Perfecto. Voy a transcribir todo el contenido del seminario práctico 3 sobre OpenMP de forma literal y estructurada en texto plano para que puedas estudiarlo cómodamente. Te aviso en cuanto esté listo.

# Seminario 3. Herramientas de programación paralela III: Interacción con el entorno en OpenMP y evaluación de prestaciones

**Arquitectura de Computadores** 2º curso / 2º cuatr. – Grado en Ingeniería Informática

Material elaborado por Mancia Anguita

## Interacción con el entorno (v3.0 en gris)

- **Objetivos:**
    - Consultar: obtener información (p. ej. nº de threads o tipo de planificación de tareas)
    - Modificar: influir en la ejecución (p. ej. fijar nº de threads o fijar el tipo de planificación de tareas)
- **Relacionado con el entorno de ejecución:**
    - Variables de control internas
        - V2.5: nthreads-var, dyn-var, nest-var, run-sched-var, def-sched-var
        - V3.0: thread-limit-var…
    - Variables de entorno (ámbito: los códigos que se ejecuten a partir de su modificación)
        - V2.5: OMP_NUM_THREADS, OMP_DYNAMIC, OMP_NESTED, OMP_SCHEDULE
        - V3.0: OMP_THREAD_LIMIT, …
    - Funciones del entorno de ejecución (ámbito: el código que las usa)
        - V2.5: omp_get_dynamic(), omp_set_dynamic(), omp_get_max_threads(), omp_set_num_threads(), omp_get_nested(), omp_set_nested(), omp_get_thread_num(), omp_get_num_threads(), omp_get_num_procs(), omp_in_parallel()
        - V3.0: omp_get_thread_limit(), omp_get_schedule(kind, modifier), omp_set_schedule(kind, modifier) …
    - Cláusulas (no modifican variables de control) (ámbito: directiva que las usa)
        - V2.5: if, schedule, num_threads
        - V3.0: prioridad

## Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interaccionar con el entorno

- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

## Variables de control

### Variables de control internas que afectan a una región *parallel*

| **Variable de control** | **Ámbito** | **Valor (valor inicial)** | **¿Qué controla?** | **Consultar/Modificar** |
|---|---|---|---|---|
| **dyn-var** | entorno de datos | true/false *(depende de la implementación)* | Ajuste dinámico del nº de threads | sí (f) / sí (ve, f) |
| **nthreads-var** | entorno de datos | número *(depende de la implementación)* | Nº de threads en la siguiente ejecución paralela | sí (f) / sí (ve, f) |
| **thread-limit-var** *(V3.0)* | entorno de datos | número *(depende de la implementación)* | Máximo nº de threads para todo el programa | sí (f) / sí (ve, –) |

*V3.0 en gris f: función, ve: variable de entorno*

### Variables de control internas que afectan a regiones DO/loop

| **Variable de control** | **Ámbito** | **Valor (valor inicial)** | **¿Qué controla?** | **Consultar/Modificar** |
|---|---|---|---|---|
| **run-sched-var** | entorno de datos | (kind[,chunk]) *(depende de la implementación)* | Planificación de bucles para *runtime* | sí (f) / sí (ve, f) |
| **def-sched-var** *(V3.0)* | dispositivo | (kind[,chunk]) *(depende de la implementación)* | Planificación de bucles por defecto (ámbito: todo el programa) | no / no |

*V3.0 en gris f: función, ve: variable de entorno*

## Variables de entorno

| **Variable de control** | **Variable de entorno** | **Ejemplos de modificación (shell bash/ksh)** |
|---|---|---|
| **dyn-var** | OMP_DYNAMIC | `export OMP_DYNAMIC=FALSE`<br>`export OMP_DYNAMIC=TRUE` |
| **nthreads-var** | OMP_NUM_THREADS | `export OMP_NUM_THREADS=8` |
| **thread-limit-var** | OMP_THREAD_LIMIT | `export OMP_THREAD_LIMIT=8` |

| Variable de control | Variable de entorno | Ejemplos de modificación (shell bash/ksh) |
|---|---|---|
| **run-sched-var** | OMP_SCHEDULE | `export OMP_SCHEDULE="static,4"`<br>`export OMP_SCHEDULE="nonmonotonic:static,4"`<br>`export OMP_SCHEDULE="dynamic"`<br>`export OMP_SCHEDULE="monotonic:dynamic,4"` |
| **def-sched-var** | *(No existe)* | *(No aplica)* |

*V3.0 en gris*

## Funciones del entorno de ejecución

| Variable de control | Rutina para consultar | Rutina para modificar |
|---|---|---|
| **dyn-var** | omp_get_dynamic() | omp_set_dynamic() |
| **nthreads-var** | omp_get_max_threads() | omp_set_num_threads() |
| **thread-limit-var** *(V3.0)* | omp_get_thread_limit() | *(no)* |
| **nest-var** | omp_get_nested() | omp_set_nested() |
| **run-sched-var** | omp_get_schedule(&kind, &modifier) | omp_set_schedule(kind, modifier) |
| **def-sched-var** | *(no)* | *(no)* |

*V3.0 en gris*

```
typedef enum omp_sched_t {
    // schedule kinds
    omp_sched_static = 0x1,
    omp_sched_dynamic = 0x2,
    omp_sched_guided = 0x3,
    omp_sched_auto = 0x4,
    // schedule modifier
    omp_sched_monotonic = 0x80000000u
} omp_sched_t;

omp_get_schedule(&kind, &modifier);
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

### Otras rutinas del entorno de ejecución (v2.5)

- **omp_get_thread_num()** – Devuelve al *thread* su identificador dentro del grupo de *threads*.
- **omp_get_num_threads()** – Obtiene el nº de *threads* que se están usando en una región paralela. *(Devuelve 1 en código secuencial.)*
- **omp_get_num_procs()** – Devuelve el nº de procesadores disponibles para el programa en el momento de la ejecución.
- **omp_in_parallel()** – Devuelve true si se llama a la rutina dentro de una región paralela activa (puede estar dentro de varios `parallel`, basta con que uno esté activo) y false en caso contrario.

## Cláusulas para interaccionar con el entorno

| Cláusula (tipo) | parallel | for/DO | sections | single |
|---|---|---|---|---|
| **if** (control nº threads) | X | X | X | |
| **num_threads** (control nº threads) | X | X | X | |
| **shared** (control ámbito variables) | X | X | X | X |
| **private** (control ámbito variables) | X | X | X | X |
| **lastprivate** (control ámbito variables) | | X | X | |
| **firstprivate** (control ámbito variables) | X | X | X | X |
| **default** (control ámbito variables) (1) | X | X | X | |
| **reduction** (control ámbito variables) | X | X | X | |
| **copyin** (copia de valores) | X | | | |
| **copyprivate** (copia de valores) | | | | X |
| **schedule** (planificación iteraciones) (1) | | X | | |
| **ordered** (planificación iteraciones) (1) | | X | | |
| **nowait** (no espera) | | X | X | X |

*Nota:* (1) Cláusulas que interaccionan con el entorno.

### Cláusula **if**

- **Sintaxis:** `if(scalar-exp)` (C/C++)
- No hay ejecución paralela si no se cumple la condición.
- **Precaución:** Sólo usable en construcciones *parallel*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char **argv)
{
    int i, n = 20, tid;
    int a[n], suma = 0, sumalocal;
    if(argc < 2) {
        fprintf(stderr,"[ERROR] - Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]);
    if (n > 20) n = 20;
    for (i = 0; i < n; i++) {
        a[i] = i;
    }
    #pragma omp parallel if(n > 4) default(none) private(sumalocal, tid) shared(a,
```

```
suma, n)
    {
        sumalocal = 0;
        tid = omp_get_thread_num();
        #pragma omp for private(i) schedule(static) nowait
        for (i = 0; i < n; i++) {
            sumalocal += a[i];
            printf(" thread %d suma de a[%d]=%d sumalocal=%d \n", tid, i, a[i],
sumalocal);
        }
        #pragma omp atomic
        suma += sumalocal;
        #pragma omp barrier
        #pragma omp master
        printf("thread master = %d imprime suma = %d\n", tid, suma);
    }
}
```

- En la primera ejecución, sólo trabaja el *thread* 0 porque no hay más de 4 iteraciones. *(La región parallel no se activa si la condición del if no se cumple.)*

*(Cláusula if – Salida omitida: se mostraron ejemplos de salida con uno o más threads en clase.)*

## Cláusula **schedule**

- **Sintaxis:** `schedule([modifier:]kind[,chunk])`
- **kind:** forma de asignación
  - *static* (monotonic por defecto)
  - *dynamic*
  - *guided*
  - *auto*
  - *runtime*
- **chunk:** granularidad de la distribución
- **modifier:**
  - *monotonic* – si un hilo ejecutó la iteración i, ejecutará después iteraciones mayores que i.
  - *nonmonotonic* – no hay restricciones en el orden de asignación de los *chunks*.
- **Precauciones:**
  - Sólo aplicable a bucles (`for`/`DO`).
  - Por defecto se utiliza tipo *static* (distribución en tiempo de compilación) en la mayor parte de las implementaciones.
  - Es mejor no asumir una granularidad de distribución por defecto.

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

```c
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char **argv) {
    int i, n = 7, chunk, a[n], suma = 0;
    if(argc < 2) {
        fprintf(stderr,"\nFalta chunk \n");
        exit(-1);
    }
    chunk = atoi(argv[1]);
    for (i = 0; i < n; i++)
        a[i] = i;
    #pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static,
chunk)
    for (i = 0; i < n; i++) {
        suma = suma + a[i];
        printf(" thread %d suma a[%d] suma=%d \n", omp_get_thread_num(), i, a[i],
suma);
    }
    printf("Fuera de 'parallel for' suma=%d\n", suma);
}
```

**static:**

- Uso: `schedule(static, chunk)`
- Las iteraciones se dividen en unidades de *chunk* iteraciones.
- Las unidades se asignan en *round-robin* (cíclicamente entre los *threads*).
- El parámetro de entrada *chunk* indica el nº de iteraciones por unidad de distribución.
- Si se usa `schedule(static)` (sin especificar *chunk*), se asigna un único *chunk* a cada *thread (comportamiento usual por defecto)*.

**dynamic:**

- *Kind* = dynamic (distribución en tiempo de ejecución).
- Apropiado si se desconoce el tiempo de ejecución de las iteraciones.
- La unidad de distribución tiene *chunk* iteraciones (si no se especifica *chunk*, se usa unidad de 1 iteración).
- Las unidades se asignan en tiempo de ejecución: los *threads* más rápidos ejecutan más unidades.
- Parámetros de entrada: nº de iteraciones y tamaño del *chunk*.

**guided:**

- *Kind* = guided (distribución en tiempo de ejecución).
- Apropiado si se desconoce el tiempo de ejecución o el número de iteraciones.
- Comienza con un bloque de iteraciones grande.
- El tamaño de los bloques va menguando: en cada asignación es el nº de iteraciones que quedan por ejecutar dividido por el nº de *threads*, no más pequeño que *chunk* (excepto el último bloque).

- **Precaución:** Existe una sobrecarga extra, pero menor que con *dynamic* para el mismo tamaño de *chunk*.

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char **argv) {
    int i, n = 20, chunk, a[n], suma = 0;
    if(argc < 3) {
        fprintf(stderr,"\nFalta iteraciones y/o chunk \n");
        exit(-1);
    }
    n = atoi(argv[1]);
    if (n > 20) n = 20;
    chunk = atoi(argv[2]);
    for (i = 0; i < n; i++)
        a[i] = i;
    #pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(guided,
chunk)
    for (i = 0; i < n; i++) {
        suma = suma + a[i];
        printf(" thread %d suma a[%d]=%d suma=%d \n", omp_get_thread_num(), i,
a[i], suma);
    }
    printf("Fuera de 'parallel for' suma=%d\n", suma);
}
```

**runtime:**

- *Kind* = runtime
- El tipo de distribución (*static*, *dynamic* o *guided*) se fija en tiempo de ejecución.
- El tipo de distribución efectiva depende del valor de la variable de control **run-sched-var** (p.ej. mediante `OMP_SCHEDULE`).

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

int main(int argc, char **argv)
{
    register double width, x;
    double sum, lsum;
    register int intervals, i;
    int nproc, iproc;
    MPI_Status status;
```

```
    double t;
    if(argc < 2) {
        fprintf(stderr,"\nFalta iteraciones \n");
        exit(-1);
    }
    intervals = atoi(argv[1]);
    if (intervals < 1) intervals = 1;
    t = MPI_Wtime();
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    width = 1.0 / intervals;
    lsum = 0;
    for (i = iproc; i < intervals; i += nproc) {
        x = (i + 0.5) * width;
        lsum += 4.0 / (1.0 + x * x);
    }
    lsum *= width;
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Finalize();
    t = MPI_Wtime() - t;
    if (!iproc) {
        printf("Iteraciones:\t%d\t. PI:\t%26.24f\t. Procesos:\t%d\t.
Tiempo:\t%8.6f\n", intervals, sum, nproc, t);
    }
    return(0);
}
```

*(Extraído del documento con las especificaciones de OpenMP – https://www.openmp.org/)*

## Clasificación de las funciones de la biblioteca OpenMP

- **Funciones para acceder al entorno de ejecución de OpenMP** (ya vistas: funciones de consulta/modificación de número de *threads*, anidamiento, etc.)
- **Funciones para usar sincronización con cerrojos (locks):**
  - *(OpenMP V2.5)* `omp_init_lock()`, `omp_destroy_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`
  - *(OpenMP V3.0)* `omp_init_nest_lock()`, `omp_destroy_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`
- **Funciones para obtener el tiempo de ejecución:**
  - `omp_get_wtime()` – tiempo en segundos desde algún momento (inicio de ejecución)
  - `omp_get_wtick()` – precisión (resolución) del temporizador utilizado por `omp_get_wtime`

## Funciones para obtener el tiempo de ejecución

*(OpenMP proporciona las rutinas `omp_get_wtime()` y `omp_get_wtick()` para medir tiempos de ejecución. Se emplean en los siguientes ejemplos.)*

# Ejemplo: cálculo de PI en C

**C Pi (secuencial)** – Cálculo del número π mediante integración numérica simple.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char **argv)
{
    register double width, x;
    double sum = 0;
    register int intervals, i;
    struct timespec cgt1, cgt2;
    double ncgt;  // tiempo de ejecución

    if(argc < 2) {
        fprintf(stderr,"\nFalta nº intervalos\n");
        exit(-1);
    }
    intervals = atoi(argv[1]);
    if (intervals < 1) intervals = 1;

    clock_gettime(CLOCK_REALTIME, &cgt1);
    width = 1.0 / intervals;
    for (i = 0; i < intervals; i++) {
        x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }
    sum *= width;
    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) +
           (double) ((cgt2.tv_nsec - cgt1.tv_nsec) / 1.e+9);

    printf("Iteraciones:\t%d\t. PI:\t%26.24f\t. Threads:\t1\t. Tiempo:\t%8.6f\n",
intervals, sum, ncgt);
    return 0;
}
```

**Resultados (secuencial):**

Nodos usados en la ejecución del trabajo: 1 Machines: *shn13*

```
Iteraciones: 10000000 . PI: 3.14159265358973094350858 4 . Threads: 1 . Tiempo:
0.194065
Iteraciones: 40000000 . PI: 3.14159265358880057661394 9 . Threads: 1 . Tiempo:
0.561454
```

*(Se ejecutó el programa en un único proceso, sobre un nodo con un core.)*

# Ejemplo: cálculo de PI con OpenMP/C

**OpenMP/C Pi** – Versión paralela usando OpenMP.

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

int main(int argc, char **argv)
{
    register double width, x;
    double sum = 0;
    register int intervals, i;
    double t;  // tiempo de ejecución

    if(argc < 2) {
        fprintf(stderr,"\nFalta nº intervalos\n");
        exit(-1);
    }
    intervals = atoi(argv[1]);
    if (intervals < 1) intervals = 1;

    t = omp_get_wtime();
    #pragma omp parallel for firstprivate(sum) lastprivate(sum)
    for (i = 0; i < intervals; i++) {
        x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }
    t = omp_get_wtime() - t;

    printf("Iteraciones:\t%d\t. PI:\t%26.24f\t. Threads:\t%d\t. Tiempo:\t%8.6f\n",
intervals, sum, omp_get_num_threads(), t);
    return 0;
}
```

**Resultados (OpenMP con distintos nº de *threads*):**

Nodos usados en la ejecución del trabajo: 1 Machines: *shn13*

*(Para 10,000,000 de intervalos)*

```
Iteraciones: 10000000 . PI: 3.14159265358980377413900 0 . Threads: 8 . Tiempo:
0.016534
Iteraciones: 10000000 . PI: 3.14159265358966965919762 5 . Threads: 4 . Tiempo:
0.029227
Iteraciones: 10000000 . PI: 3.14159265358984962536619  . Threads: 2 . Tiempo:
0.055943
Iteraciones: 10000000 . PI: 3.14159265358973094350858 4 . Threads: 1 . Tiempo:
0.105901
```

*(Para 40,000,000 de intervalos)*

```
Iteraciones: 40000000 . PI: 3.14159265358975181570144 7 . Threads: 8 . Tiempo:
0.058191
Iteraciones: 40000000 . PI: 3.14159265358984818305998 5 . Threads: 4 . Tiempo:
0.109995
Iteraciones: 40000000 . PI: 3.14159265358998629480424 8 . Threads: 2 . Tiempo:
0.214541
Iteraciones: 40000000 . PI: 3.14159265358880057661394 9 . Threads: 1 . Tiempo:
0.424877
```

- Datos obtenidos en un computador con cuatro procesadores de cuatro cores cada uno.
- Tiempo en segundos.

## Ejemplo: cálculo de PI en MPI/C

**MPI/C Pi** – Versión paralela usando MPI (distribución de iteraciones entre procesos MPI).

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    register double width, x;
    double sum, lsum;
    register int intervals, i;
    int nproc, iproc;
    MPI_Status status;
    double t;

    if(argc < 2) {
        fprintf(stderr,"\nFalta nº intervalos\n");
        exit(-1);
    }
    intervals = atoi(argv[1]);
    if (intervals < 1) intervals = 1;
    t = MPI_Wtime();
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

    width = 1.0 / intervals;
    lsum = 0;
    for (i = iproc; i < intervals; i += nproc) {
        x = (i + 0.5) * width;
        lsum += 4.0 / (1.0 + x * x);
    }
    lsum *= width;
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    MPI_Finalize();
    t = MPI_Wtime() - t;
    if (!iproc) {
        printf("Iteraciones:\t%d\t. PI:\t%26.24f\t. Procesos:\t%d\t.
Tiempo:\t%8.6f\n", intervals, sum, nproc, t);
    }
    return 0;
}
```

**Resultados (MPI con distintos nº de procesos):**

Nodos usados en la ejecución del trabajo: 1 Machines: *shn09*

*(Para 10,000,000 de intervalos)*

```
Iteraciones: 10000000 . PI: 3.14159265358980688276346 9 . Procesos: 8 . Tiempo:
2.281467
Iteraciones: 10000000 . PI: 3.14159265358968609049839 0 . Procesos: 4 . Tiempo:
1.116629
Iteraciones: 10000000 . PI: 3.14159265358496253661 9  . Procesos: 2 . Tiempo:
0.119861
Iteraciones: 10000000 . PI: 3.14159265358973094350858 4 . Procesos: 1 . Tiempo:
0.156071
```

*(Para 40,000,000 de intervalos)*

```
Iteraciones: 40000000 . PI: 3.14159265358968742276601 9 . Procesos: 8 . Tiempo:
1.306842
Iteraciones: 40000000 . PI: 3.14159265359017414454001 5 . Procesos: 4 . Tiempo:
1.213710
Iteraciones: 40000000 . PI: 3.14159265359017414454001 5 . Procesos: 2 . Tiempo:
0.278467
Iteraciones: 40000000 . PI: 3.14159265358880057661394 9 . Procesos: 1 . Tiempo:
0.475955
```

- Datos obtenidos en un computador con cuatro procesadores de cuatro cores cada uno.
- Tiempo en segundos.