

2º curso / 2º cuatr.

Grados en
Ing. Informática

Arquitectura de Computadores

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

Material elaborado por
Mancia Anguita López y Julio Ortega Lopera



**UNIVERSIDAD
DE GRANADA**

ETSIIT
Escuela Técnica Superior
de Ingenierías Informática
y de Telecomunicación



ATC
Departamento de Arquitectura
y Tecnología de Computadores
UNIVERSIDAD DE GRANADA



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... *Nos permiten usar variables de control internas*

➤ 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

prioridad

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

Variables de control internas que afecta 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) <i>Nos deja o no uso dinámico de hebras</i>	Ajuste dinámico del nº de threads	sí(f) /sí(ve,f)
nthreads-var	entorno de datos	número (depende de la implementación)	threads en la siguiente ejecución paralela	sí(f) /sí(ve,f)
thread-limit-var	entorno de datos	número (depende de la implementación)	Máximo nº de threads para todo el programa	sí(f) /sí(ve,-)
nest-var		true/false (false)	Paralelismo anidado	sí(f) /sí(ve,f)

V3.0 en gris

f: función, ve: variable de entorno

nest-var: en desuso a partir de 5.0

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) <small>↓ ↓ variable de entorno funciones</small>
def-sched-var	dispositivo	(kind[,chunk]) (depende de la implementación)	Planificación de bucles por defecto. Ámbito el programa.	no /no

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

Variables de entorno

Variable de control	Variable de entorno	Ejemplos de modificación (shell bash/ksh)
dyn-var	OMP_DYNAMIC	export OMP_DYNAMIC=FALSE 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
nest-var	OMP_NESTED	export OMP_NESTED=TRUE export OMP_NESTED=FALSE
run-sched-var	OMP_SCHEDULE	export OMP_SCHEDULE="static,4" export OMP_SCHEDULE="nonmonotonic:static,4" export OMP_SCHEDULE="dynamic" export OMP_SCHEDULE="monotonic:dynamic,4"
def-sched-var		

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

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	omp_get_thread_limit()	
nest-var	omp_get_nested()	omp_set_nested()
run-sched-var	omp_get_schedule(&kind, &chunk) <i>Tipo de reparto</i> → <i>Granularidad</i>	omp_set_schedule(kind, chunk)
def-sched-var	no	no

void omp_set_schedule(omp_sched_t kind, int chunk_size);

Otras rutinas del entorno de ejecución de v2.5

- `omp_get_thread_num()`
 - Devuelve al thread su identificador dentro del grupo de thread
- `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()` → Se suele usar mucho en depuración
 - Devuelve `true` si se llama a la rutina dentro de una región `parallel` activa (puede estar dentro de varios `parallel`, basta que uno esté activo) y `false` en caso contrario.

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

Cláusula que interaccionan con el entorno

TIPO	Cláusula	Directivas					
		parallel	DO/for	sections	single	parallel DO/for	parallel sections
Control nº threads	if (1)	X				X	X
	num_threads (1)	X				X	X
Control ámbito de las variables	shared	X	X			X	X
	private	X	X	X	X	X	X
	lastprivate		X	X		X	X
	firstprivate	X	X	X	X	X	X
	default (1)	X				X	X
	reduction	X	X	X		X	X
Copia de valores	copyin	X				X	X
	copyprivate				X		
Planifica. iteraciones bucle	schedule (1)		X			X	
	ordered (1)		X			X	
No espera	nowait		X	X	X		

¿Cuántos threads se usan?

➤ Orden de precedencia para fijar el nº de threads:

- 
- El nº que resulte de evaluar la *cláusula* `if`
 - El nº que fija la *cláusula* `num_threads`
 - El nº que fija la *función* `omp_set_num_threads()`
 - El contenido de la *variable de entorno* `OMP_NUM_THREADS`
 - Fijado por defecto por la implementación: normalmente el nº de cores de un nodo

```
#pragma omp <directive> [<clause> <clause> ...]  
#pragma omp    parallel  num_threads(8) if(N>20)
```

↓
Si se cumple, se ejecuta
en paralelo con 8 hebras

Cláusula `if`



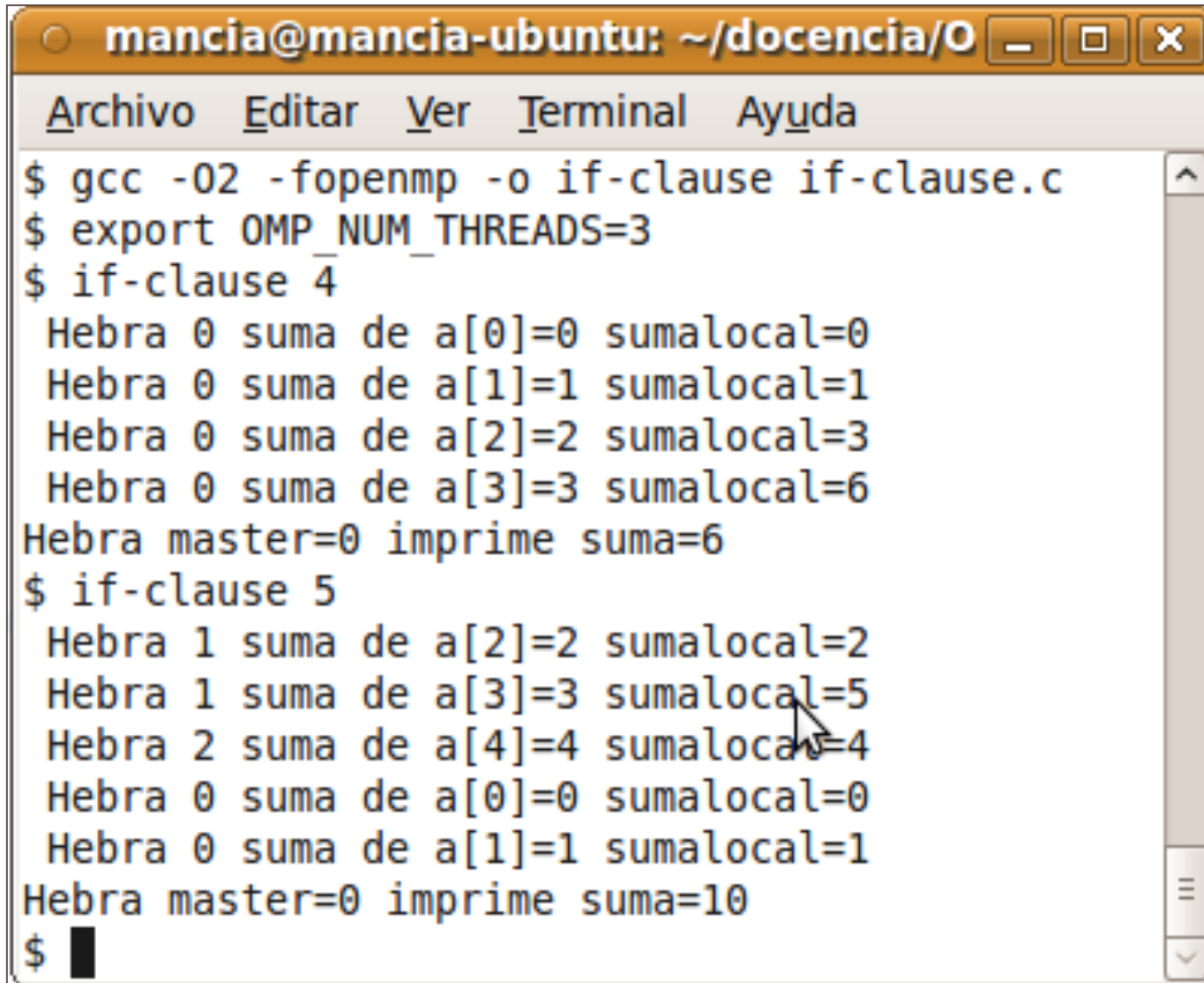
- Sintaxis:
 - `if (scalar-exp)`
(C/C++)
- No hay ejecución paralela si no se cumple la condición
- Precaución:
 - Sólo en construcciones `con parallel`

```
#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);
    }
}
```

Cláusula `if`. Salida



```
mancia@mancia-ubuntu: ~/docencia/O
Archivo  Editar  Ver  Terminal  Ayuda
$ gcc -O2 -fopenmp -o if-clause if-clause.c
$ export OMP_NUM_THREADS=3
$ if-clause 4
Hebra 0 suma de a[0]=0 sumalocal=0
Hebra 0 suma de a[1]=1 sumalocal=1
Hebra 0 suma de a[2]=2 sumalocal=3
Hebra 0 suma de a[3]=3 sumalocal=6
Hebra master=0 imprime suma=6
$ if-clause 5
Hebra 1 suma de a[2]=2 sumalocal=2
Hebra 1 suma de a[3]=3 sumalocal=5
Hebra 2 suma de a[4]=4 sumalocal=4
Hebra 0 suma de a[0]=0 sumalocal=0
Hebra 0 suma de a[1]=1 sumalocal=1
Hebra master=0 imprime suma=10
$
```

- En la primera ejecución sólo trabaja el thread 0 porque no hay más de 4 iteraciones

Cláusula schedule

schedule-clause.c



➤ Sintaxis:

- `schedule (kind[, chunk])`
 - Es un parámetro optativo
- `kind`: forma de asignación
 - static
 - dynamic
 - guided
 - auto
 - runtime
- `chunk`: granularidad de la distribución

➤ Precauciones:

- Sólo bucles
- Por defecto tipo `static` (distribución en tiempo de compilación) en la mayor parte de las implementaciones.
 - Se gasta menos tiempo en ejecución
- Mejor no asumir una granularidad de distribución por defecto → Depende de la versión de OpenMP

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

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, suma);
    }

    printf("Fuera de 'parallel for' suma=%d\n", suma);
}
```

V3.0 en gris

Cláusula `schedule.static`

➤ Usa

- `schedule(static, chunk)`
- Las iteraciones se dividen en unidades de chunk iteraciones.
- Las unidades se asignan en round-robin

➤ La entrada es el nº de iteraciones de la unidad de distribución (chunk)

➤ Usando

`schedule(static)` se asigna un único chunk a cada thread (comportamiento usual por defecto)

```
mancia@mancia-ubuntu: ~/docencia/OpenMP/lec
Archivo  Editar  Ver  Terminal  Ayuda

$ gcc -O2 -fopenmp -o schedule-clause schedule-clause.c
$ schedule-clause 1
Hebra 1 suma a[1] suma=1
Hebra 1 suma a[4] suma=5
Hebra 2 suma a[2] suma=2
Hebra 2 suma a[5] suma=7
Hebra 0 suma a[0] suma=0
Hebra 0 suma a[3] suma=3
Hebra 0 suma a[6] suma=9
Fuera de 'parallel for' suma=9
$ schedule-clause 2
Hebra 1 suma a[2] suma=2
Hebra 1 suma a[3] suma=5
Hebra 2 suma a[4] suma=4
Hebra 2 suma a[5] suma=9
Hebra 0 suma a[0] suma=0
Hebra 0 suma a[1] suma=1
Hebra 0 suma a[6] suma=7
Fuera de 'parallel for' suma=7
$ schedule-clause 3
Hebra 2 suma a[6] suma=6
Hebra 0 suma a[0] suma=0
Hebra 0 suma a[1] suma=1
Hebra 0 suma a[2] suma=3
Hebra 1 suma a[3] suma=3
Hebra 1 suma a[4] suma=7
Hebra 1 suma a[5] suma=12
Fuera de 'parallel for' suma=6
```

Cláusula `schedule` `dynamic`

scheduled-clause. C



- 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
 - Nº unidades $O(n/\text{chunk})$
- Precauciones:
 - Añade sobrecarga adicional

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

main(int argc, char **argv) {
    int i, n=200, chunk, a[n], suma=0;
    if(argc < 3) {
        fprintf(stderr, "\nFalta iteraciones o chunk \n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>200) n=200; chunk = atoi(argv[2]);

    for (i=0; i<n; i++)    a[i] = i;

    #pragma omp parallel for firstprivate(suma) \
        lastprivate(suma) schedule(dynamic, 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);
}
```

Cláusula `schedule.dynamic`

➤ Usa

- `schedule(dynamic, chunk)`
- Las iteraciones se dividen en unidades de `chunk` iteraciones.
- Las unidades se asignan en tiempo de ejecución. Los threads más rápidos ejecutan más unidades.
- Si no se especifica `chunk` se usan unidades de una iteración

➤ Entradas: nº de iteraciones y tamaño de la unidad de distribución

```
mancia@mancia-ubuntu: ~/docencia/OpenMP/lecci
Archivo  Editar  Ver  Terminal  Ayuda
$ export OMP_NUM_THREADS=2
$ gcc -O2 -fopenmp -o scheduled-clause scheduled-clause.c
$ scheduled-clause 7 3
Hebra 0 suma a[0]=0 suma=0
Hebra 0 suma a[1]=1 suma=1
Hebra 0 suma a[2]=2 suma=3
Hebra 0 suma a[6]=6 suma=9
Hebra 1 suma a[3]=3 suma=3
Hebra 1 suma a[4]=4 suma=7
Hebra 1 suma a[5]=5 suma=12
Fuera de 'parallel for' suma=9
$ scheduled-clause 7 2
Hebra 0 suma a[0]=0 suma=0
Hebra 0 suma a[1]=1 suma=1
Hebra 0 suma a[4]=4 suma=5
Hebra 0 suma a[5]=5 suma=10
Hebra 0 suma a[6]=6 suma=16
Hebra 1 suma a[2]=2 suma=2
Hebra 1 suma a[3]=3 suma=5
Fuera de 'parallel for' suma=16
$
```

Cláusula `schedule` guided

[scheduleg-clause.c](#)



- Kind = guided
 - Distribución en tiempo de ejecución
 - Apropiado si se desconoce el tiempo de ejecución de las iteraciones o su número
 - Comienza con bloque largo
 - El tamaño del bloque va menguando (**nº iteraciones que restan dividido por nº threads**), no más pequeño que chunk (excepto la última)
- Precauciones:
 - Sobrecarga extra, pero menos que `dynamic` para el mismo *chunk*

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

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);
}
```

Cláusula `schedule` `runtime`

`scheduler-clause.c`



- Kind = runtime
 - El tipo de distribución (static, dynamic o guided) se fija en tiempo de ejecución
 - El tipo de distribución **depende** de la variable de control **run-sched-var**

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

main(int argc, char **argv) {
    int i, n=20, a[n], suma=0;
    if(argc < 2) {
        fprintf(stderr, "\nFalta 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 for firstprivate(suma) \
                          lastprivate(suma) schedule(runtime)
    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);
}
```

Cláusula `schedule.runtime`. Salida

➤ Usa

- `schedule(runtime)`
- `OMP_SCHEDULE`
para fijar el tipo de
distribución

➤ Entrada: n° de iteraciones

```
mancia@mancia-ubuntu: ~/docencia/OpenMP/lecci
Archivo  Editar  Ver  Terminal  Ayuda
$ gcc -O2 -fopenmp -o scheduler-clause scheduler-clause.c
$ export OMP_SCHEDULE="static"
$ scheduler-clause 8
Hebra 0 suma a[0]=0 suma=0
Hebra 0 suma a[2]=2 suma=2
Hebra 0 suma a[4]=4 suma=6
Hebra 0 suma a[6]=6 suma=12
Hebra 1 suma a[1]=1 suma=1
Hebra 1 suma a[3]=3 suma=4
Hebra 1 suma a[5]=5 suma=9
Hebra 1 suma a[7]=7 suma=16
Fuera de 'parallel for' suma=16
$ export OMP_SCHEDULE="static,2"
$ scheduler-clause 8
Hebra 0 suma a[0]=0 suma=0
Hebra 0 suma a[1]=1 suma=1
Hebra 0 suma a[4]=4 suma=5
Hebra 0 suma a[5]=5 suma=10
Hebra 1 suma a[2]=2 suma=2
Hebra 1 suma a[3]=3 suma=5
Hebra 1 suma a[6]=6 suma=11
Hebra 1 suma a[7]=7 suma=18
Fuera de 'parallel for' suma=18
```

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

Funciones de la biblioteca OpenMP

- Funciones para acceder al entorno de ejecución de OpenMP
- Funciones para usar sincronización con cerrojos
 - V2.5 `omp_init_lock()`, `omp_destroy_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`
 - V3.0: `omp_destroy_nest_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock`, `omp_test_nest_lock`
- Funciones para obtener tiempos de ejecución
 - `omp_get_wtime()`, `omp_get_wtick()`

Contenido

- Variables de control
- Variables de entorno
- Funciones del entorno de ejecución
- Cláusulas para interactuar con el entorno
- Clasificación de las funciones de la biblioteca OpenMP
- Funciones para obtener el tiempo de ejecución

Ejemplo: cálculo de PI en C

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

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

    struct timespec cgt1,cgt2;
    double ncgt;

    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);
}
```

Ejemplo PI

- Compilar con `-lrt` para incluir librería **real time** (no siempre es necesario)
- Datos obtenidos en un computador con cuatro procesadores de cuatro cores cada uno.
- Tiempo en segundos

pruebaPI.o19562

Nodos usados en la ejecución del trabajo: 1

Machines:

shn13

Iteraciones: 10000000 .

PI: 3.141592653589730943508584 .

Threads: 1 . Tiempo: 0.194065

Iteraciones: 40000000 .

PI: 3.141592653588800576613949 .

Threads: 1 . Tiempo: 0.561454

Ejemplo: cálculo de PI con OpenMP/C

OpenMP/C Pi

```
#include <stdlib.h>
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_max_threads() 1
#endif

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

    double t;

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

```
t=omp_get_wtime();

width = 1.0 / intervals;
#pragma omp parallel
{
    #pragma omp for reduction(+:sum) private(x)
    for (i=0; i<intervals; i++) {
        x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }
}
sum *= width;

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_max_threads(),t);

return(0);
}
```

Ejemplo PI con OpenMP

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

pruebaPI_OMP.o19561

Nodos usados en la ejecución del trabajo: 1

Machines: shn10 *Aquí el tiempo de comunicación se mantiene constante*

Iteraciones:	10000000	.
PI:	3.141592653589803774139000	.
Threads:	8	.Tiempo: 0.016534
Iteraciones:	10000000	.
PI:	3.141592653589669659197625	.
Threads:	4	.Tiempo: 0.029227
Iteraciones:	10000000	.
PI:	3.141592653589922790047240	.
Threads:	2	.Tiempo: 0.055943
Iteraciones:	10000000	.
PI:	3.141592653589730943508584	.
Threads:	1	.Tiempo: 0.105901
Iteraciones:	40000000	.
PI:	3.141592653589751815701447	.
Threads:	8	.Tiempo: 0.058191
Iteraciones:	40000000	.
PI:	3.141592653589848183059985	.
Threads:	4	.Tiempo: 0.109995
Iteraciones:	40000000	.
PI:	3.141592653589986294804248	.
Threads:	2	.Tiempo: 0.214541
Iteraciones:	40000000	.
PI:	3.141592653588800576613949	.
Threads:	1	.Tiempo: 0.424877

Ejemplo: cálculo de PI en MPI/C

MPI/C Pi

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

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);
}
```

Ejemplo PI con MPI

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

pruebaPI_MPI2.o19560

Nodos usados en la ejecución del trabajo: 1

Machines: shn09 *Aquí el tiempo de comunicación añade mucho retardo*

Iteraciones:	10000000	.
PI:	3.141592653589806882763469	.
Procesos:	8	.Tiempo: 2.281467
Iteraciones:	10000000	.
PI:	3.141592653589686090498390	.
Procesos:	4	.Tiempo: 1.116629
Iteraciones:	10000000	.
PI:	3.141592653589984962536619	.
Procesos:	2	.Tiempo: 0.119861
Iteraciones:	10000000	.
PI:	3.141592653589730943508584	.
Procesos:	1	.Tiempo: 0.156071
Iteraciones:	40000000	.
PI:	3.141592653589687422766019	.
Procesos:	8	.Tiempo: 1.306842
Iteraciones:	40000000	.
PI:	3.141592653589948547221411	.
Procesos:	4	.Tiempo: 1.213710
Iteraciones:	40000000	.
PI:	3.141592653590174144540015	.
Procesos:	2	.Tiempo: 0.278467
Iteraciones:	40000000	.
PI:	3.141592653588800576613949	.
Procesos:	1	.Tiempo: 0.475955