

Grai2º curso / 2º
cuatr.
Grado Ing. Inform.
Doble Grado Ing.
Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Miguel Ángel Cantarero López

Grupo de prácticas:

Fecha de entrega: 19 / 04 / 17

Fecha evaluación en clase:

Ejercicios basados en los ejemplos del seminario práctico

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? (añada una captura de pantalla que muestre lo que ocurre) **(b)** Resuelva el problema generado sin eliminar `default(none)`. Añada el código con la modificación al cuaderno de prácticas.

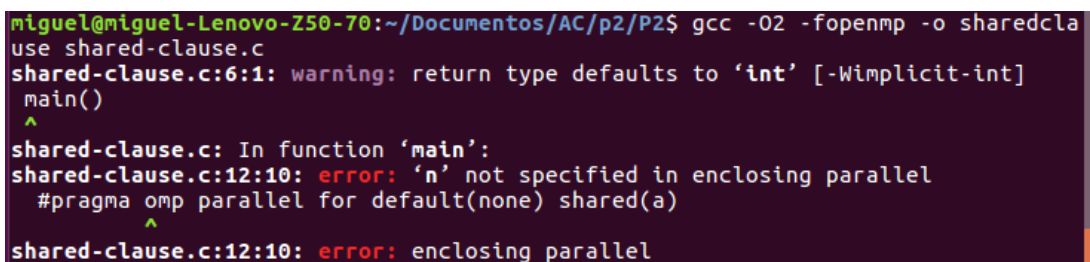
RESPUESTA: Cuando añades `default(none)`, debemos especificar el ámbito de todas las variables de la region `parallel`, por lo que el compilador, nos indica que tenemos que definir dicho ámbito de la variable `n`.

CÓDIGO FUENTE: `shared-clauseModificado.c`

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

int main(int argc, char ** argv)
{
    int i, n = 7;
    int a[n];
    for (i=0; i<n; i++)
        a[i] = i+1;
    #pragma omp parallel for default(none) shared(a,n)
        for (i=0; i<n; i++) a[i] += i;
    printf("Después de parallel for:\n");
    for (i=0; i<n; i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

CAPTURAS DE PANTALLA:



```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2/P2$ gcc -O2 -fopenmp -o sharedcla
use shared-clause.c
shared-clause.c:6:1: warning: return type defaults to 'int' [-Wimplicit-int]
main()
^
shared-clause.c: In function 'main':
shared-clause.c:12:10: error: 'n' not specified in enclosing parallel
#pragma omp parallel for default(none) shared(a)
^
shared-clause.c:12:10: error: enclosing parallel
```

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? (inicialice `suma` a un valor distinto de 0 dentro y fuera de `parallel`) Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA: Como la variable es privada dentro de la región `parallel`, si la inicializamos fuera, esta variable cogerá un valor aleatorio en nuestro pc. Por eso siempre que tengamos una variable privada hay que darle un valor “inicializarla” dentro de la región que sea privada.

CÓDIGO FUENTE: `private-clauseModificado.c`

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

int main(int argc, char ** argv)
{
    int i, n = 7;
    int a[n], suma=8;

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

    #pragma omp parallel private(suma)
    {
        //suma=0;
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];
            printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
        }
        printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
    }
    printf("\nFuera de parallel suma= %d", suma);
    printf("\n");
}
```

CAPTURAS DE PANTALLA:

Inicializada a 1 dentro de la región parallel

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./private-clause
thread 2 suma a[4] / thread 2 suma a[5] / thread 0 suma a[0] / thread 0 suma a[1]
] / thread 1 suma a[2] / thread 1 suma a[3] / thread 3 suma a[6] /
* thread 1 suma= 6
* thread 0 suma= 2
* thread 2 suma= 10
* thread 3 suma= 7
Fuera de parallel suma= 8
```

Inicializada a 1 fuera de la región parallel

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./private-clause
thread 2 suma a[4] / thread 2 suma a[5] / thread 3 suma a[6] / thread 1 suma a[2]
] / thread 1 suma a[3] / thread 0 suma a[0] / thread 0 suma a[1] /
* thread 0 suma= 5
* thread 3 suma= 4196598
* thread 2 suma= 4196601
* thread 1 suma= 4196597
Fuera de parallel suma= 1
```

3. ¿Qué ocurre si en `private-clause.c` se elimina la cláusula `private(suma)`? ¿A qué cree que es debido?

RESPUESTA: Deja de ser una variable privada, y esto implica que todos los threads usan la misma variable, almacenándola todos en el mismo espacio de memoria.

CÓDIGO FUENTE: `private-clauseModificado3.c`

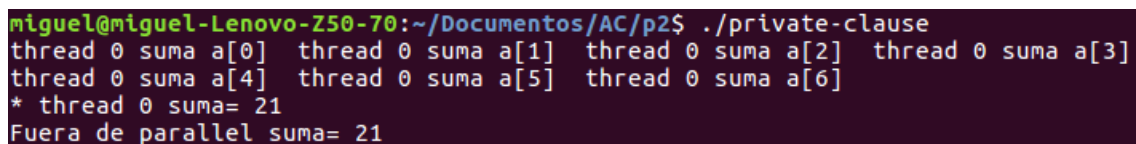
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char ** argv)
{
    int i, n = 7;
    int a[n], suma=8;

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

    #pragma omp parallel private(suma)
    {
        //suma=0;
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];
            printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
        }
        printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
    }
    printf("\nFuera de parallel suma= %d", suma);
    printf("\n");
}
```

CAPTURAS DE PANTALLA:



```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./private-clause
thread 0 suma a[0] thread 0 suma a[1] thread 0 suma a[2] thread 0 suma a[3]
thread 0 suma a[4] thread 0 suma a[5] thread 0 suma a[6]
* thread 0 suma= 21
Fuera de parallel suma= 21
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta.

RESPUESTA:

Sí, porque al usar la cláusula `last`, se devuelve el valor local de la hebra que realiza la última iteración del `for`, como podemos ver en la siguiente captura.

CAPTURAS DE PANTALLA:

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ export OMP_NUM_THREADS=4
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 3 suma a[6] suma=6
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9

Fuera de la construcción parallel suma=6
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 3 suma a[6] suma=6
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9

Fuera de la construcción parallel suma=6
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=6
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./firstlastprivate-clause
thread 3 suma a[6] suma=6
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=6
```

5. ¿Qué ocurre si en copyprivate-clause.c se elimina la cláusula copyprivate(a) en la directiva single? ¿A qué cree que es debido?

RESPUESTA: El resto de hebras no copian la variable privada a, y por tanto no la inicializan, utilizando un valor aleatorio

CÓDIGO FUENTE: copyprivate-clauseModificado.c

```

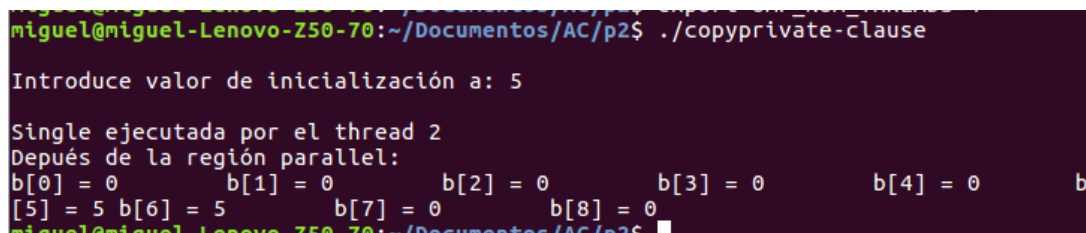
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char ** argv)
{
    int n = 9, i, b[n];

    for (i=0; i<n; i++) b[i] = -1;

    #pragma omp parallel
    {
        int a;
        #pragma omp single
        {
            printf("\nIntroduce valor de inicialización a: ");
            scanf("%d", &a );
            printf("\nSingle ejecutada por el thread
%d\n", omp_get_thread_num());
        }
        #pragma omp for
        for (i=0; i<n; i++) b[i] = a;
    }
    printf("Después de la región parallel:\n");
    for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
    printf("\n");
}

```

CAPTURAS DE PANTALLA:


6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado

RESPUESTA:**CÓDIGO FUENTE:** `reduction-clauseModificado.c`

```

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

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

    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
}

```

```

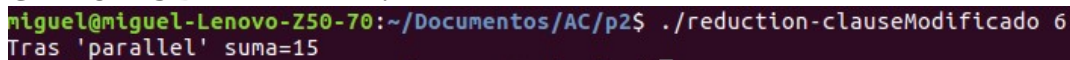
n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d",n);}

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

#pragma omp parallel for reduction(+:suma)
for (i=0; i<n; i++) suma += a[i];

printf("Tras 'parallel' suma=%d\n",suma);
}
}

```

CAPTURAS DE PANTALLA:


```

miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./reduction-clauseModificado 6
Tras 'parallel' suma=15

```



```

miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./reduction-clauseModificado 6
Tras 'parallel' suma=25

```

7. En el ejemplo reduction-clause.c, elimine reduction() de #pragma omp parallel for reduction(+:suma) y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector a en paralelo sin usar directivas de trabajo compartido.

RESPUESTA: He creado una variable local con atomic para cada hebra, y luego sumo todas estas variables locales.

CÓDIGO FUENTE: reduction-clauseModificado7.c

```

#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, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d\n",n);}

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

    #pragma omp parallel
    {
        int sumalocal=0;
        #pragma omp for schedule(static)
        for (i=0; i<n; i++) {
            sumalocal += a[i];
        }
        #pragma omp atomic
        suma+=sumalocal;
    }
    printf("Tras 'parallel' suma=%d\n",suma);
}

```

}

CAPTURAS DE PANTALLA:

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./reduction-clauseModificado7 5
Tras 'parallel' suma=10
```

Resto de ejercicios

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1 (implemente una versión para variables globales y otra para variables dinámicas, use una de estas versiones en los siguientes ejercicios):

$$v2 = M \bullet v1; v2(i) = \sum_{k=0}^{N-1} M(i, k) \bullet v(k), i = 0, \dots, N-1$$

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código paralelo que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CÓDIGO FUENTE: pmv-secuencial.c

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

// #define TIMES
#define PRINTF_ALL

main(int argc, char **argv) {
    // 1. Lectura valores de entrada
    if(argc < 2) {
        fprintf(stderr, "Falta num\n");
        exit(-1);
    }
    int n = atoi(argv[1]);
    if (n > 10000) {
        n = 10000;
        printf("n=%d", n);
    }
    int i, j;
    struct timespec ini, fin; double transcurrido;

    // 2. Creación e inicialización de vector y matriz
    // Creación
    int *v1, *v2;
    v1 = (int*) malloc(n*sizeof(double));
    v2 = (int*) malloc(n*sizeof(double));
```

```

int **M;
M = (int**) malloc(n*sizeof(int*));
for(i=0;i<n;i++)
    M[i] = (int*)malloc(n*sizeof(int));

// Inicialición
for(i=0;i<n;i++)
    v1[i]=i;

for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        (M[i])[j]=v1[j]+n*i;
    }
}

// 3. Impresión de vector y matriz
#ifdef TIMES
#ifdef PRINTF_ALL
    printf("Vector inicial:\n");
    for (i=0; i<n; i++) printf("%d ",v1[i]);
    printf("\n");

    printf("Matriz inicial:\n");
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            if(M[i][j]<10) printf(" %d ",M[i][j]);
            else printf("%d ",M[i][j]);
        }
        printf("\n");
    }
#endif
#endif

// 4. Cálculo resultado
clock_gettime(CLOCK_REALTIME,&ini);
for (i=0; i<n; i++) {
    v2[i]=0;
    for (j=0; j<n; j++) {
        v2[i]+=M[i][j]*v1[j];
    }
}
clock_gettime(CLOCK_REALTIME,&fin);
transcurrido=(double) (fin.tv_sec-ini.tv_sec)+(double)
((fin.tv_nsec-ini.tv_nsec)/(1.e+9));

// 5. Impresión de vector resultado
#ifdef TIMES
    printf("%d %11.9f\n",n,transcurrido);
#else
#ifdef PRINTF_ALL
    printf("Tiempo: %11.9f\n",transcurrido);
    printf("Vector resultado (M x v1):\n");
    for (i=0; i<n; i++) printf("%d ",v2[i]);
    printf("\n");
#else
    printf("Tiempo: %11.9f\n",transcurrido);
#endif
#endif

```



```

        printf("v2[0]: %d, v2[n-1]: %d\n",v2[0],v2[n-1]);
    #endif
#endif

// 6. Eliminar de memoria
free(M);
free(v1);
free(v2);
}

```

CAPTURAS DE PANTALLA:

```

miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./pmv-secuencial 5
Vector inicial:
0 1 2 3 4
Matriz inicial:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
Tiempo: 0.000000558
Vector resultado (M x v1):
30 80 130 180 230

```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for`. Debe implementar dos versiones del código (consulte la lección 5/Tema 2):
- una primera que paralelice el bucle que recorre las filas de la matriz y
 - una segunda que paralelice el bucle que recorre las columnas.

Use las directivas que estime oportunas y las cláusulas que sean necesarias **excepto la cláusula `reduction`**. Se debe paralelizar también la inicialización de las matrices. Respecto a este ejercicio:

- Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
- Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, $v3$, para tamaños pequeños de los vectores (por ejemplo, $N = 8$ y $N=11$); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CÓDIGO FUENTE : pmv-OpenMP-a.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef _OPENMP
    #include <omp.h>
#else

```

```

#define omp_get_thread_num() 0
#endif

// #define TIMES
#define PRINTF_ALL

main(int argc, char **argv) {
    // 1. Lectura valores de entrada
    if(argc < 2) {
        fprintf(stderr, "Falta num\n");
        exit(-1);
    }
    int n = atoi(argv[1]);
    if (n>50000) {
        n=10000;
        printf("n=%d",n);
    }
    int i, j;
    struct timespec ini,fin; double transcurrido;

    // 2. Creación e inicialización de vector y matriz
    // Creación
    int *v1,*v2;
    v1 = (int*) malloc(n*sizeof(double));
    v2 = (int*) malloc(n*sizeof(double));

    int **M;
    M = (int**) malloc(n*sizeof(int*));
    for(i=0;i<n;i++)
        M[i] = (int*)malloc(n*sizeof(int));

    // Inicialición
    for(i=0;i<n;i++)
        v1[i]=i;

    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            (M[i])[j]=v1[j]+n*i;
        }
    }

    // 3. Impresión de vector y matriz
    #ifndef TIMES
    #ifndef PRINTF_ALL
        printf("Vector inicial:\n");
        for (i=0; i<n; i++) printf("%d ",v1[i]);
        printf("\n");

        printf("Matriz inicial:\n");
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                if(M[i][j]<10) printf(" %d ",M[i][j]);
                else printf("%d ",M[i][j]);
            }
            printf("\n");
        }
    }
}

```

```

    #endif
#endif

// 4. Cálculo resultado
clock_gettime(CLOCK_REALTIME,&ini);
#pragma omp parallel for default(none) private(i,j)
shared(n,v1,M,v2)
for (i=0; i<n; i++) {
    v2[i]=0;
    for (j=0; j<n; j++) {
        v2[i]+=M[i][j]*v1[j];
    }
}
clock_gettime(CLOCK_REALTIME,&fin);
transcurrido=(double) (fin.tv_sec-ini.tv_sec)+(double)
((fin.tv_nsec-ini.tv_nsec)/(1.e+9));

// 5. Impresión de vector resultado
#ifdef TIMES
    printf("%d %11.9f\n",n,transcurrido);
#else
    #ifdef PRINTF_ALL
        printf("Tiempo: %11.9f\n",transcurrido);
        printf("Vector resultado (M x v1):\n");
        for (i=0; i<n; i++) printf("%d ",v2[i]);
        printf("\n");
    #else
        printf("Tiempo: %11.9f\n",transcurrido);
        printf("v2[0]: %d, v2[n-1]: %d\n",v2[0],v2[n-1]);
    #endif
#endif
#endif

// 6. Eliminar de memoria
free(M);
free(v1);
free(v2);
}

```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```

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

// #define TIMES

```

```

#define PRINTF_ALL

main(int argc, char **argv) {
    // 1. Lectura valores de entrada
    if(argc < 2) {
        fprintf(stderr, "Falta num\n");
        exit(-1);
    }
    int n = atoi(argv[1]);
    if (n>10000) {
        n=10000;
        printf("n=%d",n);
    }
    int i, j, sumalocal;
    struct timespec ini,fin; double transcurrido;

    // 2. Creación e inicialización de vector y matriz
    // Creación
    int *v1,*v2;
    v1 = (int*) malloc(n*sizeof(double));
    v2 = (int*) malloc(n*sizeof(double));

    int **M;
    M = (int**) malloc(n*sizeof(int*));
    for(i=0;i<n;i++)
        M[i] = (int*)malloc(n*sizeof(int));

    // Inicialición
    for(i=0;i<n;i++)
        v1[i]=i;

    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            (M[i])[j]=v1[j]+n*i;
        }
    }

    // 3. Impresión de vector y matriz
    #ifndef TIMES
    #ifdef PRINTF_ALL
        printf("Vector inicial:\n");
        for (i=0; i<n; i++) printf("%d ",v1[i]);
        printf("\n");

        printf("Matriz inicial:\n");
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                if(M[i][j]<10) printf(" %d ",M[i][j]);
                else printf("%d ",M[i][j]);
            }
            printf("\n");
        }
    #endif
    #endif

    // 4. Cálculo resultado

```

```

clock_gettime(CLOCK_REALTIME,&ini);
for (i=0; i<n; i++) {
    v2[i]=0;
    #pragma omp parallel default(none) shared(i,n,v1,M,v2)
private(sumalocal)
    {
        sumalocal=0;
        #pragma omp for schedule(static)
        for (j=0; j<n; j++) {
            sumalocal+=M[i][j]*v1[j];
        }
        #pragma omp atomic
        v2[i]+=sumalocal;
    }
}
clock_gettime(CLOCK_REALTIME,&fin);
transcurrido=(double) (fin.tv_sec-ini.tv_sec)+(double)
((fin.tv_nsec-ini.tv_nsec)/(1.e+9));

// 3. Impresión de vector resultado
#ifdef TIMES
    printf("%d %11.9f\n",n,transcurrido);
#else
    #ifdef PRINTF_ALL
        printf("Tiempo: %11.9f\n",transcurrido);
        printf("Vector resultado (M x v1):\n");
        for (i=0; i<n; i++) printf("%d ",v2[i]);
        printf("\n");
    #else
        printf("Tiempo: %11.9f\n",transcurrido);
        printf("v2[0]: %d, v2[n-1]: %d\n",v2[0],v2[n-1]);
    #endif
#endif

// 6. Eliminar de memoria
free(M);
free(v1);
free(v2);
}

```

RESPUESTA: Al principio pensé en introducir una directiva `single` dentro del `for`, pero conforme lo iba programando vi que no tenía sentido, aparte que no iba a compilar, por tanto cambie a `private` en las filas y `shared` en las columnas. Salvo eso , un par de errores sintácticos y el tamaño máximo de vector que me daba error y tuve que cambiarlo.

CAPTURAS DE PANTALLA:

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./pmv-OpenMP-a 10
Vector inicial:
0 1 2 3 4 5 6 7 8 9
Matriz inicial:
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
Tiempo: 0.005283254
Vector resultado (M x v1):
285 735 1185 1635 2085 2535 2985 3435 3885 4335
```

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./pmv-OpenMP-b 10
Vector inicial:
0 1 2 3 4 5 6 7 8 9
Matriz inicial:
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
Tiempo: 0.003349898
Vector resultado (M x v1):
285 735 1185 1635 2085 2535 2985 3435 3885 4335
```

10. A partir de la segunda versión de código paralelo desarrollado en el ejercicio anterior, implementar una versión paralela del producto matriz por vector con OpenMP que use para comunicación/sincronización la cláusula `reduction`. Respecto a este ejercicio:

- Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
- Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

CÓDIGO FUENTE: pmv-OpenMMP-reduction.c

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

// #define TIMES
#define PRINTF_ALL

main(int argc, char **argv) {
    // 1. Lectura valores de entrada
    if(argc < 2) {
        fprintf(stderr, "Falta num\n");
        exit(-1);
    }
    int n = atoi(argv[1]);
    if (n > 500000) {
        n = 100000;
        printf("n=%d", n);
    }
    int i, j, sumalocal;
    struct timespec ini, fin; double transcurrido;

    // 2. Creación e inicialización de vector y matriz
    // Creación
    int *v1, *v2;
    v1 = (int*) malloc(n*sizeof(double));
    v2 = (int*) malloc(n*sizeof(double));

    int **M;
    M = (int**) malloc(n*sizeof(int*));
    for(i=0; i<n; i++)
        M[i] = (int*) malloc(n*sizeof(int));

    // Inicialización
    for(i=0; i<n; i++)
        v1[i] = i;

    for(i=0; i<n; i++){
        for(j=0; j<n; j++){
            (M[i])[j] = v1[j] + n*i;
        }
    }

    // 3. Impresión de vector y matriz
```

```

#ifndef TIMES
#ifdef PRINTF_ALL
printf("Vector inicial:\n");
for (i=0; i<n; i++) printf("%d ",v1[i]);
printf("\n");

printf("Matriz inicial:\n");
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        if(M[i][j]<10) printf(" %d ",M[i][j]);
        else printf("%d ",M[i][j]);
    }
    printf("\n");
}
#endif
#endif

// 4. Cálculo resultado
clock_gettime(CLOCK_REALTIME,&ini);
for (i=0; i<n; i++) {
    sumalocal=0;
    #pragma omp parallel for reduction(+:sumalocal)
    for (j=0; j<n; j++) {
        sumalocal+=M[i][j]*v1[j];
    }
    v2[i]=sumalocal;
}
clock_gettime(CLOCK_REALTIME,&fin);
transcurrido=(double) (fin.tv_sec-ini.tv_sec)+(double)
((fin.tv_nsec-ini.tv_nsec)/(1.e+9));

// 5. Impresión de vector resultado
#ifdef TIMES
printf("%d %11.9f\n",n,transcurrido);
#else
#ifdef PRINTF_ALL
printf("Tiempo: %11.9f\n",transcurrido);
printf("Vector resultado (M x v1):\n");
for (i=0; i<n; i++) printf("%d ",v2[i]);
printf("\n");
#else
printf("Tiempo: %11.9f\n",transcurrido);
printf("v2[0]: %d, v2[n-1]: %d\n",v2[0],v2[n-1]);
#endif
#endif

// 6. Eliminar de memoria
free(M);
free(v1);
free(v2);
}

```

RESPUESTA:**CAPTURAS DE PANTALLA:**


```
miguel@miguel-Lenovo-Z50-70:~/Documentos/AC/p2$ ./pmv-OpenMP-reduction 10
Vector inicial:
0 1 2 3 4 5 6 7 8 9
Matriz inicial:
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
Tiempo: 0.004442749
Vector resultado (M x v1):
285 735 1185 1635 2085 2535 2985 3435 3885 4335
```

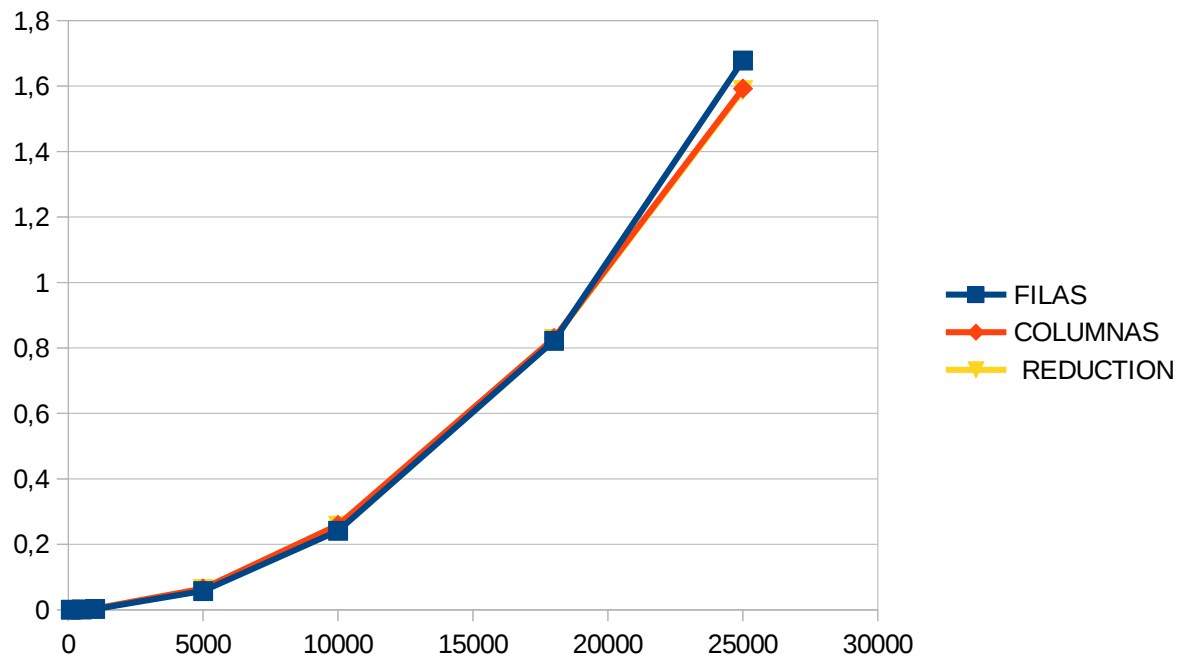
11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en el PC local del mejor código paralelo de los tres implementados en los ejercicios anteriores para dos tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar -O2 al compilar. Justificar por qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

TABLA Y GRÁFICA (por ejemplo para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N: alguno del orden de cientos de miles):

Primero he realizado una ejecución de los tres códigos en mi PC local y en ATCGRID para comprobar cual era el mejor de los 3, obteniendo los siguientes resultados:

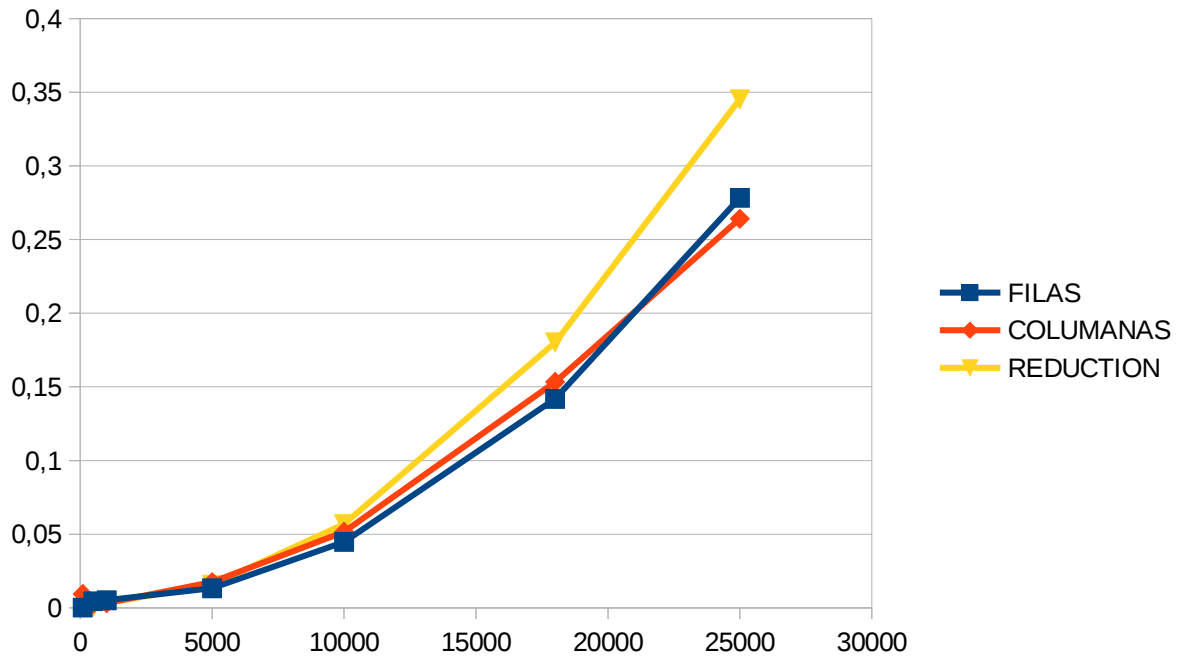
ATCGRID:

Tamaño	FILAS	COLUMNAS	REDUCTION
100	0,000036038	0,000135493	0,000087163
500	0,000752338	0,001180609	0,000986449
1000	0,002486935	0,003499086	0,003093444
5000	0,057285274	0,065281627	0,063984529
10000	0,241406474	0,259107517	0,256709732
18000	0,821642632	0,830555313	0,827506947
25000	1,678354066	1,592602576	1,587455129



PC LOCAL :

Tamaño	FILAS	COLUMNAS	REDUCTION
100	0,000144058	0,009436072	0,000081373
500	0,004500423	0,002906111	0,000911741
1000	0,005135648	0,003151866	0,003152619
5000	0,013315871	0,017455015	0,015645139
10000	0,044760444	0,051573935	0,056835508
18000	0,141865652	0,153387322	0,180181376
25000	0,278320946	0,264070573	0,345236563

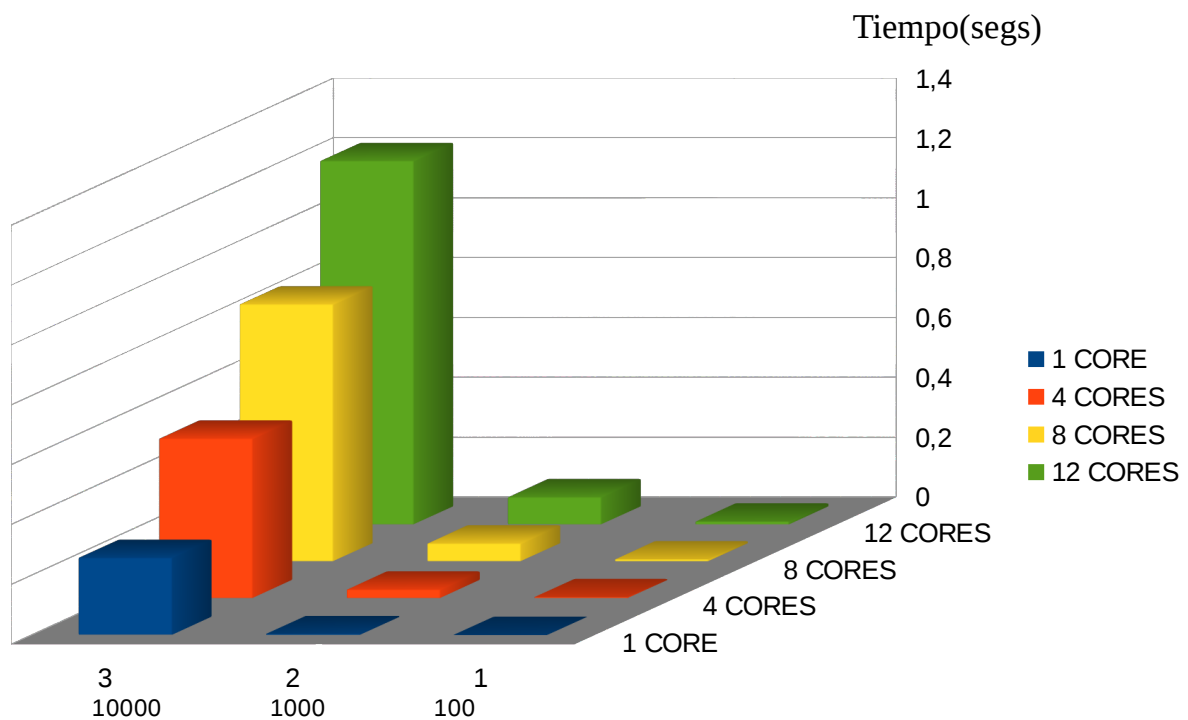


Vemos que en atcgrid, no se aprecia apenas diferencia entre los códigos, aunque el de columnas es el más rápido de los 3, pero en el pc local, sin aplicar ninguna restricción a los cores, vemos de forma más clara la eficiencia de los 3 códigos, reafirmandose que el pmv-OpenMP-b , que calcula el producto por columnas es un poco más rápido que el resto.

Ahora ejecuto con un script el ejecutable pmv-OpenMP-b modificando la variable OMP_NUM_THREADS según nos sea conveniente para ver qué ganancias obtenemos cuando aumentamos el número de cores.

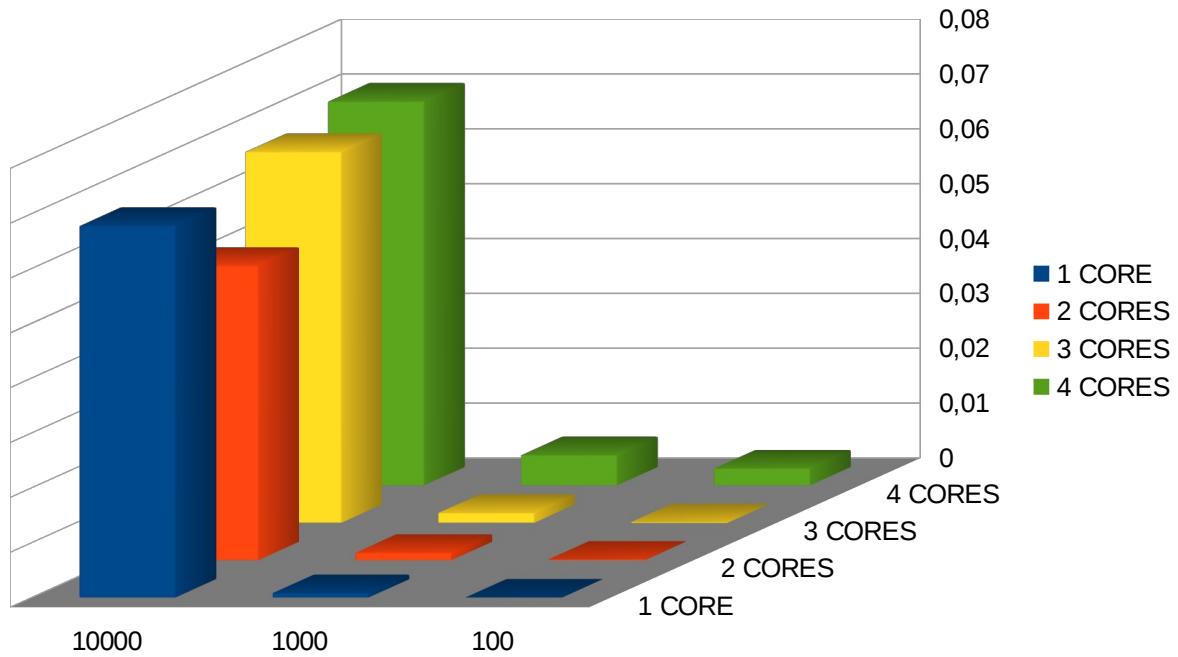
ATCGRID pmv-OpenMP-b.c:

Tamaño	1 CORE	4 CORES	8 CORES	12 CORES
100	0,000112586	0,003788791	0,007181437	0,010619341
1000	0,003366107	0,029240066	0,060145446	0,091824116
10000	0,258444861	0,53376188	0,859371589	1,215466229



PC-local:

Tamaño	1 CORE	2 CORES	3 CORES	4 CORES
100	0,000041939	0,000149494	0,000153041	0,003171698
1000	0,000868739	0,00140924	0,001867591	0,005590494
10000	0,067831606	0,053723283	0,067709593	0,070089754



COMENTARIOS SOBRE LOS RESULTADOS: Los scripts que he utilizado para la ejecución de los programas están adjuntos en el .zip de la práctica.

La verdad me puede resultar un poco extraño que al aumentar el número de cores, aumente el tiempo de ejecución de la función, pero por otra parte, pienso que con un tamaño de matriz y vector grandes, el tiempo aumenta ya que se tiene que dividir mucho más trabajo entre las hebras, cosa que penaliza mucho la ejecución, aunque luego el cálculo sea más rápido al tenerlo más dividido. Y para valores pequeños no se aprecia mucho la mejora, ya que el tiempo de cálculo es inapreciable con respecto al de E/S entre hebras.