

2° curso / 2° cuatr.  
Grado Ing. Inform.  
Doble Grado Ing.  
Inform. y Mat.

# Arquitectura de Computadores (AC)

## Cuaderno de prácticas.

### Bloque Práctico 3. Programación paralela III: Interacción con el entorno en OpenMP

Estudiante (nombre y apellidos): Christian Andrades Molina

Grupo de prácticas: B2

Fecha de entrega: 20/05/14

Fecha evaluación en clase: 21/05/14

1. Usar la cláusula `num_threads(x)` en el ejemplo del seminario `if_clause.c`, y añadir un parámetro de entrada al programa que fije el valor `x` que se va a usar en la cláusula. Incorporar en el cuaderno de trabajo de esta práctica volcados de pantalla con ejemplos de ejecución que ilustren la funcionalidad de esta cláusula y explicar por qué lo ilustran.

#### CÓDIGO FUENTE: `if-clauseModificado.c`

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

int main(int argc, char **argv)
{
    int i, n=20, tid, x;
    int a[n], suma=0, sumalocal;

    if(argc < 3) {
        fprintf(stderr, "[ERROR]- Falta iteraciones / Falta valor de x\n");
        exit(-1);
    }

    n = atoi(argv[1]); if (n>20) n=20;
    x = atoi(argv[2]);
    for (i=0; i<n; i++){
        a[i] = i;
    }

    #pragma omp parallel num_threads(x) if(n>4) default(none) \
        private(sumalocal,tid) shared(a,suma,n) \

    { sumalocal=0;
      tid=omp_get_thread_num();

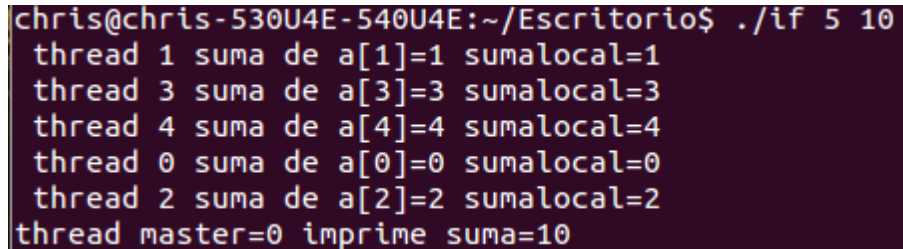
    #pragma omp for private(i) schedule(static) nowait

    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);

}
}
```

**CAPTURAS DE PANTALLA:**


```
chris@chris-530U4E-540U4E:~/Escritorio$ ./if 5 10
thread 1 suma de a[1]=1 sumalocal=1
thread 3 suma de a[3]=3 sumalocal=3
thread 4 suma de a[4]=4 sumalocal=4
thread 0 suma de a[0]=0 sumalocal=0
thread 2 suma de a[2]=2 sumalocal=2
thread master=0 imprime suma=10
```

**RESPUESTA:** El número que resulta de evaluar la cláusula `if` tiene mayor prioridad que el número que fijamos para la cláusula `num_threads (x)`, dando lugar a que el valor de `x` no sea evaluado al ejecutar el código.

2. (a) Rellenar la tabla (se debe poner en la tabla el *thread* que ejecuta cada iteración) ejecutando los ejemplos del seminario `schedule-clause.c`, `scheduled-clause.c` y `scheduleg-clause.c` con dos *threads* (0,1) y unas entradas de:

- iteraciones: 16 (0,...15)
- chunk= 1, 2 y 4

	schedule-clause.c			scheduled-clause.c			scheduleg-clause.c		
ITERACIÓN	1	2	4	1	2	4	1	2	4
0	0	0	0	1	0	1	0	0	0
1	1	0	0	0	0	1	0	0	0
2	0	1	0	1	1	1	0	0	0
3	1	1	0	1	1	1	0	0	0
4	0	0	1	1	0	0	0	0	0
5	1	0	1	1	0	0	0	0	0
6	0	1	1	1	1	0	0	0	0
7	1	1	1	1	1	0	0	0	0
8	0	0	0	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	1
10	0	1	0	1	0	1	1	1	1
11	1	1	0	1	0	1	1	1	1
12	0	0	1	0	0	0	0	0	0
13	1	0	1	0	0	0	0	0	0
14	0	1	1	0	0	0	1	0	0
15	1	1	1	0	0	0	1	0	0

**(b)** Rellenar otra tabla como la de la figura pero esta vez usando cuatro *threads* (0,1,2,3).

	schedule-clause.c			scheduled-clause.c			scheduleg-clause.c		
ITERACIÓN	1	2	4	1	2	4	1	2	4
0	0	0	0	2	0	0	2	3	3
1	1	0	0	3	0	0	2	3	3
2	2	1	0	0	2	0	2	3	3
3	3	1	0	1	2	0	2	3	3
4	0	2	1	1	1	1	3	0	0
5	1	2	1	1	1	1	3	0	0
6	2	3	1	1	3	1	3	0	0
7	3	3	1	1	3	1	1	1	0
8	0	0	2	1	3	3	1	1	2
9	1	0	2	1	3	3	1	1	2
10	2	1	2	1	3	3	0	2	2
11	3	1	2	1	3	3	0	2	2
12	0	2	3	1	2	2	2	0	1
13	1	2	3	1	2	2	2	0	1
14	2	3	3	1	2	2	2	0	1
15	3	3	3	1	2	2	2	0	1

- Añadir al programa `scheduled-clause.c` lo necesario para que imprima el valor de las variables de control `dyn-var`, `nthreads-var`, `thread-limit-var` y `run-sched-var` dentro (debe imprimir sólo un thread) y fuera de la región paralela. Realizar varias ejecuciones usando variables de entorno para modificar estas variables de control antes de la ejecución. Incorporar en su cuaderno de prácticas volcados de pantalla de estas ejecuciones. ¿Se imprimen valores distintos dentro y fuera de la región paralela?

**CÓDIGO FUENTE:** `scheduled-clauseModificado.c`

```
#include<stdio.h>

#include<stdlib.h>

#include<time.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;
    srand(time(NULL));

    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("\nthread %d suma a[%d]=%d suma=%d\n",
                omp_get_thread_num(), i, a[i], suma/*, omp_get_schedule(&
kind, n)*/);

            if (i==1)
                printf("VALORES (hebra :%d) --> dyn-var = %d --
thread-max = %d -- thread-limit-var = %d ", omp_get_thread_num(),
omp_get_dynamic(), omp_get_max_threads(), omp_get_thread_limit());

        }

    printf("\n-----");
    printf("\nFuera de 'parallel for' suma=%d\n", suma);

```

```
printf("VALORES (hebra :%d) --> dyn-var = %d -- thread-max = %d --
thread-limit-var = %d \n",omp_get_thread_num(),
omp_get_dynamic(),omp_get_max_threads(),omp_get_thread_limit());
}
```

### CAPTURAS DE PANTALLA:

```
chris@chris-530U4E-540U4E:~/Escritorio$ export OMP_THREAD_LIMIT=8
chris@chris-530U4E-540U4E:~/Escritorio$ export OMP_DYNAMIC=FALSE
chris@chris-530U4E-540U4E:~/Escritorio$ export OMP_NUM_THREADS=6
chris@chris-530U4E-540U4E:~/Escritorio$ gcc -O2 -fopenmp -o she2 she2.c
chris@chris-530U4E-540U4E:~/Escritorio$ ./she2 6 2

thread 2 suma a[0]=0 suma=0

thread 2 suma a[1]=1 suma=1
VALORES (hebra :2) --> dyn-var = 0 -- thread-max = 6 -- thread-limit-var = 8
thread 3 suma a[2]=2 suma=2

thread 3 suma a[3]=3 suma=5

thread 4 suma a[4]=4 suma=4

thread 4 suma a[5]=5 suma=9

-----
Fuera de 'parallel for' suma=9
VALORES (hebra :0) --> dyn-var = 0 -- thread-max = 6 -- thread-limit-var = 8
```

**RESPUESTA:** Se dan los mismos valores.

4. Usad en el ejemplo anterior las funciones `omp_get_num_threads()`, `omp_get_num_procs()` y `omp_in_parallel()` dentro y fuera de la región paralela. Imprimir los valores que obtienen estas funciones dentro (lo debe imprimir sólo uno de los threads) y fuera de la región paralela. Incorporar en su cuaderno de prácticas volcados de pantalla con los resultados de ejecución obtenidos. Indicad en qué funciones se obtienen valores distintos dentro y fuera de la región paralela.

**CÓDIGO FUENTE:** `scheduled-clauseModificado4.c`

```
#include <errno.h>

#include <time.h>

#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;
srand(time(NULL));

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("\nthread %d suma a[%d]=%d suma=%d\n",
            omp_get_thread_num(), i, a[i], suma /*, omp_get_schedule(&
kind, n) */);

        if (i==1)

            printf("VALORES (hebra :%d) --> N°threads r.p.= %d --
N°procesadores= %d -- Es parallel? = %d", omp_get_thread_num(),
omp_get_num_threads(), omp_get_num_procs(), omp_in_parallel());

    }

printf("\n-----");
printf("\nFuera de 'parallel for' suma=%d\n", suma);

printf("VALORES (hebra :%d) --> N°threads r.p.= %d -- N°procesadores= %d -- Es
parallel? = %d\n", omp_get_thread_num(),
omp_get_num_threads(), omp_get_num_procs(), omp_in_parallel());
}
```

## CAPTURAS DE PANTALLA:

```

chris@chris-530U4E-540U4E:~/Escritorio$ ./she2 6 2

thread 2 suma a[0]=0 suma=0

thread 2 suma a[1]=1 suma=1
VALORES (hebra :2) --> N°threads r.p.= 6 -- N°procesadores= 4 -- Es parallel? = 1
thread 1 suma a[4]=4 suma=4

thread 1 suma a[5]=5 suma=9

thread 3 suma a[2]=2 suma=2

thread 3 suma a[3]=3 suma=5

-----
Fuera de 'parallel for' suma=9
VALORES (hebra :0) --> N°threads r.p.= 1 -- N°procesadores= 4 -- Es parallel? = 0

```

**RESPUESTA:** Varía en el número de threads disponibles (1 al estar en zona secuencial) y si estamos o no en región parallel.

5. Añadir al programa `scheduled-clause.c` lo necesario para modificar las variables de control `dyn-var`, `nthreads-var` y `run-sched-var` y para poder imprimir el valor de estas variables antes y después de dicha modificación. Incorporar en su cuaderno de prácticas volcados de pantalla con los resultados de ejecución obtenidos.

**CÓDIGO FUENTE:** `scheduled-clauseModificado5.c`

```

#include <errno.h>

#include <time.h>

#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;
    int dyn, nthreads, dyn1, nthreads1;
    srand(time(NULL));

    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("\nthread %d suma a[%d]=%d suma=%d\n",
            omp_get_thread_num(),i,a[i],suma/*,omp_get_schedule(&
kind,n)*/);

        if (i==1)
            printf("VALORES (hebra :%d) --> dyn-var = %d --
thread-max = %d ",omp_get_thread_num(),
omp_get_dynamic(),omp_get_max_threads());

        if (i==3){
            printf("\nModificamos variables:\n");
            omp_set_dynamic(1);
            omp_set_num_threads(14);

            printf("VALORES (hebra :%d) --> dyn-var = %d --
thread-max = %d ",omp_get_thread_num(),
omp_get_dynamic(),omp_get_max_threads());
        }
    }

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

```

### CAPTURAS DE PANTALLA:

```

chris@chris-530U4E-540U4E:~/Escritorio$ ./she55 6 2

thread 0 suma a[0]=0 suma=0

thread 0 suma a[1]=1 suma=1
VALORES (hebra :0) --> dyn-var = 0 -- thread-max = 4
thread 2 suma a[2]=2 suma=2

thread 2 suma a[3]=3 suma=5

Modificamos variables:
VALORES (hebra :2) --> dyn-var = 1 -- thread-max = 14
thread 3 suma a[4]=4 suma=4

thread 3 suma a[5]=5 suma=9

-----
Fuera de 'parallel for' suma=9

```

6. Implementar un programa secuencial en C que multiplique una matriz triangular por un vector.

NOTAS: (1) el número de filas/columnas debe ser un argumento de entrada; (2) se debe inicializar las matrices antes del cálculo; (3) se debe imprimir siempre



las componentes (0,0) y (N-1, N-1) del resultado antes de que termine el programa.

**CÓDIGO FUENTE:** pmtv-secuencial.c

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

int main (int argc, char** argv){

    int i,j;

    if (argc<2){
        printf("Faltan componentes del vector\n");
        exit(1);
    }

    const int N = atoi(argv[1]);

    ////////// DECLARACIÓN DE VECTORES Y MATRIZ

    int **matriz = (int **) malloc (N*sizeof(int *));
    for (i=0;i<N;i++)
        matriz[i]=(int *) malloc (N*sizeof(int));
    int **matriz2 = (int **) malloc (N*sizeof(int *));
    for (i=0;i<N;i++)
        matriz2[i]=(int *) malloc (N*sizeof(int));

    int *v1;
    v1 = (int *) malloc (N*sizeof(int*));
    int *v2;
```

```
v2 = (int *) malloc (N*sizeof(int*));

if ( (v1==NULL) || (v2==NULL) ){
printf("Error en la reserva de espacio para vectores/matriz\n");
exit(-2);
}

////////// INICIALIZAR VECTOR 1

for (i=0;i<N;i++)
    v1[i]=i+1;

////////// INICIALIZAR VECTOR 2 a ceros

for (i=0;i<N;i++)
    v2[i]=0;

////////// INICIALIZAR MATRIZ triangular

for (i=0; i<N; i++){
    for(j=0;j<N;j++){
        if(i<=j)
            matriz[i][j]=i+j;
        else
            matriz[i][j]=0;
    }
}

////////// MOSTRAR VECTOR MATRIZ Y VECTOR 1

printf("\nMATRIZ:\n");

for (i=0;i<N;i++){
    printf("|");
```

```

        for (j=0;j<N;j++) {
            printf(" %d ",matriz[i][j]);

        }
        printf("|");
        printf("\n");
    }

    printf("\nVECTOR:\n");
    printf("[");
    for (i=0;i<N;i++){
        printf(" %d ",v1[i]);
    }

    printf("]\n\n");

////////// MULTIPLICACIÓN MATRIZ X VECTOR Y RESULTADO

for (i=0;i<N;i++)
    for (j=0;j<N;j++){
        if (matriz[i][j]==0) {
            v2[i]+=0;
            j++;
        }
        v2[i]+=(matriz[i][j]*v1[j]);
    }

    printf("Resultado de la multiplicación : [");

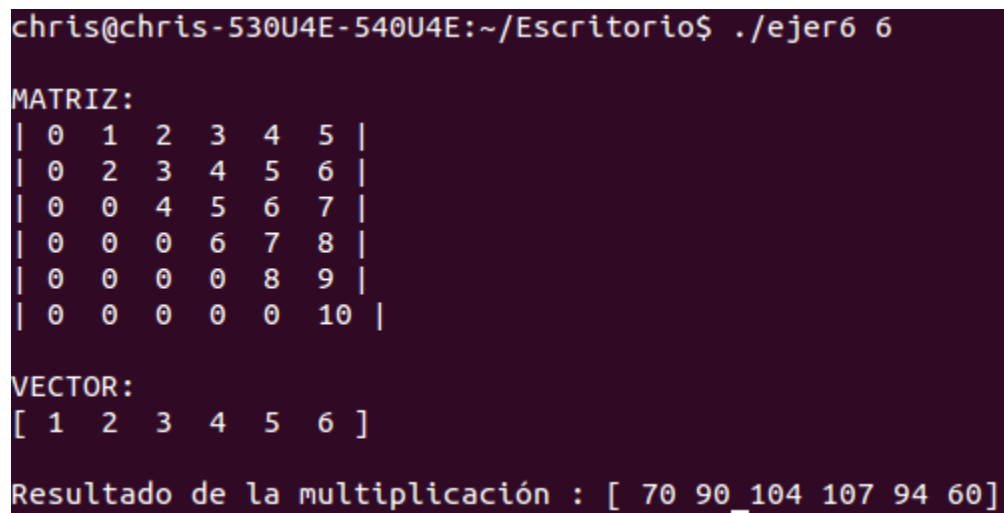
for (i=0;i<N;i++)
    printf(" %d",v2[i]);

```

```
printf("]\n");

return 0;

}
```

**CAPTURAS DE PANTALLA:**


```
chris@chris-530U4E-540U4E:~/Escritorio$ ./ejer6 6
MATRIZ:
| 0  1  2  3  4  5 |
| 0  2  3  4  5  6 |
| 0  0  4  5  6  7 |
| 0  0  0  6  7  8 |
| 0  0  0  0  8  9 |
| 0  0  0  0  0 10 |
VECTOR:
[ 1  2  3  4  5  6 ]
Resultado de la multiplicación : [ 70 90 104 107 94 60]
```

- Implementar en paralelo la multiplicación de una matriz triangular por un vector a partir del código secuencial realizado para el ejercicio anterior utilizando la directiva `for` de OpenMP. Dibujar en el cuaderno de prácticas la descomposición de dominio utilizada (Lección 4/Tema 2) en el código paralelo implementado para asignar tareas a los threads (Lección 5/Tema 2). Añadir lo necesario para que el usuario pueda fijar la planificación de tareas usando la variable de entorno `OMP_SCHEDULE`. Obtener en `atcggrid` los tiempos de ejecución del código paralelo que multiplica una matriz triangular por un vector con las alternativas de planificación `static`, `dynamic` y `guided` para `chunk` de 2, 64, 128, 1024 y el `chunk` por defecto para la alternativa. No use vectores mayores de 32768 componentes ni menores de 4096 componentes. El número de threads en las ejecuciones debe coincidir con el número de cores. Rellenar la Error: No se encuentra la fuente de referencia con los tiempos obtenidos, ponga en la tabla el número de threads que utilizan las ejecuciones. Representar el tiempo para `static`, `dynamic` y `guided` en función del tamaño del `chunk` en una gráfica. Rellenar la tabla y realizar la gráfica también para el PC local. ¿Qué alternativa ofrece mejores prestaciones? Razone por qué.

**CÓDIGO FUENTE:** `pmtv-OpenMP.c`

```
#include <errno.h>
```

```

#include <time.h>
#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=0, j=0;
int chunk = 2;
int aux = 0;
int kind; //0,1,2

if ((argc)<2) {
    exit(1);
}

const int N = atoi(argv[1]);

///////// DECLARACIÓN DE VECTORES Y MATRIZ

int **matriz = (int **) malloc (N*sizeof(int *));

#pragma omp parallel for shared(i) schedule(runtime)
for (i=0;i<N;i++)
    matriz[i]=(int *) malloc (N*sizeof(int));

int *v1;        //Declaramos vector
v1 = (int *) malloc (N*sizeof(int*));
int *v2;
v2 = (int *) malloc (N*sizeof(int*));

////////// INICIALIZAR VECTOR 1

#pragma omp parallel for shared(i) schedule(runtime)
for (i=0;i<N;i++)
    v1[i]=i;

////////// INICIALIZAR VECTOR 2 a ceros

#pragma omp parallel for shared (i) schedule(runtime)
for (i=0;i<N;i++)
    v2[i]=0;

////////// INICIALIZAR MATRIZ triangular

#pragma omp parallel for private (i,j) schedule(runtime)
for (i=0; i<N; i++){
    v1[i]=i+1;
    for(j=0;j<N;j++){
        if(i<=j)
            matriz[i][j]=i+j;
        else
            matriz[i][j]=0;
    }
}

```

```

}

////////// MOSTRAR VECTOR MATRIZ Y VECTOR 1

if (N<100){
printf("\nMATRIZ:\n");
for (i=0;i<N;i++){
    printf("|");
    for (j=0;j<N;j++) {
        printf(" %d ",matriz[i][j]);
    }
    printf("|");
    printf("\n");
}
}

if (N<100){
    printf("\nVECTOR:\n");
    printf("[");
        for (i=0;i<N;i++){
            printf(" %d ",v1[i]);
        }
    printf("]\n\n");
}

//MULTIPLICACIÓN DE LA MATRIZ

#pragma omp parallel for private (i,j) schedule (runtime) reduction (+:aux)
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        if (matriz[i][j]==0) {
            v2[i]+=0;
            j++;
        }
        aux=(matriz[i][j]*v1[j]);
        v2[i]+=aux;
        printf(" thread %d\n",omp_get_thread_num());
    }
}

if (N<100){
printf("Resultado de la multiplicación : ");

for (i=0;i<N;i++)
    printf(" %d",v2[i]);
printf("\n");
}

if (N>=100){
    printf("\nPrimer valor: %d y ultimo valor: %d\n",v2[0],v2[N-1]);
    printf("\n\n");
}

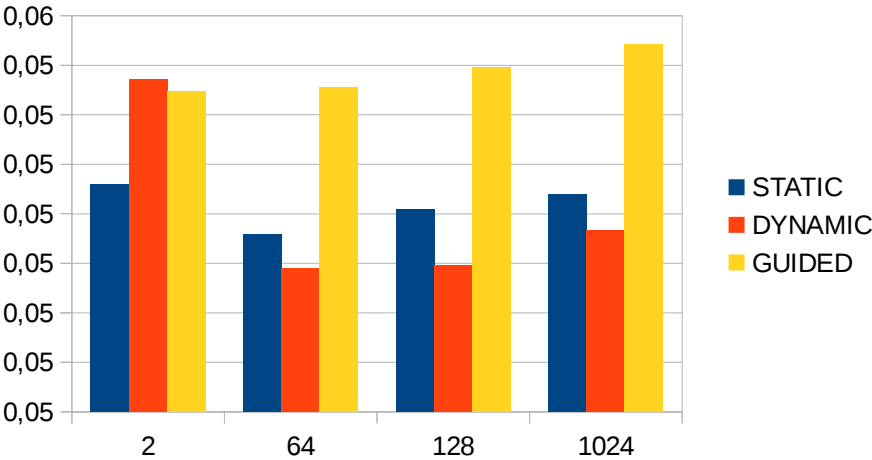
return (0);
}

```

**CAPTURAS DE PANTALLA:  
(ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

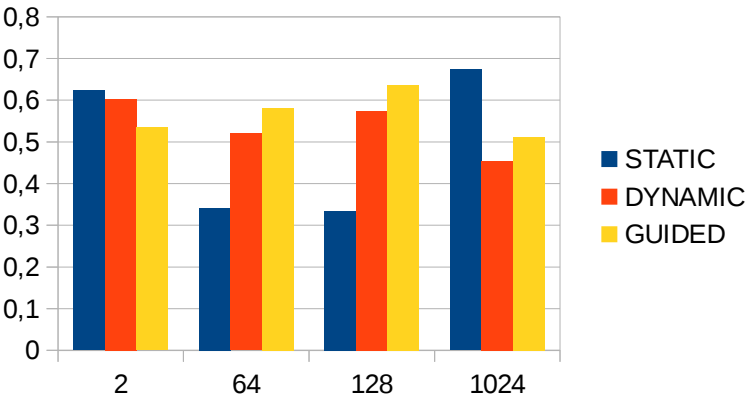
**LOCAL - 4 HEBRAS**

CHUNK	STATIC	DYNAMIC	GUIDED
2	0,051589191	0,053724616	0,05346711
64	0,050593583	0,049898556	0,053546426
128	0,051101182	0,049958453	0,053957887
1024	0,051394211	0,050662435	0,054419016



**ATCGRID - 4 HEBRAS**

CHUNK	STATIC	DYNAMIC	GUIDED
2	0,623	0,603	0,534
64	0,342	0,52	0,581
128	0,334	0,574	0,636
1024	0,674	0,454	0,511



En ambas gráficas se observa que el planificador más regular en ambos ordenadores es dynamic al realizar en tiempo de ejecución el reparto. Static consigue unos resultados similares y guided el peor rendimiento.

8. Implementar un programa secuencial en C que calcule la multiplicación de matrices cuadradas, B y C:

NOTAS: (1) el número de filas/columnas debe ser un argumento de entrada; (2) se deben inicializar las matrices antes del cálculo; (3) se debe imprimir siempre las componentes (0,0) y (N-1, N-1) del resultado antes de que termine el programa.

**CÓDIGO FUENTE:** pmm-secuencial.c

```
#include <errno.h>

#include <time.h>

#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,j;
    if (argc<2){
        printf("Faltan componentes del vector\n");
        exit(1);
    }
    const int N = atoi(argv[1]);
    /////////// DECLARACIÓN DE VECTORES Y MATRIZ

    int **matriz = (int **) malloc (N*sizeof(int *));
    for (i=0;i<N;i++)
        matriz[i]=(int *) malloc (N*sizeof(int));

    int **matriz2 = (int **) malloc (N*sizeof(int *));
    for (i=0;i<N;i++)
        matriz2[i]=(int *) malloc (N*sizeof(int));

    int **matriz3 = (int **) malloc (N*sizeof(int *));
    for (i=0;i<N;i++)
        matriz3[i]=(int *) malloc (N*sizeof(int));

    if ( (matriz==NULL) || (matriz2==NULL) || (matriz3==NULL) ){
```



```

printf("Error en la reserva de espacio para matrices\n");
exit(-2);

}

////////// INICIALIZAR MATRIZ, 2, 3

for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        matriz[i][j]=0;
        matriz2[i][j]=i+j;
        matriz3[i][j]=i+j;
    }
}

printf("\nMATRIZ 2:\n");
for (i=0;i<N;i++){
    printf("|");
    for (j=0;j<N;j++) {
        printf(" %d ",matriz2[i][j]);
    }
    printf("|");
    printf("\n");
}
printf("\nMATRIZ 3:\n");
for (i=0;i<N;i++){
    printf("|");
    for (j=0;j<N;j++) {
        printf(" %d ",matriz3[i][j]);
    }
    printf("|");
    printf("\n");
}

////////// MULTIPLICACIÓN MATRIZ2 X MATRIZ3

int h=0;

for (i=0;i<N;i++)
    for (j=0;j<N;j++){
        for (h=0;h<N;h++)
            matriz[i][j]+=((matriz2[i][h])*(matriz3[h][j]));
    }

printf("\nRESULTADO MATRIZ2 X MATRIZ3:\n");
for (i=0;i<N;i++){
    printf("|");
    for (j=0;j<N;j++) {
        printf(" %d ",matriz[i][j]);
    }

    printf("|");
    printf("\n");
}

printf("\n\n");
return 0;

```

}

**CAPTURAS DE PANTALLA:**

```

chris@chris-530U4E-540U4E:~/Escritorio$ ./ejer8 6

MATRIZ 2:
| 0 1 2 3 4 5 |
| 1 2 3 4 5 6 |
| 2 3 4 5 6 7 |
| 3 4 5 6 7 8 |
| 4 5 6 7 8 9 |
| 5 6 7 8 9 10 |

MATRIZ 3:
| 0 1 2 3 4 5 |
| 1 2 3 4 5 6 |
| 2 3 4 5 6 7 |
| 3 4 5 6 7 8 |
| 4 5 6 7 8 9 |
| 5 6 7 8 9 10 |

RESULTADO MATRIZ2 X MATRIZ3:
| 55 70 85 100 115 130 |
| 70 91 112 133 154 175 |
| 85 112 139 166 193 220 |
| 100 133 166 199 232 265 |
| 115 154 193 232 271 310 |
| 130 175 220 265 310 355 |

```

9. Implementar en paralelo la multiplicación de matrices cuadradas con OpenMP a partir del código escrito en el ejercicio anterior. Use las directivas, las cláusulas y las funciones de entorno que considere oportunas. Se debe paralelizar también la inicialización de las matrices. Dibuje en su cuaderno de prácticas la descomposición de dominio que ha utilizado en el código paralelo implementado para asignar tareas a los threads (Leción 4/Tema 2, Lección 5/Tema 2).

**CÓDIGO FUENTE:** pmm-OpenMP.c

```

#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,j;
int chunk = 1;

if (argc<2){
    printf("Faltan componentes del vector\n");
    exit(1);
}

const int N = atoi(argv[1]);

```

```

////////// DECLARACIÓN DE VECTORES Y MATRIZ
int **matriz = (int **) malloc (N*sizeof(int *));

#pragma omp parallel for shared(i) schedule (dynamic,chunk)
for (i=0;i<N;i++){
    matriz[i]=(int *) malloc (N*sizeof(int));
}

int **matriz2 = (int **) malloc (N*sizeof(int *));
#pragma omp parallel for shared(i) schedule (dynamic,chunk)
for (i=0;i<N;i++){
    matriz2[i]=(int *) malloc (N*sizeof(int));
}

int **matriz3 = (int **) malloc (N*sizeof(int *));

#pragma omp parallel for shared(i) schedule (dynamic,chunk)
for (i=0;i<N;i++){
    matriz3[i]=(int *) malloc (N*sizeof(int));
}

if ( (matriz==NULL) || (matriz2==NULL) || (matriz3==NULL) ){
    printf("Error en la reserva de espacio para matrices\n");
    exit(-2);
}

////////// INICIALIZAR MATRIZ1,2,3

#pragma omp parallel for private(i,j) schedule (dynamic,chunk)
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        matriz[i][j]=0;
        matriz2[i][j]=i+j;
        matriz3[i][j]=i+j;
    }
}

if (N<150){
    printf("\nMATRIZ 2:\n");
    for (i=0;i<N;i++){
        printf("|");
        for (j=0;j<N;j++) {
            printf(" %d ",matriz2[i][j]);
        }
        printf("|");
        printf("\n");
    }
}

if (N<150){
    printf("\nMATRIZ 3:\n");
    for (i=0;i<N;i++){
        printf("|");
        for (j=0;j<N;j++) {
            printf(" %d ",matriz3[i][j]);
        }
    }
}

```

```

        printf("|");
        printf("\n");
    }
}

////////// MULTIPLICACIÓN MATRIZ2 X MATRIZ3

int k=0; //indice

#pragma omp parallel for schedule (dynamic,chunk) private (i, j, k)
    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            for(k=0; k<N; k++) {
                matriz[i][j] += matriz2[i][k] * matriz3[k][j];
            }
        }
    }

printf("\nRESULTADO MATRIZ2 X MATRIZ3:\n");

if (N<150){
    for (i=0;i<N;i++){
        printf("|");
        for (j=0;j<N;j++) {
            printf(" %d ",matriz[i][j]);
        }
        printf("|");
        printf("\n");
    }
    printf("\n\n");
}

else
    printf("\n - Primer valor: %d, - Ultimo valor: %d ",matriz[0][0],matriz[N-1]
[N-1]);

    printf("\n\n");

return 0;

}

```

### CAPTURAS DE PANTALLA:

```

chris@chris-530U4E-540U4E:~/Escritorio$ gcc -fopenmp -O2 ejer9.c -o ejer9
chris@chris-530U4E-540U4E:~/Escritorio$ ./ejer9 4

MATRIZ 2:
| 0 1 2 3 |
| 1 2 3 4 |
| 2 3 4 5 |
| 3 4 5 6 |

MATRIZ 3:
| 0 1 2 3 |
| 1 2 3 4 |
| 2 3 4 5 |
| 3 4 5 6 |

RESULTADO MATRIZ2 X MATRIZ3:
| 21 32 43 54 |
| 32 58 84 110 |
| 43 84 125 166 |
| 54 110 166 222 |

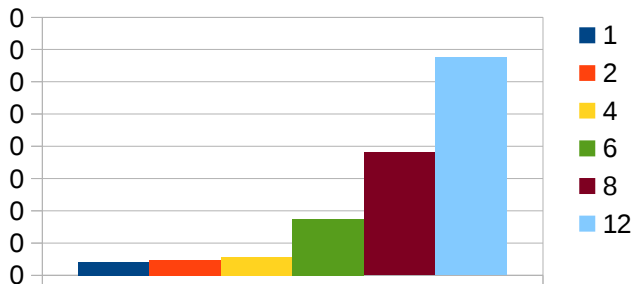
```

10. Hacer un estudio de escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en el PC local del código paralelo implementado para tres tamaños de las matrices ( $N = 10, 100$  y  $500$ ). Presente los resultados del estudio en tablas de valores y en gráficas. Escoger los tamaños de manera que se observe diferentes curvas de escalabilidad en las gráficas que entregue en su cuaderno de prácticas. Consulte la Lección 6/Tema 2.

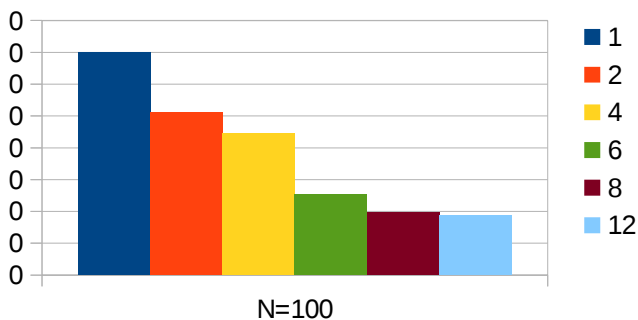
### PC - LOCAL

CORES	1	2	4	6	8	12
N=10	0,000008394	0,000009451	0,000011577	0,000034668	0,000076193	0,000135096
N=100	0,003494576	0,002561652	0,002229683	0,001265722	0,000981462	0,00093328
N=500	0,324729793	0,209718499	0,206025778	0,202249203	0,156235578	0,144549297

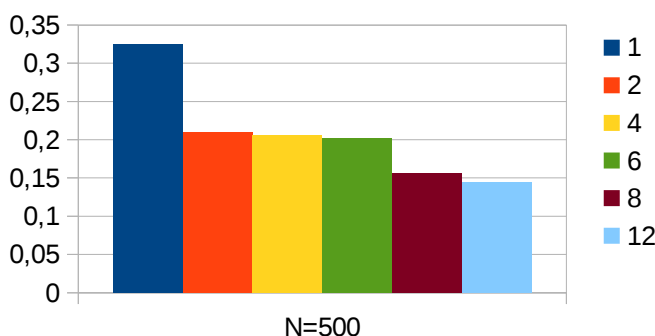
LOCAL N=10



LOCAL N=100



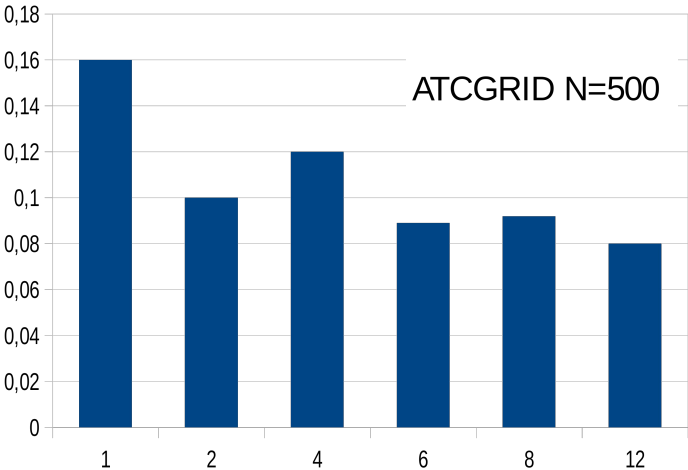
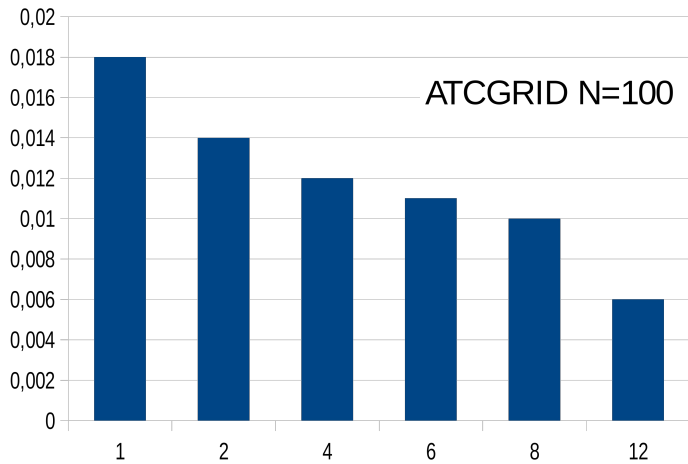
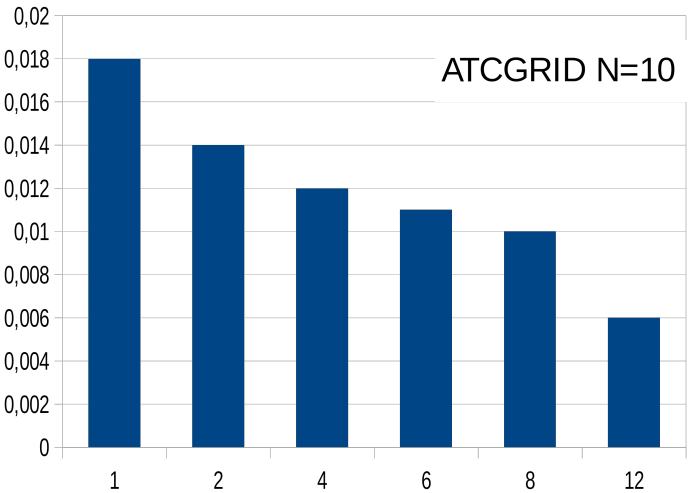
LOCAL N=500



En estos gráficos para PC-LOCAL, podemos comprobar que para un tamaño pequeño ( $N=10$ ), el aumento de cores provoca un rendimiento peor al repartir una tarea simple para una hebra en varias. Sin embargo, para tamaños más significantes, el aumento de hebras consigue mejores resultados en tiempo.

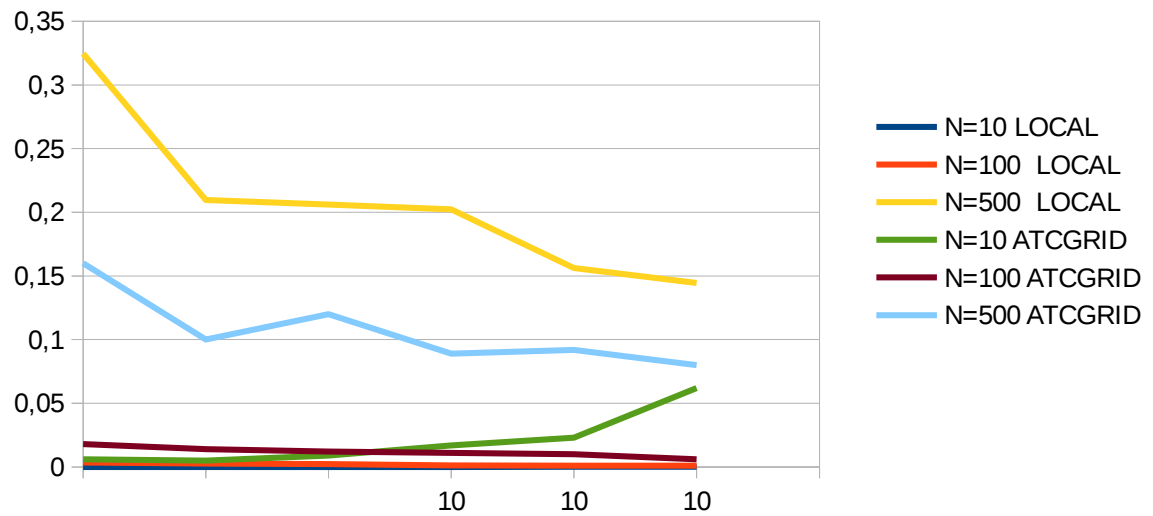
PC - ATCGRID

CORES	1	2	4	6	8	12
N=10	0,006	0,005	0,009	0,017	0,023	0,062
N=100	0,018	0,014	0,012	0,011	0,01	0,006
N=500	0,16	0,1	0,12	0,089	0,092	0,08



En estos gráficos para ATC-GRID, podemos comprobar que para un tamaño pequeño (N=10), el aumento de cores si provoca un rendimiento mejor al repartir una tarea simple para una hebra en varias.Y ademas , para tamaños más significantes, el aumento de hebras consigue mejores resultados en tiempo.

### Comparativa



**Para N=10, el rendimiento en el pc-local y en atcgrid es parejo salvo al aumentar el numero de hebras, donde se dispara el tiempo en atcgrid.**  
**Para N=100, si conseguimos unos tiempos similares en ambos sistemas, siendo ligeramente mejor con menos hebras e igualándose con más.**  
**Para N=500, atcgrid consigue unos mejores resultados desde 1 hebra hasta 12.**