Cuaderno de prácticas
de Arquitectura de Computadores
Grado en Ingeniería Informática

Memoria
Bloque Práctico 2

Alumno: Miguel Sánchez Tello
DNI: 75574961C
Grupo: *B1*

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? Si se plantea algún problema, resuélvalo sin eliminar `default(none)`.
   RESPUESTA: Dependiendo del lugar donde pongamos la cláusula *default*, tendrá un efecto u otro:
   -Si ponemos *shared(a) default(none)*, estamos anulando el efecto de *shared(a)* al poner todas las variables no-compartidas inmediatamente después con *default(none)*.
   -Si ponemos *default(none) shared(a)*, primero estamos poniendo a no-compartidas todas las variables, a excepción de *a*. Dicha excepción se consigue poniendo la cláusula *shared(a)* inmediatamente después.

   CÓDIGO FUENTE: `shared-clauseModificado.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
main()
{
		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)
		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:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio1$ ./shared-claus
e
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio1$ gcc -O2 -o sha
red-clauseMOD shared-clauseMOD.c
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio1$ ./shared-claus
eMOD
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
```

En este caso, al ponerlo en el orden correcto, no se produce ningún contratiempo.

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? Razone su respuesta.
   RESPUESTA: Dicho valor se perderá, pues *private(suma)* convierte la variable *suma* en privada para cada hebra, pero sin inicializar. Es decir, no importa que la inicialicemos antes, su valor será tomado en cuenta igualmente.

CÓDIGO FUENTE: `private-clauseModificado.c`

```
#include <stdio.h>

#ifdef _OPENMP

#include <omp.h>

#else

#define omp_get_thread_num() 0

#endif

main()

{

        int i, n = 7;

        int a[n], suma;
```

```
		for (i=0; i<n; i++)
			a[i] = i;
		suma=0;
		#pragma omp parallel private(suma)
		{
			#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("\n");
}
```

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio2$ ./private-clau
se
thread 1 suma a[2] /thread 1 suma a[3] /thread 3 suma a[6] /t
hread 2 suma a[4] /thread 2 suma a[5] /thread 0 suma a[0] /th
read 0 suma a[1] /
* thread 1 suma= 5
* thread 2 suma= 9
* thread 0 suma= 1
* thread 3 suma= 6
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio2$ gcc -O2 -fopen
mp -o private-clauseMOD private-clauseMOD.c
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio2$ ./private-clau
seMOD
thread 0 suma a[0] /thread 0 suma a[1] /thread 1 suma a[2] /t
hread 1 suma a[3] /thread 2 suma a[4] /thread 2 suma a[5] /th
read 3 suma a[6] /
* thread 1 suma= 4196517
* thread 3 suma= 4196518
* thread 2 suma= 4196521
* thread 0 suma= -1229545791
```

3. ¿Qué ocurre si en `private-clause.c` se elimina la cláusula `private(suma)`? ¿A qué cree que es debido?
   RESPUESTA: *suma* ahora se convierte en una variable compartida. Su valor será ahora la suma total de todas las "pequeñas sumas" que obteníamos en la versión *private-clause.c* original. Todas las hebras dejan su resultado acumulado en *suma*, así hasta llegar a 15 (con 4 hebras), en lugar de quedarse su propio resultado para ellas.

   CÓDIGO FUENTE: `private-clauseModificado3.c`

```
#include <stdio.h>

#ifdef _OPENMP

#include <omp.h>

#else

#define omp_get_thread_num() 0

#endif

main()

{

        int i, n = 7;

        int a[n], suma;



        for (i=0; i<n; i++)

                a[i] = i;



        #pragma omp parallel

        {

                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");
                                    }
                                    printf("\n* thread %d suma= %d", omp_get_thread_num(),
suma);
                    }
                    printf("\n");
}
```

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio3$ gcc -O2 -fopenm
p -o private-clauseMOD2 private-clauseMOD2.c
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio3$ gcc -O2 -fopenm
p -o private-clause private-clause.c
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio3$ ./private-claus
e
thread 1 suma a[2] /
thread 2 suma a[4] /
thread 2 suma a[5] /
thread 0 suma a[0] /
thread 0 suma a[1] /
thread 3 suma a[6] /
thread 1 suma a[3] /

* thread 0 suma= 1
* thread 2 suma= 9
* thread 1 suma= 5
* thread 3 suma= 6
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio3$ ./private-claus
eMOD2
thread 1 suma a[2] /
thread 1 suma a[3] /
thread 3 suma a[6] /
thread 2 suma a[4] /
thread 2 suma a[5] /
thread 0 suma a[0] /
thread 0 suma a[1] /

* thread 2 suma= 15
* thread 0 suma= 15
* thread 1 suma= 15
* thread 3 suma= 15
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6. ¿El código imprime siempre 6? Razone su respuesta.
   RESPUESTA: *Lastprivate* guarda el valor de la última iteración en la que se modifique la variable implicada, en nuestro caso *suma*.

   Siempre imprime 6 porque es el valor de la suma de la última iteración: 0 + 6
   6 es el valor de la última posición del vector.

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio4$ export OMP_DYNAMIC=FALSE
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio4$ ./firstlastprivate-clause
 thread 0 suma a[0] suma=0
 thread 3 suma a[3] suma=3
 thread 5 suma a[5] suma=5
 thread 4 suma a[4] suma=4
 thread 1 suma a[1] suma=1
 thread 2 suma a[2] suma=2
 thread 6 suma a[6] suma=6

Fuera de la construcción parallel suma=6
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio4$ export OMP_DYNAMIC=TRUE
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio4$ ./firstlastprivate-clause
 thread 3 suma a[3] suma=3
 thread 2 suma a[2] suma=2
 thread 6 suma a[6] suma=6
 thread 0 suma a[0] suma=0
 thread 5 suma a[5] suma=5
 thread 4 suma a[4] suma=4
 thread 1 suma a[1] suma=1

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: Lo que ocurrirá es que *a* tomará el valor que me indiquemos en la hebra que ejecute el *single*, pero en el resto de hebras, *a* seguirá valiendo lo que quiera que valga anteriormente.

CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```
#include <stdio.h>

#include <omp.h>



main() {

            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("Depués de la región parallel:\n");
            for (i=0; i<n; i++){
                        printf("b[%d] = %d\t",i,b[i]);
                        printf("\n");
            }
                        printf("\n");
}
```

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio5$ ./copyprivate-clause

Introduce valor de inicialización a: 8

Single ejecutada por el thread 1
Depués de la región parallel:
b[0] = 8
b[1] = 8
b[2] = 8
b[3] = 8
b[4] = 8
b[5] = 8
b[6] = 8
b[7] = 8
b[8] = 8

loadge@loadge-K53SV:~/AC/Práctica2/ejercicio5$ ./copyprivate-clauseMOD

Introduce valor de inicialización a: 8

Single ejecutada por el thread 1
Depués de la región parallel:
b[0] = 32767
b[1] = 32767
b[2] = 8
b[3] = 8
b[4] = 0
b[5] = 0
b[6] = 0
b[7] = 0
b[8] = 0
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado
   RESPUESTA: Como cabe esperar, ahora la suma se inicia desde 10, por lo que se imprimirá un resultado 10 unidades más alto.

   CÓDIGO FUENTE: `reduction-clauseModificado.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=10;

            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);
}



main(int argc, char **argv) {



            int i, n=20, a[n],suma=10;



            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:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio6$ ./reduction-clauseMOD 20
Tras 'parallel' suma=200
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio6$ ./reduction-clause 20
Tras 'parallel' suma=190
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio6$ ./reduction-clause 14
Tras 'parallel' suma=91
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio6$ ./reduction-clauseMOD 14
Tras 'parallel' suma=101
```

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.
   RESPUESTA: Simplemente, convertimos *suma* en una variable compartida para todas las hebras al no poner ninguna cláusula en el *parallel* que le involucre. Así, *suma* irá almacenando el valor de cada posición del vector.

   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);}

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

            #pragma omp parallel
            {
            #pragma omp for
                        for (i=0; i<n; i++){
                                    suma += a[i];
                                    //printf("\nSuma del thread %d que
contiene %d de la posicion %d: %d", omp_get_thread_num(), a[i], i,
suma);
                        }
            }
            printf("\nTras 'parallel' suma=%d\n",suma);
}
```

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio7$ export OMP_DYNAMIC=FALSE
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio7$ export OMP_NUM_THREADS=3
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio7$ ./reduction-clauseMOD2 10

Suma del thread 0 que contiene 0 de la posicion 0: 0
Suma del thread 0 que contiene 1 de la posicion 1: 13
Suma del thread 0 que contiene 2 de la posicion 2: 15
Suma del thread 0 que contiene 3 de la posicion 3: 18
Suma del thread 1 que contiene 4 de la posicion 4: 12
Suma del thread 1 que contiene 5 de la posicion 5: 23
Suma del thread 1 que contiene 6 de la posicion 6: 29
Suma del thread 1 que contiene 7 de la posicion 7: 36
Suma del thread 2 que contiene 8 de la posicion 8: 8
Suma del thread 2 que contiene 9 de la posicion 9: 45
Tras 'parallel' suma=45
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio7$ ./reduction-clauseMOD2 10

Suma del thread 0 que contiene 0 de la posicion 0: 0
Suma del thread 0 que contiene 1 de la posicion 1: 13
Suma del thread 0 que contiene 2 de la posicion 2: 15
Suma del thread 0 que contiene 3 de la posicion 3: 18
Suma del thread 2 que contiene 8 de la posicion 8: 12
Suma del thread 2 que contiene 9 de la posicion 9: 27
Suma del thread 1 que contiene 4 de la posicion 4: 4
Suma del thread 1 que contiene 5 de la posicion 5: 32
Suma del thread 1 que contiene 6 de la posicion 6: 38
Suma del thread 1 que contiene 7 de la posicion 7: 45
Tras 'parallel' suma=45
```

```
Suma del thread 2 que contiene 14 de la posicion 14: 14
Suma del thread 2 que contiene 15 de la posicion 15: 120
Suma del thread 2 que contiene 16 de la posicion 16: 136
Suma del thread 2 que contiene 17 de la posicion 17: 153
Suma del thread 2 que contiene 18 de la posicion 18: 171
Suma del thread 2 que contiene 19 de la posicion 19: 190
Tras 'parallel' suma=190
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio7$ ./reduction-clauseMOD2 20

Suma del thread 0 que contiene 0 de la posicion 0: 0
Suma del thread 0 que contiene 1 de la posicion 1: 22
Suma del thread 0 que contiene 2 de la posicion 2: 24
Suma del thread 0 que contiene 3 de la posicion 3: 27
Suma del thread 0 que contiene 4 de la posicion 4: 31
Suma del thread 0 que contiene 5 de la posicion 5: 36
Suma del thread 0 que contiene 6 de la posicion 6: 42
Suma del thread 1 que contiene 7 de la posicion 7: 7
Suma del thread 1 que contiene 8 de la posicion 8: 50
Suma del thread 1 que contiene 9 de la posicion 9: 59
Suma del thread 1 que contiene 10 de la posicion 10: 69
Suma del thread 1 que contiene 11 de la posicion 11: 80
Suma del thread 1 que contiene 12 de la posicion 12: 92
Suma del thread 1 que contiene 13 de la posicion 13: 105
Suma del thread 2 que contiene 14 de la posicion 14: 21
Suma del thread 2 que contiene 15 de la posicion 15: 120
Suma del thread 2 que contiene 16 de la posicion 16: 136
Suma del thread 2 que contiene 17 de la posicion 17: 153
Suma del thread 2 que contiene 18 de la posicion 18: 171
Suma del thread 2 que contiene 19 de la posicion 19: 190
Tras 'parallel' suma=190
```

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por vector, v1:

$$\text{v2= M·v1; } \text{v2}(i)=\sum_{k=0}^{N-1} M(\text{i,k})\cdot v(k),\ \text{i=0,…N-1}$$

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`

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
// gcc -O2 -fopenmp paralelo1.c -o paralelo1

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

int main(int argc, char **argv){
            int i=0,j=0,n=0, suma=0;
            double cgt1,cgt2,ncgt;

            if(argc < 2) {
                        fprintf(stderr,"\nNumero de argumentos
incorrecto.\n");
                        exit(-1);
            }

            n = atoi(argv[1]);

            double **matriz = (double**)malloc(n*sizeof(double));

            for(i=0;i<n;i++)
                        matriz[i] = (double*)malloc(n*sizeof(double));

            double *vector = (double*) malloc(n*sizeof(double));

            double resultado[n];

            #pragma omp parallel
            {

            //inicializamos la matriz
            #pragma omp for
            for(i = 0;i < n;i++)
                        for(j = 0;j < n;j++)
                                    matriz[i][j] = i+j;


            //inicializamos el vector
```

```
            #pragma omp for
            for(i = 0;i < n;i++)
                        vector[i] = i;

            //inicializamos el vector resultados
            #pragma omp for
            for(i = 0;i < n;i++)
                        resultado[i] = 0;

            }

            /*//muestra la matriz
            for(i = 0;i < n;i++){
                        for(j = 0;j < n;j++)
                                    printf("%d ",matriz[i][j]);
                        printf("\n");
            }*/
            #pragma omp parallel private(j)
            {

            cgt1 = omp_get_wtime();

            #pragma omp for
            for(i = 0;i < n;i++){
                        for(j = 0;j < n;j++){
                                    resultado[i] += (matriz[i][j] *
vector[j]);
                        }
            }

            cgt2 = omp_get_wtime();

            ncgt= cgt2-cgt1 + ((cgt2 - cgt1)/(1.e+9));

            }
            //muestra el resultado
            for(i = 0;i < n;i++)
                        printf("\n Elemento %i %f ",i,resultado[i]);


            printf("\nTiempo(seg.):%11.9f\t\n", ncgt);
}
```

CAPTURAS DE PANTALLA:
PC LOCAL:

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

CÓDIGO FUENTE : `pmv-OpenMP-a.c`
(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

```
// gcc -O2 -fopenmp paralelo1.c -o paralelo1

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

int main(int argc, char **argv){
            int i=0,j=0,n=0, suma=0;
            double cgt1,cgt2,ncgt;

            if(argc < 2) {
                        fprintf(stderr,"\nNumero de argumentos
incorrecto.\n");
                        exit(-1);
            }

            n = atoi(argv[1]);

            double **matriz = (double**)malloc(n*sizeof(double));

            for(i=0;i<n;i++)
                        matriz[i] = (double*)malloc(n*sizeof(double));

            double *vector = (double*) malloc(n*sizeof(double));

            double resultado[n];
```

```
            #pragma omp parallel
            {

            //inicializamos la matriz
            #pragma omp for
            for(i = 0;i < n;i++)
                        for(j = 0;j < n;j++)
                                    matriz[i][j] = i+j;


            //inicializamos el vector
            #pragma omp for
            for(i = 0;i < n;i++)
                        vector[i] = i;

            //inicializamos el vector resultados
            #pragma omp for
            for(i = 0;i < n;i++)
                        resultado[i] = 0;

            }

            /*//muestra la matriz
            for(i = 0;i < n;i++){
                        for(j = 0;j < n;j++)
                                    printf("%d ",matriz[i][j]);
                        printf("\n");
            }*/
            #pragma omp parallel private(j)
            {

            cgt1 = omp_get_wtime();
            omp_set_num_threads(n);

            #pragma omp for
            for(i = 0;i < n;i++){
                        for(j = 0;j < n;j++){
                                    resultado[i] += (matriz[i][j] *
vector[j]);
                        }
            }

            cgt2 = omp_get_wtime();

            ncgt= cgt2-cgt1 + ((cgt2 - cgt1)/(1.e+9));

            }
            //muestra el resultado
            for(i = 0;i < n;i++)
                        printf("\n Elemento %i %f ",i,resultado[i]);


            printf("\nTiempo(seg.):%11.9f\t\n", ncgt);
}
```

CÓDIGO FUENTE: pmv-OpenMP-b.c
(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

int main(int argc, char **argv){
            int i=0,j=0,n=0, suma=0;
            double cgt1,cgt2,ncgt;

            if(argc < 2) {
                        fprintf(stderr,"\nNumero de argumentos
incorrecto.\n");
                        exit(-1);
            }

            n = atoi(argv[1]);

            double **matriz = (double**)malloc(n*sizeof(double));

            for(i=0;i<n;i++)
                        matriz[i] = (double*)malloc(n*sizeof(double));

            double *vector = (double*) malloc(n*sizeof(double));

            double resultado[n];

            #pragma omp parallel
            {

            //inicializamos la matriz
            #pragma omp for
            for(i = 0;i < n;i++)
                        for(j = 0;j < n;j++)
                                    matriz[i][j] = i+j;


            //inicializamos el vector
            #pragma omp for
            for(i = 0;i < n;i++)
                        vector[i] = i;

            //inicializamos el vector resultados
            #pragma omp for
            for(i = 0;i < n;i++)
                        resultado[i] = 0;

            }

            /*//muestra la matriz
            for(i = 0;i < n;i++){
                        for(j = 0;j < n;j++)
                                    printf("%d ",matriz[i][j]);
                        printf("\n");
            }*/
            #pragma omp parallel private(i)
```

```
            {

            cgt1 = omp_get_wtime();
            omp_set_num_threads(n);
            for(i = 0;i < n;i++){
                        #pragma omp for
                        for(j = 0;j < n;j++){
                                    #pragma omp atomic
                                    resultado[i] += (matriz[i][j] *
vector[j]);
                        }
            }

            cgt2 = omp_get_wtime();

            ncgt= cgt2-cgt1 + ((cgt2 - cgt1)/(1.e+9));

            }
            //muestra el resultado
            for(i = 0;i < n;i++)
                        printf("\n Elemento %i %f ",i,resultado[i]);


            printf("\nTiempo(seg.):%11.9f\t\n", ncgt);
}
```

RESPUESTA: Paralelizaremos, en ambas versiones, las inicializaciones y las operaciones del bucle principal.

-Para paralelizar las filas, procedemos tal y como aparece en la diapositiva de la Lección 5 del Tema 2, privatizando la variable j del bucle y asignando una hebra por fila.

-Para paralelizar las columnas, debemos cambiar de sitio la cláusula *#pragma omp for* y ponerla anidada en el primer bucle. Además, se debe poner la línea de la operación principal del programa (la que ejecuta el 2º bucle *for*) en una sección crítica para no falsear los datos.

He recibido ayuda externa de un compañero.

CAPTURAS DE PANTALLA:

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=9.000000 /
vSuma[1]=6.000000 /
vSuma[2]=15.000000 /
vSuma[3]=18.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=4.000000 /
vSuma[1]=20.000000 /
vSuma[2]=0.000000 /
vSuma[3]=18.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=10.000000 /
vSuma[1]=14.000000 /
vSuma[2]=11.000000 /
vSuma[3]=4.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=4.000000 /
vSuma[1]=6.000000 /
vSuma[2]=3.000000 /
vSuma[3]=14.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=14.000000 /
vSuma[1]=8.000000 /
vSuma[2]=18.000000 /
vSuma[3]=10.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo2 4
vSuma[0]=13.000000 /
vSuma[1]=12.000000 /
vSuma[2]=8.000000 /
vSuma[3]=10.000000 /
```

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo1 4
vSuma[0]=14.000000 /
vSuma[1]=20.000000 /
vSuma[2]=26.000000 /
vSuma[3]=32.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo1 4
vSuma[0]=14.000000 /
vSuma[1]=20.000000 /
vSuma[2]=26.000000 /
vSuma[3]=32.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo1 4
vSuma[0]=14.000000 /
vSuma[1]=20.000000 /
vSuma[2]=26.000000 /
vSuma[3]=32.000000 /
```

```
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo1 8
vSuma[0]=140.000000 /
vSuma[1]=168.000000 /
vSuma[2]=196.000000 /
vSuma[3]=224.000000 /
vSuma[4]=252.000000 /
vSuma[5]=280.000000 /
vSuma[6]=308.000000 /
vSuma[7]=336.000000 /
loadge@loadge-K53SV:~/AC/Práctica2/ejercicio9$ ./paralelo1 8
vSuma[0]=140.000000 /
vSuma[1]=168.000000 /
vSuma[2]=196.000000 /
vSuma[3]=224.000000 /
vSuma[4]=252.000000 /
vSuma[5]=280.000000 /
vSuma[6]=308.000000 /
vSuma[7]=336.000000 /
```

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`
    (RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

    RESPUESTA:

    CAPTURAS DE PANTALLA:

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 del aula de prácticas de los tres códigos implementados en los ejercicios anteriores para tres tamaños (N) distintos (consulte la Lección 6/Tema 2).

    TABLA Y GRÁFICA (por *ejemplo* para 1-4 hebras PC aula, y para 1-12 hebras en atcgrid, tamaños-N-: 100, 1.000,

10.000):

-Ejecución en ATGRID con 1 núcleo:

Secuencial: tamaño 10	0.000064 unidades de tiempo
Secuencial: tamaño 1000	0.008622 unidades de tiempo
Secuencial: tamaño 10000	0.619298 unidades de tiempo
Paralelo filas: 10	0.000000942 unidades de tiempo
Paralelo filas: 1000	0.000146720 unidades de tiempo
Paralelo filas: 10000	0.029421273 unidades de tiempo
Paralelo columnas: 10	0.000018009 unidades de tiempo
Paralelo columnas: 1000	0.099953471 unidades de tiempo
Paralelo columnas: 10000	8.848635722 unidades de tiempo

-Ejecución en ATGRID con 12 núcleos:

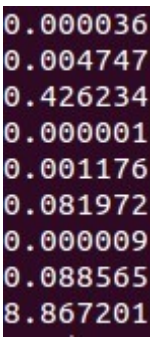
Secuencial: tamaño 10	0.000064 unidades de tiempo
Secuencial: tamaño 1000	0.008579 unidades de tiempo
Secuencial: tamaño 10000	0.620046 unidades de tiempo
Paralelo filas: 10	0.000000911 unidades de tiempo
Paralelo filas: 100	0.000141544 unidades de tiempo
Paralelo filas: 1000	0.029327670 unidades de tiempo
Paralelo columnas: 10	0.000019373 unidades de tiempo
Paralelo columnas: 100	0.099654421 unidades de tiempo
Paralelo columnas: 1000	8.851861363 unidades de tiempo

-Ejecución en el PCLOCAL con 1 núcleo:

Secuencial: tamaño 10	0.000036
Secuencial: tamaño 1000	0.004821
Secuencial: tamaño 10000	0.432979
Paralelo filas: 10	0.000001
Paralelo filas: 1000	0.001419
Paralelo filas: 10000	0.120947
Paralelo columnas: 10	0.000004
Paralelo columnas: 1000	0.016671
Paralelo columnas: 10000	1.648635

-Ejecución en el PCLOCAL con 4 núcleos:

Secuencial: 10
Secuencial: 1000
Secuencial: 10000
Paralelo filas: 10
Paralelo filas: 1000
Paralelo filas: 10000
Paralelo columnas: 10
Paralelo columnas: 1000
Paralelo columnas: 10000



COMENTARIOS SOBRE LOS RESULTADOS:

Los resultados son prácticamente iguales en cada máquina (1 hebra en atcgrid = 12 hebra, y así con el PCLocal). Esto es debido, muy posiblemente, al script utilizado para la ejecución de la batería de pruebas. Con certeza podríamos afirmar que llegados a un determinado número de núcleos/hebra, cada uno de los programas, salvo el secuencial, se vuelve inviable debido al gran coste de comunicación entre hebras, creación, destrucción... Se alcanzaría el máximo beneficio en un número determinado de núcleos.

Script de ejecución en ATCGRID de 12 núcleos/hebras:

```
export OMP_NUM_THREADS=12
echo 'Practicas2/secuencial 10' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/secuencial 1000' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/secuencial 10000' | qsub -q ac

sleep 5

export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo1 10' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo1 1000' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo1 10000' | qsub -q ac

sleep 5

export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo2 10' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo2 1000' | qsub -q ac
sleep 1
export OMP_NUM_THREADS=12
echo 'Practicas2/paralelo2 10000' | qsub -q ac
```

Algo similar para la ejecución con 1 hebra.

Script de ejecución en PCLocal de 4 núcleos/hebras:

```
export OMP_NUM_THREADS=4
./secuencial 10
sleep 1
export OMP_NUM_THREADS=4
./secuencial 1000
sleep 1
export OMP_NUM_THREADS=4
./secuencial 10000

sleep 5

export OMP_NUM_THREADS=4
./paralelo1 10
sleep 1
export OMP_NUM_THREADS=4
./paralelo1 1000
sleep 1
export OMP_NUM_THREADS=4
./paralelo1 10000

sleep 5

export OMP_NUM_THREADS=4
./paralelo2 10
sleep 1
export OMP_NUM_THREADS=4
./paralelo2 1000
sleep 1
export OMP_NUM_THREADS=4
./paralelo2 10000
```