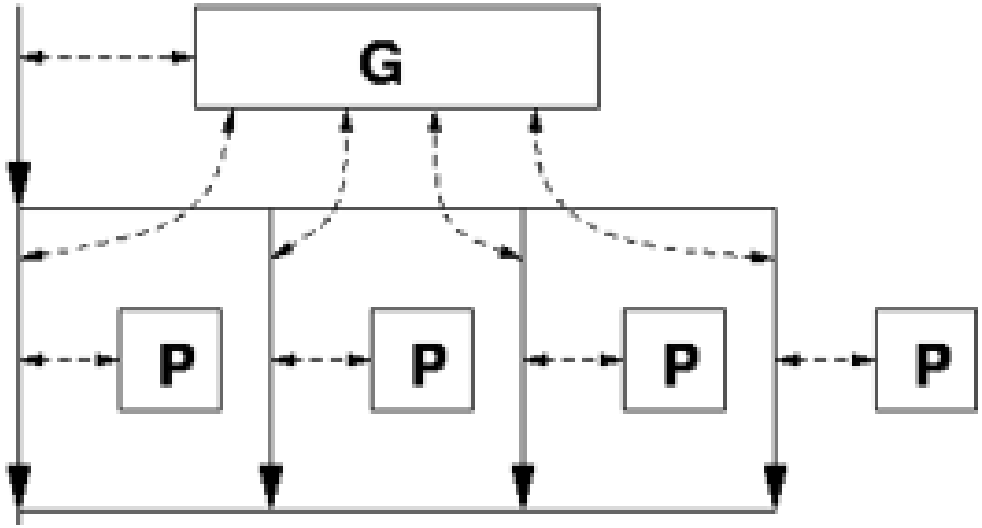
- Ejemplo: inicialización de una matriz

```
void foo(int *m, int N, int M){
int i, j;
#pragma omp parallel for private(j)
    for (i=0;i<N;i++){
            for (j=0;j<M;j++);
                    m[i*N+j] = 0;
    }
```



OpenMP: modelo de datos

- Variables globales (compartidas) y locales (privadas)
  - Las variables globales están disponibles en un espacio de direccionamiento global y son accesibles por todos los threads
  - Las variables locales están disponibles en un espacio de direccionamiento privado de cada thread y solo son accesibles por ese thread.
  - Varias opciones, dependiendo de los valores iniciales y si los resultados se copian fuera de una region compartida.



G – espacio de direccionamiento global

P – variables privadas

OpenMP: alcance de variables

- Variables globales (compartidas) y locales (privadas)

```
#include <stdio.h>
#include <omp.h>
double PI = 3.14159265359;  // compartida
void main (void){
    int id = 0;             // privada
    #pragma omp parallel num_threads(4) shared(PI) private(id){
        id = omp_get_thread_num();
        printf ("Hello world! I am thread %d. I like %f\n”,id,PI);
    }
}
```

```
$ ./myProgram
Hello world! I am thread 2. I like 3.141593.
Hello world! I am thread 0. I like 3.141593.
Hello world! I am thread 1. I like 3.141593.
Hello world! I am thread 3. I like 3.141593.
```

OpenMP: privatizar variables

- Las variables dentro del constructor son nuevas variables
  - Tienen el mismo tipo que la variable original
  - En un constructor parallel, todos los threads tienen diferentes variables
  - Pueden ser accedidas sin necesidad de sincronizaciones
- Las cláusulas private (almacena) y firstprivate (almacena y copia)

```
#pragma omp parallel {private|firstprivate}(list)
{bloque estructurado}
```

  - Variables private tienen valor definido al iniciar el bloque
  - Firstprivate inicializa las variables con el valor de la variable original

```
double A;
#pragma omp parallel private(A){
        A =<expresión>;
        ...
}
printf("A = %f \n",A);
```

El código paralelo no afecta a A

```
double A = 3.14159265259;
#pragma omp parallel firstprivate(A){
        A = f(A);
        ...
}
printf("A = %f \n",A);
```

El código paralelo no afecta a A

OpenMP: crear copias de variables

- Creación de copias de variables globales por thread

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

  - Aplicable a variables globales o estáticas
  - El almacenamiento de threadprivate es persistente

```
char buffer[SIZE];
#pragma omp threadprivate(buffer)
void main(void){
        #pragma omp parallel{
                buffer = <expresión>;
                ...
        }
}
```

Cada thread pasa a tener su propia variable buffer (privada)

Crear copias de variables

- Con variable estática (static)

```
char * fun(void){
    static char buffer[SIZE];
    #pragma omp threadprivate(buffer)
    ...
    return(buffer)
}
void main(void){
    #pragma omp parallel{
        char *a = fun();
        ...
    }
}
```

  - fun() puede ser invocada simultáneamente por más de un thread
  - Cada thread pasa a tener su propia variable `a`/`buffer` (privada)
  - El valor de la variable persiste (no cambia por multiples invocaciones)

Compartir variables en una región paralela

- La variable es la misma dentro y fuera del constructor
  - Todos los threads ven la misma variable (la misma dirección de memoria)
  - Pero no necesariamente ven el mismo valor (problemas de consistencia)
  - Se necesitan sincronizaciones para la actualización correcta de la variable

```
#include <stdio.h>
double PI = 3.14159265359
void main(void){
int id = 0;
        #pragma omp parallel num_threads(4) shared(PI){
                PI = 3;
        }
        printf ("PI = %f\n", PI);
}
```

  - Todos los threads leen la misma variable
  - Luego de la region paralela, las modificaciones aún son visibles

```
$ ./myProgram
PI = 3.000000;
```

Modificar variables en una región paralela

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Modificar las variables a y b (compartidas) dentro de la region paralela

```
#include <stdio.h>
#include <assert.h>
#include <omp.h>
int a = 0, b = 0, ITERS = 100;
void main(void){
        int NT = 4;
        #pragma omp parallel num_threads(NT) shared(a,b,NT,ITERS){
                #pragma omp master
                a = NT*ITERS;
                for(int i=0;i<ITERS;i++){
                        b = b + 1;
                }
        }
        assert(a == NT*ITERS,"Value of a is incorrect !!!") // correcto
        assert(b == NT*ITERS,"Value of b is incorrect !!!") // incorrecto
}
```

Modificar variables en una región paralela

```
int a = 0, b = 0, ITERS = 100;
void main(void){
        int NT = 4;
        #pragma omp parallel num_threads(NT) shared(a,b,NT,ITERS){
                #pragma omp master
                a = NT*ITERS;
                for(int i=0;i<ITERS;i++){
                        b = b + 1;
                }
        }
        assert(a == NT*ITERS,"Value of a is incorrect !!!") // correcto
        assert(b == NT*ITERS,"Value of b is incorrect !!!") // incorrecto
}
```

- Las variables a, NT e ITERS no tienen problemas de sincronización
- La variable b puede generar resultados inconsistentes
- Si dos threads acceden al mismo tiempo a b = 5

Reg-1	Thread-1	b	Thread-2	Reg-2
r1=5	load b, r1	5		r1=X
r1=6	increment r1	5		r1=X
r1=6		5	load b, r1	r1=5
r1=6	store r1, b	6		r1=5
r1=6		6	increment r1	r1=6
r1=6		6	store r1, b	r1=6

El constructor critical

- Permite definir regiones de exclusión mutua

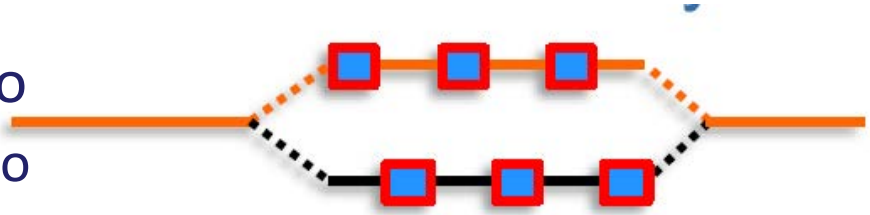
```
#pragma omp critical [(name)[hint(hint expression)]
    {bloque estructurado}
```

- Semántica del constructor critical:
  - Solo un thread puede ejecutar la region crítica en un instante de tiempo
  - Por defecto, todas las regiones críticas están sincronizadas todas con todas
  - Si se especifica un nombre, solo las regiones que comparten un nombre se sincronizan

El constructor critical

```
#include <assert.h>
#include <omp.h>
int a = 0, b = 0, ITERS = 100;
void main(void){
        int NT = 2;
        #pragma omp parallel num_threads(NT) shared(a,b,NT,ITERS){
                #pragma omp master
                a = NT*ITERS;
                for(int i=0;i<ITERS;i++){
                        #pragma omp critical
                        b = b + 1;
                }
        }
        assert(a == NT*ITERS,"Value of a is incorrect !!!") // correcto
        assert(b == NT*ITERS,"Value of b is incorrect !!!") // incorrecto
```

- El patrón de exclusion mutua es correcto
  - Pero se obtiene un desempeño muy bajo



El constructor critical

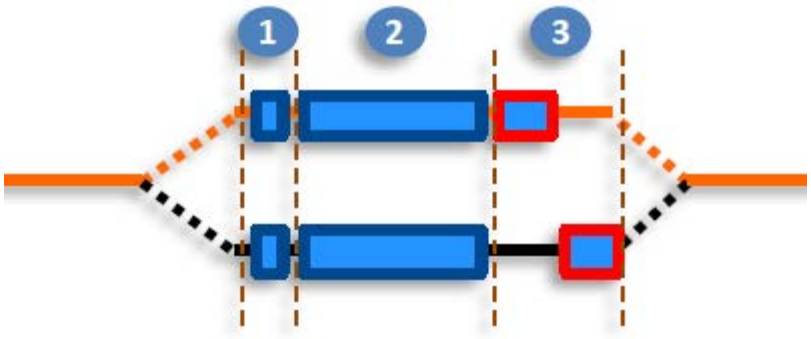
- Cómo mejorar el desempeño?

```
void main(void){
        int b = 0, NT = 2, ITERS = 100;
        omp_set_num_threads(NT);
        #pragma omp parallel shared(b,NT){
                for(int i=0;i<ITERS;i++){
                        b = b + 1;
                }
        }
        assert(b == NT*ITERS,“Valor de b incorrecto!")
}
```

```
        #pragma omp parallel shared(b,NT){
                int p_b = 0; (1)
                for(int i=0;i<ITERS;i++){
                        p_b = p_b +1; (2)
                }
                #pragma omp critical
                        b = b + p_b; (3)
        }
        assert(b == NT*ITERS,"Valor de b incorrecto!")
```

Solución:

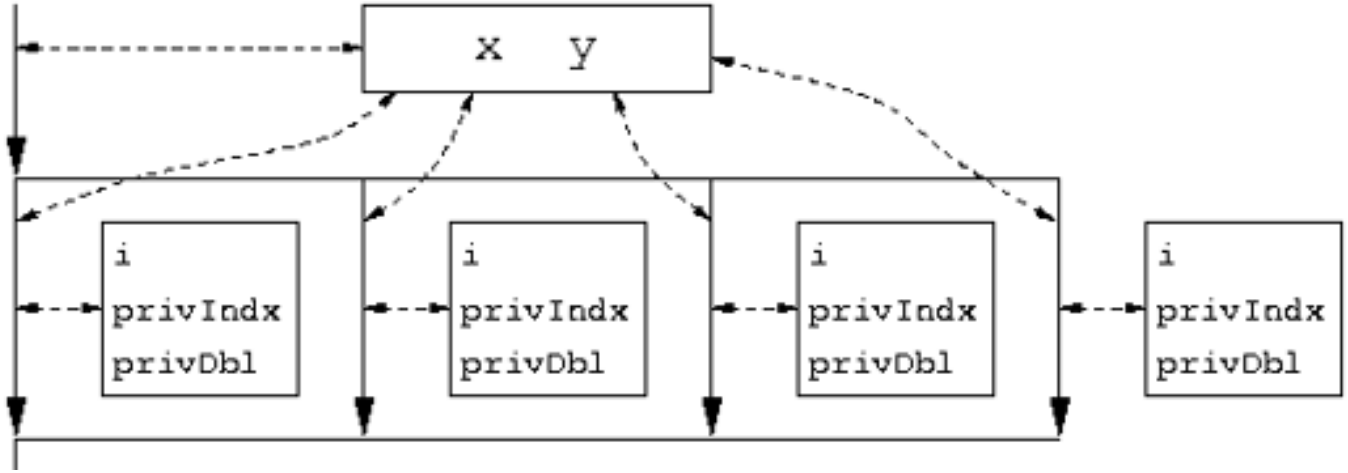
1. Se crea una copia por thread para acumular resultados parciales
2. Se usa la region crítica para acumular en la variable original



OpenMP: variables globales y privadas

- Un ejemplo más complejo

```
#pragma omp parallel for private(privIndx, privDbl)
for (i=0; i<arraySize; i++) {
      for (privIndx = 0; privIndx < 16; privIndx++) {
            privDbl = ((double) privIndx)/16;
            y[i] = sin(exp(cos(-exp(sin(x[i]))))) + cos(privDbl);
      }
}
```



La cláusula de reducción

- Aplica el patrón de reducción sobre resultados parciales

```
#pragma omp parallel reduction(operador:lista)
    {bloque estructurado}
```

- Puede aplicarse al caso de estudio previo

```
void main(void){
        int b = 0, NT = 2, ITERS = 100;
        omp_set_num_threads(NT);
        #pragma omp parallel reduction(+:b){
                for(int i=0;i<ITERS;i++){
                        b = b + 1;
                }
        }
        assert(b == NT*ITERS,“Valor de b incorrecto!")
}
```

  - El compilador crea una copia privada de la(s) variable(s) y la inicializa
  - Se garantiza que la variable compartida es apropiadamente actualizada
  - Operadores de reducción: +,-,*,|,&,^, min, max o definidos por el usuario

El constructor single

- Permite ejecutar secuencialmente una parte de una region paralela

```
#pragma omp single [cláusula[[,] clausula ... ]
    {bloque estructurado}
```

- Las cláusulas pueden ser: private(lista), firstprivate(lista), nowait, copyprivate(lista)

```
#include omp.h
void main (void){
    #pragma omp parallel {
            do_parallel_work_1();
            #pragma omp single
                printf("Hello world!\n");
            do_parallel_work_2();
    }
}
```

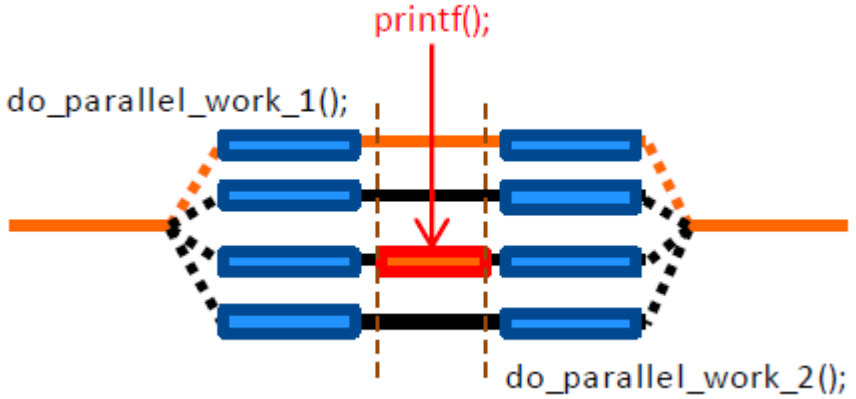
Solo escribe un "Hello world!\n"

- Solo un thread ejecuta el bloque estructurado
- Muy útil para operaciones de lectura/escritura

Barreras implícitas y nowait

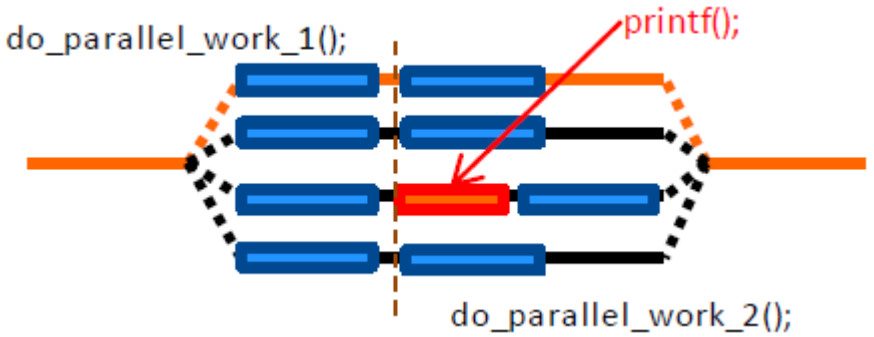
- El constructor parallel incorpora una barrera implícita al final del bloque

```
#pragma omp parallel {
    do_parallel_work_1();
    #pragma omp single
        printf("Hellow world!\n");
    do_parallel_work_2();
}
```



- La clausula nowait elimina la barrera al final del constructor

```
#pragma omp parallel {
    do_parallel_work_1();
    #pragma omp single nowait
        printf("Hellow world!\n");
    do_parallel_work_2();
}
```



Broadcasting con copyprivate

- Copiar variables desde el constructor single

```
#pragma omp single copyprivate(variables)
    {bloque estructurado}
```

- Permite el broadcast de valores de variables a otros threads
- Solo aplica a variables private, firstprivate o threadprivate
- El broadcast se realiza luego de ejecutar el bloque estructurado, pero antes de que los threads abandonen la barrera al final del constructor

```
void main(void){
float x,y;
    #pragma omp parallel private(x,y){
    ...
    #pragma omp single copyprivate(x,y){
        scanf ("%f %f”,&x,&y);
    }
    ...
}
```

Single vs. master

- En ambos casos el bloque estructurado lo ejecuta un único thread

```
#pragma omp single
    {bloque estructurado}
```

```
#pragma omp master
    {bloque estructurado}
```

- single tiene mayor overhead (requiere sincronizaciones adicionales) …
  - Qué thread lo ejecuta y la barrera implícita al final
- … pero es más flexible (cualquier thread puede ejecutar el bloque)
- master tiene menor overhead …
  - Solo involucra una comparación (`if thread_id == 0`) y no tiene una barrera implícita al final
- … pero es más restrictiva (solo el master puede ejecutar el bloque)
- Regla práctica: si todos los threads alcanzan el bloque estructurado al mismo tiempo, usar master, en otro caso usar single

El constructor sections

- Permite definir un conjunto de bloques estructurados que se distribuyen entre los threads

```
#pragma omp sections [cláusula[[,] clausula ... ]{
    [#pragma omp section]
            {bloque estructurado}
    [#pragma omp section]
            {bloque estructurado}
```

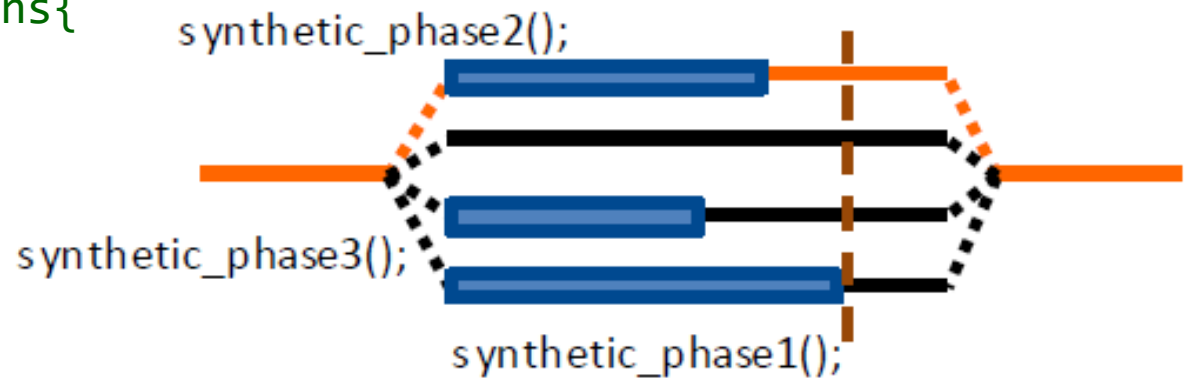
- Las cláusulas pueden ser: private(lista), firstprivate(lista), lastprivate(lista), nowait, reduction

El constructor sections

- Cada bloque estructurado se precede por una directiva section (salvo en el primer bloque, que es opcional)
- La asignación de bloques a threads la define la implementación de OpenMP
- sections puede combinarse con parallel

```
#pragma omp parallel sections [cláusula[[,] clausula ... ]
    {bloque estructurado: sections}
```

```
void main(void){
#pragma omp parallel sections{
    synthetic_phase1();
    #pragma omp section
    synthetic_phase2();
    #pragma omp section
    synthetic_phase3();
}
```



Privatizar variables dentro de un constructor

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Cada variable dentro del constructor es una nueva variable
  - Tiene el mismo tipo de la variable original
  - Cada thread tiene su propia variable, que puede acceder sin necesidad de sincronizaciones.
- La clausula lastprivate se usa con sections/parallel sections para determinar el valor final de una variable

```
#pragma omp sections lastprivate(lista)
    {bloque estructurado: sections}
```

- lastprivate declara un entorno privado de cada thread y actualiza la variable original con el ultimo valor que tuvo la variable privada
  - Considera el valor de la variable en la última sección léxica definida

Privatizar variables con lastprivate

```
void main(void){
int v=0;
#pragma omp parallel sections lastprivate(v){
    #pragma omp section{v=1; synthetic_phase1(v);}
    #pragma omp section{v=2; synthetic_phase1(v);}
    #pragma omp section{v=3; synthetic_phase1(v);}
    printf("v=%d\n");
}
```

- Las variables v en cada section son privadas
- La última sección léxica en declararse es la que actualiza el valor de la variable declarada con lastprivate
  - No importa el orden de ejecución